TEXbook

naruby

Toto je text knihy v „klikací“ verzi ve formátu PDF. Další informace o knize naleznete na adrese http://math.feld.cvut.cz/olsak/tbn.html.

Verze tohoto textu přesně odpovídá textu druhého vydání knihy v nakladatelství Konvoj, spol. s r. o., Berkova 22, 612 00, Brno. Knihu je možné u tohoto nakladatelství objednat, viz http://www.konvoj.cz, e-mail: konvoj@konvoj.cz, tel./fax: +420 5 740233.

Jsem si vědom některých nedostatků, které zůstaly ve formátu PDF neodstraněny. Systém záložek, který obsahuje strukturovaný obsah, je zapsán česky, což se nemusí na všech implementacích Acroreaderu správně zobrazovat. Přesto zůstává systém záložek celkem čitelný a tedy použitelný. Podrobněji o problematice „klikací“ verze této knihy, viz článek *Jak jsem dělal knihu klikací* například v souboru `tbnklik.tex`.

PDF formát knihy byl připraven pomocí volně dostupné modifikace TEXu `pdftex`. Viz například http://www.cstug.cz/pdftex/.

TEXbook naruby

Petr Olšák

Konvoj, CSTUG

Brno 2001

Autorem programů TEX a METAFONT je profesor Donald Knuth.

TEX je ochranná známka American Mathematical Society.

METAFONT je ochranná známka Addison Wesley Publishing Company.

Ostatní v knize použité názvy programových produktů, firem apod. mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Kniha vyšla za přispění Československého sdružení uživatelů TEXu.

KATALOGIZACE V KNIZE – NÁRODNÍ KNIHOVNA ČR

Olšák, Petr
TEXbook naruby / Petr Olšák. – 2. vyd. – Brno : Konvoj
2001. – 468 s.
ISBN 80-7302-007-6

004.42TEX * 004.912 * 655.26
• TEX
• počítačová typografie
• příručky

ISBN 80-7302-007-6 (2. vyd.)
ISBN 80-85615-64-9 (1. vyd.)

Úvod

TEX je volně dostupný počítačový program na pořizování vysoce kvalitní elektronické sazby. Tento program se od svých stejně zaměřených komerčních kolegů liší v mnoha ohledech. Mezi nejpodstatnější odlišnost zřejmě patří *otevřenost* systému. Šikovnému uživateli se v takovém systému dostává do rukou nástroj na vytvoření nadstavby podle vlastní představy a potřeby. K tomu je ale potřeba o vlastnostech systému poměrně hodně vědět.

Chcete-li nahlédnout pod pokličku TEXovského hrnce a postupně se podívat až na jeho samotné dno, abyste se dozvěděli, co se v něm vaří, je tato kniha určena pro vás. Na každém pracovišti, kde se používá TEX, by měl být aspoň jeden odborník, který TEXu na této úrovni rozumí. Ostatní spolupracovníci pak mohou postupovat podle jednoduchých příruček typu „napíšete-li `\alpha`, dostanete na výstupu α“. Pokud si tito spolupracovníci nebudou vědět s něčím rady, mají ve svých řadách odborníka, který jim poradí. Idea, že se všechno zvládne pomocí LATEXových příruček, se při náročnějších požadavcích na sazbu většinou rozplyne před očima jako jarní sníh.

Referenční manuál k TEXu se jmenuje *The TEXbook* [4]. Tato základní kniha o TEXu není mezi našinci příliš rozšířena zvláště pro její vysokou cenu. Pro mnohé je navíc tato kniha dosti obtížná ke čtení, protože TEX není nic jednoduchého. Pokusil jsem se tedy napsat k tomuto základnímu manuálu alternativní text. Text ale není jen odvarem TEXbooku. Užitečné asi budou zcela původní příklady řešené přímo v textu. Nikdy jsem se nesnažil o doslovný překlad žádné pasáže z TEXbooku, ba dokonce jsem do tohoto manuálu při psaní nahlížel jen výjimečně a snažil se celou problematiku podat svými slovy podle svých vlastních zkušeností. Měl jsem potom radost, pokud se mi něco podařilo říci přehledněji nebo stručněji než v TEXbooku. Představil jsem si, že text píši pro čtenáře, který již má základní zkušenosti s pořizováním jednoduchých textů a který tuto knihu otevírá třeba proto, že chce v TEXu programovat složitější makra. Rozhodl jsem se tedy výklad obrátit. V TEXbooku se začíná popisem použití již hotových maker plainu a teprve potom, podstatně později, se čtenář dozví, na jakém principu jsou tato makra postavena. Naproti tomu, v této knize je výklad veden „naruby“. Nejprve je vždy popsán vnitřní algoritmus TEXu a vlastnosti souvisejících primitivů a potom jsou tyto vlastnosti ilustrovány na příkladech maker. Z této koncepce pramení název knihy *TEXbook naruby*.

Tato kniha není příručkou uživatele, který si chce pouze osvojit základní metody při pořizování textů pro TEX. K tomu účelu slouží mnoho jiných učebnic. Je tedy patrné, že kniha není určena začátečníkovi. Ke zvládnutí tohoto textu by ale měly stačit znalosti získané četbou *Jemného úvodu do TEXu* [3]. Užitečné informace lze též čerpat z knížky *Typografický systém TEX* [7], která na rozdíl od této knihy není úzce specializovaná na TEX samotný, ale rozebírá „softwarové souvislosti programu“. Tam čtenář najde rozbor práce TEXu s virtuálními fonty, s PostScriptem, s obrázky a například se dozví, jak vypadá jazyk WEB zdrojového textu TEXu.

Text knihy, kterou právě otvíráte, je rozdělen na dvě části. Část A (Algoritmy) popisuje v jednotlivých kapitolách a sekcích systematicky všechny algoritmy TEXu. Část B (Reference) obsahuje slovník všech primitivů a maker plainu, přičemž u každého hesla je poměrně rozsáhlý vysvětlující text. Představte si rejstřík z TEXbooku v dodatku I a nechť každé heslo z tohoto rejstříku „expanduje“ na zhruba půlstránkový výklad o heslu. Pak dostáváte část B této knihy.

Obě části knihy jsou bohatě ilustrovány příklady. Jedná se především o ukázky z formátu plain a dále o úlohy, které jsem v době své praxe s TEXem někdy řešil a které mohou být pro čtenáře zajímavé. Aby čtenář nemusel ručně opisovat do svého počítače texty ukázek, které ho zaujmou, vystavuji na síti Internet soubor `tbn.mac`. V něm jsou veškeré ukázky z této knihy obsaženy. Každý zájemce si tento soubor může zkopírovat z adresy `ftp://math.feld.cvut.cz/pub/olsak/tbn/`.

Nikdo není neomylný. I při sebevětší péči se neubráním možnému zavlečení chyb do textu. Budu proto na stejné adrese v Internetu průběžně obnovovat soubor `errata.txt`, ve kterém budou zaneseny všechny chyby, o kterých vím a které nemusí být ještě opraveny v tomto textu. K odhalení a zveřejnění chyb může přispět kterýkoli čtenář, pokud mi o chybě napíše na adresu `olsak@math.feld.cvut.cz`.

Při psaní této knížky se též objevilo plno jazykových problémů. Jazykoví puristé mi doufám odpustí, že používám termíny jako token, expand procesor, box apod. Mohl bych sice mluvit o amuletu, proceduře na rozklad balíčků povelů a krabičce. Myslím si ale, že by takový text vzhledem k návaznosti na anglickou literaturu nebyl příliš čitelný. My také klikáme myší a díváme se na video, místo abychom poklepávali po tlačítku polohovacího zařízení a pouštěli magnetoskop. S jazykem to je těžké.

Chtěl bych ze srdce poděkovat především své ženě Ludmile, která mi byla velkou oporou v dobách, kdy jsem nevnímal svět kolem sebe a myšlenkami jsem byl u svých příkladů `\output` rutin. Má žena pomáhala též organizaci $\mathcal{CS}$TUGu v době, kdy měla tato organizace poměrně velké problémy. Dále děkuji všem dobrovolným korektorům, kteří se přihlásili v hojném počtu, za podnětné připomínky při závěrečných úpravách textu.

Přeji všem čtenářům této knihy mnoho hezkých zážitků se skvělým programem, který vznikl v dílně profesora Donalda Knutha jako „labour of love“ (práce z radosti) a byl poskytnut veřejnosti zdarma.

28. 9. 1996 *Petr Olšák*

29. 11. 2000. Ve druhém vydání jsou jen opraveny chyby podle `errata.txt`. Děkuji touto cestou všem čtenářům, kteří přispěli k odhalení chyb a tím k vylepšení kvality knihy. Snažil jsem se důsledně zachovat stránkování i číslování, aby případné odkazy na knihu, které jste již dříve použili ve svých makrech, zůstaly v platnosti.

Obsah

Část A

Algoritmy

1. Vstupní části TEXu

1.1. Koncept vstupní brány TEXu

Vstupní soubor TEXu je textový a obsahuje jednotlivé znaky, které se mají vysázet. Mimoto vstupní soubor obsahuje tzv. *řídicí sekvence* (anglicky control sequences). Řídicí sekvence určují způsob, jakým se má sazba provést. Například řídicí sekvence `\TeX` způsobí vysázení loga „TEX“. Při startu (kdy není načten formát) je v TEXu implementováno zhruba 300 *primitivních* řídicích sekvencí (zkráceně *primitivy*). Kromě toho se dají deklarovat nové řídicí sekvence (například prostřednictvím primitivu `\def`), které nahrazují většinou skupiny jiných řídicích sekvencí. Nově deklarovaným řídicím sekvencím říkáme *makra*. Například řídicí sekvence `\TeX` je makro, které se opírá o primitivy `\hbox`, `\kern` a `\lower` (viz část B).

Formáty TEXu (například plain, LATEX atd.) deklarují nové řídicí sekvence, které jsou většinou uživatelsky mnohem přijatelnější než přímé použití primitivů. Autor textu se pak nemusí vyjadřovat jen pomocí primitivů (to by bylo asi příšerně komplikované) a není nucen si všechna potřebná makra v úvodu svého textu deklarovat sám. Vesměs všechny formáty vycházejí ze základního formátu plain, který je popsán v TEXbooku. I v této knize se zaměříme na tento formát. Všechna makra plainu jsou popsána v části B. Pokud chceme začít psát vlastní makra, bude pro nás formát plain dobrým startovním bodem, o který se jednak můžeme opřít a jednak v něm můžeme hledat inspiraci. □

TEX na svém vstupu čte textový soubor a na výstupu vzniká (mimo jiné) binární soubor `dvi`. Pro dobré pochopení jednotlivých funkcí TEXu je užitečné rozdělit cestu datových struktur od vstupního textového souboru do výstupního `dvi` na úseky. TEX jako program tím pomyslně rozdělíme na samostatné procesory, které si předávají mezi sebou konkrétním způsobem definované datové struktury. Knuth přirovnal tento proces zpracování vstupní informace k „trávení“ TEXu a mluví o očích, ústech, hltanu, žaludku atd. My budeme mluvit o input procesoru, token procesoru, expand procesoru a hlavním procesoru.

Input procesor čte jednotlivé řádky vstupního souboru a zpracovává je do podoby řádků, které jsou nezávislé na použitém operačním systému.

Token procesor čte zpracované řádky z input procesoru a vytváří posloupnost *tokenů*, což jsou buď řídicí sekvence nebo jednotlivé znaky nesoucí svou „kategorii“ (pojem zavedeme v sekci 1.3). Výstupní posloupnost tokenů už není členěna na řádky.

Expand procesor čte jednotlivé tokeny a rozlišuje mezi tokeny, které bude expandovat a které ponechá beze změny. Beze změny ponechá například jednotlivé znaky. Také primitivní řídicí sekvence zůstávají (až na výjimky) beze změny. Naopak procesor expanduje všechny řídicí sekvence, které byly deklarovány jako makra. Expandování tokenu znamená (zhruba řečeno) záměnu tokenu za posloupnost jiných tokenů. V nové sekvenci tokenů mohou některé podléhat expanzi znova, tj. proces expanze se opakuje tak dlouho, až jsou všechny tokeny primitivními řídicími sekvencemi nebo jednotlivými znaky, které nepodléhají expanzi.

Hlavní procesor rozlišuje na svém vstupu přesně definovanou množinu tzv. povelů. Například samostatný znak znamená povel na vysázení tohoto znaku. Nebo primitivní řídicí sekvence se stává v hlavním procesoru povelem k vykonání konkrétního úkonu souvisejícího se sazbou. Takové povely mohou mít své parametry, které přicházejí za povelem. Například v zápise `\vskip2mm` považujeme `\vskip` za povel a `2mm` za jeho parametr. Hlavní procesor vytváří vlastní sazbu, provádí výstup do `dvi` a rovněž nastavuje prostřednictvím povelů hodnoty registrů, jež ovlivňují algoritmy na všech úrovních zpracování TEXu.

Jednotlivé procesory spolu úzce spolupracují a jejich činnost je provázána. Nelze tedy považovat za správnou představu, při které nejprve celý vstupní soubor projde input procesorem, pak token procesorem atd. Změny parametrů provedené v hlavním procesoru (například `\endlinechar`, `\catcode`, nová makra deklarovaná pomocí `\def`) okamžitě mohou ovlivnit následující činnost ostatních vstupních procesorů. Proto je v každém okamžiku jednotlivým procesorem zpracováno jen tolik informace, kolik bylo pro uspokojení požadavku následujícího procesoru nezbytně potřeba. Lze si představit, že hlavní procesor řídí celou činnost. Vyžádá si od expand procesoru povel. Expand procesor si vyžádá od token procesoru jeden token, který případně expanduje. Token procesor při vytváření tokenu musí mít ve vstupním bufferu řádek vstupního textu, který byl připraven input procesorem. Nový řádek načte input procesor až v okamžiku, kdy je to nezbytně nutné.

Vstupní bránu hlavního procesoru budeme též označovat jako *čtecí fronta*. Jedná se o místo v paměti, kam přichází výstup z expand procesoru a odkud si bere hlavní procesor povely a parametry. Později ukážeme, že hlavní procesor za určitých okolností vrací načtený povel do čtecí fronty (jakoby proti proudu dat), pak udělá nějakou konkrétní činnost a potom znovu čte ze čtecí fronty odložený povel.

Přesněji si o jednotlivých procesorech povíme v následujících sekcích. □

1.2. Input procesor

Input procesor čte ze vstupního souboru postupně řádky textu, upravuje je a na jeho výstupu jsou znovu řádky textu připravené pro token procesor. Pojmem *řádek textu* na vstupu rozumíme konkrétní část textového souboru, která je (bohužel) v různých operačních systémech odlišně definována. Řádek textu na výstupu z input procesoru je již vnitřní datová struktura TEXu, která je stejná ve všech implementacích TEXu. Algoritmy input procesoru jsou tedy závislé na systému a starají se o odstínění zvláštností jednotlivých operačních systémů tak, aby všechny ostatní algoritmy TEXu již mohly být implementovány nezávisle na operačním systému.

Input procesor postupně provádí se vstupními řádky textu dvě aktivity:

- Případně překóduje obsah řádku z kódování operačního systému do ASCII.
- Upraví znak konce řádku smluveným způsobem.

1. Překódování. Představme si dvě implementace TEXu; jednu na operačním systému s kódováním ASCII a druhou na systému s kódováním Pišvejcovým. Pokud si uživatelé předávají mezi těmito systémy textové soubory, provádějí konverze mezi těmito dvěma kódováními. Proto uživatel A pracující se systémem s ASCII vidí v textovém editoru totéž, co uživatel B, pracující se systémem podle Pišvejce. Uvažujme, že se oba dívají na tento text:

```
1 \catcode64=13 % Znak "@" má ASCII 64 a bude mít kategorii 13
2 Zde je použit @.
```

Kdyby input procesor neprovedl konverzi z Pišvejce do ASCII, pak by se tento zdrojový text mohl chovat ve zmíněných implementacích TEXu různě, ačkoli oba uživatelé vidí ve svém editoru totéž. Knuth tedy rozhodl, že vnitřní kódování TEXu bude podle ASCII a vše se do tohoto kódování bude konvertovat na úrovni input procesoru. Když v TEXu řekneme `\catcode64`, pak je jednoznačně jasné, co se tím myslí. Poznamenejme, že místo zápisu `\catcode64` asi použijeme `\catcode‘\@`, takže pak by zmíněný problém nenastal. Nastal by ovšem na úrovni textových fontů TEXu, které se obvykle ve větší části kryjí s ASCII kódem. Například při výskytu písmene `A` v „obyčejném textu“ TEX vysází znak aktuálního fontu z pozice, která odpovídá kódu tohoto písmene. Musíme tedy přesně vědět, jaký to je kód. TEX má přitom ve svých instalacích své vlastní fonty společné všem instalacím TEXu a nezávislé na operačním systému.

2. Úprava konce řádku. Textový soubor se člení na řádky. Oddělovače mezi jednotlivými řádky jsou definovány v různých operačních systémech různě. V systémech dnes už historických měly například všechny řádky stejnou délku (třeba

80 znaků) a byly zprava doplňovány mezerami. Dnešní systémy mají vesměs definován jeden znak nebo skupinu znaků jako oddělovač mezi řádky. Například v systémech typu UNIX je tímto oddělovačem znak s kódem ASCII 10 (LF, Ctrl-J), v DOSu se používá dvojice CR LF (Ctrl-M Ctrl-J) a na systémech počítačů Macintosh je oddělovačem pro změnu jen CR. Pokud si uživatelé mezi těmito systémy vyměňují textové soubory, musí provádět konverze. □

Input procesor postupně načítá řádky textu ze vstupního souboru tak, jak je tento pojem definován v použitém operačním systému. Pak provede případné překódování do ASCII a odstraní případnou značku konce řádku definovanou systémem. Dále odstraní z konce řádku (tedy zprava) všechny případné mezery (ASCII 32, nikoli token mezera) až po první znak, který není mezerou. Konečně připojí na konec řádku znak, definovaný v registru `\endlinechar`. Teprve takto upravený řádek vstupuje do token procesoru.

IniTEX nastavuje `\endlinechar` na hodnotu `^^M`, která ve formátech obvykle nebývá měněna. Symbolem `^^M` označujeme kód Ctrl-M, což je ASCII 13 neboli CR. K pochopení funkce `\endlinechar` si vyzkoušíme:

```
{ \endlinechar=‘\*  Tady je první řádek,
  tady druhý,
  tu třetí}
a konečně poslední.
```

Na výstupu dostaneme:

Tady je první řádek, tady druhý,*tu třetí*a konečně poslední.

Vidíme, že za první řádek byla připojena ještě původní hodnota `\endlinechar`, která je rovna `^^M` a která na úrovni token procesoru vyprodukuje (obvykle) mezeru. Je to z toho důvodu, že v okamžiku, kdy hlavní procesor provede nové přiřazení do registru `\endlinechar`, byl již input procesorem celý první řádek načten a kompletován. Input procesor se tedy začne chovat jinak až od druhého řádku. Jakmile hlavní procesor na třetím řádku uzavře skupinu (`}`), registr `\endlinechar` se vrátí k původní hodnotě, ovšem input procesor už na konec třetího řádku hvězdičku připojil. Teprve na čtvrtém řádku bude vše v původním stavu. □

Uvedeme příklad na využití `\endlinechar`. Nechť máme soubor (například `data.txt`), ve kterém jsou data systematicky členěna do řádků. Třeba údaje pro vyplnění formuláře vysvědčení:

```
Ferdinand Mravenec
Údolní 14
1 1 1 2 1 1 3 2 2 1 1

```

```
Karel Mařík
bratří Maříků 13
1 2 2 1 2 4 3 3 2 1 2
... atd.
```

Vidíme, že data jsou členěna do trojic řádků. První řádek obsahuje jméno, druhý ulici a třetí známky. Vidíme též, že zde není jediná TEXovská značka (řídicí sekvence) a nám se nechce žádné značky do souboru dopisovat. K načtení takového souboru TEXem tak, aby zůstala zachována členěná struktura, můžeme použít třeba toto makro:

```
\def\jmeno#1{{\it Jméno\/}: #1\par}
\def\ulice#1{{\it Ulice\/}: #1\par}
\def\znamky#1{{\it Známky\/}: #1\par}
\catcode`\*=13 \let*=\space
\def\udaj#1*#2*#3*{{\everypar={}\jmeno{#1}\ulice{#2}\znamky{#3}
    \vfil\eject}}
\endlinechar=`\* \everypar={\udaj}
\input data.txt
```

Trik s `\everypar` vyžaduje znalost módů hlavního procesoru, takže přesně se o něm zmíníme až v sekci 3.4. Zde jen rámcově: Jakmile TEX narazí na první písmeno, které by se mělo sázet (písmeno F v našem souboru `data.txt`), spustí se `\everypar`, tj. v tomto případě se provede makro `\udaj`. Všimneme si, že konec každého řádku máme ohraničen hvězdičkou, protože `\endlinechar` má hodnotu hvězdičky. Hvězdička je navíc aktivní (kategorie 13) a kdyby náhodou na ni přišla řada, bude pracovat jako mezera (`\space`). Takže nám nebude na koncích řádků překážet. Teď už vidíme, proč makro `\udaj` nabere do parametru `#1` první řádek se jménem, do `#2` druhý řádek s ulicí a do `#3` třetí řádek se známkami. Makra `\jmeno`, `\ulice`, `\znamky` jsou zde pro jednoduchost uvedena jen v „ladicí podobě“. Prakticky by tato makra měla řešit přesné usazení textu do konkrétního místa na formuláři. V případě `\znamky` by se asi použil cyklus přes všechny známky (viz sekci 2.3) a jednotlivé číslovky by se nahradily slovy „výborný“, „chvalitebný“ atd. V tuto chvíli není účelem zde takové věci předvádět. □

Pokud uděláme ve vstupním souboru změny, které v obvyklých textových editorech nejsou vidět, měl by se TEX chovat pokud možno stejně. K tomu účelu je potřeba udělat trochu práce na úrovni formátu. Pokud třeba uživatel použije tabelátor místo mezery (nebo skupiny mezer), většinou to v editoru explicitně nevidí. *Tabelátor* je znak (ASCII 9), který v některých editorech způsobí vložení „smluveného“ množství mezer. Plain definuje kategorii tabelátoru shodně s kategorií mezery. Také primitivní sekvence `\␣` (pro explicitní mezeru) se v plainu ztotožňuje jednak s `\^^I` (čti „backslash tabelátor“) a také s `\^^M` (čti „backslash CR“). Pak se tedy sekvence `\␣` chová stejně uvnitř řádku i na konci řádku. Uvedeme si příklad, ve kterém předefinujeme chování uvedených řídicích sekvencí:

```
\def\ {SPACE} \def\^^M{CR}
A nyní vyzkouším mezeru\   a také mezeru na konci\␣␣
```

Na výstupu máme: „A nyní vyzkouším mezeruSPACEa také mezeru na konciCR“. Vidíme, že ačkoli jsme za posledním znakem „\“ výslovně napsali dokonce dvě mezery (tato skutečnost je vyznačena pomocí vaničky) a případně jsme si překontrolovali binárním prohlížečem, že tam ty mezery skutečně máme, nepomohlo to. Input procesor totiž po odebrání znaku konce řádku zlikviduje tyto mezery a teprve pak přidá `\endlinechar`. Kdybychom ale napsali

```
... a také mezeru na konci\␣␣⟨tabelátor⟩
```

pak skutečně dostaneme: „...mezeru na konciSPACE“. Input procesor totiž přidá `\endlinechar` za znak ⟨*tabelátor*⟩, protože to není mezera. Skutečnost, že ⟨*tabelátor*⟩ má stejnou kategorii jako mezera, nemá pro input procesor žádný význam. □

Uvedené příklady se mohou zdát extrémně neužitečné. Na druhé straně je velmi užitečné přesně vědět, co input procesor vlastně provádí. Může se třeba stát, že jsme převzali textový soubor pořízený v DOSu (na konci řádku jsou CR LF) a bez konverze jsme jej použili v TEXu na UNIXu. Co se stane? Input procesor odebere z každého řádku LF (to je značka konce řádku v UNIXu), ponechá CR a přidá `\endlinechar`, což bývá obvykle `^^M`, neboli jinými slovy CR. Do token procesoru tedy vstupuje na konci každého řádku CR CR (tj. `^^M^^M`). Kategorie znaku `^^M` je většinou nastavena na 5, což v token procesoru způsobí ukončení zpracování vstupního řádku při prvním výskytu `^^M` a vložení mezery. K druhému výskytu `^^M` se token procesor vůbec nedostane. UNIXový TEX se tedy chová stejně, jako kdyby byl textový soubor pořízen přímo na UNIXu. Ale pozor! Pokud změníme kategorii znaku `^^M`, začne se UNIXový TEX na našem souboru chovat odlišně. Přestává být jedno, zda máme na koncích řádků CR nebo CR CR. K takové situaci dojde například ve verbatim prostředí. Tam je znak `^^M` nastaven jako aktivní a je definován jako makro pro přechod na nový řádek. Druhé `^^M` v každém vstupním řádku nám tedy vloží do výstupu ve verbatim prostředí prázdný řádek.

Některé DOSové editory způsobují UNIXovému TEXu ještě jednu starost. Tvrdohlavě vkládají na konec souboru znak `^^Z`. Uvedeme doporučený postup, kterým naučíme UNIXový TEX číst DOSem připravené soubory. Na úrovni algoritmu překódování (o něm se zmíníme později v této sekci) provedeme konverzi znaku `^^M` na `^^Z` a do formátu přidáme řádeček:

```
\catcode`\^^Z=9 % Znak ^^Z bude ignorován
```

Potom na konci každého řádku budeme mít `^^Z` (to je konvertované CR z DOSového konce řádku), za něj input procesor přidá `^^M` z `\endlinechar`, které už nepodléhá

konverzi. Na konci řádku tedy máme dvojici: `^^Z^^M`. Token procesor bude znak `^^Z` ignorovat a se znakem `^^M` naloží obvyklým způsobem.

Dodejme, že v této věci se DOSový TEX chová podstatně lépe (mluvím o emTEXu). Tam je v input procesoru naprogramováno, že konec řádku v systému může končit buď CR LF nebo jen LF. Obě situace vstupní procesor interpretuje správně, tj. odstraní CR LF, případně jen LF. Po odstranění mezer na konci přidá `\endlinechar`. Je tedy naprosto jedno, zda DOSovým TEXem zpracováváme soubory pořízené v DOSu nebo na UNIXu. UNIXové soubory poznáme v *některých* DOSových editorech podle toho, že vidíme jen jeden hrozně dlouhý řádek, který v sobě obsahuje znaky LF, promítané na obrazovce většinou jako tmavý obdélníček s kolečkem. □

Výše zmíněné problémy nejsou bohužel nejpodstatnější. Horší je, že nové textové editory berou do svých rukou i starosti o formátování. Na základě toho si definují své vlastní značky pro konec řádků. Jistě si vzpomenete na mnoho DOSových editorů, které aspirují na textový procesor nebo dokonce DTP a rozlišují mezi tzv. „tvrdým“ a „měkkým“ koncem řádku. „Tvrdým“ koncem řádku je přitom skutečně míněn ten konec řádku, který je definován v systému. „Měkký“ konec řádku je značka dosti často závislá na konkrétním editoru.

V této záležitosti šel bohužel vývoj jiným směrem, než odpovídá velmi dobré koncepci z TEXu, kde konec řádku podléhá dalšímu formátování a konec odstavce se zapíše na vstupu jako prázdný řádek. V editorech, které používají dva druhy značek pro konec řádku, jsou „tvrdé“ konce řádku míněny jako konec odstavce a „měkké“ konce řádku podléhají dalšímu formátování. Tedy úplně jinak než v TEXu.

Pokud je „měkký“ konec řádku například skryt pod kódem `"8D` (hexadecimálně), pak můžeme zkusit na začátek dokumentu uvést:

```
27 \catcode"8D=5 % Měkký konec řádku bude konec řádku
28 \catcode`\^^M=13 \def^^M{\par}  % Tvrdý konec ukončí odstavec
```

Například program T602 vytváří v místě měkkých konců řádků dvojici `"8D` (hexa) následovanou znakem LF. V takovém případě náš kód umožní načítat texty pořízené programem T602, který je již dlouhou dobu v našich končinách nejužívanějším prostředkem pro pořizování textů na počítači. Samozřejmě je potřeba „odkrojit“ hlavičku, kterou si program vytváří na začátku souboru. Hlavička má na začátku každého řádku zavináč. Takže stačí v TEXu před načtením souboru označit tento znak jako komentářový: `\catcode`\@=14`.

Uvedené řešení může narazit na problémy. Pokud je totiž vstupní znak s kódem `"8D` na úrovni input procesoru překódován, ve vnitřním kódování TEXu už vystupuje pod úplně jiným kódem. Například při transformaci z kódování Kamenických do vnitřního kódu TEXu je znak s kódem `"8D` překódován na znak Í, tedy slovenské l s čárkou. Nepracujeme-li se slovenskými texty, nemusí nám to vadit. Pak ovšem

místo řádku 27 pišme `\catcode‘\Í=5`. Poznamenejme, že u kódu PC Latin2 ani ISO8859-2 není znak `"8D` transformován, takže bychom neměli mít problémy. Ovšem jen do té doby, než autor v programu T602 použije nějakou funkci, která tento procesor odlišuje od obyčejného ASCII editoru; například zvolí jiný druh písma. Tím vznikají ve zpracovávaném souboru další privátní značky textového procesoru.

Bývá obvyklejší při zpracování dokumentů z programu T602 a jemu podobných provést nejprve záměnu všech „tvrdých“ konců řádků za dva „tvrdé“ a všech „měkkých“ za jeden „tvrdý“. To můžeme provést před zpracováním v TEXu například na úrovni inteligentního editoru nebo ještě lépe UNIXového filtru využívajícího například dávkový editor `sed`. Také můžeme ošetřit výskyty přepínačů druhu písma apod. Dá se totiž očekávat, že když pracujeme s TEXem, pak nepracujeme s programem T602. Proto nám bude stačit provést konverzi jen jednou — v okamžiku, kdy obdržíme rukopis od autora textu. □

Nyní si povíme o problematice překódování vstupu na úrovni input procesoru TEXu, což je pro nás v případě češtiny a rozličných operačních systémů záležitost bytostně důležitá.

Knuth rozhodl, že vnitřní kódování TEXu je ASCII a všechny ostatní kódy se do tohoto kódu budou transformovat na úrovni input procesoru. V souladu s tímto rozhodnutím též připravil fonty Computer Modern, které na toto ASCII kódování navazují. Svým rozhodnutím ovšem definoval jen prvních 128 pozic v kódu a významy znaků s kódem nad 128 nechal nedefinovány. Původní verze TEXu (do r. 1988) dokonce ani tyto znaky na vstupu nedovolovala. Dnes je volba vnitřního kódování znaků v TEXu nad hranicí 128 v rukou implementátorů národních instalací TEXu. Rozhoduje většinou kód základních národních fontů, použitých v TEXovské instalaci. Pokud mají další fonty odlišné kódování, implementují se do instalace prostřednictvím tzv. „virtuálních fontů“ (viz [7]).

V DOSu v emTEXu je překódování na úrovni input procesoru implementováno pomocí tzv. `tcp` tabulek, které se načítají do TEXu při inicializaci formátu. Jakmile je formát vytvořen, je v něm už konverzní tabulka zahrnuta. V $\mathcal{CS}$TEXu se používají formáty s `tcp` tabulkami, které konvertují například z kódování Kamenických, nebo z kódování PC Latin2 nebo jiného Pišvejcova kódování do kódu $\mathcal{CS}$-fontů. Prvních 128 pozic (základní ASCII) je ponecháno beze změny.

V UNIXu je potřeba překódovací algoritmus začlenit do zdrojového textu TEXu před kompilací programu. Většinou se to dělá pomocí změnového souboru k `tex.web`. Problematika se týká sekcí 20 až 24 ve zdrojovém kódu TEXu a jedná se o vektory `xord` a `xchr`. V případě, že se ponechá vnitřní kódování TEXu podle $\mathcal{CS}$-fontů, pak není obvykle potřeba dělat žádné zásahy do zdrojového kódu TEXu, protože čeština v UNIXu je kódována výhradně podle ISO 8859-2 a $\mathcal{CS}$-fonty na toto kódování navazují. Je to analogie Knuthova postupu, kdy byl pro

angličtinu zvolen vnitřní kód ASCII a kódování Computer Modern fontů na tento (sedmibitový) kód navazovalo.

V UNIXu lze též použít „patch“ pana Škarvady. Jeho algoritmy doplňují jinak standardní UNIXový změnový soubor pro `tex.web`. Po kompilaci TEXu je pak možné volit mezi různými kódovacími postupy prostřednictvím nastavení systémové proměnné. Není-li příslušná systémová proměnná nastavena, TEX neprovádí žádnou konverzi, což (jak již jsme uvedli) vyhovuje vnitřnímu kódu TEXu podle $\mathcal{CS}$-fontů a vstupnímu kódu podle ISO 8859-2. □

Algoritmy input procesoru jsou v TEXu implementovány i v „inverzní“ podobě. Pokud TEX potřebuje něco zapsat na terminál, do souboru `log` nebo do jiného souboru prostřednictvím `\write` nebo `\message`, jsou v činnosti tyto inverzní algoritmy. Jednotlivé řádky jsou na výstupu členěny způsobem, kterému rozumí příslušný operační systém. Provádí se samozřejmě též případné zpětné překódování. Ačkoli tedy v TEXu probíhá vše podle ASCII, na terminálu a v souborech `log`, `aux` apod. čteme při použití systému s Pišvejcovým kódováním veškeré texty podle Pišvejce.

Pokud se v textu pro `\write` použije znak s kódem shodným s obsahem registru `\newlinechar`, pak se na výstupu místo tohoto znaku ukončí řádek a výstup dále pokračuje na novém řádku. Například:

```
29 \newlinechar`\^^J \immediate\write16{Druhý^^Jřádek}
```

Jedná se tedy o jakýsi „protějšek“ k registru `\endlinechar`.

Doporučuje se, aby všechny netisknutelné znaky, které vystupují na terminál nebo do souborů `log`, `aux` apod. byly nahrazeny sekvencí `^^`⟨*něco*⟩, například `^^Z` nebo `^^e1`. (Podrobněji o tomto formátu, viz následující sekci o token procesoru.) Tím máme zaručeno, že se terminály nezačnou chovat po obdržení nějakého řídicího znaku nedefinovaným způsobem. Mezi lidem počítačovým se takovému jevu říká „rozsypaný čaj“. Proto je rozumné, že například veškeré řídicí znaky z tabulky ASCII vystupují na terminál tímto způsobem. Třeba při tisku znaku ASCII 7 (BELL, česky zvonek) nám nebude terminál pískat, ale objeví se text `^^G`. Při opětovném načtení (například ze souboru `aux`) si TEX na úrovni token procesoru zpětně konvertuje sekvenci `^^G` na ASCII 7. Takže žádný problém.

Problém ovšem nastává, pokud si chceme přečíst na terminálu nebo v souborech `log`, `aux` český text. Například místo slova „řádek“ si přečteme „`^^f8^^e1dek`“, což není příliš čitelné. Samozřejmě, TEXu to nevadí. Ten si takový text při zpětném načtení překonvertuje. Ale jak k tomu přijde člověk? Zde je tedy třeba před kompilací TEXu ze zdrojového textu `tex.web` rozhodnout, zda operační systém snese na terminálu znaky s kódy nad 128 a nedojde k efektu „rozsypaný čaj“. Pokud ano, pak je možné (zásahem do změnového souboru k `tex.web`, sekce 49) potlačit

výstup znaků nad 128 ve formátu `^^`⟨*něco*⟩ a místo toho podporovat formát přímý. To je výhodné i pro TEX samotný. Stačí totiž předefinovat kategorii znaku „^“ a TEX po sobě texty v pomocných souborech obsahující `^^`⟨*něco*⟩ nepřečte.

V případě emTEXu v DOSu je přímý výstup znaků nad 128 do souborů typu `log` podporován. Stejně je tomu v UNIXu po použití zmíněného „patch“ pana Škarvady. Řídicí znaky ASCII (typu BELL) samozřejmě zůstávají ve formátu pomocí „`^^`“. □

1.3. Token procesor

Každý znak má v TEXu svoji *kategorii*, což je celé číslo v intervalu ⟨0, 15⟩. Token procesor bere ze vstupu řádky upravené input procesorem a na jeho výstupu je posloupnost tzv. *tokenů*. Token (čti toukn, přeložili bychom jako symbol, známka, znak, žeton, ale překládat nebudeme) je buď uspořádaná dvojice (ASCII kód, kategorie), nebo řídicí sekvence. Ve všech algoritmech, které následují za token procesorem se už nikdy nemluví o znacích vstupního textu, ale pouze o tokenech.

Kdybychom chtěli vyložit chování token procesoru uživatelům, kteří se nechtějí stát programátory maker, mohli bychom pojem „token“ zatajit a uvést jednoduše následující vlastnosti:

- Konec řádku je brán jako mezera a přechází se na další řádek.
- Od znaku `%` až do konce řádku je vše ignorováno (tzv. komentář).
- Mezeru z konce řádku lze „zamaskovat“ komentářem (`%`).
- Více mezer vedle sebe se rovná jedné mezeře.
- Mezery zleva na řádku jsou zcela ignorovány.
- Prázdný řádek ukončuje odstavec.
- Řídicí sekvence jsou tvaru `\`⟨*identifikátor*⟩.
- ⟨*identifikátor*⟩ obsahuje buď písmena, nebo jen jediný jiný znak.
- Za sekvencí typu `\slovo` jsou všechny následující mezery nevýznamné.
- Za sekvencí typu `\$` je mezera významná.

Čtenář této knížky ovšem nebude s tímto výkladem spokojen. Při programování maker TEXu se totiž bez pojmu „token“ neobejde. Než se pustíme do přesného popisu činnosti token procesoru (jedná se o stavový automat), nastíníme hrubou ideu. Token procesor plní tyto úkoly:

- Jednotlivé řídicí sekvence interpretuje jako jeden token (např. `slovo`).
- Skupiny mezer interpretuje jako jeden token „mezera“ (␣$_{10}$).
- Prázdný řádek převádí do tokenu `par`.
- Ostatní znaky se stanou tokenem typu uspořádaná dvojice (např. `A`$_{11}$).
- Některé kategorie interpretuje token procesor ve své režii (např. `%`$_{14}$).

Rozlišujeme dva různé typy tokenů. Prvním typem tokenu je *uspořádaná dvojice* (ASCII hodnota, kategorie). Kategorie udává další chování tohoto tokenu buď přímo v algoritmech token procesoru, nebo později. Kategorii přiděluje token procesor vstupním znakům podle „tabulky kategorií“, která se v průběhu zpracování může měnit (viz níže). Jakmile je token vytvořen, v dalším zpracování se už jeho kategorie nemění, ani v případě, že je změněna tabulka kategorií. Token typu uspořádaná dvojice budeme značit pomocí „škatulky“, např. `A`$_{11}$ znamená ASCII hodnotu znaku „`A`“ a kategorii 11.

Druhým typem tokenu je typ *řídicí sekvence*. Například sekvenci `\slovo` zpracuje token procesor jako token `slovo`. Tím jsme také uvedli způsob značení těchto tokenů; do rámečku budeme psát identifikátor řídicí sekvence. Upozorňujeme, že je třeba rozlišovat mezi tokenem `$` a `$`$_3$. První uvedený je token typu „řídicí sekvence“, který vzniká zápisem `\$`, zatímco druhý je typu „uspořádaná dvojice“ a vznikl prostým zápisem znaku `$` za předpokladu, že tento znak měl v tabulce kategorií hodnotu 3. V textu v této knize budeme pro tokeny typu řídicí sekvence používat jak značení `\slovo`, tak značení `slovo`. Druhou alternativu použijeme v případě, kdy chceme zdůraznit, že se jedná o token. □

Tabulka kategorií přiřazuje každému ASCII znaku právě jednu číselnou hodnotu v rozsahu 0 až 15, tj. *kategorii*. Následuje seznam všech kategorií TEXu. U každé je stručně uveden význam a dále seznam znaků, které mají obvykle tuto kategorii přiřazenu. Je-li u čísla kategorie hvězdička, pak to znamená, že tato kategorie má význam jen v algoritmech token procesoru a neobjeví se nikdy na jeho výstupu. Je-li za znakem uvedeno slovo (plain), pak příslušná kategorie není tomuto znaku nastavena implicitně (v iniTEXu), ale je nastavena až ve formátu plain.

kategorie	*význam*	*výchozí přiřazení*
0*	uvození řídicí sekvence	`\`
1	otevření skupiny	`{` (plain)
2	zavření skupiny	`}` (plain)
3	přepínač matematického módu	`$` (plain)
4	separátor v tabulkách	`&` (plain)
5*	konec řádku	`^^M` (ASCII 13)
6	označení parametrů maker	`#` (plain)
7	konstruktor mocniny	`^` (plain)
8	konstruktor indexu	`_` (plain)
9*	znak, který se ignoruje	`^^@` (ASCII 0) (plain)
10	mezera	␣
11	písmeno	`A` až `Z`, `a` až `z`
12	ostatní znaky	*zbylé znaky*
13	aktivní znaky	`~`, `^^L` (plain)
14*	uvození komentáře na řádku	`%`
15*	nedovolený znak	`^^?` (ASCII 127)

Plain nastavuje ještě znak ⟨*tabelátor*⟩ (ASCII 9) na kategorii 10 (mezera) a dále deklaruje alternativní konstruktory pro mocninu a index (ASCII 1 a 11), protože se Knuth setkal s klávesnicemi, které po zmáčknutí šipky nahoru a dolu vloží tento kód a v editoru jsou vidět vykreslené šipky. Formát `csplain` navíc nastavuje všem znakům s kódy nad 128, které mají v ISO 8859-2 význam písmene z české a slovenské abecedy, kategorii 11 (písmeno). □

Nastavit jinou než výchozí kategorii nějakému znaku lze pomocí primitivu `\catcode`, za nímž napíšeme ASCII hodnotu znaku, pak (nepovinné) rovnítko a pak číslo kategorie. ASCII hodnotu znaku přitom můžeme zapisovat ve formátu `‘\`⟨*znak*⟩ (přesněji $\boxed{\texttt{‘}}_{12}\,\boxed{\langle znak\rangle}$, viz syntaktické pravidlo ⟨*number*⟩ v části B). Například:

```
30 \catcode‘\*=13  \catcode‘\>=0  >catcode‘>#=12  \catcode‘\%=12
```

znamená, že hvězdička bude aktivní znak, dále symbol `>` může také uvozovat řídicí sekvence a konečně znaky „vězení“ (`#`) a procento budou interpretovány jako obyčejné znaky. Nebudou mít tedy speciální význam. Jakmile hlavní procesor provede přiřazení typu `\catcode`, token procesor vezme tuto změnu na vědomí a bude se podle toho případně chovat jinak. Proto v naší ukázce můžeme třetí a čtvrtý příkaz `\catcode` již psát jako `>catcode`. Můžeme, ale nemusíme. Zatím jsme totiž nezměnili kategorii znaku „\“. □

Při čtení formátů a stylových souborů je zvykem přechodně nastavit kategorii znaku „`@`“ na 11 (písmeno), zatímco při zpracování dokumentu má tento znak kategorii 12. Důsledek: Ve formátech a stylech je možno používat identifikátory, které obsahují znak „`@`“. Na druhé straně ve vlastním dokumentu nelze takové identifikátory použít přímo, ale až po nastavení `\catcode‘\@=11`. Tím je možno „zakrýt“ před nepoučeným uživatelem řídicí sekvence, které nejsou určeny k přímému použití. Máme tedy zaručeno, že nám tyto pomocné řídicí sekvence nebude uživatel nevědomky předefinovávat.

Osobně nemám použití znaku „`@`“ v identifikátorech řídicích sekvencí příliš v oblibě. Takové makro je podle mého názoru hůře čitelné. Budu se proto snažit v celé knize takovým zvykům v ukázkách maker pokud možno vyhnout. Jestliže někdo chce později zvýšit odolnost svých maker před nepoučeným uživatelem, určitě si do názvů identifikátorů nějaké zavináče rád dodělá. Pro LaTeXovské styly se dokonce doporučuje používat pro pomocné řídicí sekvence vyhrazený název tvaru `\`⟨*název stylu*⟩`@`⟨*můj název*⟩. Tím je možno zavádět do dokumentu styly různých autorů a snižuje se riziko, že budou tyto styly ve vzájemném sporu. Existuje ještě mnoho dalších nebezpečí, která mohou přivést styly různých autorů ke sporu, takže ani na tuto konvenci nelze příliš spoléhat. □

Pusťme se do výkladu algoritmů token procesoru. Upozorňujeme, že token procesor důsledně rozlišuje vstupní znaky podle kategorií a ASCII kódy bere na vědomí jen

v případě, kdy to je výslovně řečeno. Budeme-li v dalším textu mluvit například o mezeře, budeme tím mít na mysli znak s kategorií 10, přitom ASCII kód tohoto znaku není podstatný.

(a) Dvojitá stříška. Je-li na vstupu znak s kategorií 7 (například $\boxed{\texttt{\^{}}}_7$), TEX se podívá, zda následuje znak se stejným ASCII kódem jako zrovna načtený (například zase znak `^`). Pokud ano, mluvíme o výskytu dvojité stříšky. V takovém případě TEX konvertuje dvojitou stříšku s následným jedním nebo dvěma znaky takto: (1) Je-li následující znak číslice `0` až `9` nebo písmeno `a` až `f` (písmena jsou malá a bere se v úvahu ASCII hodnota znaků, ignoruje se kategorie), pak se dvojice znaků za dvojitou stříškou interpretuje jako hexadecimální zápis ASCII hodnoty výsledného znaku. Například čtveřice `^^f8` je konvertována na jediný znak ASCII 248, který v ISO 8859-2 znamená písmeno `ř`. (2) Má-li následující znak kód menší než 128 a je různý od výše jmenovaných, pak se tento jediný znak za dvojitou stříškou konvertuje na znak s ASCII hodnotou, která se od ASCII hodnoty stávajícího znaku liší o 64, a přitom zůstává v rozsahu $\langle 0, 127\rangle$. Například `^^M` je ASCII 13, protože $13 + 64 = 77$, což je ASCII hodnota znaku `M`. V běžných editorech nejsme schopni znak s kódem 13 přímo zapsat. Méně praktický příklad: `^^+` je totéž, jako znak `k` (ASCII 107), protože $107 - 64 = 43$, což je ASCII hodnota znaku „`+`“. Výsledný znak podléhá v token procesoru dalšímu zpracování podle stavu, ve kterém se token procesor nalézá.

(b) Sestavování řídicích sekvencí. Je-li na vstupu znak s kategorií nula, začne TEX sestavovat z následujících znaků identifikátor, který potom vystupuje jako token typu řídicí sekvence. Přitom se postupuje dvěma různými způsoby: (1) Má-li první znak identifikátoru kategorii 11 (písmeno), pak se sestaví řídicí sekvence ze všech znaků s kategorií 11 až po konec řádku nebo po první znak, který má jinou kategorii (tento odlišný znak už není do identifikátoru zahrnut). Poté token procesor přechází do stavu S (viz níže). (2) Má-li první znak identifikátoru jinou kategorii, než 11, pak identifikátor bude obsahovat jen tento znak. Token procesor přechází do stavu M, výjimečně při sestavení řídicí sekvence `\␣` přechází do stavu S. O stavech token procesoru, viz níže.

(c) Sestavování tokenů typu dvojice. Není-li zrovna v činnosti algoritmus (a) nebo (b), pak se vstupní znak konvertuje do tokenu typu dvojice za předpokladu, že kategorie tohoto znaku je 1, 2, 3, 4, 6, 7, 8, 11, 12 nebo 13. Například na vstupu máme znak `A`, který má zrovna kategorii 11, proto na výstupu dostáváme $\boxed{\texttt{A}}_{11}$. Má-li znak jinou kategorii než zde uvedenou, chová se token procesor speciálním způsobem (viz níže).

(d) Ignorování znaku. Objeví-li se na vstupu znak kategorie 9 nebo 15, je tento znak ignorován, tj. ve výstupu z token procesoru se neobjeví. V případě kategorie 15 je navíc připojeno chybové hlášení `Text line contains an invalid character`.

(e) Komentář. Objeví-li se na vstupu znak kategorie 14 (komentář), TEX ignoruje zbytek řádku a přejde na nový řádek.

Pro další popis chování token procesoru je nutné rozlišovat tři stavy tohoto procesoru. Stav N (nový řádek), dále stav M (zpracování uvnitř řádku) a konečně stav S (přeskakování mezer). Stav N je na začátku řádku. Z něj TEX (obvykle) přechází do stavu M a občas udělá krátkou exkurzi do stavu S, kdy přeskakuje mezery. Pak se většinou znovu vrací do stavu M. A nyní podrobněji:

(f) Stav N, nový řádek. Na začátku každého řádku se token procesor bez výjimek nastaví do stavu N. V tomto stavu TEX ignoruje všechny mezery a znaky, které se ignorují podle algoritmu (d), až se objeví jiný znak. Má-li tento znak kategorii 5 (konec řádku), vytvoří token procesor na výstupu token `par`, ukončí čtení řádku (případný zbytek řádku je ignorován) a přejde na další řádek. Jinak token procesor přechází do stavu M.

Všimneme si, že výše zmíněný algoritmus způsobuje (obvykle) vložení sekvence `par` v místě každého vizuálně prázdného řádku, tj. řádku obsahujícího jen mezery. Je to proto, že na konci takového řádku je `^^M` z `\endlinechar` a tento znak má kategorii 5.

(g) Stav S, přeskakování mezer. Do tohoto stavu se TEX dostává například po načtení řídicí sekvence typu `\slovo`. Ve stavu S ignoruje všechny mezery a znaky, které se ignorují podle bodu (d), až narazí na jiný znak. Má-li tento znak kategorii 5 (konec řádku), ignoruje případný zbytek řádku a přejde na další řádek. Jinak se vrací do stavu M.

(h) Stav M, vlastní práce token procesoru. V tomto stavu TEX spouští jednotlivé algoritmy uvedené v bodech (a) až (e). Kromě toho jsou ošetřeny ještě následující situace: Objeví-li se na vstupu znak kategorie 5 (konec řádku), token procesor vloží do výstupu token $\boxed{\sqcup}_{10}$ (tj. ASCII 32, kategorie 10), ignoruje zbytek řádku a přejde na další řádek. Objeví-li se na vstupu znak kategorie 10 (mezera), token procesor vloží do výstupu token $\boxed{\sqcup}_{10}$, (tj. ASCII 32 bez závislosti na ASCII hodnotě „mezery“) a přechází do stavu S. Tím je zaručeno, že více mezer se chová jako jedna mezera.

Proveďme sumarizaci toho, co se stane, když token procesor narazí na znak kategorie 5 (znak konce řádku): Ve stavu N (nový a prázdný řádek) vytvoří token `par`, ve stavu S (ignorování mezer) nevytvoří nic a ve stavu M vytvoří token $\boxed{\sqcup}_{10}$. To odpovídá požadavkům, které na konec řádku v jednotlivých situacích máme.

(i) Fyzický konec řádku. V popisech algoritmů jsme zamlčeli, jak se token procesor zachová, pokud narazí na fyzický konec řádku. Především uveďme, že taková situace je možná tehdy, pokud znak z `\endlinechar` má jinou kategorii než 5. Ve všech stavech při dosažení fyzického konce řádku token procesor přejde na nový

řádek. Do výstupu nevkládá nic. Pokud je znak s kategorií 0 posledním znakem na řádku, pak má výsledná řídicí sekvence prázdný identifikátor.

Uvedené algoritmy mají sestupnou prioritu. Pokud třeba píšeme `\%`, pak znak `%` nezpůsobí konec načítání řádku podle bodu (e), protože je zpracováván jako identifikátor řídicí sekvence podle bodu (b), což má vyšší prioritu. Jiný příklad: pokud token procesor sestavuje řídicí sekvenci a objeví se dvojitá stříška, spustí se algoritmus podle bodu (a) a jeho výstup se použije při sestavování řídicí sekvence. Uvedeme jednu neužitečnou a jednu užitečnou ukázku: (1) `\vs^^+ip` znamená totéž co `\vskip` a (2) `\^^M` se nekonvertuje do $\boxed{\texttt{\^{}}}\ \boxed{\texttt{\^{}}}_7\ \boxed{\texttt{M}}_{11}$, ale vznikne jediný token typu řídicí sekvence, jejíž identifikátor obsahuje znak ASCII 13. □

Po možná trochu nepřehledném, ale dostatečně přesném, popisu algoritmu si odpočineme uvedením příkladu. Uvažujme následujících pět vstupních řádků:

```
␣␣Pokus%␣Tady␣je␣komentář␣␣
ný␣␣⟨tabelátor⟩text⟨tabelátor⟩␣␣v~\TeX␣␣␣
␣␣⟨tabelátor⟩␣u.
␣␣␣␣⟨tabelátor⟩␣␣
^^e8^^edslo␣2.\end
```

Předpokládejme, že je použito standardní nastavení kategorií z plainu. Input procesor postupně u každého řádku odstraní mezery zprava a přidá `\endlinechar`, což je ASCII 13 (má kategorii 5). Tento znak označíme $\langle CR \rangle$. Jednotlivé řádky po zpracování input procesorem vypadají takto:

```
␣␣Pokus%␣Tady␣je␣komentář⟨CR⟩
ný␣␣⟨tabelátor⟩text⟨tabelátor⟩␣␣v~\TeX⟨CR⟩
␣␣⟨tabelátor⟩␣u.⟨CR⟩
␣␣␣␣⟨tabelátor⟩⟨CR⟩
^^e8^^edslo␣2.\end⟨CR⟩
```

První dvě mezery v ukázce jsou přeskočeny ve stavu N. Pak TEX přejde do stavu M a probíhá sestavování tokenů $\boxed{\texttt{P}}_{11}\ \boxed{\texttt{o}}_{11}\ \boxed{\texttt{k}}_{11}\ \boxed{\texttt{u}}_{11}\ \boxed{\texttt{s}}_{11}$ podle bodu (c). Pak se podle bodu (e) ignoruje zbytek řádku s komentářem. Na dalším řádku okamžitě ze stavu N přecházíme do stavu M a vytváříme tokeny $\boxed{\texttt{n}}_{11}\ \boxed{\texttt{ý}}_{11}$. Pak se zpracuje mezera jako token $\boxed{\texttt{␣}}_{10}$ a ostatní mezery se ignorují (ve stavu S). Znovu ve stavu M se vytvoří tokeny $\boxed{\texttt{t}}_{11}\ \boxed{\texttt{e}}_{11}\ \boxed{\texttt{x}}_{11}\ \boxed{\texttt{t}}_{11}$. Protože $\langle tabelátor \rangle$ má kategorii mezery, vytvoří se token $\boxed{\texttt{␣}}_{10}$ a ostatní mezery se ignorují ve stavu S. Opět se TEX vrátí do stavu M, aby vytvořil tokeny $\boxed{\texttt{v}}_{11}\ \boxed{\texttt{\~{}}}_{13}\ \boxed{\texttt{TeX}}$. Na znak $\langle CR \rangle$ narazí ve stavu S, protože do tohoto stavu se dostal po sestavení řídicí sekvence. Ukončí tedy řádek bez vložení čehokoli do výstupu. Na třetím řádku se ve stavu N přeskočí všechny mezery až po písmeno `u`. Pak se přejde do stavu M a vytvoří se tokeny $\boxed{\texttt{u}}_{11}\ \boxed{\texttt{.}}_{12}\ \boxed{\texttt{␣}}_{10}$. Poslední token byl vytvořen díky výskytu znaku $\langle CR \rangle$ kategorie 5 ve stavu M. Na čtvrtém řádku se dospěje ke znaku $\langle CR \rangle$ již ve stavu N, proto se vloží do výstupu token

`par`. Uvedeme si konečně posloupnost tokenů, kterou v naší ukázce dostaneme na výstupu z token procesoru:

P$_{11}$ o$_{11}$ k$_{11}$ u$_{11}$ s$_{11}$ n$_{11}$ ý$_{11}$ ␣$_{10}$ t$_{11}$ e$_{11}$ x$_{11}$ t$_{11}$ ␣$_{10}$

v$_{11}$ ~$_{13}$ TeX u$_{11}$.$_{12}$ ␣$_{10}$ par č$_{11}$ í$_{11}$ s$_{11}$ l$_{11}$ o$_{11}$ ␣$_{10}$ 2$_{12}$.$_{12}$ end

Může se zdát, že jsme šli dělovou koulí proti mouše. Skutečně, v těchto jednoduchých případech si vystačíme s pravidly, která zná každý uživatel TEXu a která jsme shrnuli na začátku této sekce. Pokud se ale pustíme do tvorby složitějších maker, pak se obvykle stane, že se bez podrobné znalosti algoritmů token procesoru neobejdeme. □

• **Příklady.** Uvedeme jednoduché makro, které nám poví hodnotu kategorie libovolného znaku. Vyzkoušejte si:

```
\escapechar=-1
\def\kat #1{%
    \message{Znak "\string#1" má kategorii \the\catcode`#1.}}
\kat\\ \kat\{ \kat\} \kat\$ \kat\& \kat\^^M \kat\#
\kat\^ \kat\_ \kat\^^@ \kat\  \kat\A \kat\č \kat\^^+ \kat\:
\kat\0 \kat\~ \kat\%  \kat\^^?
```

Vidíme, že primitiv `\catcode` lze použít nejen ve smyslu definování nové kategorie znaku, ale v jiném kontextu (zde po primitivu `\the`) nám poslouží pro výzvědné účely: řekne nám hodnotu kategorie daného znaku. Při použití makra `\kat` píšeme pro jistotu všechny znaky ve tvaru jednoznakové řídicí sekvence. Tím můžeme zapsat znaky, které bychom jinak nebyli schopni samostatně použít (například `%`). Primitiv `\string` nám do zprávy vypíše obsah identifikátoru `#1`, přičemž díky hodnotě `\escapechar=-1` nepřipojuje před identifikátor znak „\“. □

Uvedeme jedno nebezpečí, které číhá skoro na každého uživatele, který začne poprvé experimentovat se změnou kategorií znaků. V následujícím příkladě předpokládejme, že si chceme zjednodušit psaní položek typu `\item`, přitom víme, že v textu položek nikdy nepoužijeme hvězdičku. Přidělíme tedy hvězdičce aktivní kategorii (13) a následně ji definujeme jako makro startující položku ve výčtovém seznamu:

```
Tady je normální text.
    \par\begingroup \leftskip=2em \catcode`\*=13
    \def*{\par\noindent\llap{$\bullet$ }\ignorespaces}
* První údaj.
* Zde mluvím o druhém.
* A konečně poslední.
    \par\endgroup
A zase další text.
```

Toto skutečně bude fungovat. Okamžitě se tedy budeme snažit o ještě větší přehlednost a schováme kód na začátku a na konci výčtového seznamu do maker, například `\begitems` a `\enditems`. Mohlo by to vypadat třeba takto:

```
\def\begitems{\par\begingroup \leftskip=2em \catcode`\*=13
   \def*{\par\noindent\llap{$\bullet$ }\ignorespaces}}
\def\enditems{\par\endgroup}
Tady je normální text.
  \begitems
* První údaj.
* Zde mluvím o druhém.
* A konečně poslední.
  \enditems
A zase další text.
```

Toto ovšem už fungovat nebude. Než si řekneme proč, dovolte mi krátkou filosofickou úvahu: Tato ukázka je příkladem, že na makra se nemůžeme dívat jen jako na pouhé zkratky nějakého delšího kódu, ale spíš jako na živý organismus. Je potřeba podrobně vědět, jak věci fungují.

Proč to přestalo fungovat? V době, kdy se TEX „učí“ makro `\begitems`, jednotlivé příkazy v těle tohoto makra nevykonává, ale jako obsah makra se uloží pouze posloupnost tokenů, jak ji vytvořil token procesor. Proto se do makra `\begitems` uloží tato posloupnost tokenů:

`par` `begingroup` `leftskip` `=`$_{12}$ `2`$_{12}$ `e`$_{11}$ `m`$_{11}$ `␣`$_{10}$

`catcode` `‘`$_{12}$ `*` `=`$_{12}$ `1`$_{12}$ `3`$_{12}$ `␣`$_{10}$ `def` `*`$_{12}$ `{`$_{1}$ `par` `noindent` atd.

Důležité je, že hlavní procesor v době „učení“ definice nevykonal povel `\catcode` a důsledkem toho nám token procesor vytvořil `def` `*`$_{12}$ a nikoli požadované `def` `*`$_{13}$. Token procesor pracuje jen na vstupní straně (řádek po řádku) a k jednou vytvořené posloupnosti tokenů se už nikdy nevrací. Můžeme říci, že jednou vytvořený token je pevný a neměnný. Výjimku tvoří například použití primitivu `\string`, ovšem to je úplně o něčem jiném. Jakmile se hlavní procesor v případě použití makra `\begitems` pokusí vykonat `def` `*`$_{12}$, samozřejmě nám vynadá, že v povelu `\def` chybí řídicí sekvence nebo aktivní znak.

Problém opravíme použitím povelu `\catcode` na dvou místech — jednak při „učení“ definice a jednak v těle definice. Fungující makro tedy vypadá takto:

```
{\catcode`\*=13
 \gdef\begitems{\par\begingroup \leftskip=2em \catcode`\*=13
   \def*{\par\noindent\llap{$\bullet$ }\ignorespaces}}}
\def\enditems{\par\endgroup}
```

Nastavení aktivní kategorie hvězdičky jsme provedli lokálně uvnitř skupiny. Jinde totiž chceme, aby se hvězdička chovala „normálním způsobem“, tedy ne jako aktivní znak. Dále musíme zaměnit `\def` za `\gdef`, abychom po opuštění skupiny makro `\begitems` okamžitě nezapomněli. Konečně primitiv `\catcode` uvnitř těla makra zachováme, protože bude pracovat v místě použití makra. Následující hvězdičky, které použije uživatel namísto obvyklého `\item`, budou tedy aktivní. □

V TEXu lze sestavit token typu dvojice až dodatečně prostřednictvím primitivu `\uppercase`. Vytvoříme třeba makro `\definujaktivni`, které načte do svého parametru znak, nastaví jej jako aktivní a hned jej nějak definuje. Pokud napíšeme:

```
69 \def\definujaktivni #1{\catcode`#1=13 \def #1{udělej něco...}}
```

tak se se zlou potážeme. Jakmile je třeba po `\definiujaktivni C` načten znak `C` jako parametr, je jednou pro vždy vytvořen token $\boxed{\texttt{C}}_{11}$. Pomocí `\def #1` se tedy snažíme definovat token kategorie 11, což skončí s chybou. Řešení je následující:

```
70 \def\definujaktivni #1{\catcode`#1=13 \bgroup \uccode`~=`#1
71    \uppercase{\egroup\def~}{udělej něco, #1 je zde neaktivní}}
```

Primitiv `\uppercase` pozmění na řádku 71 ASCII kód znaku `~` na hodnotu podle `#1`, ale token zůstane aktivní. Současně odstraní závorky kolem zápisu `\egroup\def`⟨*znak*⟩. Pak se tento zápis provede. To znamená, že se uzavře skupina (`\uccode`~` se vrátí k původní hodnotě) a vlastní ⟨*znak*⟩ se definuje. □

Pusťme se do dalšího příkladu. Vytvoříme pro uživatele „verbatim“ prostředí. Napíše-li uživatel například:

```
72 \begtt
73 Tady píšu: $ {# \ahoj
74 a další    řádek %.
75 \endtt
```

dostane na výstupu:

```
Tady píšu: $ {# \ahoj
a další    řádek %.
```

Uživatel může v oblasti mezi sekvencemi `\begtt` a `\endtt` psát cokoli a vše se mu věrně přepíše do výstupu. Jediné, co zde uživatel nesmí napsat, je sekvence šesti znaků: `\endtt`, která prostředí ukončuje. Vlastní makro může vypadat takto:

```
76 \def\setverb{\def\do##1{\catcode`##1=12}\dospecials}
77 \def\begtt{\par\bgroup \setverb\obeyspaces\obeylines\startverb}
78 {\catcode`\|=0 \catcode`\\=12
```

```
79    |gdef|startverb#1\endtt{|tt#1|egroup}}
```

Funkci makra si podrobně vysvětlíme. Pomocná sekvence `\setverb` nastaví kategorie všech „speciálních“ znaků (viz tabulku na straně 20) na kategorii 12 (obyčejný znak). Zde využíváme makro plainu `\dospecials`, které provede opakovaně `\do` s parametry `\\`, `\{` atd. Přitom `\do` je definováno tak, že nastaví `\catcode` svého parametru na 12. O zdvojení znaku parametru (`##`) viz stranu 38.

Makro `\begtt` otevře skupinu (`\bgroup`), takže následné `\setverb` nastaví kategorie jen lokálně. Makra plainu `\obeyspaces` a `\obeylines` si čtenář může projít v části B, zde jen stručně. První z nich nastaví mezeru (ASCII 32) jako aktivní znak a definuje ji jako `\space`. Tím je zaručeno, že každý výskyt mezery bude samostatně interpretován a expanduje se na token $\boxed{\sqcup}_{10}$ (token procesor nebude přecházet do stavu S). Podobně `\obeylines` nastaví znak `^^M` jako aktivní a definuje jej jako `\par`. Takže se na konci každého řádku uzavře odstavec.

Čtenář už jistě ví, proč lze za sekvencí `\setverb` (na řádku 77) bez obav psát další sekvence (`\obeylines`, atd.), ačkoli `\setverb` mění kategorii znaku „\“. Je to z toho důvodu, že sekvence `obeylines` a další jsou již token procesorem zpracovány v době, kdy se TEX „učí“ tělo makra `\begtt`. Kategorie znaku „\“ (a dalších) je pomocí `\setverb` změněna teprve v okamžiku, kdy uživatel použije makro `\begtt`.

Nejzajímavější v naší ukázce je pomocné makro `\startverb`, které dělá vlastní práci. Toto makro má parametr se separátorem: $\boxed{\backslash}_{12}\boxed{e}_{11}\boxed{n}_{11}\boxed{d}_{11}\boxed{t}_{11}\boxed{t}_{11}$. Není možné použít separátor `endtt`, protože takový separátor se po spuštění `\begtt` token procesorem nikdy nevytvoří. Skutečně, uživatelovo ukončení verbatim prostředí formou zápisu `\endtt` je token procesorem zpracováno do šesti tokenů a ne do jedné řídicí sekvence. Abychom mohli při definování makra `\startverb` napsat zmíněný šestitokenový separátor, musíme nastavit kategorii znaku „\“ na 12. Abychom se ale vůbec mohli vyjadřovat, musíme si nastavit kategorii jiného znaku na 0. V našem případě jsme použili znak „|“. Po spuštění makra `\startverb` se pak vloží do parametru `#1` celé prostředí „verbatim“ až po ukončující `\endtt`. Makro zopakuje tento parametr hned za sekvencí `tt`. Tisk bude tedy proveden strojopisem. Nakonec se pomocí `egroup` ukončí skupina otevřená na začátku makra `\begtt`. Od té chvíle začne token procesor znovu pracovat „normálně“.

Pokud bychom chtěli místo bílého místa každou mezeru tisknout jako vaničku (viz ukázku na řádcích 31 až 40), pak stačí místo `\obeyspaces` na řádku 77 psát `\catcode‘\ =12`. Nyní bude každá mezera zpracována jako obyčejný znak a vytiskne se přímo symbol z pozice 32 fontu `\tentt`. Tam je kresba vaničky. Chceme-li v našem verbatim prostředí číslovat řádky (podobně, jako v této knize), stačí použít `\everypar`. Každý řádek v našem prostředí je totiž sázen jako samostatný odstavec (díky `\obeylines`).

Po chvíli experimentování s naším makrem narazíme na problémy. V případě příliš dlouhého řádku obdržíme hlášení `Overfull` a někdy se řádek dokonce zlomí na dva, jako každý jiný odstavec. S tímto problémem nic nenaděláme. Musíme volit takovou velikost fontu a tak „široké“ ukázky, aby se nám do šířky zrcadla vešly.

Také nás překvapí, že se ztratí mezery ze začátku každého řádku. Přitom mezi slovy máme správný počet mezer. Je to tím, že `\obeyspaces` nastavuje mezeru na aktivní a tato aktivní mezera je v plainu ztotožněná s makrem `\space`, které expanduje na povel ␣$_{10}$. Tento povel udělá mezeru jen v horizontálním módu, zatímco ve vertikálním módu je ignorován (o módech a mezerách viz sekce 3.1, 3.4 a 3.6). Na začátku každého řádku našeho verbatim prostředí je TEX ve vertikálním módu. Tím jsme si vysvětlili důvod ztráty mezer a nyní provedeme nápravu. Nahradíme `\obeyspaces` z řádku 77 voláním makra `\activespace`, které definujeme takto:

```
80 {\obeyspaces \gdef\activespace{\obeyspaces\let =\ }}
```

Dvojí použití `\obeyspaces` nás už nesmí překvapit. První nastavuje aktivní mezeru při čtení těla definice a druhé `\obeyspaces` (v těle definice) nastavuje aktivní mezeru v okamžiku spuštění našeho makra `\activespace`. Toto makro ztotožňuje aktivní mezeru s primitivem `\␣`, který v případě použití ve vertikálním módu zahájí odstavcový mód.

Cvičení: na řádku 80 vidíme celkem tři mezery. První za slovem `\obeyspaces`, druhou za slovem `\let` a poslední těsně před zavíracími závorkami. Jak jsou tyto mezery interpretovány token procesorem? Řešení: První mezera je ignorována, protože ukončuje sestavení tokenu `obeyspaces` v době, kdy ještě není mezera aktivní. Druhá mezera vytvoří token ␣$_{13}$ a poslední je součástí identifikátoru řídicí sekvence a vytvoří společně s předchozím zpětným lomítkem token ␣.

Posledním nedostatkem našeho makra `\begtt` je skutečnost, že prázdný řádek na vstupu zmizí a na výstupu není. Je to z toho důvodu, že dvě `\par` za sebou pracují jako jedno, protože druhé je ve vertikálním módu ignorováno. Nápravou může být předefinování `\par` tak, aby makro `\par` ošetřilo, zda za ním nenásleduje nové `\par` a pokud ano, pak založí nejprve prázdný řádek. Shrneme všechny opravy do nové „verze“ našeho makra. Pro ilustraci do kódu navíc přidáme číslování řádků:

```
81 \newcount\num
82 {\obeyspaces \gdef\activespace{\obeyspaces\let =\ }}
83 \def\setverb{\def\do##1{\catcode`##1=12}\dospecials}
84 \def\begtt{\par\bgroup \setverb \activespace
85   \everypar={\global\advance\num1 \llap{\sevenrm\the\num\quad}}
86   \def\par##1{\endgraf\ifx##1\par\leavevmode\fi ##1}
87   \obeylines \startverb}
88 {\catcode`\|=0 \catcode`\\=12
89   |gdef|startverb#1\endtt{|tt#1|egroup}}                    □
```

Na závěr této sekce se zmíníme o problému zavlečených mezer v makrech. Předpokládejme blíže neurčené makro, ve kterém jsme použili cyklus. Přitom z každého průchodu cyklem vypadne z expand procesoru jeden zapomenutý token $\boxed{\sqcup}_{10}$. Pokud makro pracuje jen ve vertikálním módu, pak zavlečené mezery nevadí. Jakmile uživatel použije takové makro v horizontálním módu, dočká se nemilého překvapení: na výstupu bude mnoho mezer vedle sebe. Na mezery je nutno dávat pozor též při sestavování tabulek s linkami, které musí na sebe přesně navazovat. Studenti mi v mnoha případech předváděli svá makra, kde tabulky vypadaly uspokojivě, ale za cenu toho, že do kódu vkládali různá vyrovnávací `\hskip` opatřená experimentálně zjištěnými Pišvejcovými čísly. Takto se skutečně nedá postupovat.

Na konci řádku, který je dosažen ve stavu M, vyprodukuje token procesor $\boxed{\sqcup}_{10}$. Proto musíme být na taková místa mimořádně citliví a projít všechny konce řádku. Máme-li na konci řádku napsáno „`{`“ nebo „`}`“, pak za těmito závorkami skoro jistě přichází zavlečená mezera. Pišme proto raději „`{%`“ nebo „`}%`“. Proč jsme nemuseli takový zápis použít v naší poslední ukázce? Na konci řádku 88 se sice token $\boxed{\sqcup}_{10}$ vytvoří, ale je součástí syntaktického pravidla ⟨*number*⟩. Takže nevadí ani v případě, že by taková konstrukce byla použita v horizontálním módu. Na konci řádku 85 by sice zavlečená mezera v horizontálním módu mohla vadit, ale my jsme ve vertikálním módu (viz `\par` na řádku 88). Ze stejných důvodů nám nevadí mezera z konce řádku 86. Konečně na řádku 87 nám rovněž nevadí zavlečená mezera, protože proces „učení“ makra `\begtt` provedeme ve vertikálním módu.

Navrhujeme-li makro pro použití i v horizontálním módu, musíme vymýtit všechny zavlečené mezery. Je-li makro dosti složité, pak to může být i velmi komplikovaný problém. Doporučuji nejprve projít všechny výskyty „`}`“ a „`{`“ na koncích řádků a zvážit, zda tam není zavlečená mezera. Pak se ještě hodí projít výskyty „`#1`“, „`#2`“ apod. například v podmínkách typu `\if`. Za těmito parametry také nesmí být mezera, kterou bychom si tam možná hodně přáli, abychom zvýšili čitelnost konstrukce `\if` a oddělili podmínku od těla konstrukce. Bohužel, mezeru tam nelze psát. Na druhé straně za řídicími sekvencemi typu `\slovo` a za čísly podle syntaktického pravidla ⟨*number*⟩ se mezer nemusíme bát. V prvním případě je mezera zničena rovnou v token procesoru a v druhém případě je ignorována hlavním procesorem jako součást syntaktického pravidla.

Může se stát, že jsme vyčerpali všechny možnosti, a přitom se na nás zavlečená mezera někde v makru stále skrytě směje. Pak je možné pomocí `\showlists` zjistit, kde se přesně v horizontálním seznamu mezera vyskytuje a pomocí nastavení `\tracingcommands=2` trasovat činnost všech primitivních povelů hlavního procesoru. Mezi nimi bude určitě povel označený jako „`blank space`“. Také můžeme přistoupit na primitivní ladicí postupy: někam do konkrétního místa v makru napíšeme třeba znak `A` a jinam znak `B`. Bude-li na výstupu mezi nimi mezera, budeme dále půlit tento „interval“. Tak postupně dospíváme ke stále menšímu kódu v makru, ve kterém se ta mezera skrývá. Tím se postupně blížíme k okamžiku, kdy se chytneme za hlavu a vykřikneme podobně jako bystrozraký v pohádce: „Už ji vidím!“ □

2. Expand procesor

2.1. Definování maker

Ačkoli k definování maker je v TEXu implementováno jen málo primitivů (\def, \edef, \gdef, \xdef, prefixy \global, \long a \outer), jedná se o poměrně širokou problematiku. Začneme proto velmi jednoduchým příkladem. Na příkladě si podrobně rozebereme, co se v TEXu odehrává.

```
1 \def\pokus{to je {\it pokusné\/} makro v~\TeX u}
2 Použití makra: \pokus, a ještě jednou \pokus.
```

Celá věc má dvě fáze. Fáze „učení“ makra je realizována primitivem \def. Zde hlavní procesor přiřadí řídicí sekvenci \pokus význam makra a uloží do paměti posloupnost tokenů uzavřenou v konstrukci \def mezi závorkami {$_1$ a }$_2$. Na ASCII hodnotách těchto závorek nezáleží. Této posloupnosti tokenů říkáme *tělo definice*.

Tělo definice může obsahovat další závorky {$_1$ a }$_2$, ovšem pouze párované. Znamená to, že TEX interpretuje jako konec těla definice teprve takovou }$_2$, která párově odpovídá {$_1$ uvozující tělo definice. Říkáme, že tělo definice musí být *balancovaným textem*. Na řádku 1 ukončuje tedy tělo definice až druhá závorka „}“.

Druhá fáze „použití“ makra se odehrává v expand procesoru. V našem příkladě na řádku 2 při obdržení sekvence \pokus zamění expand procesor tuto sekvenci za posloupnost tokenů, která byla zapamatována ve fázi učení. Této činnosti říkáme *expanze* řídicí sekvence či tokenu. Expanze našeho makra \pokus spočívá v záměně:

pokus $\longrightarrow$ t$_{11}$ o$_{11}$ ␣$_{10}$ j$_{11}$ e$_{11}$ ␣$_{10}$ {$_1$ it p$_{11}$ o$_{11}$ k$_{11}$ u$_{11}$ s$_{11}$ n$_{11}$ é$_{11}$

/ }$_2$ ␣$_{10}$ m$_{11}$ a$_{11}$ k$_{11}$ r$_{11}$ o$_{11}$ ␣$_{10}$ v$_{11}$ ~$_{13}$ TeX u$_{11}$

Tato posloupnost je znovu zpracována expand procesorem, tj. pokud se v ní vyskytuje nějaká řídicí sekvence ve významu makra, je tato sekvence rovněž zaměněna za odpovídající posloupnost podle těla definice. Zde se tedy ještě expanduje aktivní znak ~$_{13}$ a tokeny it a TeX. Tato makra byla definována ve formátu plain. Podrobněji k těmto makrům, viz část B.

Tímto způsobem expand procesor expanduje tokeny tak dlouho, až jsou všechny tokeny neexpandovatelné, nebo nemají být expandovány (viz sekci 3.2). Teprve taková posloupnost tokenů přichází do hlavního procesoru. □

Je dobré si uvědomit, že závorky {}$_1$ a }$_2$ mají v TEXu tři poněkud nesouvisející významy: (1) V kontextu povelu hlavního procesoru závorky otevírají či zavírají skupiny, uvnitř kterých jsou (vesměs) všechny změny parametrů sazby lokální. (2) Závorky vymezují tělo definice v konstrukcích typu `\def`. (3) Závorky mohou ovlivnit načítání parametru makra. Ve všech případech záleží pouze na kategorii závorky a nikoli na ASCII hodnotě.

Význam závorek podle (3) podrobně probereme v této sekci později. Zdvojení významu závorek podle (1) a (2) asi nebude činit programátorovi maker potíže. Pouze je vhodné upozornit na drobnou začátečnickou chybičku:

```
3 \def\priklad{\bf Příklad: }
```

způsobí, že po použití sekvence `\priklad` se sazba přepne do fontu `\bf` (polotučný řez) a makro se nepostará o návrat do původního řezu písma. Závorky ohraničující tělo definice totiž nejsou součástí těla definice. Správně je třeba psát:

```
4 \def\priklad{{\bf Příklad: }}
```

Pokud v nějaké speciální aplikaci potřebujeme v těle definice použít pouze povel k otevření skupiny a nikoli k jejímu zavření, musíme místo tokenu {}$_1$ psát zástupnou sekvenci. Viz například stranu 340, řádky 164–166. □

Ve fázi „učení“ makra se neprovádí expanze těla definice (výjimku tvoří použití primitivu `\edef` a `\xdef`, o kterých pohovoříme později). Proto při definování maker nezáleží na pořadí jejich definic a můžeme je uspořádat jednak „shora dolů“ (tj. od cílových maker k pomocným makrům, která jsou v tělech definic předchozích maker použita) a jednak „zdola nahoru“.

V TEXu lze později předefinovat některé části maker nebo makra celá:

```
5  \def\A{ABC\B}
6  \def\B{bbb}
7  \A % se expanduje na ABC\B a to se expanduje na ABCbbb
8  \def\B{ccc}
9  \A % nyní vede na ABC\B a to na ABCccc
10 \def\A{XYZ}
11 \A % se nyní expanduje na XYZ
```

Na druhé straně, pokud přiřadíme sekvenci nějaký význam pomocí `\let`, zůstává tento význam zachován, i když je význam vzoru později změněn. Například:

```
12 \def\A{XYZ}
13 \let\B=\A  % \B získává význam makra s tělem definice "XYZ"
14 \def\C{\A} % \C je makro s tělem definice "\A"
```

```
\A \B \C   % Všechny tři expandují na XYZ
\def\A{PQR}
\B  % stále expanduje na XYZ
\C  % expanduje na \A a to expanduje na PQR
```
□

• **Parametry maker.** Makra je možné definovat s *parametry*. Proto je obecná konstrukce povelu `\def` tvaru:

`\def`⟨*řídicí sekvence*⟩⟨*maska parametrů*⟩`{`⟨*tělo definice*⟩`}`

kde ⟨*řídicí sekvence*⟩ je nově definovaná řídicí sekvence a ⟨*maska parametrů*⟩ je nepovinná a určuje způsob práce nově definovaného makra s parametry. Touto vlastností se budeme podrobně zabývat v následujícím textu. Text masky parametrů je ukončen prvním výskytem závorky $\boxed{\{}_1$. Konečně ⟨*tělo definice*⟩ je balancovaný text, jak jsme se o tom již zmínili výše.

Aby byl výklad poněkud přehlednější, začneme maskou parametrů, která deklaruje použití jediného parametru. Tento parametr se v masce parametrů i v těle definice označuje pomocí `#1`. Přesněji: $\boxed{\#}_6\,\boxed{1}_{12}$, kde na ASCII hodnotě tokenu $\boxed{\#}_6$ nezáleží zatímco u tokenu $\boxed{1}_{12}$ je významná jednak kategorie a jednak ASCII hodnota.

O parametru budeme mluvit v různých souvislostech na třech místech. Za prvé *deklarace parametru* se píše v masce parametrů. Za druhé o *formálním parametru* mluvíme při výskytu `#1` v těle definice. Konečně *aktuální parametr* je posloupnost tokenů, se kterou se pracuje jako s parametrem v době činnosti makra. Uvedeme si příklad:

```
\def\sekce #1.{% Zde je deklarován #1 v masce parametrů
   \bigskip\noindent{\bf #1}\par\nobreak} % Zde je formální #1

\sekce O~problematice chroustů.
% Aktuálním parametrem je text: "O~problematice chroustů"
```

V masce parametrů je možné kromě výskytu samotného parametru `#1` zapsat tokeny, které při použití makra budou aktuální parametr obklopovat. V našem příkladě před parametrem nebudou žádné tokeny, zatímco za ním musí být tečka. Při expanzi makra `\sekce` je tedy aktuálním parametrem veškerý text, zapsaný za tímto makrem až po první tečku. Tomuto procesu říkáme, že se text *načítá* do parametru. Tečku v našem příkladě považujeme za *separátor* parametru.

Expanze makra s parametrem probíhá přesně takto: Do parametru se načte veškerý text až po separátor (mimo tento separátor). Přitom se neprovádí expanze. Dále je token s názvem makra včetně zapsaného parametru a použitého separátoru zaměněn za posloupnost tokenů v těle definice. Pokud se v těle definice vyskytuje formální parametr `#1`, je tento zápis zaměněn za aktuální parametr. Takových

výskytů formálních parametrů může být v těle definice i více, nebo také žádný. Nově vytvořená posloupnost tokenů pak podléhá případné další expanzi v expand procesoru. □

V masce parametrů můžeme použít jako separátor celé skupiny tokenů, nikoli jen jediný token. Například po definici:

```
24 \def\sekce:\param=#1-.{...}
```

může použití makra `\sekce` vypadat takto:

```
25 \sekce :\param=text aktuálního parametru-.
```

Co se stane, pokud použití makra neodpovídá definici podle masky parametrů? Jestliže bychom v našem příkladě nenapsali bezprostředně za `\sekce` text „`:\param=`“, obdrželi bychom chybu: `Use of \sekce doesn't match its definition`, což znamená, že použití makra nesouhlasí s jeho definicí. Pokud se při načítání parametru „nikdy nedosáhne“ separátoru (v našem příkladě textu $\boxed{-}_{12}\,\boxed{.}_{12}$), TEX vypíše chybu `Paragraph ended before \sekce was complete`, což znamená, že se do aktuálního textu parametru měl zavést token `par` a to se TEXu nelíbilo. TEX je totiž vybaven „pojistkou“, která nedovolí kvůli zapomenutému separátoru v místě použití makra načíst do parametru více než jeden odstavec textu. Přesněji: TEX nedovolí vložit do aktuálního parametru token `par`, pokud to není výslovně dovoleno při definování makra prefixem `\long`. Takže kdybychom psali:

```
26 \long\def\sekce:\param=#1-.{...}
```

pak by při zapomenutém separátoru v místě použití probíhalo načítání parametru přes hranice odstavců a havarovalo by to až kvůli překročení kapacity TEXu nebo z důvodu dosažení konce souboru před ukončením kompletace parametru: `File ended while scanning use of \sekce`. Při haváriích tohoto typu navíc TEX lehce nadhodí: `Runaway argument?` (přetažený argument?), čímž dává najevo, kde asi hledat příčinu nehody.

Separátor v masce parametrů se nikdy neexpanduje. Proto například na řádku 25 nevadí, že sekvence `\param` není nikde definována.

Separátor v masce musí zcela odpovídat skutečnému separátoru aktuálního parametru. Pod pojmem „zcela“ máme na mysli, že nestačí shodnost ASCII hodnot tokenů, ale musí se shodovat též kategorie. U tokenů typu řídicí sekvence se musí shodovat identifikátory. □

Při načítání parametru TEX sice neprovádí expanzi, ale všímá si přítomnosti závorek $\boxed{\{}_1$ a $\boxed{\}}_2$, které mohou ovlivnit načítání. Pravidlo říká, že text shodný se

separátorem parametru separuje parametr pouze tehdy, jestliže se nevyskytuje ve vnořené úrovni závorek v rámci načítání parametru. Například:

```
\def\sekce #1.{\bigskip\noindent{\bf #1}\par\nobreak}
\sekce {J. K.} Tyl.
```

Zde se do parametru `#1` načte text „`{J.␣K.}␣Tyl`“, tedy první dvě tečky netvoří separátor, protože nejsou při načítání parametru ve vnější úrovni závorek. Teprve třetí tečka ukončuje parametr.

Knuthovi je vyčítáno, že tento nový význam závorek zbytečně koliduje s významem otevření a zavření skupiny. Skutečně, v předchozí ukázce jsem nechtěl otevírat a zavírat skupinu, nicméně při provádění sazby (po úplné expanzi) k této činnosti dojde. Ve většině případů to naštěstí nevadí.

TEX je tedy ochoten do parametru načíst pouze balancovaný text. Pokud se při načítání parametru vyskytne závorka $\boxed{\}}_2$, která neodpovídá žádné otevírací závorce v již načteném textu parametru, TEX nekompromisně ohlásí chybu `Argument of \macro has an extra }`. □

Parametr makra může být *separovaný* nebo *neseparovaný*. V obou případech probíhá načítání do parametru trošku odlišně. Separovaný parametr je takový, který má za svou deklarací v masce parametrů vyznačen separátor. Neseparovaný parametr je takový, který nemá za svou deklarací v masce parametrů žádný separátor. Například v naší ukázce makra `\sekce` se vždy jednalo o separovaný parametr.

Je-li parametr separovaný, je při použití makra načten do parametru text až po separátor podle pravidel, která jsme uvedli před chvílí. Pokud je takto načtený parametr ve tvaru $\boxed{\{}_1$ *balancovaný text* $\boxed{\}}_2$, TEX vnější závorky odstraní.

Je-li parametr neseparovaný, pak se načte první nemezerový token (tj. přeskočí se všechny případné tokeny $\boxed{␣}_{10}$). Pokud je načtený token různý od $\boxed{\{}_1$ nebo $\boxed{\}}_2$ (na ASCII hodnotě nezáleží), je tento token aktuálním parametrem. Jedná-li se však o $\boxed{\{}_1$, do parametru se načte balancovaný text až po odpovídající závorku $\boxed{\}}_2$. Vymezující závorky tedy pracují podobně jako separátory parametru. Ve vlastním textu parametru nejsou tyto závorky zahrnuty. S takovými množinovými závorkami se v parametrech maker setkávají hlavně uživatelé LATEXu. Příklad:

```
\def\:#1{\message{Parametr je "#1".}} % neseparovaný parametr
\: a    % přeskočí se mezera a načte se "a"
\:abc   % načte se "a" a zbytek ("bc") není makrem zpracován
\:{abc{d}ef}  % načte se "abc{d}ef"
\: {abc}      % přeskočí se mezera a načte se "abc"
\: }    % dojde k chybě "Argument of \: has an extra }"
```
□

Jediné makro může mít v masce parametrů deklarováno až devět parametrů současně. Některé mohou být separované a některé nikoli. Jednotlivé parametry se postupně označují `#1` až `#9` a musí stát v masce parametrů vzestupně za sebou bez přeskakování čísel. Maska parametrů má tedy obecně tvar:

⟨*separátor0*⟩`#1`⟨*separátor1*⟩`#2`⟨*separátor2*⟩`#3`⟨*separátor3*⟩ *atd.*

Přitom separátory jsou nepovinné. Například:

```
35 \def\a :#1#2. #3,#4#5 {...tělo definice}
```

definuje makro s pěti parametry. Uživatel je povinen za použitím makra napsat nejprve dvojtečku (viz ⟨*separátor0*⟩), dále `#1` a `#4` jsou neseparované, zatímco `#2` je separováno textem .$_{12}$ ␣$_{10}$, `#3` je separováno čárkou a konečně `#5` mezerou. Pokud třeba použijeme toto makro v kontextu:

```
36 \tracingmacros=1
37 \a: {první} druhý. třetí{,} stále třetí, 4 další
```

dostaneme do `log`u přehlednou zprávu (díky `\tracingmacros=1`):

```
\a :#1#2. #3,#4#5 ->...tělo definice
#1<-první
#2<- druhý
#3<-třetí{,} stále třetí
#4<-4
#5<-
```

Uvedeme některé záludnosti tohoto příkladu. Do `#1` se zavede neseparovaný parametr až po přeskočení mezery (za dvojtečkou). Mezera před slovem „`druhý`“ je ovšem do druhého parametru zahrnuta. Do `#4` se načte po přeskočení mezery jediný token „`4`“ a mezera za touto číslovkou pracuje jako separátor pátého parametru, který tedy zůstává prázdný. Text „`další`“ už nepodléhá expanzi našeho makra.

V těle definice se mohou vyskytovat jen takové formální parametry, které byly deklarovány v masce parametrů. Dá rozum, že pokud třeba máme v masce parametrů jen `#1` až `#5`, nemůžeme v těle definice pracovat s formálním parametrem `#6`. □

Ukážeme jednoduchý příklad, kdy nám schopnost TEXu pracovat se separovanými parametry pomůže ke zjišťování přítomnosti nějakého znaku ve stringu. Připravíme makro `\setfilename`, které načte název souboru až po mezeru a vloží tento název do makra `\filename`. Pokud je v názvu souboru tečka, zůstává `\filename` shodné s načteným názvem. Není-li v názvu tečka, makro připojí příponu `.tex`. Takový algoritmus je například v činnosti při načítání souborů primitivem `\input`.

```
\def\setfilename #1 {\def\filename{#1 }\checkdot #1.\end}
\def\checkdot #1.#2\end {\def\temp{#2}%
  \ifx\temp\empty \def\filename{#1.tex }\fi}
```

Sekvenci `\end` používáme v tomto makru nikoli ve významu povelu `\end`, který ukončuje činnost TEXu, ale jako separátor druhého parametru makra `\checkdot`. Podle toho, zda je tento parametr prázdný zjistíme, zda v názvu je či není tečka. □

Můžeme narazit na situaci, kdy nám 9 parametrů v jednom makru nebude stačit. V takovém případě řešíme problém vnořenými makry. Například pro 12 parametrů separovaných mezerou můžeme použít tento trik:

```
\def\nactidvanact #1 #2 #3 #4 #5 #6 #7 #8 {\def\prvni{#1}%
   \def\druhy{#2}\def\treti{#3} ... atd. \def\osmy{#8}\dalsi}
\def\dalsi #1 #2 #3 #4 {\def\devaty{#1}\def\desaty{#2}%
   \def\jedenacty{#3}\def\dvanacty{#4}\pouzijnactene}
```

Toto je poněkud těžkopádné řešení. Lepší bude, když načteme dvanáct parametrů do řídicích sekvencí `param1` až `param12` takto:

```
\newcount\tempnum
\def\nactidvanact {\tempnum=0 \let\next=\nactijeden \next}
\def\nactijeden #1 {\advance\tempnum by 1
   \expandafter\def \csname param\the\tempnum\endcsname {#1}%
   \ifnum\tempnum=12 \let\next=\pouzijnactene \fi
   \next}
```

Vysvětlíme si v této ukázce některé obraty. Na řádku 45 deklarujeme numerickou „proměnnou“ `\tempnum`, kterou makro nastavuje na nulu. Pomocí obratu `\let\next` je zajištěn cyklus. Dvanáctkrát pracuje `\next` jako `\nactijeden` a po třinácté se změní jeho význam na `\pouzijnactene`. Toto je zařízeno na řádku 47, kde je vždy pomocí `\advance` zvětšena hodnota `\tempnum` o jedničku. Dále na řádku 49 je test, zda už dosáhla hodnota `\tempnum` dvanáctky a pokud ano, je změněn význam `\next`. Na konci makra se pomocí `\next` makro buď opakuje nebo přechází do `\pouzijnactene`.

Na řádku 48 je definována řídicí sekvence `param1` až `param12` (podle momentální hodnoty `\tempnum`) jako text právě načteného parametru `#1`. Jak to pracuje, se čtenář jistě dozví po studiu sekce o tricích s `\expandafter` (2.2) a po seznámení s primitivy `\csname` a `\endcsname` v části B.

Výhodou tohoto řešení je skutečnost, že nyní už snadno změníme makro s dvanácti parametry za makro se 120 parametry a navíc můžeme jednoduchou úpravou

testovat konec parametrů, pokud je jejich počet proměnlivý. Při proměnlivém počtu parametrů se musíme dohodnout na značce, která bude znamenat konec textu s parametry. Nechť je to třeba hvězdička. Pak můžeme psát:

```
\def\hvezda{*}
\def\nactiparametry {\tempnum=0 \let\next=\nactijeden \next}
\def\nactijeden #1 {\advance\tempnum by 1 \def\param{#1}%
  \ifx\param\hvezda \let\next=\pouzijnactene
  \else \expandafter\def \csname param\the\tempnum\endcsname{#1}%
  \fi \next}
```

Testem `\ifx\param\hvezda` zjišťujeme, zda je načtena koncová značka (zde hvězdička). Jestliže ano, cyklus načítání parametrů je ukončen.

Pokud bychom chtěli pro koncovou značku použít konec řádku, máme u parametrů separovaných mezerou trošku více práce, protože za `\endlinechar` už nevpravíme mezeru. Na konci této sekce ukážeme poněkud složitější příklad, ve kterém tento úkol řešíme. Makro bude zpracovávat tabulku údajů, členěnou na vstupu do nestejně dlouhých řádků. □

• **Zdvojení symbolů pro parametr „#“.** V těle definice může být další konstrukce `\def`. Tato vnitřní definice může mít též parametry. Ovšem pro parametr vnitřní definice nemůžeme použít zápis `#1`, protože toto je zápis formálního parametru vnější definice. Proto je TEX vybaven následující vlastností. Pokud je v těle definice token $\boxed{\#}_6$ (na ASCII hodnotě tokenu nezáleží), pak v případě, že je následován číslicí (například $\boxed{1}_{12}$), je tato dvojice tokenů interpretována jako formální parametr a v okamžiku expanze je nahrazena příslušným aktuálním parametrem. S touto vlastností jsme se už setkali. Pokud za tokenem $\boxed{\#}_6$ následuje znovu token téže kategorie, je tato dvojice tokenů v okamžiku expanze nahrazena jediným tokenem $\boxed{\#}_6$. Například:

```
\def\obklop #1{\def\separuj ##1#1{používám ##1 a #1}\separuj}
\obklop :abc:   \obklop .text.
```

Zápis „`\obklop :abc:`“ expanduje na:

```
\def\separuj #1:{používám #1 a :}\separuj abc:
```

a tento zápis konečně expanduje na text: „`používám abc a :`“. Kdybychom chtěli mít parametr v definici, která je uvnitř už vnitřní definice, píšeme místo dvou znaků `#` čtyři tyto znaky. To se vyskytuje velmi zřídka. S ještě hlubším vnořením definic (kdy by bylo potřeba použít osm znaků `#`) jsem se ještě nesetkal. □

V závěru našeho povídání o parametrech maker shrneme, co lze a co nelze použít jako separátor parametru. Víme, že separátorem může být libovolná posloupnost

tokenů, která se neexpanduje a u níž se kontroluje úplná shoda. Tj. shodovat se musí ASCII hodnoty i kategorie a v případě tokenů typu řídicí sekvence se musí shodovat názvy. Tyto řídicí sekvence mohou, ale nemusí být definovány.

V separátoru nelze použít tokeny kategorie 1, 2 a 6. Tokeny ostatních kategorií použít lze. Proč nelze použít {}$_1$, nás asi napadne. Je to především ze syntaktických důvodů, protože v konstrukci \def tento znak ukončuje ⟨*masku parametrů*⟩ a zahajuje ⟨*tělo definice*⟩. Podobně token #$_6$ lze použít jen v kontextu parametru.

Existuje jedna výjimka, která programátorovi maker umožní použít separátor {}$_1$, ovšem zcela jinak se zapisuje a poněkud jinak se chová, než běžný separátor. Pokud je v masce parametrů zcela posledním tokenem masky znak kategorie 6 (tj. vidíme obvykle zapsanou dvojici „#{“), pak se tento zápis chová jako separátor shodný s otevírací závorkou těla definice. TEX tuto závorku ovšem přidá při expanzi na konec rozvoje makra, což s běžnými separátory nedělá. Srovnejte:

```
\def\A #1#{...}  \def\B #1[{...}
\A abc{---} % #1 je "abc" a řádek expanduje na "...{---}"
\B abc[---] % #1 je "abc" a řádek expanduje na "...---]"
```

• **\edef a přátelé.** V dalším textu se zaměříme na alternativy k primitivu \def. Začneme primitivem \edef. Zatímco u konstrukce \def při fázi „učení“ neprobíhá expanze, tj. tokeny z těla definice se ukládají do paměti v neexpandovaném tvaru, při použití \edef probíhá expanze už při fázi učení a do paměti se ukládají tokeny ve tvaru, v jakém by nakonec po úplné expanzi přicházely do hlavního procesoru. Například:

```
\def\A  {\the\pageno}  % \A je makro s tělem "\the\pageno"
\edef\B {\the\pageno}  % \B je makro s tělem "39"
```

Konstrukce \edef má stejné možnosti, jako \def. Umožňuje tedy i deklarování parametrů, ovšem tato vlastnost se obvykle nepoužívá. Uvedeme si jednoduchý příklad použití \edef. Pomocí makra \naber načítáme postupně číslice a střádáme je do jedné posloupnosti ukryté v makru \celkem. Pokud třeba napíšeme:

```
\naber 012 \naber 03456 \naber 13
```

bude tělo makra \celkem obsahovat „0120345613“. To zařídíme následujícím kódem:

```
\def\celkem{}
\def\naber #1 {\edef\celkem{\celkem #1}}
```

Kdybychom místo \edef použili \def, makro \celkem by obsahovalo ve svém těle token celkem, což by při použití tohoto makra odstartovalo rekurzivní kolotoč

končící havárií typu `TeX capacity exceeded`. Na druhé straně při `\edef` je v těle `\celkem` nejprve původní *obsah* makra `\celkem` a k němu jsou připojeny nové hodnoty z `#1`. □

Povely `\def` i `\edef` vlastně přiřazují definované řídicí sekvenci význam makra, které je určeno svým tělem a maskou parametrů. Toto přiřazení je lokální (pokud samozřejmě není nastaveno kladné `\globaldefs`, což nebývá obvyklé). Pod pojmem *lokální přiřazení* rozumíme přiřazení, které je platné jen uvnitř skupiny, ve které bylo provedeno. *Skupiny* se v TEXu otevírají pomocí povelu {$_1$ (na ASCII hodnotě nezáleží) nebo primitivem `\begingroup` nebo alternativou definovanou v plainu `\bgroup`. Skupiny zavíráme povelem }$_2$ nebo primitivem `\endgroup` nebo alternativou z plainu `\egroup`. Po uzavření skupiny se všechna lokální přiřazení vracejí k původním hodnotám, jaké byly před vstupem do skupiny. Například:

```
68 \def\A{ABC}
69 { \def\A{XYZ} \def\B{PQR} }
70 zde \A je znovu makro "ABC" a \B je nedefinováno.
```

Pokud bychom chtěli definovat *globálně* (tj. aby po ukončení skupiny definice makra zůstala nezměněna), je potřeba před primitiv `\def` nebo `\edef` psát prefix `\global`, například `\global\def\B{PQR}`.

TEX je vybaven primitivními zkratkami, které lze použít místo prefixu global:

- `\gdef` je totéž jako `\global\def`,
- `\xdef` je totéž jako `\global\edef`. □

Už jsme se seznámili s prefixy `\long` (na straně 34) a s prefixem `\global` (právě nyní). Zbývá zmínit ještě poslední prefix `\outer`. Pokud se použije tento prefix, je definovaná řídicí sekvence použitelná jen jako „vnější makro“, tj. není dovoleno ji použít v těle jiných definic.

Například makro plainu `\newdimen` je definováno s prefixem `\outer`, takže toto makro, které deklaruje novou „proměnnou“ typu ⟨*dimen*⟩, nelze použít v těle definice, ale jen na „vnější úrovni“. Kdybychom psali:

```
71 \def\deklarujBLA{\newdimen\BLA}
72 \deklarujBLA
```

tak nám to havaruje už na řádku 71 s konstatováním: `Forbidden control sequence found while scanning definition of \deklarujBLA`. Je to proto, že řídicí sekvence `\newdimen` není dovoleno psát v těle žádné definice, neboť byla definována s prefixem `\outer`.

Naskýtá se otázka, proč se Knuth rozhodl implementovat takový prefix a dokonce jej v plainu použil. Vždyť to jen omezuje možnosti definované řídicí sekvence! Použití tohoto prefixu ovšem může zpřehlednit ladění maker. Stačí totiž někde zapomenout jedinou závorku typu „}“ a tělo definice nám končí úplně někde jinde, než bychom předpokládali. Pokud občas „mezi definicemi“ používáme deklarátory typu `\newdimen`, `\newcount` atd., pak můžeme takové opomenutí závorky včas rozpoznat, a o to, myslím, autorovi TEXu šlo. Pokud se nám to nelíbí, pišme při generování plainu `\let\outer=\relax` a teprve potom generujme formát. □

• **Příklad.** V závěrečném příkladě této sekce předvedeme makro, které čte své parametry ze vstupní tabulky. Jednotlivé údaje jsou od sebe odděleny mezerou a je zachováno členění do řádků. Přitom na každém řádku může být nestejné množství údajů. Uživatel makra napíše třeba:

```
\tabulka
 11 112  15  18
 32   4   9  15  17  27
142  17 321  92 141
\konectabulky
```

a makro `\tabulka` postupně načítá údaje z prvního řádku (11, 112, 15, 18), nějakým způsobem je zpracovává, pak načítá údaje z druhého řádku, atd. Způsob tisku údajů nebudeme v makru řešit, to záleží na konkrétní aplikaci. Pro nás je nyní důležité správně naprogramovat vstupní stranu makra `\tabulka`, aby zůstalo zachováno členění vstupní informace na řádky a údaje. Může se nám to hodit například při programování maker pro jízdní řády.

Makro `\tabulka` může vypadat třeba takto:

```
\newcount\tempnum
{\catcode`\^^M=13 % budeme pracovat s koncem řádku "^^M"
 \gdef\tabulka{\bgroup \catcode`\^^M=13 \let^^M=\jedenradek}%
 \gdef\jedenradek #1^^M{\def\temp{#1}%
    \ifx \temp\posledniradek \def\next ##1 ^^M {\konectab}%
    \else \message{Další řádek:}%
          \tempnum=0 \let\next=\polozka %
    \fi \next #1 ^^M }%
 \gdef\poslednipolozka{^^M} \gdef\posledniradek{\konectabulky}%
}
\def\polozka #1 {\advance\tempnum by 1 \def\temp{#1}%
   \ifx \temp\poslednipolozka \let\next=\jedenradek
   \else \message{údaj č. \the\tempnum: #1.}%
   \fi \next}
\def\konectab{\egroup}
```

Makro pracuje se znakem `^^M`, který nám na konec každého řádku vloží input procesor. Na řádcích 79–87 je pro něj přechodně nastavena kategorie 13. Všechny definice, které nechceme po opuštění skupiny na řádku 87 zapomenout, musíme proto psát jako `\gdef` a nikoli `\def`. Všechny řádky v oblasti definic, na kterých je `^^M` aktivní, musíme ukončit procentem, abychom zakryli tento znak a jeho nežádoucí činnost.

Makro `\tabulka` otevře skupinu (`\bgroup`), nastaví znak `^^M` jako aktivní a tento znak bude pracovat jako makro `\jedenradek`. Protože uživatel napíše sekvenci `\tabulka` na samostatný řádek, máme jistotu, že znak `^^M` z konce tohoto řádku začne pracovat jako `\jedenradek`.

Makro `\jedenradek` přečte celý řádek až do konce řádku (`^^M`) jako parametr a testuje tento parametr na shodnost s makrem `\posledniradek`. Na posledním řádku totiž uživatel napsal sekvenci `\konectabulky`. Nastala-li tato situace, je `\next` na řádku 82 definováno jako „přeskoč tuto sekvenci včetně konce řádku a přejdi do závěrečných rutin definovaných v `\konectab`“. Jinak je `\next` nastaveno na význam `\polozka`. Na řádku 85 je důležitý trik. Makro tam vrací obsah celého řádku do čtecí fronty (za tokenem `\next`), přidá mezeru, přidá konec řádku a ještě jednu mezeru.

Makro `\polozka` nyní z takto připravené čtecí fronty ubere jen jeden údaj končící mezerou a zpracovává jednotlivé údaje cyklem podobně, jako jsme ukázali na straně 38 na řádcích 51–56. Mezi jednotlivými údaji je jediná mezera, ačkoli jich tam uživatel napsal pro přehlednost třeba více (o to se postaral token procesor). Koncovou značkou řádku je znak `^^M`. Je-li dosažena, je zpětně nastaveno `\next` jako `\jedenradek` a cyklus přes řádky se opakuje.

Z ukázky je patrné, že pokud dobře umíme makrojazyk TEXu, nebudeme nutit uživatele při zápisu rozsáhlých a často se opakujících tabulek separovat údaje obvyklým znakem `&`, řádky pomocí `\cr` a údaje maker pomocí složených závorek `{...}`. Tím bychom popularitě TEXu u nic netušících uživatelů asi moc nepomohli. Raději přizpůsobíme vstupní stranu našeho makra tak, aby dokázalo číst libovolně strukturovaný vstupní text. □

2.2. Triky s `\expandafter`

V TEXovské literatuře (i zde v části B) se můžeme dočíst, že `\expandafter` je primitiv, který mění pořadí expanze následujících tokenů. Posloupnost:

93 `\expandafter`⟨*token1*⟩⟨*token2*⟩

se po prvním průchodu změní v posloupnost:

94 ⟨*token1*⟩⟨*výsledek expanze token2*⟩

a tato posloupnost znovu vstupuje do expand procesoru v druhém průchodu. Znamená to, že se v druhém průchodu případně expanduje ⟨*token1*⟩ (je-li expandovatelný), který může přijmout jako parametry ⟨*výsledek expanze token2*⟩. □

Může se stát, že ⟨*token2*⟩ je znovu primitiv \expandafter. Ten se v rámci prvního průchodu expanduje, tj. přeskočí se následující ⟨*token3*⟩ a expanduje ⟨*token4*⟩. Druhý průchod pak pracuje s posloupností:

```
95  ⟨token1⟩⟨token3⟩⟨výsledek expanze token4⟩
```

Jako příklad si uvedeme tisk římského čísla velkými písmeny. Nechť v registru \num máme hodnotu čísla. Primitiv \romannumeral vrací hodnotu následujícího ⟨*number*⟩ jako římské číslo, ovšem vyjádřeno malými písmeny. Na velká písmena konvertuje primitiv \uppercase.

Kdybychom napsali:

```
96  \uppercase{\romannumeral\num}
```

dočkáme se na výstupu malých písmen. Je to proto, že \uppercase konvertuje posloupnost tokenů uzavřenou v závorkách. V našem případě tato posloupnost má dva tokeny romannumeral a num. Toto nejsou samostatné znaky (přesněji tokeny typu uspořádaná dvojice), proto je primitiv \uppercase nechá nezměněny. Teprve poté se expanduje primitiv \romannumeral, ovšem výsledkem zůstanou malá písmena.

Abychom dostali požadovaný výsledek, musíme provést:

```
97  \expandafter\uppercase \expandafter {\romannumeral\num}
```

V prvním průchodu se přeskočí uppercase a {$_1$. Dále se \romannumeral\num expanduje podle hodnoty \num řekněme na viii. Do druhého průchodu tedy vstupuje posloupnost:

```
98  \uppercase {viii}
```

Tato posloupnost se konvertuje na VIII, což je přesně to, co jsme potřebovali. Kdybychom nezapsali druhé \expandafter, ale použili pouze:

```
99  \expandafter\uppercase {\romannumeral\num}
```

příkaz \expandafter by se snažil expandovat token {$_1$. To ovšem není expandovatelný token, proto jej nechá beze změny a do druhého průchodu půjde totéž, co jsme napsali v řádku 96. Takže se velkých písmen nedočkáme.

Protože `\uppercase` je povel, který se zpracovává až na úrovni hlavního procesoru, je možné si jedno `\expandafter` ušetřit a psát:

```
100 \uppercase \expandafter {\romannumeral\num}
```

Povel `\uppercase` totiž dostane token $\boxed{\{}_1$ až po provedení všech aktivit expand procesoru. Kdyby ale mělo být `\uppercase` makro s jedním parametrem uzavřeným do závorek, je samozřejmě nutné použít řešení z řádku 97. □

Uveďme další příklad. Chceme změnit pořadí expanze tří maker `\A\B\C` zapsaných takto za sebou. Přitom chceme, aby se nejprve expandovalo `\C`, pak `\B` (které může vzít do parametrů výsledek expanze `\C`) a nakonec `\A` (které může vzít do parametrů výsledek expanze `\B`). Řešení úlohy je následující:

```
101 \let\ex=\expandafter
102 \ex \ex \ex \A \ex \B \C
```

Pro větší přehlednost si podtrhneme, co se expanduje v prvním průchodu (`\ex` je zkratka za `\expandafter`).

```
103 \ex \ex \ex \A \ex \B \C
```

Do druhého průchodu tedy vstupuje:

```
104 \ex \A \B ⟨výsledek expanze \C⟩
```

Dále už je situace jasná. □

Ukažme si nyní makro `\letcs`, které má dva parametry `#1` a `#2`, přičemž se provede:

```
105 \let#1=#2
```

kde `#1` a `#2` jsou řídicí sekvence, jejichž identifikátory jsou určeny obsahem parametrů `#1` a `#2`. Takové řídicí sekvence se dají vyprodukovat pomocí dvojice primitivů `\csname ... \endcsname`. Makro vypadá následovně:

```
106 \let\ex=\expandafter
107 \def\letcs #1#2{\ex\ex\ex \let
108   \ex\ex \csname#1\endcsname \csname#2\endcsname}
```

Při použití makra třeba ve tvaru `\letcs{s1}{s2}` se při prvním průchodu expandují podtržené tokeny:

```
109 \ex\ex\ex \let \ex\ex \csname s1\endcsname
```

Do druhého průchodu tedy vstupuje:

```
110 \ex \let \ex s1 \csname s2\endcsname
```

V tomto případě se po druhé expanzi TEX znovu vrátí, tentokrát k třetímu průchodu, do kterého už vstupuje posloupnost pro vykonání požadované akce:

```
111 \let s1 s2
```

Existuje ještě jedno řešení našeho problému, které se zdá být velmi elegantní. Jak to pracuje si už čtenář jistě rozmyslí sám:

```
112 \def\letcs #1#2{\expandafter \let\csname#1\expandafter\endcsname
113                                \csname#2\endcsname}
```

□

Pomocí \expandafter se dá „odejít“ z konstrukce podmínky typu \if dříve, než se provede vyhodnocené makro nebo povel v podmínce. Uvedeme si příklad. Představme si, že po zjištění, zda je makro \minus použito v odstavcovém módu, chceme expandovat makro \testnextchar. Makro odebere jeden token, který následuje za zápisem makra \minus, a provede podle toho nějakou akci. V ostatních módech se \minus expanduje na znak „-“. Nejprve uvedeme řešení, které pracuje s \let\next:

```
114 \def\minus{\let\next=\relax
115    \ifhmode
116       \ifinner -%
117       \else  \let\next=\testnextchar
118       \fi
119    \else
120       -%
121    \fi \next}
122 \def\testnextchar #1{...}
```

Toto řešení se ovšem opírá o povely hlavního procesoru (\let), což někdy není žádoucí. Tentýž problém lze řešit pouze na úrovni expand procesoru takto:

```
123 \def\minus{%
124    \ifhmode
125       \ifinner -%
126       \else  \expandafter\expandafter\expandafter \testnextchar
127       \fi
128    \else
129       -%
130    \fi}
131 \def\testnextchar #1{...}
```

Kdybychom nenapsali řadu tří `\expandafter`, `\testnextchar` by odebralo následující `\fi` z řádku 127, a to my nechceme. Kdybychom použili jen jedno `\expandafter`, pak sice ukončíme `\fi`, ale makro `\testnextchar` by odebralo následující `\else` z vnější podmínky (viz řádek 128), což také nechceme.

Má-li se vykonat `\testnextchar`, v prvním průchodu se provede první a třetí `\expandafter` a ukončí se vnitřní podmínka (`\fi`). V druhém průchodu se provede prostřední `\expandafter` a ukončí se `\else` vnější podmínky, a tedy celá konstrukce makra `\minus`. Ve třetím průchodu se teprve expanduje makro `\testnextchar`, které nyní skutečně odebere token, jež uživatel zapsal za sekvenci `\minus`. □

Poslední ukázka použití `\expandafter` se týká definic aktivních znaků. Přitom nechceme, aby tyto znaky zůstaly aktivní pořád. Běžný postup pak vypadá takto:

```
\def\hvezda{*}
{\catcode`\*=13
 \gdef*{tady něco proveď a občas expanduj neaktivní \hvezda}
}
```

Toto řešení má dvě mírné nevýhody. Především význam aktivního znaku je definován globálně. Dále se v těle makra odvoláváme na neaktivní znak `*` prostřednictvím makra `\hvezda`, a nikoli přímo. Ukažme si řešení, které využívá `\expandafter`:

```
{\catcode`\*=13
\expandafter }\expandafter \def\noexpand*{%
   tady něco proveď a občas použij neaktivní *}
```

Poté, co je nastavena kategorie znaku „`*`“ na aktivní, se v prvním průchodu provedou podtržené sekvence:

```
\expandafter }2 \expandafter def \noexpand *13
```

Vidíme tedy, že si `\noexpand` potřebovalo „sáhnout“ na následující token $\boxed{*}_{13}$. V tomto okamžiku mu tedy byla přidělena kategorie. V druhém průchodu se uzavře skupina pomocí $\boxed{\}}_2$ a tím je ukončena aktivita znaku „`*`“. Definovaný token už ale má kategorii 13. V těle definice se zase pracuje s tokeny $\boxed{*}_{12}$. □

2.3. Podmínky typu `\if`

Pro větvení „výpočtu“ podle podmínek jsou na úrovni expand procesoru zavedeny primitivy typu `\if`. Jedná se o tyto primitivy: `\if`, `\ifx`, `\ifcat`, `\ifnum`, `\ifodd`, `\ifdim`, `\ifeof`, `\iffalse`, `\iftrue`, `\ifhbox`, `\ifvbox`, `\ifvoid`, `\ifhmode`, `\ifvmode`, `\ifmmode`, `\ifinner` a `\ifcase`. Podrobněji o každém z nich viz část B.

Seznam tokenů, které se mají vykonávat v závislosti na splnění podmínky, je oddělen separátory \else, \fi a v případě \ifcase ještě \or. Konstrukce podmínek má (s výjimkou \ifcase) obecný tvar:

```
\if...⟨podmínka⟩ ⟨je splněna⟩ \else ⟨není splněna⟩ \fi
```

Část „\else ⟨*není splněna*⟩“ se může vynechat. Závěrečné \fi je povinné. Uvnitř textu ⟨*je splněna*⟩ a ⟨*není splněna*⟩ se mohou vyskytovat další vložené konstrukce typu \if. □

Negaci zapíšeme pomocí \else takto:

```
\if...⟨podmínka⟩\else ⟨není-li splněna⟩\fi
```

Znamená to tedy, že lze vynechat část před \else. Logické AND implementujeme následovně:

```
\if...⟨podmínka A⟩
  \if...⟨podmínka B⟩ ⟨zde platí A AND B⟩
  \else ⟨zde platí A AND (NON B)⟩
  \fi
\fi
```

Nebo:

```
\if...⟨podmínka A⟩ \else
   \if...⟨podmínka B⟩ ⟨zde platí (NON A) AND B⟩
   \else ⟨zde platí NON (A OR B)⟩
   \fi
\fi
```

Logické OR dá trochu více práce:

```
\def\AorB{⟨co vykonat při A OR B⟩}
\if... ⟨podmínka A⟩ \AorB
\else \if... ⟨podmínka B⟩ \AorB
      \fi
\fi
```

Ostatní logické operace se vytvoří podobným způsobem. □

Pomocí konstrukcí s \if... se implementují též cykly v makrech. Sestavme například makro \provedvsechny, které aplikuje makro \delej postupně na všechny tokeny, které následují. Mezery se ignorují, protože vnitřní makro \opakuj odebírá

jednotlivé tokeny do parametru `#1`, který není separován. Posledním tokenem, po jehož načtení se cyklus ukončí, je dvojtečka.

```
157 \def\provedvsechny{\let\next=\opakuj \next}
158 \def\opakuj #1{\if :#1\let\next=\relax
159                \else \delej{#1}%
160                \fi \next}
161 % Použití makra:
162 \provedvsechny a c d e + 5 6 8 j 9 t r K J :
```

Taková konstrukce na první pohled vypadá jako rekurze. Rekurze je samozřejmě v makrech možná, ovšem případ v naší ukázce je svým způsobem výjimečný. Algoritmus pro start rekurzivního volání makra nejprve ošetří, zda je nezbytně nutné si „zapamatovat“, kam se po ukončení rekurzivního volání vrátit. V naší ukázce takové „pamatování“ není nutné, protože za aplikací sekvence `\next` (ve významu `\opakuj`) nenásledují žádné další tokeny, které se v rámci (vnějšího) makra `\opakuj` mají dále vykonávat. TEXovská paměť tedy v tomto případě není zatížena údaji o místě návratu z rekurzivního volání, a proto na uvedenou konstrukci můžeme pohlížet jako na obyčejný cyklus. Viz též stranu 297, heslo *stack_size*.

Podobným způsobem je implementováno makro plainu `\loop`, viz část B. □

Jedním z častých omylů začínajících adeptů na programování maker v TEXu je skutečnost, že si nedají pozor na možnost expanze maker uvnitř podmínky. Přesněji: ⟨*podmínka*⟩ se sestaví z tokenů následujících za konstrukcí `\if...` *až po provedení úplné expanze*. Výjimkou z tohoto pravidla je pouze použití primitivu `\ifx`, kde následující ⟨*token1*⟩ a ⟨*token2*⟩ mohou být makra a testuje se, zda jsou tato makra shodně definovaná. V tomto případě se jedná vlastně o porovnávání stringů, kde porovnávané stringy tvoří obsah definice maker ⟨*token1*⟩ a ⟨*token2*⟩.

Uvedeme příklad, jak se to nemá dělat: Chceme zjistit třeba shodnost kategorií dvou tokenů pomocí `\ifcat`⟨*token1*⟩⟨*token2*⟩ a máme:

```
163 \def\a{Abc}
164 \ifcat \a 0 ⟨co provést⟩ \fi
```

pak se neporovnává `\a` s `0`, ale porovná se `A` s `b`. Protože mají (obvykle) stejnou kategorii, provede se „`c0` ⟨*co provést*⟩“. Všimněte si, že znak `0` se provádí a není součástí podmínky. Abychom zabránili nežádoucí expanzi, můžeme místo řádku 164 psát:

```
165 \ifcat \noexpand \a 0 ⟨co provést⟩ \fi
```

Praktický smysl má tento příklad teprve tehdy, když třeba testujeme token, který byl načten do parametru (viz například řádek 180). □

Jiný nesprávný příklad:

```
\def\a#1 {\if #1:\message{parametr je dvojtečka}\fi}
\a 12  \a :  \a 11
```

Zde je chybou skutečnost, že parametr #1 je separován mezerou, tudíž se do něj může při použití makra \a zavést více tokenů než jeden. Při použití \a 12 se testuje shodnost jedničky s dvojkou. Nejsou shodné, proto se :\message neprovede. V případě \a : se testuje shodnost dvojtečky z parametru s dvojtečkou v makru. Jsou shodné, provede se tedy \message. V posledním případě \a 11 platí shodnost jedničky s jedničkou, proto se provede :\message, tj. vytiskne se dvojtečka z makra a na terminálu nám to lže.

Upozorňujeme na skutečnost, že ani v případě, že parametr makra obsahuje jediný token, nemůžeme si být absolutně jisti konstrukcí \if #1⟨*token*⟩. Je to proto, že v parametru #1 může být řídicí sekvence, která nám po expanzi může zcela změnit testovací podmínku. V takovém případě je vhodné před parametr #1 psát \noexpand nebo použít \ifx:

```
\def\param{#1}\def\testtoken{⟨token⟩}%
\ifx\param\testtoken ...
```

Na druhé straně, pokud víme určitě, z jaké množiny jsou možné parametry, pak použití primitivu \ifx s předchozími \def není nutné. Například testujeme, zda je #1 prázdný. Můžeme postupovat (poněkud těžkopádně) třeba takto:

```
\def\param{#1}\ifx\param\empty ...
```

Toto řešení vyžaduje akci na úrovni hlavního procesoru (provedení \def). Může se stát, že potřebujeme vyřešit celý úkol na úrovni expand procesoru. V případě testu, zda je #1 prázdný, stačí psát:

```
\if:#1: ⟨je prázdný⟩ \else ⟨není prázdný⟩ \fi
```

Zde se předpokládá, že parametr nikdy nebude obsahovat znak „:“. Jak to funguje? Není-li #1 prázdný, pak dvojtečka určitě nebude shodná s prvním tokenem parametru, takže se další tokeny přeskakují až po \else. Je-li #1 prázdný, pak se sejde první dvojtečka s druhou, ty jsou stejné, a provede se část ⟨*je prázdný*⟩. □

Uvedeme si další příklad, ve kterém implementujeme cyklus typu for.

```
\def\for #1#2\endfor{\def\forbody##1{#2}\let\next=\forcycle
   \next #1^^X}
\def\forcycle#1{\if #1^^X\let\next=\relax
                \else \forbody #1%
```

```
176                 \fi \next}
177 % V těle cyklu \for se může vyskytnout #1.
178 % Ukázka použití makra \for:
179 \for{ABC123bc}\message{zpracovávám #1}\endfor
```

Zde samozřejmě vycházíme z předpokladu, že uživatel makra `\for` nikdy mezi parametry nepoužije znak Ctrl-X. Také použití řídicích sekvencí v parametrech bude působit potíže. Pokud dovolíme používat v parametrech cyklu i řídicí sekvence, pak místo `\if #1^^X` na řádku 174 pišme:

```
180 \if \noexpand #1^^X
```

a pro vyzkoušení třeba napišme:

```
181 \for{A B C \A \B \C}\message{zpracovávám \string#1}\endfor
```

□

V závěrečné části této sekce se budeme zabývat dalším problémem, který souvisí s algoritmem expand procesoru. V konstrukcích typu `\if...` se po vyhodnocení platnosti podmínky přeskakují tokeny, které se nemají provádět. Tyto řady tokenů jsou separovány sekvencemi `\else` nebo `\fi`. Například:

```
182 \iftrue  ⟨toto se provede⟩  \else ⟨toto se přeskočí⟩ \fi
183 \iffalse ⟨toto se přeskočí⟩ \else ⟨toto se provede⟩  \fi
```

Při přeskakování tokenů se *neprovádí expanze*. Algoritmus si všímá jen tokenů, které mají význam primitivů `\if...` (viz seznam všech těchto primitivů na začátku sekce, str. 46) nebo význam `\else` nebo `\fi`. Pod pojmem „token má význam primitivu“ zde rozumíme, že token je primitivem samotným nebo sdílí s ním společný význam prostřednictvím dříve provedeného `\let`. Pod pojmem „všímat si tokenů“ myslíme, že algoritmus rozpoznává vložené konstrukce typu `\if...` i uvnitř přeskakovaného textu. Například:

```
184 \iffalse ⟨přeskakuji⟩
185     \if... % tady je vnořený \if
186         ⟨pořád přeskakuji⟩
187     \else % ten patří ke vnořenému \if
188         ⟨pořád přeskakuji⟩
189     \fi % ten patří ke vnořenému \if
190 \fi % to je konec původního \iffalse
```

Skutečnost, že při přeskakování se neprovádí expanze, ale jistých tokenů si algoritmus všímá, může zpočátku působit potíže. Uvedeme si ukázku implementace dalšího typu testu. Test nazveme `\ifdigit`. Tvar `\ifdigit` ⟨*token*⟩ se expanduje na `\iftrue` v případě, že ⟨*token*⟩ je číslice, a na `\iffalse` v případě, že číslice není.

V následujících ukázkách budeme záměrně dělat chyby a postupně je opravovat, až v závěru dospějeme k uspokojivému řešení.

První, co by nás mohlo napadnout, načrtneme třeba takto:

```
% V ASCII jsou číslice v pozicích 48 až 57
\def\ifdigit #1{\ifnum`#1>47 \ifnum`#1<58 \let\next=\iftrue
                             \else \let\next=\iffalse
                             \fi
                \else \let\next=\iffalse
                \fi \next}
```

Z popisu algoritmu přeskakování tokenů v konstrukcích typu `\if...` okamžitě plyne, že toto řešení nemůže vůbec fungovat. Sekvencí `\iftrue` a `\iffalse`, které jsou uvnitř konstrukcí `\ifnum`, si totiž bude všímat algoritmus na přeskakování a bude k nim hledat `\else` a `\fi`. Pokud nahradíme sekvenci `\let\next=\iftrue` pomocí:

```
\expandafter\let \expandafter\next \csname iftrue\endcsname
```

a podobně se zachováme k `\iffalse`, tyto problémy odpadnou. Při přeskakování si TEX takových posloupností všímat nebude.

Elegantnější je zřejmě připravit si provedení akce `\let\next=\iftrue` do separátního makra a v těle konstrukce `\ifnum` volat toto makro. Pro tyto účely je vytvořen v plainu alokátor `\newif`. Po alokaci:

```
\newif\ifnext
```

máme k dispozici makro `\nexttrue`, které provede `\let\ifnext=\iftrue`, a dále makro `\nextfalse`, které provede `\let\ifnext=\iffalse`. Naše ukázka tedy vypadá takto:

```
\newif\ifnext
\def\ifdigit #1{\ifnum`#1>47 \ifnum`#1<58 \nexttrue
                             \else \nextfalse
                             \fi
                \else \nextfalse
                \fi \ifnext}
```

Toto už bude fungovat, ale při použití ve vnějším prostředí se můžeme setkat se situací, kdy se makro nebude chovat korektně. Ukážeme si to na příkladě makra `\countdigits`, které v posloupnosti tokenů počítá, kolik tokenů je typu číslice:

```
205 \newcount\tempnum
206 \def\countdigits #1{\tempnum=0 \let\next=\cykldigits \next#1^^X}
207 \def\cykldigits #1{\if #1^^X\let\next=\relax
208    \else \ifdigit #1\advance\tempnum by1
209          \fi
210    \fi \next}
211 % Použití makra \countdigits:
212 \countdigits{12345dghjhg12} \message{Počet číslic: \the\tempnum}
```

Pokud tento kód vyzkoušíme, obdržíme chybové hlášení:

```
! Extra \fi.
\next ...#1 \advance \tempnum by1 \fi \fi
                                          \next
l.13 \countdigits{12345dghjhg12}
                                  \message{Počet číslic: ...
?
```

Jednotlivé otočky cyklu proběhly bez problémů. K chybě došlo až při zjištění, že je třeba cyklus opustit. Je to proto, že v této situaci podmínka `\if #1^^X` (řádek 207) vyšla jako true a bylo potřeba přeskočit tokeny za `\else` (řádek 208–209). Tam ale token `\ifdigit` nemá žádný z významů primitivů typu `\if...`, a proto `\fi` vztažené k `\ifdigit` (na řádku 209) je chápáno jako konec k `\if #1^^X` a další `\fi` na řádku 210 způsobí chybu.

Pro překonání tohoto problému se mi osvědčilo použít makro `\test`, které vezme první token zapsaný za ním (a případně další) a expanduje na `\iftrue` nebo `\iffalse`. Přitom ten první token je sekvence `\ifdigit`, pro kterou je na začátku zpracování řečeno `\let\ifdigit=\iffalse`. Potom posloupnost

```
213 \test \ifdigit #1
```

pracuje dvěma způsoby podle toho, zda je přeskakována nebo vykonávána. Je-li vykonávána, expanduje se tato posloupnost na `\iftrue` nebo `\iffalse` v závislosti na tom, zda `#1` je číslice nebo ne. Je-li přeskakována, je token `\test` nezajímavý a token `\ifdigit` má význam `\iffalse`, tj. struktura vnořených konstrukcí typu `\if...` zůstane zachována. Vlastní test na číslici pak nemůžeme řešit v makru `\ifdigit`, protože tato sekvence plní jinou úlohu. Místo toho použijeme makro `\testdigit`. Makro `\test` je vhodné udělat univerzálně tak, že dvojice

```
214 \test\ifcokoli
```

se expanduje na `\testcokoli`. Tím můžeme snadno přidávat další testy na speciální případy. Například `\ifundefined` a `\ifaccented` bychom řešili v makrech

\testundefined a \testaccented. Náš příklad s počítáním číslic v řadě tokenů (nyní už v definitivní podobě) vypadá takto:

```
\def\test #1{\expandafter\maskif\string#1:}
\expandafter\def\expandafter\maskif\string\if #1:{%
   \csname test#1\endcsname}
\newif\ifnext
\let\ifdigit=\iffalse
\def\testdigit #1{\ifnum‘#1>47 \ifnum‘#1<58 \nexttrue
                               \else \nextfalse
                               \fi
                  \else \nextfalse
                  \fi \ifnext}
\newcount\tempnum
\def\countdigits #1{\tempnum=0 \let\next=\cykldigits \next#1^^X}
\def\cykldigits #1{\if ^^X#1\let\next=\relax
                    \else \test\ifdigit #1\advance\tempnum by1
                                \fi
                    \fi \next}
\countdigits {12345dghjhg12} \message{\the\tempnum}
```

V této ukázce je zajímavá činnost makra \test a \maskif. Věnujme jim pár slov. Při použití například \test\ifcokoli se provede

```
maskif \12 i12 f12 c12 o12 k12 o12 l12 i12 :12
```

protože byl nejprve na sekvenci \ifcokoli aplikován primitiv \string. Makro \maskif je definováno se separátorem $\boxed{\backslash}_{12}\boxed{\texttt{i}}_{12}\boxed{\texttt{f}}_{12}$ následovaným parametrem #1 se separátorem $\boxed{:}_{12}$. Do parametru se tedy zavede vše mezi $\boxed{\backslash}_{12}\boxed{\texttt{i}}_{12}\boxed{\texttt{f}}_{12}$ a dvojtečkou, tj. „cokoli“. Makro se pak prostřednictvím \csname a \endcsname expanduje na \testcokoli. Vidíme, že makro \test tedy vyžaduje jako parametr řídicí sekvenci, jejíž identifikátor musí začínat na if. Podobně funguje například alokátor \newif v plainu. □

2.4. Registry pro uchování posloupností tokenů (\toks)

TEX při své činnosti pracuje s tzv. *registry*, což jsou rezervovaná místa v paměti, kam lze vložit určitý typ informace a později ji znovu použít. O registrech se podrobněji zmíníme v sekci 3.3. V této sekci se zaměříme jen na jeden typ registrů: ⟨*tokens*⟩. S tímto typem se pracuje zvláště na úrovni expand procesoru.

Do registru typu ⟨*tokens*⟩ lze vložit libovolnou posloupnost tokenů. Délka této posloupnosti je omezena jen velikostí paměti TEXu. Kromě pojmenovaných registrů tohoto typu, které nějak souvisejí s dalšími algoritmy TEXu (např. \everypar,

`\output`), máme k dispozici registry `\toks0` až `\toks255`, kterými se budeme nyní zabývat.

Uložení do registru a získání informace z registru se provede tímto způsobem:

```
\toks0={abc\cosi {&$}: } % Uložení do registru
\the\toks0               % expanduje na obsah registru
```

V tomto příkladě jsme (při běžném nastavení kategorií z plainu) do registru `\toks0` uložili posloupnost:

a$_{11}$ b$_{11}$ c$_{11}$ cosi {$_{1}$ &$_{4}$ $$_{3}$ }$_{2}$:$_{12}$ ␣$_{10}$

Vidíme, že obklopující závorky nejsou do obsahu registru zařazeny. Při ukládání do registru se bere v úvahu balancovaný text, podobně jako u těla definic v konstrukcích typu `\def`.

Skoro nikdy se nepracuje s registry ve tvaru `\toks13` nebo `\toks117` přímo, ale používají se zástupné řídicí sekvence, které jsou deklarovány primitivem `\toksdef`. Aby v přidělování „adres“ (tj. čísel registrů `\toks`) byl pořádek, používají programátoři maker plainovské makro `\newtoks`, které pomocí primitivu `\toksdef` přidělí deklarované řídicí sekvenci význam konkrétního registru `\toks`. Přidělování čísel registrů je v kompetenci makra `\newtoks` a programátor se o to nemusí starat. Naše předchozí ukázka by tedy mohla vypadat takto:

```
\newtoks\promenna
\promenna={abc\cosi {&$}: } % Uložení do registru
\the\promenna               % expanduje na obsah registru
```

Pokud se nám někdy hodí pracovat přímo s primitivem `\toks` následovaným číslem (místo deklarace vlastní proměnné), používejme jen čísla 0 až 9. Těchto prvních deset registrů není makrem `\newtoks` alokováno, takže nemůže dojít ke kolizi. □

Můžete namítnout, že registry typu ⟨*tokens*⟩ jsou zbytečný přežitek, protože stejného výsledku dosáhneme pomocí definice maker, například:

```
\def\promenna {abc\cosi {&$}: } % Uložení do proměnné
\promenna                       % expanduje na obsah proměnné
```

S touto námitkou lze jen souhlasit, ovšem existuje tu jisté ALE, které nás občas přinutí registr typu ⟨*tokens*⟩ použít. Tímto ALE je jedna výjimka v chování primitivu `\edef`, kterou nyní uvedeme: Ve fázi „učení“ se tělo definice při `\edef` ukládá do paměti až po úplné expanzi. Výjimkou je použití `\the`⟨*registr typu tokens*⟩, které

expanduje pouze na posloupnost tokenů odpovídající obsahu registru, ale tyto tokeny už v těle definice ve fázi „učení“ neexpandují a ukládají se do paměti tak, jak jsou.

Tuto vlastnost ilustrujeme na příkladě. Připravíme makro `\pridejtoken`, které přečte jeden parametr a ten přidá do proměnné `\buffer`. Nejprve uvedeme, jak bychom to udělali, kdybychom se chtěli obejít bez typu ⟨*tokens*⟩:

```
240 \def\buffer{}
241 \def\pridejtoken #1{\edef\buffer{\buffer #1}}
242 % test:
243 \pridejtoken a    \pridejtoken b
244 \pridejtoken \A   \pridejtoken \B
```

Na řádku 243 nám to ještě bude fungovat, ale na řádku 244 nám to havaruje. Povel `\edef` se snaží expandovat parametr `#1`, který je v tuto chvíli `\A`. Ztroskotáme na `Undefined control sequence`. Použití `\noexpand` na řádku 241:

```
245 \def\pridejtoken #1{\edef\buffer{\buffer \noexpand #1}}
```

katastrofu jen oddálí. Nyní už `\pridejtoken \A` proběhne bez problémů, ale při činnosti `\pridejtoken \B` se `\edef` snaží zcela expandovat `\buffer`, který ovšem obsahuje nedefinovanou řídicí sekvenci `\A`. Řešením může být buď použití poměrně dost velkého množství `\expandafter`:

```
246 \def\pridejtoken #1{\expandafter\def
247    \expandafter\buffer\expandafter{\buffer #1}}
```

nebo použijeme typ ⟨*tokens*⟩:

```
248 \newtoks\buffer % na začátku je \buffer={}
249 \def\pridejtoken #1{\edef\act
250   {\buffer={\the\buffer \noexpand #1}}\act}
```

Pomocné makro `\act` se zde definuje jako:

```
251 \buffer={⟨neexpandovaný obsah registru \buffer⟩⟨nový token⟩}
```

a toto přiřazení se provede. Ukážeme ještě jedno řešení našeho úkolu s přidáváním tokenů:

```
252 \def\pridejtoken #1{%
253   \expandafter\buffer\expandafter{\the\buffer #1}}
```

Protože rovnítko v zápise `\buffer={`⟨*přiřazovaný obsah*⟩`}` je nepovinné, ušetřili jsme v této ukázce jeden zápis primitivu `\expandafter`. □

Nejčastější použití registrů typu ⟨*tokens*⟩ je v součinnosti s povelem `\write`. TEX totiž provádí expanzi vstupního textu v době, kdy teprve probíhá sestavování tiskového materiálu odstavce. Tehdy ještě vůbec není známo, na jaké straně se nakonec zpracovávaný text objeví, protože zpracovávaný odstavec se nakonec může rozdělit do několika stran. Proto je povel `\write`, který se používá pro zápis do pomocných souborů, vybaven schopností pozdržet svůj argument až do doby, kdy jsou sestaveny jednotlivé strany. Teprve potom se provede expanze argumentu `\write` a zápis do pomocného souboru. To už je známo, na jaké straně je sledovaný objekt umístěn. Protože se zápis do pomocných souborů primitivem `\write` používá velmi často v souvislosti se zajištěním křížových referencí nebo sestavení obsahu, je popsaná vlastnost tohoto primitivu víceméně nutností. Podrobněji o tom budeme mluvit v sekci 7.1, takže zde se jen stručně zmíníme o jednom problému, kde je použití typu ⟨*tokens*⟩ velmi užitečné.

Představme si, že do pomocného souboru připravujeme podklady pro sestavení obsahu. Jednotlivé sekce jsou automaticky číslovány a v pomocném souboru `toc.tex` chceme mít ke každému řádku obsahu tyto tři údaje: číslo sekce, název sekce a číslo strany, kde sekce začíná. Například jeden řádek pracovního souboru by mohl vypadat takto:

```
254 \tocline{2.13}{O~problematice chroustů}{73}
```

tj. číslo sekce 2.13, název „O problematice chroustů“ a strana 73. Uživatel napíše

```
255 \sec O~problematice chroustů
256
257 Zde pokračuje text sekce ...
```

v místě, kde je začátek sekce. TEX by měl vysázet název sekce vhodným písmem a do souboru `toc.tex` uložit údaje pro následné sestavení obsahu. K tomu účelu v makru deklarujeme numerické registry `\chapnum` a `\secnum` pro práci s čísly stávající kapitoly a sekce. Pro zápis do souboru použijeme primitiv `\write`. Při každém použití makra `\sec` zvedneme číslo sekce o jedničku.

Pokud v makru `\sec` napíšeme:

```
258 \write\toc
259   {\string\tocline{\the\chapnum.\the\secnum}{#1}{\the\pageno}}
```

máme zajištěnu správnost zápisu čísla strany, protože expanze argumentu `\write` (jmenovitě `\the\pageno`) probíhá se zpožděním až v době, kdy je celá strana sestavena a číslo strany známé. Ale chybička se vloudila: kdyby byly v textu natolik

krátké sekce, že se na jedné straně sejdou nadpisy dvou sekcí, pak zpoždění expanze \the\secnum povede k chybě. Registr \secnum v době sestavení strany už nemusí obsahovat správnou hodnotu. Zde tedy zpoždění není na místě. Chtěli bychom, aby jistá část argumentu \write se expandovala okamžitě (např. \the\secnum), zatímco jiná se zpožděním (např. \the\pageno). K tomu nám právě pomůže registr typu ⟨*tokens*⟩ v součinnosti s \edef:

```
260 \newtoks\pagetoks \pagetoks={\the\pageno}
261 \newcount\chapnum \newcount\secnum
262 \newwrite\toc
263 \immediate\openout\toc=toc.tex
264
265 \def\sec #1 \par{\advance\secnum by1
266    \bigskip\noindent{\bf \the\chapnum.\the\secnum. #1}\par
267    \nobreak\medskip
268    \edef\act{\write\toc{\noexpand\string\noexpand\tocline
269       {\the\chapnum.\the\secnum}{#1}{\the\pagetoks}}}\act}
```

Na řádku 265 se zvětšuje číslo sekce o jedničku (podobně bychom zvětšovali číslo kapitoly o jedničku v analogickém makru \chap). Dále uvádíme jen jednoduchou sazbu nadpisu (řádek 266), protože tato problematika teď není předmětem našeho zájmu. Pracovní makro \act má díky \edef (například) hodnotu:

```
270 \write\toc
271   {\string\tocline{2.13}{O problematice chroustů}{\the\pageno}}
```

a toto makro se provede. Už v okamžiku činnosti makra \sec je tedy do argumentu \write uloženo skutečné číslo sekce, zatímco číslo strany bude expandováno až po sestavení strany. O to nám šlo.

Pohledem do souboru toc.tex zažijeme asi nemilé překvapení. Protože za písmenem „O“ v textu „O~problematice chroustů“ je obvykle napsána vlnka, tato vlnka se nám na cestě do souboru toc.tex expandovala a máme tam tedy napsáno:

```
272 \tocline{2.13}{O\penalty \@M \ problematice chroustů}{73}
```

Tento problém souvisí s takzvanými křehkými (fragile) nebo bytelnými (robust) příkazy, které jsou rozlišovány v LaTeXových příručkách. Pokud bychom při zpracování obsahu (načítání souboru toc.tex) nenastavili kategorii znaku „@“ na 11 (písmeno), došlo by u zpracování sekvence \@M k havárii. Raději se proto postaráme o to, aby se text do souboru ukládal v neexpandovaném tvaru, ale tím se budeme zabývat až v sekci 7.1. □

• **Hrátky s typem ⟨*tokens*⟩.** Zkusíme si implementovat datovou strukturu podobnou poli indexovaných údajů a vytvoříme makra, která hledají v tomto poli údaj

podle indexu nebo hledají index k danému údaji nebo hledají, zda je testovaný údaj v poli přítomen.

Nejprve vytvoříme makro, které nám pole údajů naplní. Nechť uživatel deklaruje libovolnou proměnnou typu ⟨*tokens*⟩ a naplní ji jako pole údajů třeba takto:

```
273 \newtoks\seznam
274 \declare\seznam={ABC{123}\A\B $:.\uf}
```

Tím je míněno, že údaj A má index 1, údaj B má index 2, údaj C index 3, dále „hromadný“ údaj 123 má index 4, atd. Konečně údaj `\uf` má index 10. Aby se nám dobře vyhledávaly údaje podle indexů, vložíme údaje do proměnné `\seznam` tak, že budou odděleny řídicí sekvencí `\:`, kterou později budeme různě definovat pro různé účely. Naším cílem tedy je, aby po použití makra `\declare` z řádku 274 měla proměnná `\seznam` tento obsah:

```
275 \:{A}\:{B}\:{C}\:{123}\:{\A}\:{\B}\:{$}\:{:}\:{.}\:{\uf}
```

Stanovený úkol může splnit následující makro:

```
276 \newtoks\buffer  \newtoks\temptok
277 \newcount\tempnum
278
279 \def\declare#1=#2{\buffer={#2}\onetok #1}
280 \def\onetok#1{\edef\temp{\the\buffer}%
281   \ifx\temp\empty \def\next##1{} \else
282   \edef\act{\noexpand\separe\the\buffer\noexpand\end}\act
283   \edef\act{#1={\the#1\noexpand\:{\the\temptok}}}\act
284   \let\next=\onetok
285   \fi \next#1}
286 \def\separe#1#2\end{\temptok={#1}\buffer={#2}}
```

Pomocné makro `\onetok` přerovná z přechodného místa `\buffer` vždy jeden údaj do cílové proměnné `#1` (v našem příkladě to je proměnná `\seznam`). Na řádku 281 se testuje, zda pomocná proměnná `\buffer` už není náhodou prázdná. Pokud ano, změníme význam `\next`, aby byl cyklus „přerovnávání“ ukončen. Jinak pomocí dvou pomocných `\act` provedeme vlastní přerovnání. Tato pomocná makra mají (například) tento obsah:

```
287 \separe ⟨obsah bufferu⟩\end                    % první \act
288 \seznam={⟨obsah \seznam⟩\:{⟨přenesený údaj⟩}} % druhé \act
```

Pomocné makro `\separe` skutečně oddělí jeden údaj ze stávajícího obsahu `\buffer` a přenese jej do `\temptok`. Proměnné `\buffer` je tento údaj odebrán.

Nyní vytvoříme makro \index, které nám vrátí do makra \out údaj, odpovídající požadovanému indexu. Například po:

```
\index\seznam{7} \message{na pozici 7 je údaj \out}
```

máme makro \out definováno jako $. Makro \index vypadá takto:

```
\def\index#1#2{\tempnum=0 \def\out{}
  \def\:##1{\advance\tempnum by1
    \ifnum\tempnum=#2 \def\:####1{}\def\out{##1}\fi}
  \the#1}
```

Zde definujeme sekvenci \: vhodným způsobem a pak na řádku 293 expandujeme obsah použité proměnné. Makra \:, která jsou v obsahu proměnné vložena, provedou svoji práci. Až po požadovaný index počítají počet přeskočených údajů, pak se definuje výsledek do \out a ostatní makra \: jsou předefinována tak, že nedělají nic. Je třeba si uvědomit, že ve výstupním \out může být makro, takže použití v \message (jako na řádku 289) nemusí vždy fungovat. Jak ovšem s výstupem \out naložíme, záleží na konkrétní aplikaci. Vždy je možné pomocí \expandafter, \noexpand a podobných triků dosáhnout kýženého, ovšem tím se teď nebudeme zabývat.

Nyní řešme obrácenou úlohu: hledáme index daného údaje. Pokud třeba napíšeme:

```
\position\seznam{C} \message{C je na pozici: \the\tempnum}
```

máme v registru \tempnum index údaje „C“. Ukažme makro \position:

```
\def\position#1#2{\tempnum=0 \def\test{#2}
  \def\:##1{\advance\tempnum by1
    \def\temp{##1}%
    \ifx\temp\finaltok \tempnum=-1 \fi
    \ifx\temp\test \def\:####1^^X{}\fi}
  \the#1\:^^X}
\def\finaltok{^^X}
```

Zde postupně probíráme argumenty maker \: a srovnáváme je s předlohou (\test). Pokud náhodou narazíme na konec (údaj ^^X), údaj nebyl nalezen a vrátíme tedy \tempnum jako -1. Pokud je údaj nalezen, pak je následující \: definována s parametrem se separátorem tak, že je přeskočen celý zbytek ještě neprobraných údajů.

Poslední cvičeníčko je tvorba makra, které vrátí logickou hodnotu podle toho, zda údaj v seznamu je či není. Například:

```
\ifin\seznam{:}\message{: je ve stringu}
\else          \message{: neni ve stringu}  \fi
```

Ukážeme, jak je makro `\ifin` implementováno:

```
\newif\ifnext
\def\ifin#1#2{\nextfalse\def\test{#2}%
  \def\:##1{\def\temp{##1}%
    \ifx\temp\test \nexttrue\def\:####1{}\fi}%
  \the#1\ifnext}
```

□

• **Příklad.** V závěrečné části této sekce ukážeme poněkud komplikovanější příklad, ve kterém pracujeme s registry typu ⟨*tokens*⟩. Čtenář si může všimnout, že v části B této knihy jsou u každého hesla použity zajímavé odkazy na stránky a čísla řádků. Jak toho bylo dosaženo? Nejprve je výskyt každého hesla zaznamenán do pracovního souboru `tbn.ref`. Vedle hesla je uvedena strana a číslo řádku, na které je heslo použito v ukázce. Není-li heslo v ukázce, je číslo řádku nula. Pak (ve druhém průchodu zpracování TEXem) je soubor `tbn.ref` načten a ke každému heslu jsou všechny údaje o něm kumulovány do makra `r:heslo`. V takovém makru můžeme mít třeba uloženo:

```
1.1,1.2,1.0,1.3,1.5,2.0,3.0,3.0,4.0,5.0,5.1,5.1,
```

což znamená, že námi zkoumané heslo se vyskytuje na straně 1 na řádku 1, dále na straně 1 na řádku 2, dále na straně 1 v textu atd. Formát je ve tvaru strana.řádek. Vidíme, že se údaje opakují, protože na jedné straně a jednom řádku ukázky může být heslo zapsáno vícekrát. Našim úkolem nyní bude tyto údaje zpracovat tak, aby se neopakovaly a dokonce, pokud je heslo například na stranách 1, 2, 3, mělo by se tisknout 1–3.

Úkol budeme řešit ve dvou průchodech. V prvním průchodu pouze vyloučíme opakující se údaje. Takže po zápise

```
\setref 1.1,1.2,1.0,1.3,1.5,2.0,3.0,3.0,4.0,5.0,5.1,5.1,0.0,
```

makro `\setref` vloží do proměnné `\buffer` typu ⟨*tokens*⟩ tento obsah:

```
\,1(1,2,3,5,)\,2()\,3()\,4()\,5(1,)
```

Vidíme, že údaje se už neopakují a jsou formátovány trochu přehledněji: Nejprve číslo strany a k ní všechna čísla řádků v závorce. Jednotlivé strany jsou v `\buffer` odděleny makrem `\,`. V následujících ukázkách na řádcích 310–372 rozebereme činnost makra `\setref`, použitého v této knížce. Někde přerušíme text makra i uprostřed definice, protože bude potřeba vložit vysvětlující výklad.

Nejprve uvedeme deklarace a pomocná makra:

```
\newcount\tempnum   % pomocná proměnná
\newif\ifdashbase   % stav, kdy je zaneseno "-" pro strany
\newif\ifdashind    % stav, kdy je zaneseno "-" pro řádky
\newif\iffirst      % údaj je zpracován poprvé
\newif\iffirstbase  % údaj o straně je zpracován poprvé
\newcount\basenum   % aktuálně zpracovávaná strana
\newcount\indnum    % aktuálně zpracovávaný řádek
\newtoks\buffer     % seznam tokenů, kde se všechno mele
\def\nullbuf{\buffer={}}
\def\addbuf #1{\edef\act{\buffer={\the\buffer#1}}\act}
```

Například `\addbuf{abc}` přidá do proměnné `\buffer` tokeny `abc`.

```
\def\setref{\begingroup
    \let\,=\relax \nullbuf \basenum=0 \firsttrue
    \let\next=\cycleref \next}
```

Makro `\cycleref` postupně čte údaje typu strana.řádek a zpracovává je. Koncová značka `0.0` ukončuje cyklus. Makro vloží poslední znak „)“ a přejde do `\finalref`.

```
\def\cycleref #1.#2,{\ifnum #1=0 \addbuf{)}\let\next=\finalref
  \else \ifnum #1=\basenum % zůstali jsme u stejné strany?
              \ifnum #2=0 \else
                 \ifnum\indnum=#2 \else\addbuf{#2,}\indnum=#2 \fi
              \fi
```

Pokud jsme zůstali u stejné strany (`\ifnum #1=\basenum`), přidáváme pouze čísla (nenulových) řádků. Následuje `\else` pro `\ifnum #1`, tj. je zpracována nová strana. Není-li to zcela první strana (`\iffirst`), vložíme za čísla řádků předchozí strany závorku.

```
        \else  \iffirst \firstfalse \else \addbuf{)}\fi
              \addbuf{\,#1(}\basenum=#1
              \ifnum #2=0 \else \addbuf{#2,}\indnum=#2 \fi
```

U nové strany dále přidáme „`\,`⟨*číslo strany*⟩`(`“, a pokud hned následuje číslo řádku, přidáme i číslo řádku. Právě vložené číslo řádku je zaznamenáno v `\indnum`, takže příště budeme vědět, zda se číslo řádku opakuje. V takovém případě bychom jej znovu nezapisovali.

```
        \fi
  \fi \next}
```

Tím máme ukončeno makro `\cycleref` a v `\buffer` máme informace už ve formátu, kdy se neopakují. Tím je splněn první průchod. V proměnné `\buffer` tedy máme například „`\,1(1,2,3,5,)\,2()\,3()\,4()\,5(1,)`“. V druhém průchodu bychom měli tento text konvertovat na seznam tokenů, který by se mohl snadno uplatnit v matematickém seznamu (indexy budou čísla řádků). Navíc bude nahrazena souvislá posloupnost stran i řádků typu 1,2,3 za text typu `1-3`. Data z našeho příkladu budeme konvertovat na „`1_{1-3,5},2-4,5_{1}`“.

```
333 \def\finalref {%
334    \let\,=\cref \firstbasetrue \dashbasefalse \basenum=-1
335    \expandafter\nullbuf \the\buffer
336    \ifdashbase \addbuf{\the\basenum}\fi
337    \mathcode`\-="007B \thinmuskip=5mu plus 3mu minus 2mu
338    $\displaystyle\the\buffer$\endgroup}
```

Na řádku 335 nejprve „položíme“ do čtecí fronty obsah proměnné `\buffer`, pak tuto proměnnou vynulujeme a nakonec projdeme její bývalý obsah ve čtecí frontě. V této chvíli pracuje makro `\,` jako `\cref` a provede nové naplnění proměnné `\buffer` požadovaným způsobem. Jak to dělá, to si necháme jako bonbónek nakonec. V závěrečné části makra definujeme znak minus jako pomlčku (`\mathcode`), prodloužíme mezeru mezi čárkami (`\thinmuskip`) a „vypustíme“ obsah proměnné `\buffer` do matematického seznamu. Tím dosáhneme sazby čísel stran s indexy, které značí čísla řádků.

V makru `\cref` se nejprve zaměříme na zpracování čísel stran. Všímáme si, zda následující strana je jen o jedničku vyšší než předchozí. Pokud ano, vložíme do `\buffer` znak „`-`“ a nastavíme `\dashbasetrue`, abychom příště při změně strany jen o jedničku neudělali nic.

```
339 \def\cref #1(#2){%
340    \if :#2:\tempnum=#1 \advance\tempnum by-1
341          \ifnum \tempnum=\basenum % postoupili jsme o jedničku
342                 \ifdashbase % nedělej nic, už je zapsané "-"
343                 \else \addbuf{-}\dashbasetrue
344                 \fi
345          \else  \addnewbase{#1}% Je třeba zapsat novou stranu
346          \fi
347          \basenum=#1
348    \else \addnewbase{#1}% nová strana bude mít index
349          \addbuf{_\bgroup}\convertind #20,\addbuf{\egroup}%
350          \basenum=#1 \advance\basenum by-1 % aby nenásledovalo -
351    \fi }
```

Makro `\cref` se dělí na dvě hlavní větve podle toho, zda je parametr `#2` prázdný nebo ne. Prázdný parametr znamená, že heslo je přítomno na stránce, ale nikoli

na konkrétním řádku v ukázce. V takovém případě nemáme problémy s indexy (čísly řádků) a staráme se jen o návaznost čísel stránek. Makro `\addnewbase` vloží do `\buffer` nové číslo strany a nastaví příslušné stavové proměnné na správnou hodnotu. Ukážeme ho za chvíli.

Je-li parametr `#2` neprázdný, vložíme za číslo strany konstruktor pro index následovaný otevírací závorkou skupiny (viz řádek 349). Tato závorka je zapsána nikoli přímo jako „{“, ale pomocí zástupné sekvence `\bgroup`. To je proto, že samotnou závorku do seznamu v `\buffer` není možno vpravit. Seznam tokenů totiž musí být balancovaný text. Dále (viz stále řádek 349) makro `\convertind` vloží do `\buffer` jednotlivá čísla řádků společné na jedné straně. Seznam čísel řádků je ukončen číslem 0, aby makro vědělo, kdy má skončit. Konečně je do `\buffer` vložena zavírací závorka jako `\egroup`.

```
352 \def\addnewbase #1{%
353    \ifdashbase \addbuf{\the\basenum}\dashbasefalse \fi
354    \iffirstbase \firstbasefalse \else \addbuf{,}\fi
355    \addbuf{#1}}
```

Makro `\convertind` má vložit do `\buffer` seznam čísel řádků, které zrovna leží ve čtecí frontě oddělené čárkou a ukončené nulou. Tento seznam rovněž podléhá konverzi typu $1, 2, 3 \longrightarrow 1–3$. Pro zpracování jednotlivého údaje se roztočí cyklus `\cycind`, jehož jeden průchod se velmi podobá makru `\cref`.

```
356 \def\convertind {\let\next=\cycind \dashindfalse
357    \firsttrue \indnum=-1 \next}
358 \def\cycind #1,{%
359    \ifnum #1=0 \let\next=\relax
360                \ifdashind \addbuf{\the\indnum}\dashindfalse \fi
361    \else \tempnum=#1 \advance\tempnum by-1
362          \ifnum \tempnum=\indnum % postoupili jsme o jedničku
363                \ifdashind %nedělej nic, už máme otevřené "-"
364                \else  \addbuf{-}\dashindtrue
365                \fi
366          \else % Je třeba zapsat nové číslo
367                \ifdashind \addbuf{\the\indnum}\dashindfalse \fi
368                \iffirst \firstfalse \else \addbuf{,}\fi
369                \addbuf{#1}%
370         \fi
371         \indnum=#1
372    \fi \next}
```

□

3. Základy hlavního procesoru

3.1. Povely a parametry hlavního procesoru

Hlavní procesor řídí celou činnost TEXu. Po startu si vyžádá od expand procesoru první token a ten interpretuje jako *povel hlavního procesoru* (angl. command). Povel může mít parametry. V takovém případě si hlavní procesor vyžádá od expand procesoru další tokeny, kterými se vyplní parametry povelu. Pak se povel provede. Následující token (za parametry) je pak interpretován jako další povel hlavního procesoru. Tato činnost se opakuje tak dlouho, dokud hlavní procesor nedostane k vykonání povel `\end`.

Povelem hlavního procesoru se realizují různé aktivity, které nakonec vedou k sestavení tiskového materiálu a jeho výstupu do `dvi`. Stejně formulované povely mohou způsobit rozdílné aktivity podle toho, v jakém *módu* se zrovna hlavní procesor nalézá. O módech si povíme v sekci 3.4. Nejběžnějším povelem (v horizontálním módu) je zřejmě povel „vysázej znak zrovna nastaveným fontem; znak odpovídá ASCII hodnotě tokenu“. Například, pokud hlavní procesor obdrží token $\boxed{\texttt{p}}_{11}$ v kontextu povelu, vysází písmeno „p“.

Jiným příkladem je token mezera ($\boxed{\sqcup}_{10}$), který v případě, že je hlavním procesorem realizován jako povel, způsobí v horizontálním módu vložení pružného výplňku typu ⟨*glue*⟩ (viz sekci 3.6, mezery v horizontálním módu), zatímco ve vertikálním nebo matematickém módu se neprovede nic (tj. mezera je ignorována).

Často je povel formulován pomocí primitivu nebo jiné řídicí sekvence. Například primitiv `\par`, je-li realizován hlavním procesorem jako povel, uzavře načítání textu odstavce a odstavec zpracuje.

Ne každá řídicí sekvence se stává povelem. V příkladě se zaměříme na řídicí sekvenci `\par`:

```
1 \def\nadpis#1\par{\medskip\noindent{\bf #1}\par\nobreak\medskip}
2 % A zde je použití makra \nadpis:
3 \nadpis Úvod
4
5 V článku se zaměříme na problematiku ...
```

V této ukázce se řídicí sekvence `\par` vyskytla třikrát. Poprvé v masce parametrů definice makra `\nadpis`, podruhé v těle definice před sekvencí `\nobreak` a třetí `\par` vyprodukuje token procesor, když zpracovává prázdný řádek 4. Jenom jedna sekvence `\par` se při zpracování stala povelem hlavního procesoru. Při použití makra

`\nadpis` se totiž sekvence `\par` z prázdného řádku 4 stává separátorem parametru, takže do `#1` se vloží text „Úvod␣“ (vyznačili jsme mezeru, kterou token procesor vloží na konci řádku). Tím tato sekvence `\par` svoji roli splnila. Po expanzi makra `\nadpis` se sekvence `\par` uvedená v jeho těle stává skutečně povelem hlavního procesoru. □

Povelem může být i jiná řídicí sekvence než primitiv. Nechť máme řídicí sekvenci `\mujfont` deklarovánu primitivem `\font` takto:

```
6  \font\mujfont=csr10 scaled\magstep3
```

Je-li pak použita řídicí sekvence `\mujfont` ve významu povelu, způsobí přepnutí běžného fontu na `csr10` ve zvětšení `\magstep3`. □

• **Interpretace řídicích sekvencí.** Všechny řídicí sekvence, se kterými TEX umí v danou chvíli pracovat, jsou uloženy ve vnitřním slovníku identifikátorů. V něm je každé řídicí sekvenci přiřazen její *význam*. TEX dokáže tyto významy měnit a přidávat do slovníku nové řídicí sekvence, jejichž význam se zrovna „naučil“. Významy řídicích sekvencí rozdělíme do pěti skupin:

1. Registr. Například `\baselineskip`, `\pageno`. Viz sekci 3.3.
2. Přepínač fontu. Třeba `\tenrm`. Nebo `\mujfont` z předchozího příkladu.
3. Jiný povel hlavního procesoru. Například `\par`, `\end`, `\def`.
4. Primitiv pracující na úrovni expand procesoru. Třeba `\the`, `\expandafter`.
5. Makro definované pomocí `\def`, `\edef`, `\gdef` nebo `\xdef`.

TEX interpretuje právě načtenou řídicí sekvenci nejprve v expand procesoru a teprve pokud sekvence projde beze změn do hlavního procesoru, je interpretována tam. Expand procesor ponechává beze změny sekvence s významem skupiny 1, 2 nebo 3. Má-li sekvence význam skupiny 4, spustí se odpovídající vestavěný algoritmus expand procesoru. Je-li sekvence makrem (skupina 5), nahradí ji TEX tělem definice makra podle pravidel vysvětlených v sekci 2.1. Nemá-li řídicí sekvence žádný význam, objeví se chyba `Undefined control sequence`. Výjimky z tohoto pravidla jsou uvedeny v sekci 3.2.

Hlavní procesor zpracovává už jen řídicí sekvence s významem typu 1–3. Tyto řídicí sekvence interpretuje podle kontextu jako povel nebo jako parametr. Zde záleží na momentálním stavu hlavního procesoru (zda čte povel nebo parametr). Například řídicí sekvence `\mujfont` z příkladu z řádku 6 je obvykle interpretována jako povel (přepíná font), ale v kontextu `\hyphenchar\mujfont` je čtena jako parametr.

Význam jakékoli řídicí sekvence je možné nechat vypsat primitivem `\meaning` nebo `\show` (viz část B).

Na počátku je iniTEX vybaven výchozím slovníkem identifikátorů, který obsahuje všechny primitivní řídicí sekvence. Slovník se může při činnosti TEXu dále rozšiřovat. Nové řídicí sekvence lze zanést do slovníku pomocí *deklaračních primitivů*. Jedná se o následující primitivy: `\let`, `\futurelet`, `\def`, `\gdef`, `\edef`, `\xdef`, `\chardef`, `\mathchardef`, `\countdef`, `\dimendef`, `\skipdef`, `\muskipdef`, `\toksdef`, `\read` a `\font`. Tyto primitivy jednak zanášejí do slovníku identifikátorů novou položku a jednak přidělují identifikátoru konkrétní význam.

Deklarační primitivy ovšem nemusí nutně vytvářet ve slovníku identifikátorů novou položku. Pomocí těchto primitivů je též možné přidělit význam i takové řídicí sekvenci, která už ve slovníku existuje. Tím sekvence ztrácí svůj původní význam. Můžeme si třeba zcela znemožnit přístup k některým primitivním registrům. Například:

```
7 \gdef\parindent{aha}
```

zcela uzavře přístup k registru, který byl původně označován primitivní sekvencí `\parindent` a jehož hodnota rozhoduje o velikosti odstavcové zarážky. Od této chvíle s registrem nemůžeme pracovat. Registr nepřestává existovat a s jeho hodnotou pracují nadále vestavěné algoritmy TEXu. Například algoritmus pro zahájení odstavce bude nadále vkládat na začátek seznamu v odstavcovém módu prázdný box, jehož šířka je rovna hodnotě tohoto registru.

V této knize většinou (pokud nehrozí nebezpečí nedorozumění) nerozlišujeme mezi řídicí sekvencí a jejím významem. Ze stejných důvodů spojujeme pojem povel hlavního procesoru s primitivní řídicí sekvencí, která při nezměněném významu tento povel v hlavním procesoru spouští. Například mluvíme o povelu `\par`, místo abychom přesně řekli něco takového: „povel, který odpovídal významu řídicí sekvence `\par` v době, kdy tato sekvence ještě nebyla (případně) předefinovaná“. □

• **Parametry povelů hlavního procesoru**. Povely mohou obsahovat parametry. Například povel `\font` na řádku 6 má tyto parametry:

```
8 \mujfont=csr10 scaled1728
```

Vidíme, že některé parametry přicházejí do povelu z expand procesoru již v expandovaném tvaru. Třeba sekvence `\magstep` v našem příkladě se expandovala na tokeny `1728`. Také ovšem existují parametry, které se berou v neexpandovaném tvaru. Přesněji, hlavní procesor v jistých situacích požádá expand procesor, aby mu vydal další token bez expanze. V našem příkladě povel `\font` bere první parametr v neexpandovaném tvaru (`\mujfont`) a ostatní parametry jsou expandované. Obecné pravidlo zní, že všechny parametry jsou expandované, výjimky jsou jmenovitě popsány v sekci 3.2.

Náš příklad s povelem `\font` způsobí v hlavním procesoru následující akci: Do slovníku identifikátorů zařadí nový identifikátor `\mujfont` s významem přepínače fontu `csr10` s koeficientem zvětšení 1,728. Existoval-li identifikátor `\mujfont` ve slovníku už dříve, je pouze změněn význam identifikátoru.

Parametry povelů hlavního procesoru musí přesně odpovídat *syntaktickým pravidlům*, která v této knize zapisujeme do ⟨*těchto závorek*⟩. V části B máme nejprve abecední slovníček všech syntaktických pravidel, použitých v této knize. Pak teprve následuje slovník primitivů a maker. U primitivu vždy uvádíme jeho parametry pomocí zápisu ⟨*syntaktického pravidla*⟩. Třeba u našeho příkladu s primitivem `\font` jsou parametry popsány takto:

9 `\font` ⟨*control sequence*⟩⟨*equals*⟩⟨*file name*⟩⟨*at clause*⟩

V úvodu části B se pak dočteme, co může být ⟨*control sequence*⟩, co znamená ⟨*equals*⟩, ⟨*file name*⟩ apod. Například ⟨*equals*⟩ znamená zápis nepovinného znaku „=“, kterému může předcházet libovolné množství mezer.

Při vyhodnocování parametru povelu se TEX vždy snaží zahrnout do parametru maximum tokenů, které jsou v souladu se syntaktickým pravidlem parametru. Například při vyhodnocení parametru podle syntaktického pravidla ⟨*number*⟩ (tj. číslo) čte hlavní procesor jednotlivé číslice tak dlouho, až narazí na token odlišný od číslice. Teprve pak uzavře čtení parametru podle tohoto syntaktického pravidla. Tento příklad není řečen zcela přesně, protože syntaktické pravidlo ⟨*number*⟩ má poněkud bohatší vyjadřovací možnosti než jen číslo zapsané jako řada číslic. Podrobněji o pravidle ⟨*number*⟩ viz část B.

Pokud ani první token parametru vůbec neodpovídá požadovanému syntaktickému pravidlu (například místo číslice je uvedeno písmeno), TEX ohlásí chybu. Vyzkoušejte si napsat třeba `\font a=csr10`. Obdržíte chybu:

```
! Missing control sequence inserted.
<inserted text>
                \inaccessible
<to be read again>
                   a
l.2 \font a
           =csr10
?
```

Pokusíme se toto hlášení podrobně rozebrat. TEX očekává jako první parametr za povelem `\font` token podle syntaktického pravidla ⟨*control sequence*⟩. K tomu nedošlo, místo toho obdržel token $\boxed{\texttt{a}}_{11}$. Proto ohlásil chybu, vložil vnitřní zabudovanou (a nikdy nedosažitelnou) sekvenci `\inaccessible` a token $\boxed{\texttt{a}}_{11}$ bude číst

znova (viz `to be read again`). Pokud uživatel chybu přeskočí (na terminálu obvykle pomocí klávesy Enter), TEX bude vykonávat povel:

```
10 \font\inaccessible a=csr10
```

Protože v syntaktickém pravidlu ⟨*equals*⟩ je znak „=“ nepovinný, bude v tomto případě hlavní procesor brát parametr ⟨*equals*⟩ jako prázdný. Dále bude interpretovat ⟨*file name*⟩ jako text „`a=csr10`“, tj. začne hledat metriku s tímto názvem souboru. Protože takovou metriku (asi) nenajde, dočkáme se další chyby: `Font \inaccessible=a=csr10 not loadable: Metric (TFM) file not found.` □

V následujícím textu si ukážeme, že je někdy potřeba věnovat velkou péči možnému separátoru parametru, který bývá součástí syntaktického pravidla parametru. Situaci si ilustrujeme na povelu `\char`, který má jeden parametr podle pravidla ⟨*number*⟩. Například `\char92` vysází (v horizontálním módu) znak z běžného fontu z pozice 92, což při použití fontu `\tt` dává znak „backslash“ (\). Předpokládejme, že jsme definovali makro:

```
11 \def\\{\char92}
12 % Ukázka použití makra:
13 Makro {\tt \\bla} se chová tak a tak
```

Na výstupu skutečně dostaneme očekávaný text: „Makro `\bla` se chová tak a tak“. Ovšem naše makro není dobré! Pokud by chtěl uživatel napsat třeba `\1`, a přitom použil `\\1`, zažije drobné překvapení. Sekvence `\\1` se expanduje na `\char921` a povel `\char` ohlásí chybu, že ⟨*number*⟩ není v rozsahu 0 až 255. Aby se zabránilo takovým nepříjemnostem, je u syntaktického pravidla ⟨*number*⟩ řečeno, že za číslem může následovat jedna nepovinná mezera, přičemž tato mezera je brána jako součást pravidla ⟨*number*⟩ a nikoli jako následující povel. Naše makro je tedy dvakrát špatné: když uživatel napíše `\tt \\␣je␣backslash`, na výstupu mu to „sežere“ mezeru, napsanou hned za příkazem \\ a dostane `\je backslash`. Je to z toho důvodu, že uživatelova mezera je součástí syntaktického pravidla ⟨*number*⟩. Okamžitě tedy naše makro opravíme takto:

```
14 \def\\{\char92 }
```

Nyní skutečně mezera v makru je součástí (a zároveň separátorem) syntaktického pravidla ⟨*number*⟩, zatímco případná uživatelova mezera přichází jako druhý token „mezera“ do hlavního procesoru a je interpretována jako povel. □

Uvedeme nyní princip mechanismu „to be read again“. Při čtení parametru povelu hlavní procesor postupně žádá od expand procesoru tokeny a výslednou konstrukci kontroluje se syntaktickým pravidlem parametru. Jakmile přestane následující token s tímto pravidlem souhlasit, je znovu vrácen do čtecí fronty a čtení parametru

končí. Při čtení dalšího parametru nebo povelu je odložený token „čten znova“ (angl. read again).

Ilustrujme si tento fenomén na našem příkladě z řádku 13. Zde je makro ještě bez mezery za číslem `92`. Při použití `\\bla` se makro expanduje na `\char92bla` a povel `\char` zahájí čtení parametru ⟨*number*⟩. V tomto případě se hlavní procesor pokusí po načtení `9` a `2` přečíst ještě token `b`. Ten se mu nehodí do krámu, kompletuje tedy parametr ⟨*number*⟩ jako 92 a vrátí token `b` k novému načtení. Po provedení povelu `\char` hlavní procesor *znovu načte* token `b`, který se provede jako povel k vysázení písmene b. Vše tedy funguje tak, jak má. Že to ale nefunguje pro situace `\\1` a `\\␣`, jsme si uvedli výše.

Poznamenejme nakonec, že makro `\\` s požadovanými vlastnostmi z našeho příkladu se uvedeným způsobem většinou nezavádí. Místo toho se použije `\def\\{\char‘\\ }`, nebo ještě lépe `\chardef\\=‘\\`. V obou případech se využívá skutečnosti, že syntaktické pravidlo ⟨*number*⟩ umožňuje napsat číslo jako ASCII hodnotu tokenu, který (bez expanze) následuje za tokenem `‘`$_{12}$. Zde se jedná o ASCII hodnotu tokenu `\`. Ve druhém případě se navíc nemusíme starat o mezery v makru, protože

```
15 \chardef ⟨control sequence⟩⟨equals⟩⟨number⟩
```

deklaruje ⟨*control sequence*⟩ jako ekvivalent k povelu `\char` ⟨*number*⟩. □

• **Klíčová slova.** Algoritmus „to be read again“ si ukážeme ještě jednou na případě čtení tzv. *klíčových slov* parametrů. V příkladě s povelem `\font` (řádek 9) je uvedeno syntaktické pravidlo ⟨*at clause*⟩. Když prostudujeme v části B jeho syntaxi, zjistíme, že ⟨*at clause*⟩ může být buď prázdná, nebo může obsahovat klíčové slovo `scaled`, resp. `at` následované parametrem podle pravidla ⟨*number*⟩, resp. ⟨*dimen*⟩. Rozeberme si, co se stane, pokud uděláme překlep, a místo správného `scaled` v povelu `\font` napíšeme třeba `scalrd`, tedy:

```
16 \font\mujfont=csr10 scalrd\magstep3
```

Jakmile hlavní procesor začne číst parametr ⟨*at clause*⟩, přečte postupně tokeny `s`, `c`, `a`, `l`. Až dosud jsou ve shodě s klíčovým slovem `scaled`. Nyní přečte token `r` a ten už není ve shodě. Proto hlavní procesor interpretuje ⟨*at clause*⟩ jako prázdný parametr a dále vrátí do fronty ke čtení dosud zbytečně načtené tokeny „`scalr`“. Znamená to, že těchto pět tokenů je „to be read again“. Po vykonání povelu `\font` se znovu načítají tokeny „`scalr`“, které nyní jednotlivě znamenají povely k vysázení příslušných znaků. Na výstupu nakonec dostáváme text: „scalrd1728“. □

Z předchozího příkladu vidíme, že klíčová slova v parametrech povelů jsou dosti podstatnými syntaktickými objekty TEXu. Všechna klíčová slova najde čtenář v tabulce na straně 317.

V klíčových slovech se nerozlišuje mezi velkými a malými písmeny. To je svým způsobem zvláštnost, protože kdekoli jinde v TEXu je rozlišení mezi velikostí písmen důležité. Klíčové slovo je vždy psáno bez mezer mezi písmeny. Není tedy možné psát `sca␣led`. Výjimku tvoří klíčové slovo `fill` a `filll`, kde se mezi písmeny „`l`“ mohou vyskytovat mezery. Proto je možný například i takový obskurní zápis klíčového slova `filll`: `Fil␣L␣␣␣l`. Praktické a smysluplné použití této vlastnosti neznám. □

Některá klíčová slova uvozují nepovinnou část parametru. Příklad už známe se slovem `scaled`. Mezi další příklady patří třeba slova `plus` a `minus`, která umožňují vložit hodnotu pružnosti výplňku. Například jsou možné zápisy:

```
\hskip 2cm  % Vloží se výplněk s pevnou šířkou 2cm
\hskip 2cm plus 3cm minus 1cm  % Výplněk má definovanou pružnost
```

Tato skutečnost může být při tvorbě maker nebezpečná. Vytvoříme třeba pro uživatele makro `\mezera`, které vloží do textu mezeru velikosti 2 cm. Kdybychom psali:

```
\def\mezera{\hskip 2cm}
```

a uživatel použil naše makro takto:

```
Udělám mezeru.\mezera Plus další mezeru:\mezera.
```

pak mu to nebude fungovat. Jeho slovo `Plus` totiž bude bráno jako klíčové slovo povelu `\hskip` a TEX v okamžiku načtení písmene `d` (za slovem `Plus`) oznámí chybu `Missing number, treated as zero`. To může dovést uživatele našeho makra k šílenství. Proto je bezpodmínečně nutné ukončit konstrukci povelu `\hskip` v našem makru prázdným povelem `\relax` takto:

```
\def\mezera{\hskip 2cm\relax}
```

Problém z předchozího příkladu může mít podstatně záludnější charakter. Pokud uděláme naše makro s pružným výplňkem a bez `\relax`:

```
\def\mezera{\hskip 2cm plus 1fil }
A nyní makro použijeme:\mezera Láska je láska.
```

pak se nám slije `fil` s následujícím `L` od slova „`Láska`“. TEX tedy akceptuje výplněk typu `filL` a na výstupu máme jen „áska je láska“. Vidíme, že nepomohla ani mezera v makru vložená mezi `fil` a `L`, protože hlavní procesor dovolí v tomto případě výjimečně zápis slova `fill` s mezerami uvnitř: `fil␣L`. Poučeni z těchto číhajících nebezpečí, budeme vždy v makrech za parametrem podle pravidla ⟨*glue*⟩ psát důsledně prázdný povel `\relax`. □

3.2. Kdy TEX neprovádí expanzi

Hlavní procesor si postupně žádá povely a parametry povelů od expand procesoru. Expand procesor vydává jednotlivé tokeny obecně v expandovaném tvaru. Existují ovšem výjimky, kdy hlavní procesor předá expand procesoru pokyn, aby u následujícího tokenu (následujících tokenů) neprováděl expanzi. V této sekci vyjmenujeme všechny takové výjimky.

1. Pokud se zpracovává parametr podle syntaktického pravidla ⟨*control sequence*⟩, jedná se o jeden token, pro který se neprovádí expanze. Tato situace se vyskytuje v deklaračních primitivech (viz stranu 66), které definují pro ⟨*control sequence*⟩ nový význam. Například:

```
24 \let⟨control sequence⟩⟨equals⟩⟨token⟩
```

2. Pokud je povelem čten ⟨*token*⟩, bývá někdy potřeba získat jej v neexpandovaném tvaru. Například při `\let` (viz řádek 24) se vyžádá ⟨*token*⟩ v neexpandovaném tvaru, takže je možné psát třeba:

```
25 \let\novacs=\staracs
```

Tato situace se vyskytuje pouze u primitivů `\let`, `\futurelet` a `\show`.

3. Protože lze také psát `\let\novacs\staracs` (tj. lze vynechat znak „=“), vidíme, že parametr ⟨*equals*⟩ je v případě `\let` rovněž čten v neexpandovaném tvaru. Ve všech ostatních situacích je parametr ⟨*equals*⟩ čten v expandovaném tvaru. Vyzkoušejte si:

```
26 \def\rovnitko{=}
27 \let\cs\rovnitko            % \cs dostala význam makra \rovnitko
28 \font\mujbx\rovnitko csbx10 % je totéž, co \font\mujbx=csbx10
29 \font\mujtt\cs       cstt10 % je totéž, co \font\mujtt=cstt10
```

4. U povelů `\def`, `\edef`, `\gdef` a `\xdef` je v neexpandovaném tvaru načítána maska parametrů. Při povelech `\def` a `\gdef` je navíc bez expanze načteno tělo definice.

5. Při načítání posloupnosti tokenů u `\read` a při uložení nové hodnoty do proměnných typu „toks“ (např. `\toks0`, `\everymath`, `\output`) není tato posloupnost tokenů expandována.

6. Při prvním průchodu přes posloupnost tokenů u `\uppercase` a `\lowercase` a při čtení parametru `\write` nedochází k expanzi. V těchto případech se ale k posloupnosti tokenů TEX vrací obvykle znovu. V případě `\uppercase` resp. `\lowercase` se

TEX po převedení písmen na velká resp. malá znovu vrací na první token posloupnosti. V tuto chvíli už expanze probíhá. Podobně `\write` si uloží svůj parametr neexpandovaný, ale při `\shipout` provede vlastní zápis, při kterém expanduje své parametry.

7. Při čtení deklarace řádku u `\halign` a `\valign`. Výjimku tvoří token následovaný za `\span` a tokeny odpovídající syntaktickému pravidlu ⟨*glue*⟩ za `\tabskip`. Podrobněji o `\halign` a `\valign` viz sekci 4.3.

8. Je-li v syntaktickém pravidlu ⟨*number*⟩ použit zápis čísla prostřednictvím ASCII hodnoty tokenu (pomocí $\boxed{\text{‘}}_{12}$), je následný token vyžádán bez expanze. Například:

```
\count0=‘A  \count1=‘\A   % v obou případech jde o hodnotu 65.
\count2=‘\\ % vhodný způsob, jak zapsat ASCII hodnotu znaku "\".
```

9. Když TEX vstupuje do matematického módu, rozlišuje zápisy `$` a `$$`. Druhý zápis nelze nahradit například pomocí `\def\d{$}` a následného `$\d` nebo `\d\d`. Funguje ovšem `\d$`. Jinými slovy, následující token za prvním výskytem `$` je pro tuto situaci ošetřen bez expanze. Všechny zde uvedené zápisy jsou ale natolik výjimečné, že nemá smysl o tom dále diskutovat.

Abychom měli problematiku potlačení expanze v TEXu úplnou, zopakujeme si ještě případy, kdy nedochází k expanzi na úrovni samotného expand procesoru:

10. Při načítání obsahu parametru makra.

11. Je-li ⟨*token*⟩ označen příznakem „neexpanduj“ pomocí `\noexpand`.

12. Při přeskakování tokenů v podmínkách `\if`...`\fi`.

13. První dva tokeny za `\ifx` se berou neexpandované. Stejně tak první token přicházející za primitivy `\string`, `\meaning`, `\noexpand`, `\afterassignment`, `\aftergroup` vystupuje v těchto primitivech obdobně jako parametr povelu hlavního procesoru a bere se neexpandovaný. Konečně `\expandafter` v prvním průchodu přeskakuje ⟨*token1*⟩ bez expanze. Analogicky se chová `\futurelet`.

3.3. Registry, datové typy a aritmetika TEXu

Registrem v TEXu rozumíme rezervované místo, do něhož lze uložit nebo z něj vyzvednout informaci konkrétního datového typu. Přístup do registrů je v makrech realizován prostřednictvím řídicích sekvencí, přitom rozlišujeme tyto způsoby přístupu (v závorkách jsou příklady):

- Samotná primitivní sekvence (`\baselineskip`, `\spacefactor`).

- Primitivní sekvence následovaná ⟨*number*⟩
  (\catcode⟨*number*⟩, \dimen⟨*number*⟩, \ht⟨*number*⟩).
- Ještě složitější konstrukce (\fontdimen⟨*number*⟩⟨*font*⟩).
- Řídicí sekvence deklarovaná pomocí primitivu \chardef, \dimendef apod.
  (\pageno, \dimen@, nebo např. pomocí \newcount\mycount)

Z hlediska typu udržované informace dělíme registry takto:

- Typ ⟨*number*⟩ (\tolerance, \pageno, \count⟨*number*⟩, \catcode⟨*number*⟩).
- Typ ⟨*dimen*⟩ (\hsize, \dimen⟨*number*⟩, \ht⟨*number*⟩).
- Typ ⟨*glue*⟩ (\baselineskip, \skip⟨*number*⟩).
- Typ ⟨*muglue*⟩ (\thinmuskip, \muskip⟨*number*⟩).
- Typ ⟨*tokens*⟩ (\everypar, \toks⟨*number*⟩).
- Typ ⟨*box*⟩ (\box⟨*number*⟩).

K jednotlivým typům se podrobně vrátíme v této sekci později.

Všechny registry (až na pár výjimek, které jsou označeny v části B slovem global) přijímají své hodnoty lokálně. To znamená, že se po ukončení skupiny vracejí k hodnotám, které měly v okamžiku před vstupem do skupiny, pokud nebylo ve skupině přiřazení globální (pomocí prefixu \global nebo při kladném \globaldefs). □

Registry můžeme dále rozdělit na registry vázané na určitý algoritmus TEXu (\baselineskip, \tolerance, \everypar) a registry, které slouží jen jako „skladiště“ programátora. Do druhé skupiny patří tyto registry:

- \count10 až \count255 — registry typu ⟨*number*⟩.
- \dimen0 až \dimen255 — registry typu ⟨*dimen*⟩.
- \skip0 až \skip255 — registry typu ⟨*glue*⟩.
- \muskip0 až \muskip255 — registry typu ⟨*muglue*⟩.
- \toks0 až \toks255 — registry typu ⟨*tokens*⟩.
- \box0 až \box254 — registry typu ⟨*box*⟩.

\count0 až \count9 mohou nést údaje o čísle strany vkládané do dvi. Tam je místo pro deset číselných údajů pro každou stranu. Obvykle se používá jen jeden nenulový údaj — stránková číslice v \count0. Konečně \box255 má speciální význam ve výstupní rutině.

S uvedenými registry se často pracuje prostřednictvím programátorem zavedených názvů, které lze s registrem ztotožnit použitím \countdef, \dimendef, \skipdef, \muskipdef a \toksdef. Například plain deklaruje:

```
32  \countdef\pageno=0
```

což znamená, že je možné místo zápisu `\count0` psát řídicí sekvenci `\pageno`. Tento postup můžeme chápat jako deklarování názvů „proměnných“ zvoleného typu.

Kdyby si každý programátor makra deklaroval své „proměnné“ přímo pomocí zmíněných primitivů, tj. třeba `\dimendef\mujrozmer=13`, `\countdef\mecislo=117`, pak by použití více maker současně od různých autorů způsobilo chaos. Proto se tato metoda vůbec nedoporučuje a prakticky se jména proměnných alokují pomocí maker plainu (viz `\newcount`, `\newdimen`, `\newskip`, `\newmuskip`, `\newtoks` v části B). Chceme-li třeba alokovat proměnnou typu ⟨*number*⟩, pišme `\newcount\mecislo` a pro typ ⟨*dimen*⟩ použijme `\newdimen\mujrozmer`. Většinou nás vůbec nezajímá, zda je `\mujrozmer` alokován jako `\dimen13` nebo `\dimen14`, protože pracujeme pouze s nově deklarovanou sekvencí `\mujrozmer` a díváme se na ni jako na „proměnnou“, jejíž fyzickou „adresu“ by v případě vyšších programovacích jazyků přiřadil automaticky kompilátor.

Další alokační makra, která ovšem pracují na trochu jiném principu, jsou: `\newbox`, `\newinsert`, `\newhelp`, `\newread`, `\newwrite`, `\newfam`, `\newlanguage` a `\newif`.

Alokační makra nikdy nepoužívají `\dimen0` až `\dimen9` a totéž platí pro `\skip`, `\muskip`, `\toks` a `\box`. Tyto registry můžeme použít ve svých makrech přímo bez rizika, že by to kolidovalo s nějakou deklarovanou proměnnou. Obvykle se tyto registry používají pro lokální účely pro podržení hodnoty na přechodnou dobu.

Vzhledem k tomu, že se registry po ukončení skupiny vracejí k hodnotám, které měly před vstupem do skupiny, máme vlastně efektivně k dispozici daleko více registrů. Často používáme registry jen pro uchování informace na přechodnou dobu. Příkladem je třeba zdvojení významu `\count0`. Tento registr můžeme bez obav použít k nějakému výpočtu jako pomocnou proměnnou, pokud se celý výpočet odehrává uvnitř skupiny, v rámci které nedojde k zalomení strany. V takovém případě vůbec nevadí, že `\count0` alias `\pageno` má při sestavení strany význam stránkové číslice. Po ukončení skupiny se totiž tento registr k hodnotě stránkové číslice vrátí. □

• **Manipulace s registry.** Nyní uvedeme, jakým způsobem se registry v makrech používají. Pokud se registr samotný vyskytne v kontextu povelu hlavního procesoru, hlavní procesor očekává za zápisem registru ⟨*equals*⟩ ⟨*hodnota*⟩ a provede *přiřazení hodnoty* do registru. Například:

```
\parskip = 0pt plus 2pt \relax
% Protože ⟨equals⟩ zahrnuje znak "=" jako nepovinný, je možné:
\parskip 0pt plus 2pt \relax
% Registry vyjádřené větším množstvím tokenů:
\dimen0 = 12pt  \fontdimen7\tenrm = -2pt
% nebo totéž poněkud méně přehledněji:
\dimen 0 12 pt \fontdimen 7 \tenrm -2 pt
```

Pro vyzvednutí hodnoty z registru stačí napsat registr jako parametr povelu, přičemž typ tohoto registru musí souhlasit se syntaktickým pravidlem parametru. Jsou ovšem možné některé konverze, jak uvedeme za chvíli. Například povel \kern má parametr podle pravidla ⟨*dimen*⟩, takže lze psát:

```
\dimen0 = 10pt  % uložení do registru
\kern\dimen0    % použití registru, zde totéž jako \kern10pt
```

Na pravé straně přiřazení do registru nemusí stát jen konstanta, ale třeba jiný registr stejného typu:

```
\count1 = 3
\count2 = \count1
\count\count2 = 18 % to je totéž jako \count3 = 18
```

□

Pokud chceme vypsat hodnotu registru, použijeme primitiv \the, který na úrovni expand procesoru sejme následující registr a promění jej v posloupnost tokenů kategorie 12. Tato posloupnost dekadicky vyjadřuje hodnotu registru. Jako jednotka je v případech typů ⟨*dimen*⟩, ⟨*glue*⟩ použita jednotka pt. (O jednotkách viz níže v této sekci.) Například po \dimen0=1mm nám \the\dimen0 vrátí:

```
2.84526pt
```

Často se používá \the pro registry typu ⟨*number*⟩, například pro vysázení čísla kapitoly, čísla strany, čísla obrázku a čísla čehokoliv.

Poznamenejme, že uvedené manipulace s registry platí pro všechny typy registrů s výjimkou typu ⟨*box*⟩. Odlišnou práci s tímto typem registrů probereme na konci sekce. □

• **Konstanty deklarované v \chardef.** Ačkoli konstanty deklarované primitivem \chardef nejsou přímo registry, bude nyní vhodné jim věnovat pár slov. Povel \chardef\padesát=50 deklaruje sekvenci \padesát, která je ekvivalentem k \char50. Pokud je ale tato sekvence použita nikoli jako povel hlavního procesoru, ale v kontextu parametru typu ⟨*number*⟩, chová se jako konstanta 50. Například \penalty\padesát je totéž jako \penalty50.

Konstanty z \chardef reprezentují 8 bitové nezáporné číslo, tj. je povolen rozsah 0 až 255. Kromě toho existují konstanty deklarované pomocí \mathchardef, které reprezentují 15 bitové číslo, tj. číslo v rozsahu 0 až 32 767. V plainu je například makro \break definováno takto:

```
\mathchardef\@M=10000
\def\break{\penalty-\@M}
```

Vidíme, že tělo definice `\break` obsahuje jen tři tokeny, zatímco kdybychom použili `\def\break{\penalty-10000␣}`, budeme mít v těle definice osm tokenů. Pokud v makrech často používáme nějakou konstantu (jako v tomto případě 10 000), je pro úsporu místa v „makroprostoru“ užitečné takovou konstantu deklarovat uvedeným způsobem.

Další použití konstant z `\chardef` odhalíme v alokačních makrech `\newread`, `\newwrite` a dalších. Uvedeme příklad při práci se soubory. TEX pracuje s maximálně 16 soubory určenými ke čtení a 16 soubory učenými k zápisu. Tyto soubory mají svá čísla v rozmezí 0 až 15. Přiřazení mezi tímto číslem a názvem souboru se provede primitivem `\openin` nebo `\openout` a dále primitivy `\read` a `\write` pracují pouze s těmito čísly. Alokujeme-li číslo souboru pro zápis například pomocí `\newwrite\outfile`, prakticky se provede `\chardef\outfile=`„nějaké číslo“. Dále pak pracujeme s konstantou `\outfile` jako se „souborem“. Píšeme `\openout\outfile...`, `\write\outfile...` a nezajímá nás, jaké konkrétní numero se pod sekvencí `\outfile` skrývá. □

• **Typy registrů.** V další části této sekce podrobně rozebereme jednotlivé typy registrů. Začneme tím nejjednodušším typem: ⟨*number*⟩. Pro registry tohoto typu je v TEXu rezervována paměť velikosti 4 bytů. Je tedy možno uložit čísla v rozmezí $-2^{31}+1$ až $2^{31}-1$, tj. −2 147 483 647 až 2 147 483 647. Při pokusu o uložení čísla mimo tento rozsah obdržíme chybu `Number too big`. □

Informace o rozměrech se ukládají do registrů typu ⟨*dimen*⟩, které mají v TEXovské paměti rezervovány rovněž 4 byty. Vnitřně je každý rozměr v TEXu uložen jako číslo typu ⟨*number*⟩, přičemž jednotka se předpokládá tzv. `sp`, což je definováno jako $1/2^{16}$ `pt`. Můžeme si tedy uložení rozměrů typu ⟨*dimen*⟩ představit dvojím způsobem: Buď jako celočíselný násobek jednotky `sp`, nebo jako „fixed point“ datovou reprezentaci racionálních čísel pro násobky jednotky `pt`. Řádová tečka je umístěna mezi druhým a třetím bytem v čtyřbytovém registru.

Velikost jednotky `pt` je definována vztahem 1 pt = 1/72,27 in, přitom pro palec (`in`) je známý vztah 2,54 cm = 1 in. Každý rozměr typu ⟨*dimen*⟩ musí být v TEXu označen některou z těchto jednotek: `pt`, `pc`, `in`, `bp`, `cm`, `mm`, `dd`, `cc`, `sp`, `em` nebo `ex`. Hodnoty jsou při ukládání do registru okamžitě přepočítány na vnitřní datovou reprezentaci, tj. celé násobky jednotky `sp`. Při „prezentaci“ registru pomocí `\the` a podobných nástrojů TEX použije vyjádření v desetinném zápisu násobku jednotky `pt`. Přehlednou tabulku všech jednotek včetně jejich definic najde čtenář v části B u hesla ⟨*dimen*⟩.

Protože TEX rezervuje jeden bit v registru ⟨*dimen*⟩ pro svou vnitřní potřebu, zůstává rozsah od $(-2^{30}+1)$ `sp` do $(2^{30}-1)$ `sp`. To prakticky znamená (po přepočtu na metry) rozsah zhruba od −5,75 m do 5,75 m. Není pravda, že bychom tímto rozměrem byli omezeni a nemohli v TEXu dělat billboardy či nápisy na vzducholodě.

Prostřednictvím registru `\mag` se totiž při výstupu první strany vloží do `dvi` koeficient pronásobení všech použitých rozměrů v `dvi`. Tento koeficient je pokynem pro `dvi` ovladač, který v případě, že ovládá zařízení na tisk billboardů, prostě pronásobí všechny použité rozměry a může vytvářet obří výstup. □

Registry typu ⟨*glue*⟩ nesou tři *komponenty* typu ⟨*dimen*⟩: *základní velikost*, *hodnotu roztažení* a *hodnotu stažení*. Tento datový typ je použit pro vyjádření rozměrů *pružných výplňků* v sazbě. Například zápisem:

```
48 \hskip 10pt plus 5pt minus 3pt
```

vytvoříme mezeru o základní velikosti 10 pt s hodnotou roztažení 5 pt a hodnotou stažení 3 pt. Tím je míněno, že mezera může mít proměnlivou délku a může se roztáhnout až na 10 pt plus 5 pt, tj. na 15 pt. Dále se může stáhnout až na 10 pt minus 3 pt, tj. na 7 pt. Takové úpravy mezer se uplatní při zalamování odstavců do bloku nebo na mnoha jiných místech. Upozorňujeme, že uvedený význam hodnot roztažení není zcela přesný. Podrobněji viz sekci 3.5.

Jednotlivé hodnoty typu ⟨*glue*⟩ oddělujeme klíčovými slovy `plus` a `minus`. Syntaktická pravidla jsou přesně popsána v části B u hesla ⟨*glue*⟩. Hodnota roztažení i hodnota stažení může mít navíc bezrozměrný tvar s připojením „pseudo jednotky“ `fil`, `fill` nebo `filll`. Tím je naznačeno, že je dovoleno roztažení, resp. stažení na libovolnou velikost. Počet písmen „`l`“ v této jednotce udává tzv. *řád* roztažení nebo stažení. Například:

```
49 \hskip 10pt plus 1.7fil
50 \hskip 50pt plus 10pt minus 2fill
```

Na prvním řádku je zapsána mezera, jejíž základní velikost je 10 pt a má možnost se roztáhnout na libovolnou velikost (prvního řádu). Je-li v řádku taková mezera jediná, pak hodnota konstanty (zde 1,7) není podstatná. Pokud by v řádku bylo více mezer stejného řádu, pak se mezery roztáhnou v poměru, který odpovídá poměru konstant zapsaných v hodnotách roztažení. Pak tedy začne mít hodnota konstanty 1,7 smysl a měří se poměr této konstanty s dalšími konstantami řádu `fil`. Na řádku 50 je vložena mezera se základní velikostí 50 pt, s hodnotou roztažení 10 pt a s libovolnou možností stažení.

Je-li uplatněno v nějakém řádku roztažení nebo stažení mezer, provede se tato úprava rozměrů jen v mezerách, kde jsou hodnoty roztažení nebo stažení nejvyššího řádu. Ostatní mezery zůstávají jen v základní velikosti. Znovu upozorňujeme, že podrobněji a hlavně přesněji se o výpočtu velikostí pružných mezer čtenář dočte v sekci 3.5.

Pokud je použita hodnota roztažení nebo stažení s „pseudo jednotkou“ `fil`, `fill` nebo `filll`, pak se zapsaná konstanta uloží do 4 bytového registru analogicky, jako

by se jednalo o jednotku `pt`. Znamená to tedy, že největší konstanta má hodnotu 16383,9999 a nejmenší kladný zlomek má hodnotu 0,00001. □

Registry typu ⟨*muglue*⟩ mají zcela shodnou datovou strukturu, jako registry typu ⟨*glue*⟩. Také se jedná o tři rozměrové údaje: základní velikost, hodnota roztažení a hodnota stažení. Hodnoty roztažení a stažení mohou rovněž mít „pseudo jednotku“ `fil`, `fill` nebo `filll`. Rozdíl mezi ⟨*glue*⟩ a ⟨*muglue*⟩ je pouze v použité jednotce v případě rozměrového údaje. Pro ⟨*muglue*⟩ není povoleno použít žádnou z jednotek pro vyjádření hodnoty ⟨*dimen*⟩. Je dovolena jediná jednotka s označením `mu`. Například:

```
\muskip0 = 18mu plus 1fil minus 10.5mu
```

Násobky jednotky `mu` se do čtyřbytového registru ukládají analogickým způsobem, jako by se jednalo o násobky jednotky `pt` u registrů typu ⟨*dimen*⟩. Tím je určena přesnost 0,00001 a maximum 16383,9999.

Význam jednotky `mu` bude vysvětlen v sekci 5.2. Zde jen naznačíme, že ⟨*muglue*⟩ je možno použít v parametru povelu `\hskip` podobně jako ⟨*glue*⟩, ovšem jen v matematickém módu. Jednotka `mu` je pak kontextově přepočítána na skutečný rozměr podle velikosti matematického fontu. Jinak velký font se používá pro základní velikost a pro velikost indexů různých úrovní. □

Do registrů typu ⟨*tokens*⟩ lze vložit libovolně dlouhé posloupnosti tokenů. Tomuto typu jsme se podrobně věnovali v sekci 2.4. K typu ⟨*box*⟩ se vrátíme na konci této sekce. □

• **Konverze mezi typy.** Pokud napíšeme například v místě syntaktického pravidla ⟨*number*⟩ registr typu ⟨*dimen*⟩, TEX provede konverzi. Jsou možné tyto konverze:

- ⟨*dimen*⟩ ⟶ ⟨*number*⟩; ⟨*number*⟩ je násobek jednotky `sp`.
- ⟨*glue*⟩ ⟶ ⟨*dimen*⟩; vezme se jen základní velikost.
- ⟨*glue*⟩ ⟶ ⟨*number*⟩; složené zobrazení ⟨*glue*⟩ ⟶ ⟨*dimen*⟩ ⟶ ⟨*number*⟩.

Při konverzi z ⟨*glue*⟩ do ⟨*dimen*⟩ se tedy ztrácí hodnoty roztažení a stažení a při konverzi na ⟨*number*⟩ se struktura zápisu informace v paměti vůbec nemění, pouze si TEX „odmyslí“ jednotku `sp`. Například:

```
\skip0 = .5pt plus 1fill minus 10pt
\dimen0 = \skip0  % v \dimen0 máme .5pt
\kern\skip0       % totéž jako \kern 0.5pt
\count0 = \dimen0 % v \count0 je 0.5 * 2^16 = 32768
\divide \skip0 by \skip0 % argument "by \skip0" se konvertuje
   % na "by 32768", tj. nové \skip0 má hodnotu:
   % 1sp plus 0.00003fill minus 20sp, což se přibližně rovná:
```

```
    % 0.00002pt plus 0.00003fill minus 0.0003pt
\count0 = \skip0  % nyní má \count0 hodnotu 1
```

Parametr pro dělení za slůvkem „by“ u primitivu `\divide` je typu ⟨*number*⟩. Proto se provede konverze. Podrobněji o `\divide` v další části této sekce. □

• **Sčítání a odčítání.** Registry typu ⟨*number*⟩, ⟨*dimen*⟩, ⟨*glue*⟩ a ⟨*muglue*⟩ lze mezi sebou navzájem sčítat a odčítat použitím primitivu `\advance`. Přesněji, povel `\advance` zvětší hodnotu registru o požadovanou hodnotu. Odčítání implementujeme jako zvětšení o hodnotu opatřenou znaménkem minus. Primitiv `\advance` má následující syntaxi:

```
\advance ⟨numeric variable⟩⟨optional by⟩⟨value⟩
```

kde ⟨*numeric variable*⟩ je nějaký registr typu ⟨*number*⟩, ⟨*dimen*⟩, ⟨*glue*⟩ nebo ⟨*muglue*⟩ a ⟨*value*⟩ zastupuje syntaktické pravidlo ⟨*number*⟩, ⟨*dimen*⟩, ⟨*glue*⟩ nebo ⟨*muglue*⟩. Název tohoto pravidla se musí shodovat (po případné konverzi) s typem registru. Možné jsou jen konverze uvedené v předchozím odstavci. Například při `\advance\count0 by\skip0` se `\skip0` před provedením sčítání konvertuje na typ ⟨*number*⟩.

Povel `\advance` zvětší v případě registru typu ⟨*number*⟩ nebo ⟨*dimen*⟩ tento registr o stanovenou ⟨*value*⟩. V případě registru typu ⟨*glue*⟩ a ⟨*muglue*⟩ se zvětšují nezávisle všechny tři údaje o stanovenou ⟨*value*⟩, tj. zvětší se základní velikost registru o základní velikost ⟨*value*⟩, údaj o roztažení v registru se zvětší o údaj o roztažení ve ⟨*value*⟩ a totéž platí o stažení.

Uvedeme příklad implementace jednoduché aritmetiky. Povel `\součet` sčítá následující parametry takto:

```
\součet 10 + 15  % Makro zapíše hodnotu 25
\součet 8 - 12   % Makro zapíše hodnotu -4
```

Definice makra `\součet` je jednoduchá:

```
\newcount\tempnum
\def\součet #1 #2 #3 {\tempnum=#1 \advance\tempnum by #2 #3
   \the\tempnum}
```

Poznamenejme, že `#2` nese znaménko, které je součástí syntaktického pravidla ⟨*number*⟩. Dále upozorňujeme, že makro `\součet` využívá povelů hlavního procesoru a neřeší tedy celou svou úlohu na úrovni expand procesoru. Proto nelze například psát: `\count0 = \součet 1 + 1 .` □

• **Násobení na úrovni** ⟨*dimen*⟩**.** Konstantu typu ⟨*dimen*⟩ můžeme zapsat jako desetinné číslo následované jednotkou. V takovém případě TEX provede před uložením do registru pronásobení daného desetinného čísla konstantou, která definuje použitou jednotku. Toto násobení je navíc přímo přístupné programátorovi maker. TEX totiž dovoluje zápis typu ⟨*dimen*⟩ i ve tvaru desetinného čísla následovaného nikoli jednotkou, ale registrem typu ⟨*dimen*⟩. Pak se provede násobení desetinné konstanty s registrem. Například:

```
\dimen0 = 1.5 pt
\dimen1 = 2.5 \dimen0 % tj. 2.5 * 1.5 pt = 3.75 pt
```

Toto násobení je umožněno jen na úrovni typu ⟨*dimen*⟩ a jen popsaným způsobem. Pokud tedy napíšeme:

```
\baselineskip=1.2\baselineskip
```

pak se za konstantou 1,2 provede konverze typu ⟨*glue*⟩ na typ ⟨*dimen*⟩. Teprve potom se výsledek konverze násobí konstantou 1,2 a uloží do `\baselineskip`. Protože v naší ukázce nenásleduje žádné klíčové slovo `plus` ani `minus`, bude mít nová `\baselineskip` hodnoty roztažení a stažení rovny nule, ačkoli třeba před uvedenou operací mohly být tyto hodnoty nenulové. □

Ukážeme si nyní makro, které násobí mezi sebou dva registry typu ⟨*dimen*⟩. Na hodnoty těchto registrů pohlížíme nikoli jako na rozměry, ale jako na racionální čísla, která je nutno mezi sebou násobit. Racionální čísla v registrech bereme jako násobky jednotky `pt`. Například:

```
\dimen1=1.5pt \dimen2=2.5pt
\dimen0 = \součin \dimen1 * \dimen2
```

vynásobí čísla $1{,}5 \times 2{,}5 = 3{,}75$. Výsledek bude uložen v `\dimen0` znovu ve formě s jednotkou `pt`, tj. 3,75 pt. Makro `\součin` musí zapsat `\dimen1` ve formě desetinného čísla. K tomu pomůže primitiv `\the`, ovšem dá nám trochu práce odstranit z výstupu primitivu `\the` zápis jednotky „`pt`“. Pak makro připojí bez úpravy registr `\dimen2`. TEX tedy provede požadované násobení. Makro vypadá takto:

```
{\catcode`p=12 \catcode`t=12
 \gdef\noPT #1pt{#1}}
\def\thedimen #1{\expandafter \noPT \the #1}
\def\součin #1 * #2 {\thedimen{#1} #2}
```

Vidíme, že pro odstranění tokenů $\boxed{\texttt{p}}_{12}\boxed{\texttt{t}}_{12}$, které vyprodukuje primitiv `\the` (viz například řádek 45 na straně 75), bylo potřeba udělat menší tanec s kategoriemi. Makro `\noPT` má v masce separátor $\boxed{\texttt{p}}_{12}\boxed{\texttt{t}}_{12}$. Při nastavování kategorií jsme využili toho, že slovo `\catcode` neobsahuje obě písmena `p` a `t` současně. Pak stačilo změnit

kategorie uvedených písmen ve správném pořadí. Také nebylo potřeba měnit pomocí `\let` název primitivu `\gdef`, protože tento název neobsahuje žádné z písmen `p` ani `t`. □

• **Násobení a dělení pomocí `\multiply` a `\divide`.** Tyto povely umožňují jen celočíselné násobení a dělení. Jejich syntaxe je následující:

```
\multiply ⟨numeric variable⟩⟨optional by⟩⟨number⟩
\divide   ⟨numeric variable⟩⟨optional by⟩⟨number⟩
```

kde ⟨*numeric variable*⟩ je registr typu ⟨*number*⟩, ⟨*dimen*⟩, ⟨*glue*⟩ nebo ⟨*muglue*⟩. Povel `\multiply` pronásobí hodnotu registru celočíselným ⟨*number*⟩. Pokud je registr typu ⟨*glue*⟩ nebo ⟨*muglue*⟩, jsou pronásobeny všechny tři hodnoty. Povel `\divide` vydělí hodnotu registru celočíselným ⟨*number*⟩. Při ⟨*glue*⟩ a ⟨*muglue*⟩ se dělí všechny tři hodnoty. Při dělení se v případě registru typu ⟨*number*⟩ provádí zaokrouhlení výsledku na celá čísla. V případě rozměrů se provádí zaokrouhlení na celé násobky jednotky `sp`.

Uvedeme si příklad na výpočet zbytku po celočíselném dělení:

```
\newcount\citatel \newcount\jmenovatel \newcount\zbytek
\citatel    = 12345
\jmenovatel = 567   % Nějaká testovací čísla
\zbytek = \citatel
\divide \zbytek   by \jmenovatel % celočíselné dělení
\multiply \zbytek by \jmenovatel % zpětné pronásobení
\advance\zbytek by-\citatel      % - zbytek
\zbytek=-\zbytek                 % to je zbytek
\message{zbytek po dělení je: \the\zbytek}
```
□

• **Hledání poměru racionálních čísel.** V TEXu bohužel není zabudována operace dělení racionálních čísel mezi sebou. Pokud takovou funkci potřebujeme, musíme se spokojit s omezenou přesností výsledku. Představme si, že máme ve dvou registrech typu ⟨*dimen*⟩ nějaké rozměry a úkolem je zjistit poměr těchto rozměrů s přesností na tři desetinná místa. Například máme zjištěnou šířku w textu v nezvětšeném fontu a chceme zavést font tak zvětšený, aby text vyplnil stanovenou šířku g. Koeficient zvětšení fontu tedy je g/w. V povelu `\font` potřebujeme tento koeficient vyjádřit na tři desetinná místa, protože parametr `scaled` žádá celé číslo, které je tisícinásobkem koeficientu zvětšení.

Uvedeme makro, které výše formulovanou úlohu řeší. Uživatel napíše:

```
\napis to10cm {\tenbf To je nápis}
```

a TEX vytvoří \hbox široký 10 cm který vyplní textem „To je nápis“ ve fontu \tenbf zvětšeném právě tak, aby text vyplňoval na šířku celý prostor boxu.

```
\newcount\tempnum
\def\napis to#1#{\dimen0=#1\relax \dimen1=\dimen0 \zmerbox}
\def\zmerbox #1{\setbox0=\hbox{#1}\tempnum=\wd0
  % Označme g=\dimen0, w=\wd0, spočítáme V = 1000 g / w
  \divide\tempnum by1000      % t = w / 1000
  \divide\dimen0 by\tempnum   % V = g / t = g / (w/1000)
  \analyzujbasefont #1^^X%    % \let\basefont=prvni token
  \font\tempfont=\fontname\basefont\space scaled\dimen0
  \hbox to\dimen1{\hss\tempfont\text\hss}}
\def\analyzujbasefont #1#2^^X{\let\basefont=#1\def\text{#2}}
```

Makro \napis vloží do \dimen0 požadovanou šířku g a ve \zmerbox se změří šířka textu \wd0 při nezvětšeném fontu. Další aritmetika je poměrně zajímavá. Jmenovatel w konvertujeme na celé číslo (typ ⟨*number*⟩) a vydělíme ho číslem 1000. Pak provedeme celočíselné dělení pomocí \divide. Výsledek je tedy celé číslo, obsahující tisícinásobek koeficientu g/w, což je přesně to, co jsme potřebovali. Kdybychom nejprve násobili g tisícem a pak teprve dělili, skoro jistě bychom narazili na přetečení aritmetiky, protože tisícinásobek g bývá obvykle větší, než 5 metrů. Vydělením jmenovatele tisícem sice přicházíme před vlastním vyhodnocením zlomku o tři platné číslice ve jmenovateli, ale přesnost výsledku bývá obvykle postačující. Přesnost výsledku bychom možná trochu zvětšili, kdybychom například jmenovatel dělili jen číslem 100 a čitatel násobili deseti. Protože výpočet není absolutně přesný, korigujeme nepatrné odchylky vložením \hss na obě dvě strany textu ve výsledném boxu.

Na řádku 95 si všimneme, že za slovem `scaled` je použit registr typu ⟨*dimen*⟩. Protože je vyžadováno syntaktické pravidlo ⟨*number*⟩, TEX provede konverzi na ⟨*number*⟩ podle pravidel popsaných výše. Přesně k této závěrečné konverzi směřoval celý náš výpočet. □

• **Manipulace s registry typu ⟨*box*⟩.** Typ ⟨*box*⟩ má poněkud jiný způsob práce s registry. Jedná se o registry označované jako \box0 až \box255. Přitom tento zápis není přímým vyjádřením registrů. Je vhodné představit si tyto registry jako místa v paměti pro tiskový materiál charakterizovaná číslem 0 až 255. Uložení do registru provedeme pomocí \setbox⟨*number*⟩⟨*equals*⟩⟨*box*⟩, kde ⟨*number*⟩ je číslo registru. Například:

```
\setbox0=\hbox{abc}
\setbox1=\vbox to0pt{\vss \box0 \vss}
% "box1" obsahuje tedy \vbox to0pt{\vss \hbox{abc}\vss}
```

Použití registrů typu ⟨*box*⟩ se realizuje těmito povely:

- \box⟨*number*⟩ — použití registru a jeho globální vyprázdnění.
- \copy⟨*number*⟩ — použití registru a jeho zachování.
- \unhbox⟨*number*⟩ — použití horizontálního seznamu z registru.
- \unvbox⟨*number*⟩ — použití vertikálního seznamu z registru.
- \unhcopy⟨*number*⟩, \unvcopy⟨*number*⟩ — použití seznamu a jeho zachování.
- \ht⟨*number*⟩ — výška, \dp⟨*number*⟩ — hloubka, \wd⟨*number*⟩ – šířka.
- \ifvoid⟨*number*⟩ — je registr prázdný?
- \ifhbox⟨*number*⟩ — je registr naplněn jako \hbox?
- \ifvbox⟨*number*⟩ — je registr naplněn jako \vbox?
- \showbox⟨*number*⟩ — zapíše do `log` obsah registru.

Na řádku 99 v naší ukázce byl povelem \box0 zařazen do vertikálního seznamu box z registru 0. Tímto povelem byl ale obsah boxu 0 globálně vyprázdněn, což můžete ověřit pomocí následného \ifvoid0 nebo \showbox0. Pokud je vlastnost „automatického vyprázdnění v době použití“ v rozporu s naším záměrem, musíme místo povelu \box0 použít povel \copy0.

Po ukončení skupiny se registr typu ⟨*box*⟩ vrací k původní hodnotě, kterou měl před vstupem do skupiny. Pravidlo je tedy analogické jako u jiných registrů. Vyprázdnění registru pomocí povelu \box není absolutně globální, ale vztahuje se na tu hodnotu, kterou je registr zrovna naplněn. Zní to komplikovaně, ale vše by měl objasnit následující příklad:

```
101 \setbox0=\hbox{abc}
102 { \setbox0=\hbox{xyz} {\box0} % Tím se vyprázdní hodnota "xyz".
103 } % Ukončením skupiny má box0 znovu hodnotu "abc".
104 {\box0} % Tím se vyprázdní hodnota "abc"
105 % A nyní znova:
106 \setbox0=\hbox{abc}
107 { \global\setbox0=\hbox{xyz} {\box0} % Vyprázdní se "xyz".
108 } % Protože bylo použito \global, zůstává i nyní box0 prázdný.
```

Povely \unhbox a \unvbox rovněž vyprazdňují obsah registru, ale v místě použití navíc ruší „obálku“ a zavedou jen „obsah“ použitého boxu. Nikoli tedy box jako celek. Kdybychom na řádku 99 místo \box0 použili \unhbox0, pak by se do boxu 1 zavedl `\vbox to0pt{\vss abc\vss}`, což je něco jiného než v původním případě. O tom, jak dramatický je to rozdíl, se čtenář dozví po přečtení sekce 3.4 o módech hlavního procesoru.

Pomocí \unhbox nelze „odpouzdřit“ \vbox a naopak. V takovém případě se dočkáme chyby `Incompatible list can't be unboxed`. □

Registry \ht⟨*number*⟩, \dp⟨*number*⟩ a \wd⟨*number*⟩ nám umožňují jednak zjistit rozměry boxu uloženého v registru typu ⟨*box*⟩, a jednak nastavit nové rozměry. Například:

```
\setbox0=\hbox{testovaný text}
Šířka testovaného textu je: \the\wd0.
% Je též možno nastavit novou hodnotu:
\wd0=0pt % Od této chvíle při použití boxu např. pomocí \box0
         % bude text přečnívat přes hranice boxu doprava.
```

□

Makro plainu \newbox alokuje číslo postupně od 10 do 254. Toto číslo je ztotožněno s „proměnnou typu box“ prostřednictvím primitivu \chardef, takže třeba po \newbox\mybox je možno používat povely \setbox\mybox, \box\mybox, \ifhbox\mybox apod. □

Jako příklad pro použití registrů typu box poslouží úloha na očíslování jednotlivých řádků v odstavci, jak to vidíme zde. Odstavec je formátován obvyklým způsobem a až po jeho formátování makro přidá na okraj čísla řádků. Protože TEX během hledání místa zlomu v odstavci nepustí ke slovu žádné makro, musíme jej nejprve nechat „zalomit“ odstavec do pracovního boxu. Pak teprve jednotlivé řádky rozebereme, přidáme k nim čísla a znovu je vrátíme do vnějšího vertikálního seznamu. Makro, které řeší tento úkol, vypadá následovně:

```
\newcount\linenum \newcount\tempnum \newbox\allparagraph
\def\begnum{\par\begingroup\linenum=0
  \def\par{\ifhmode\completepar\fi}%
  \setbox\allparagraph=\vbox\bgroup}
\def\endnum{\par\egroup\endgroup}
\def\completepar{\endgraf \global\advance\linenum by \prevgraf
  \tempnum=\linenum \setbox0=\hbox{}
  \loop \unskip \unpenalty \setbox2=\lastbox
     \ifhbox2 \global\setbox0=
           \hbox{\llap{$\scriptstyle\the\tempnum.$\hskip.7em}%
                 \box2\penalty0\unhbox0}
       \advance\tempnum by-1
  \repeat
  \egroup \noindent\unhbox0\unpenalty \endgraf
  \setbox\allparagraph=\vbox\bgroup}
```

Popíšeme si funkci makra podrobněji. Upozorňujeme čtenáře, že některé části výkladu se opírají o znalosti práce s boxy a základní mechanismus sestavování odstavce. Tyto záležitosti jsou vysvětleny později v sekcích 3.5, 6.1 a 6.4. Nejprve uvedeme uživatelský popis makra: Uživatel napíše \begnum, pak následuje jeden nebo více odstavců, které budou mít číslovány řádky. Proces číslování je ukončen \endnum.

Makro \begnum zahájí skupinu, uvnitř které se nastaví čítač řádků \linenum na nulu a bude lokálně předefinováno \par jako \completepar. Nakonec se pro text prvního odstavce otevře \vbox\bgroup, který nevystupuje do vnějšího seznamu,

ale uloží se do boxu `\allparagraph`. Při ukončení odstavce pomocí `\completepar` se provedou tyto činnosti: čítač řádků `\linenum` se zvětší o počet řádků v odstavci a `\tempnum` se rovněž nastaví na tuto hodnotu. V cyklu `\loop` potom postupně odebereme „zespodu“ jeden řádek odstavce, dáme mu číslo `\tempnum` a `\tempnum` zmenšíme o jedničku. Odebraný řádek máme v boxu 2 a přidáme ho do boxu 0 společně s číslem řádku (v `\llap`) a s již uloženým materiálem v `\box0`. Mezi jednotlivé boxy klademe `\penalty0`, což budou místa zlomu pro opakované zpracování odstavce. Cyklus ukončíme v okamžiku, kdy se pomocí `\lastbox` už nepodařilo odebrat řádek jako `\hbox`, tj. když je vertikální seznam v `\allparagraph` vyprázdněn.

Nyní máme materiál celého odstavce ve správném pořadí v jednom „dlouhém“ boxu 0. Pokud takový materiál „vypustíme“ do vnějšího vertikálního seznamu pomocí `\noindent\unhbox0\endgraf`, TEX znovu provede řádkový zlom v místech `\penalty0` a výsledný odstavec, tentokrát už s čísly řádků, se objeví ve vnějším vertikálním seznamu. □

3.4. Šest módů hlavního procesoru

Hlavní procesor pracuje vždy v jednom ze šesti následujících módů:

- hlavní vertikální mód (vertical mode),
- vnitřní vertikální mód (internal vertical mode),
- odstavcový horizontální mód (horizontal mode),
- vnitřní horizontální mód (restricted horizontal mode),
- vnitřní matematický mód (math mode),
- display matematický mód (display math mode).

Zhruba řečeno, při vytváření hlavního tiskového materiálu, který podléhá stránkovému zlomu, pracuje TEX v hlavním vertikálním módu. Uvnitř `\vbox`u je ve vnitřním vertikálním módu, při zpracování textu odstavce je v odstavcovém horizontálním módu a uvnitř `\hbox`u je ve vnitřním horizontálním módu. Konečně mezi `$...$` je ve vnitřním matematickém módu a mezi `$$...$$` je v display matematickém módu. Zbytek této sekce se snaží problematiku módů poněkud více precizovat.

Pokud budeme v dalším textu mluvit o vlastnosti společné hlavnímu vertikálnímu módu a vnitřnímu vertikálnímu módu, budeme zkráceně říkat *vertikální mód*. Analogicky budeme mluvit o *horizontálním módu*, místo o horizontálním módu vnitřním a odstavcovém a konečně o *matematickém módu*, místo matematickém módu vnitřním a display. Také budeme zkracovat termín „odstavcový horizontální mód“ na *odstavcový mód*.

Plno povelů hlavního procesoru má význam závislý na módu, ve kterém se zpracování zrovna nalézá. Například povel `\par` ve vertikálním módu a ve vnitřním

horizontálním módu neudělá nic. V odstavcovém módu tento povel uzavře načítání textu odstavce, kompletuje jednotlivé řádky odstavce a TEX se vrací do vertikálního módu, ve kterém byl před zahájením čtení textu odstavce. Konečně v matematickém módu způsobí povel `\par` chybové hlášení.

Z uvedené vlastnosti povelu `\par` okamžitě plyne, že vyskytne-li se více povelů `\par` za sebou, pak první `\par` může provést kompletaci odstavce a ostatní `\par` už vstupují do hlavního procesoru ve vertikálním módu, a proto neprovedou nic. Důsledek: více `\par` za sebou se chovají jako jedno. Prakticky to třeba znamená, že nevadí, zda uživatel vkládá mezi odstavce jeden prázdný řádek nebo třeba deset prázdných řádků. (Připomínáme, že token procesor z každého prázdného řádku obvykle vytvoří jedno `\par`.)

Níže uvedeme všechny možnosti, při kterých hlavní procesor přechází z jednoho módu do druhého. Definujeme tím vlastně stavový automat hlavního procesoru. □

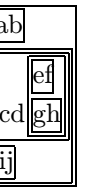
Než se pustíme do podrobného výkladu jednotlivých módů, budeme se věnovat výzkumu, k čemu ty módy jsou. Popíšeme, jak se chová tiskový materiál, který je v tom kterém módu hlavním procesorem vytvářen.

Ve vertikálním módu vytváří hlavní procesor *vertikální seznam* tiskového materiálu, v horizontálním módu vytváří *horizontální seznam* tiskového materiálu a konečně v matematickém módu vytváří *matematický seznam*. Horizontální seznam může obsahovat sazbu jednotlivých znaků, boxy a další materiál (vše je kladeno vedle sebe zleva doprava), vertikální seznam může obsahovat boxy a další materiál (vše je kladeno pod sebou shora dolů). Matematický seznam je kapitola sama pro sebe (viz sekci 5.1).

Vertikální a horizontální seznamy mohou být do sebe prostřednictvím boxů vnořené. Obsahuje-li například vertikální seznam `\hbox`, pak tento box obsahuje horizontální seznam. V něm mohou být boxy, například `\vbox` s vertikálním seznamem nebo `\hbox` s horizontálním seznamem atd. Množství vnořených úrovní těchto seznamů je omezeno pouze velikostí paměti. Například na konstrukci:

```
\vbox{\hbox{ab}
      \vbox{\hbox{cd\vbox{\hbox{ef}
                          \hbox{gh}}}
            \hbox{ij}}}
```

můžeme pohlížet jako na `\vbox`, který obsahuje vertikální seznam se dvěma elementy: `\hbox` a `\vbox`. Budou pod sebou. První element je `\hbox` obsahující horizontální seznam se dvěma znaky: „a, b“. Budou vedle sebe. Druhý element vertikálního seznamu je `\vbox` obsahující vertikální seznam se dvěma elementy: `\hbox` a `\hbox`. Budou pod sebou. A tak dále. Abychom se do toho moc nezamotali, ukážeme raději výsledek:



ab
ef
cdgh
ij

a abychom se v tom ještě lépe vyznali, dokreslíme rámečky:

Pro hloubavé čtenáře uvedeme kód, kterým jsme ty rámečky v naší ukázce vytvořili. Pokud tomu zatím nebudete rozumět, nezoufejte. Tajemství tohoto kódu odhalíte například po přečtení sekce 3.5.

```
\begingroup \offinterlineskip \def\,{\kern1pt}
\def\Vbox#1{\vbox{\,\hbox{\,\vrule\vbox{\hrule
                                        #1\hrule}\vrule\,}\,}}
\def\Hbox#1{\hbox{\,\vbox{\,\hrule\hbox{\strut\vrule
                                        #1\vrule}\hrule\,}\,}}
\Vbox{\Hbox{ab}
      \Vbox{\Hbox{cd\lower6.3pt\Vbox{\Hbox{ef}
                                     \Hbox{gh}}}
            \Hbox{ij}}}
\endgroup
```

□

Nyní si uvedeme úplný seznam všech elementů, které se mohou vyskytovat ve vertikálním a v horizontálním seznamu. V závorce uvádíme primitivy, prostřednictvím kterých tyto elementy hlavní procesor vytváří.

Vertikální seznam může obsahovat tyto elementy:

- Box (`\hbox`, `\vbox`, `\vtop`).
- Linka (`\hrule`).
- Kern, tj. pevný výplněk nepodléhající zlomu (`\kern`).
- Pružný výplněk typu ⟨*glue*⟩ (`\vskip`, `\leaders`, `\vfil`, `\vss`, ...).
- Penalta, tj. trest za zlom v daném místě (`\penalty`).
- Značka pro odkazy do plovoucího záhlaví (`\mark`).
- Odkaz na pozdější zápis do souboru (`\write`).
- Odkaz na vertikální seznam typu „insert“ (`\insert`).

Horizontální seznam může obsahovat tyto elementy:

- Znak nebo ligatura (povel pro vysázení znaku).
- Box (`\hbox`, `\vbox`, `\vtop`).
- Linka (`\vrule`).
- Kern, tj. pevný výplněk nepodléhající zlomu (`\kern`).
- Pružný výplněk typu ⟨*glue*⟩ (`\hskip`, `\leaders`, $\boxed{\sqcup}_{10}$, `\hfil`, `\hss`, ...).
- Penalta, tj. trest za zlom v daném místě (`\penalty`).

- Způsob rozdělení slova (`\discretionary`).
- Odkaz na pozdější zápis do souboru (`\write`).
- Odkaz (`\vadjust`, `\mark`, `\insert`). Později se přesune do vnějšího seznamu.

Při startu TEXu je nastaven vždy hlavní vertikální mód na zpracování vertikálního seznamu, který podléhá stránkovému zlomu. Z tohoto módu se hlavní procesor občas přepíná do jiných vnořených módů a po sestavení tiskového materiálu ve vnořených módech se hlavní procesor zásobníkovým způsobem vrací k módům, ve kterých byl v době vstupu do vnořeného módu. Na konci zpracování (při povelu `\end`) se předpokládá, že se TEX dostane zpětně do hlavního vertikálního módu. Pokud se tak nestalo (například `\end` uvnitř `\vbox`u), TEX ohlásí chybu.

Z libovolného módu přechází TEX do vnitřního vertikálního módu při povelu `\vbox` nebo `\vtop`. Přesněji, TEX si zapamatuje případné ⟨*box specification*⟩ a po otevření skupiny, která za specifikací boxu musí následovat, přejde do vnitřního vertikálního módu. Po kompletaci vertikálního seznamu tohoto boxu (při zavření skupiny na úrovni tohoto boxu) se vertikální seznam případně upraví (viz sekci 3.5) a výsledný box vystupuje jako jednolitý element ve vnějším seznamu. TEX se vrací do módu, ve kterém byl před vstupem do vnořeného módu.

Z libovolného módu přechází TEX do vnitřního horizontálního módu při povelu `\hbox`. Situace je analogická, jako ve výše uvedeném případě pro `\vbox` a `\vtop`.

Vraťme se k našemu příkladu vnořených boxů na straně 86. Na řádku 129 vstupuje TEX při zpracování povelu `\vbox` z hlavního vertikálního módu do vnitřního vertikálního módu. Znamená to, že povel `\hbox`, který následuje, bude zpracován v rámci tohoto vnitřního vertikálního módu. Povel ovšem způsobí přechod do vnitřního horizontálního módu a v rámci něj se zpracuje horizontální seznam obsahující sazbu „ab“. Po kompletaci tohoto horizontálního seznamu se TEX vrací do vnitřního vertikálního módu a v rámci něj zpracuje na řádku 130 povel `\vbox`. To způsobí přechod do vnitřního vertikálního módu o jednu úroveň hlouběji než dosud. A tak dále. □

• **Odstavcový mód.** V tomto módu TEX zpracovává horizontální seznam, který je při kompletaci rozlámán do jednotlivých řádků odstavce a tyto řádky jsou vloženy jako boxy do vnějšího vertikálního seznamu. Můžeme si představit, že TEX při startu odstavcového módu otevře neomezeně velký `\hbox` a v něm sestaví text odstavce do jednoho řádku. Při ukončení tohoto módu (prostřednictvím povelu `\par`) rozlomí TEX tento „dlouhý“ box na jednotlivé řádky.

Do odstavcového módu lze vstoupit jen z vertikálního módu. Vstup do odstavcového módu je buď *explicitní* (pomocí `\indent` nebo `\noindent`), nebo *implicitní*. Implicitní vstup do odstavcového módu proběhne například při povelu pro sazbu znaku. To je totiž povel, který v rámci vertikálního seznamu nelze provést. V následujícím příkladě:

```
143 \kern2cm
144 První odstavec.\kern3cm
145
146 \kern4cm Druhý odstavec. \end
```

se nejprve vloží výplněk `\kern2cm` ve vertikálním směru do hlavního vertikálního seznamu. Pak hlavní procesor narazí na povel P$_{11}$, který pro něj znamená sazbu znaku. To lze ovšem provést jen v horizontálním seznamu. Proto dojde k implicitnímu přechodu do odstavcového módu. V něm například pokyn `\kern3cm` způsobí vložení výplňku velikosti 3 cm v horizontálním směru. Prakticky asi nic neuvidíme, protože tento výplněk splyne s mezerou ve východovém řádku odstavce. Napsali jsme to do ukázky jen proto, abychom mírně popletli čtenáře. Dále na prázdném řádku 145 vytvoří token procesor sekvenci `\par`, která ukončí zpracování prvního odstavce. Jsme tedy zpětně v hlavním vertikálním módu. Povel `\kern4cm` nyní vloží výplněk pod prvním odstavcem ve vertikálním směru. Pak se kvůli sazbě písmene D provede implicitní přechod do odstavcového módu. Povel `\end` kompletuje poslední (zde druhý) odstavec a TEX se vrací do vnějšího vertikálního módu. V něm proběhnou závěrečné aktivity: stránkový zlom a výstup strany do `dvi`.

Zde je úplný seznam všech povelů, které si vynutí implicitní přechod z vertikálního módu do odstavcového módu: sazba znaku přímo nebo pomocí `\char` nebo pomocí sekvence definované v `\chardef`, dále primitivy `\hskip`, `\hfil`, `\hfill`, `\hss`, `\hfilneg`, `\unhbox`, `\unhcopy`, `\vrule`, `\valign`, `\accent`, `\discretionary`, `\-`, `\␣`, `\noboundary`. Do odstavcového módu se též implicitně přejde před vstupem do matematického módu (vnitřního i display), pokud se přepínač matematického módu vyskytl ve vertikálním módu.

Při explicitním přechodu pomocí `\noindent` se založí prázdný horizontální seznam pro text odstavce a expanduje se obsah proměnné `\everypar`. Při explicitním přechodu pomocí `\indent` se rovněž založí horizontální seznam, ovšem do něj TEX vloží prázdný box o šířce `\parindent` (odstavcová zarážka) a pak teprve expanduje obsah proměnné `\everypar`.

Při implicitním přechodu do odstavcového módu TEX nejprve odloží zpět do čtecí fronty token, který vyvolal implicitní přechod. Pak otevře horizontální seznam a do něj vloží prázdný box o šířce `\parindent`. Dále expanduje `\everypar`. Při prázdném `\everypar` se odložený token z čtecí fronty znovu dostává do hlavního procesoru, kde je interpretován jako povel. Při neprázdném `\everypar` je další osud odloženého tokenu v rukou maker z `\everypar`, která mohou vzít tento token do svých parametrů a naložit s ním dle libosti.

V následujícím příkladě chceme, aby každý odstavec začínal větším písmenem:

```
147 \font\vetsi=csr10 scaled\magstep5
148 \def\zvetsiznak#1{{\vetsi #1}}
149 \everypar={\zvetsiznak} \parindent=0pt
150 První odstavec. \par Druhý odstavec, atd.
```

Na řádku 150 přichází token P$_{11}$ jako povel, který způsobí implicitní přechod do odstavcového módu. Proto se tento token vrátí zpět do čtecí fronty. Pak je založen horizontální seznam s boxem o šířce `\parindent=0pt` (nechceme odstavcovou zarážku). Dále je expandováno makro `\zvetsiznak`, které načte do parametru `#1` odložený token P$_{11}$. Tím dostáváme na výstupu větší písmeno P, tedy první písmeno odstavce. Větší bude i písmeno D z druhého odstavce a první písmena ze všech případných dalších odstavců.

Uvedeme ještě jednu aplikaci `\everypar` z LaTeXu. V tomto makru bývá obvykle potlačena odstavcová zarážka u prvního odstavce pod nadpisem. Je to uděláno tak, že při sestavování nadpisu (například v makru `\section`) je řečeno `\everypar={\sejmibox\everypar={}}`. Při prvním otevření odstavce se expanduje `\everypar`. Makro `\sejmibox` odstraní z horizontálního seznamu box šířky `\parindent`, který tam byl vložen vnitřním algoritmem TeXu. Použije k tomu primitiv `\lastbox`. Dále se vyprázdní obsah `\everypar`, takže další odstavce už zase zarážku mít budou. □

Nyní se seznámíme se všemi případy, kdy dochází k ukončení odstavcového módu a návratu do vertikálního módu, ze kterého byl odstavcový mód zahájen. Nejčastěji je odstavcový mód ukončen povelem `\par`. Prázdný řádek ve vstupním textu se (obvykle) v token procesoru promění na token par a tento token (obvykle) má přímo význam povelu `\par`. Pokud je odstavec ukončen přímo povelem `\par` (nebo povelem `\par` z prázdného řádku), říkáme, že proběhlo *explicitní* ukončení odstavcového módu.

Kromě toho se může odstavcový mód ukončit *implicitně*, pokud se v tomto módu objeví některý z následujících povelů: `\vskip`, `\vfil`, `\vfill`, `\vss`, `\vfilneg`, `\end`, `\unvbox`, `\unvcopy`, `\halign`, `\hrule`, `\dump`. Za takové situace TeX vloží token, který způsobil implicitní přechod, zpět do čtecí fronty a vloží před něj do čtecí fronty sekvenci par. Pak nechá tuto sekvenci (v případě, že se jedná o makro) expandovat expand procesorem. Pokud má sekvence par původní význam primitivního povelu, vykoná se přímo povel `\par` a potom se vykoná povel, který byl odložen do čtecí fronty a který způsobil implicitní přechod. Pokud je ovšem sekvence par předefinována například ve významu makra, toto makro může vzít do svých parametrů odložený token, který způsobil implicitní přechod. Další osud odloženého tokenu je tedy zcela v rukou tohoto makra.

V předchozí ukázce ze strany 89 je na řádku 146 povel `\end`, který se nejprve objeví v odstavcovém módu. Tento povel způsobí implicitní přechod, tj. TeX vloží sekvenci par a ta se dostane do hlavního procesoru jako povel `\par`. TeX tedy kompletuje

odstavec a potom (už ve vertikálním módu) znovu obdrží povel \end, který vykoná závěrečnou práci (stránkový zlom a výstup do dvi).

Uvedeme si příklad, ve kterém předefinujeme sekvenci par. Budeme chtít, aby na konci každého odstavce byl čtvereček, který čtenář v této knize čte jako „uff, zase jsem strávil kousek uceleného textu“. V naší knize se tento čtvereček nevyskytuje za každým odstavcem, proto ve formátu knihy není sekvence par předefinována. Autor ukončiv myšlenku si vždy řekl: „uff, musím se na chvíli protáhnout“ a napsal na konec odstavce ručně sekvenci \endpar. Na druhé straně, následující makro nám vytvoří čtvereček na konci *každého* odstavce:

```
\def\par{\ifhmode\endpar\fi\endgraf}
\def\endpar{\unskip~\hfill\lower1pt\vbox{\hrule
   \hbox to 7pt{\vrule height7pt\hfil\vrule}\hrule}}
```

Makro ve skutečnosti vytvoří obdélníček (pravoúhelník je o 0,8 pt vyšší), ale z historických důvodů tomu budeme nadále říkat čtvereček. Oko to nepozná. Vidíme, že nejprve je \par předefinováno tak, aby ve vertikálním módu neudělalo nic (proto ten test \ifhmode). Kdybychom na to zapomněli, pak dva prázdné řádky by znamenaly jeden prázdný odstavec. Sekvence \endpar vyrobí příslušný čtvereček v horizontálním módu. Pak se prostřednictvím \endgraf provede povel \par v původním primitivním významu tohoto slova. Čtverečky nám to bude dělat za každým odstavcem, i v případě, že byl odstavec ukončen implicitně. Je to proto, že při implicitním ukončení se uměle vložená sekvence par zpracuje expand procesorem, takže se expanduje na kód, vedoucí k vytvoření čtverečku.

Existují ovšem odstavce, které čtvereček při použití našeho kódu nebudou mít. Nezmínili jsme se totiž ještě o posledním způsobu ukončení odstavcového módu: tzv. *vynucené ukončení*. K vynucenému ukončení dojde v případě, že hlavní procesor v odstavcovém módu obdrží povel ukončení skupiny, která odpovídá ukončení \vboxu (též \vtop nebo \vcenter), uvnitř kterého byl odstavcový mód zahájen. V takovém případě se provede vnitřní povel \par a nikoli sekvence par. Vzápětí se kompletuje vertikální seznam uzavíraného boxu. Podrobně si projdeme tento příklad:

```
\vbox{abc}
```

Zde se ve vnitřním vertikálním módu objevilo písmeno „a“. Proto TEX přešel (implicitně) do odstavcového módu. V horizontálním seznamu tohoto módu máme odstavcovou zarážku a dále sazbu „abc“. Pak se provede vynucené ukončení odstavcového módu. Do \vboxu se vloží jeden řádek odstavce o šířce \hsize (viz sekci 6.4) a kompletuje se \vbox. Tento \vbox má tedy šířku rovnou \hsize a výšku písmene „b“. Čtvereček na konci řádku (i při předefinovaném \par) není. Na konci odstavce, který zrovna čteme, čtvereček s radostí uděláme. Uff. □

Vraťme se ještě k problematice předefinování sekvence `\par`. Pokud ji předefinujeme tak, že v makru nefiguruje `\endgraf`, hrozí nebezpečí chyby `TeX capacity exceeded`. Vyzkoušejte si:

```
\def\par{konec}

\end
```

Co se stalo? Sekvence `par` z prázdného řádku se expandovala na „`konec`“. Písmeno „`k`“ způsobilo implicitní přechod do odstavcového módu. V něm tedy přichází na řadu povel `\end`. TEX proto vloží sekvenci `par` a ta se znovu expanduje na „konec“. V horizontálním seznamu už máme „koneckonec“. Pak přijde na řadu znovu `\end`, ovšem znovu v horizontálním módu. TEX tedy znovu vloží sekvenci `par`. A tak pořád dokola. Až se překročí kapacita horizontálního seznamu, která je plněna textem „koneckoneckoneckonec…“, TEX ohlásí chybu. □

Napíšeme-li ve vertikálním módu `\hbox{abc}` nebo `\indent\hbox{abc}`, okamžitě vidíme, že to pokaždé znamená něco jiného. V prvním případě se `\hbox` vloží přímo do vertikálního seznamu, zatímco v druhém případě se vloží do horizontálního seznamu v rámci odstavce. Z toho kouká jedna záludnost TEXu, na které si už mnoho lidí spálilo prsty. Ukážeme si ji na příkladě.

Představme si, že třeba nemáme ve fontu české uvozovky a sestavujeme je z anglických. Uvozovky dole vyrobíme snížením uvozovek (”), které mají ve fontu `cmr10` pozici `\char34`. Uživatel použije makro `\uv`. Toto makro je zde velmi zjednodušeno, prakticky se dělá ještě trochu jinak (viz heslo `\aftergroup` v části B).

```
\def\clqq{\vbox to0pt{\vss\hbox{\char34}\kern-1.4ex}}
\chardef\crqq=92
\def\uv #1{\clqq #1\crqq}
```

V uvedené ukázce se dopouštíme chyby. Při použití makra uvnitř odstavce vypadá vše v pořádku. Skutečně, když napíšeme `\uv{Cosi}`, dostáváme „Cosi“. Pokud ale napíšeme makro `\uv{Cosi}` na začátku odstavce, obsadí `\vbox` s otevíracími uvozovkami samostatný řádek ve vertikálním seznamu a teprve písmeno `C` způsobí přechod do odstavcového módu. Výsledek pak (při `\parindent=20pt`) vypadá takto:

”
Cosi“.

Věc opravíme tak, že před otevíracími uvozovkami přidáme makro plainu, které ve vertikálním módu způsobí implicitní přechod do odstavcového módu. Makro se jmenuje `\leavevmode`, takže pišme:

```
161 \def\uv #1{\leavevmode\clqq #1\crqq}
```

Nyní už tušíme, proč je v příručce LaTeXu zatajena před uživatelem primitivní konstrukce `\hbox`, ale místo toho se tam doporučuje důsledně používat makro `\mbox`. Toto makro je totiž definováno jako `\leavevmode\hbox`. □

• **Matematický mód.** To nejlehčí nakonec. Uvedeme případy, kdy TeX vstupuje do matematického módu a vystupuje z něj. Do vnitřního matematického módu se vstupuje prostřednictvím tokenu $[\$]_3$ a do display matematického módu se vstupuje pomocí dvojice tokenů $[\$]_3\,[\$]_3$. Na ASCII hodnotě tokenu nezáleží, ovšem vesměs se používá znak `$`, aby měl autor textu neustále na zřeteli, že kvalitní matematická sazba je drahá.

Do vnitřního matematického módu je možno vstoupit jen z horizontálního módu (odstavcového nebo vnitřního). Do display matematického módu je možné vstoupit jen z odstavcového módu. Pokud se značka vstupu do matematického módu ($[\$]_3$) vyskytne ve vertikálním módu, pak se nejprve implicitně otevře odstavcový mód a v něm teprve příslušný matematický mód.

Při prvním výskytu značky $[\$]_3$ TeX ošetří, zda následující token je také kategorie 3. Tím rozlišuje mezi vstupem do vnitřního a display módu. Tato vlastnost má výjimku ve vnitřním horizontálním módu. V tomto případě výjimečně zápis `$$` znamená nikoli vstup do display matematického módu, ale vstup a hned výstup z vnitřního matematického módu.

Uvnitř matematického módu lze přejít do vnitřního horizontálního módu pomocí `\hbox` a vnitřního vertikálního módu pomocí `\vbox`, `\vtop` a `\vcenter`. V matematickém módu lze tedy navíc použít `\vcenter`, který při kompletaci vertikálního materiálu centruje box na matematickou osu.

Uvnitř matematického módu nelze přejít do vnořeného matematického módu přímo, ale lze tam přejít prostřednictvím boxů. Například:

```
162 Odstavcový mód $a+b=\hbox{vnitřní horizontální mód $c^2$}$ znovu
163 pokračuje odstavcový mód.
```

Výstup z vnitřního matematického módu je realizován tokenem $[\$]_3$ a z display matematického módu dvojicí $[\$]_3\,[\$]_3$. TeX se v takovém případě vrací do horizontálního módu, ve kterém byl před vstupem do matematického módu. □

V souvislosti s přechody do matematického módu jsem se setkal jen s jednou záludností. Existuje rozdíl mezi těmito dvěma vstupy do display matematického módu:

```
164 Tady je výpočet:
165 $$ a+b = c $$
166 další text.
167
168 $$ a+b = c $$ \par
```

První display mód na řádku 165 se odehrává v rámci odstavcového módu. Před vložením rovnice TEX přeruší odstavec za slovem „výpočet:“ a do vnějšího vertikálního seznamu vloží řádek s textem „Tady je výpočet:“ Pak na samostatný řádek vloží výsledek matematické sazby display módu. Dále pokračuje v sestavování horizontálního seznamu, který zahájí bez odstavcové zarážky. Výsledkem je řádek s obsahem „další text.“, který se připojí pod rovnici. Jaké jsou přesně vloženy mezery nad a pod rovnicí, o tom je možné si přečíst v sekci 5.6.

Na druhé straně, v případě display módu na řádku 168 se musí nejprve založit horizontální seznam. Ten je před rovnicí kompletován jako prázdný (přesněji, obsahuje jen odstavcovou zarážku). Nad rovnicí tedy vzniká prázdný řádek! Pod tímto řádkem je standardní mezera, která se vkládá nad rovnicí. Pak následuje rovnice. Dohromady tedy nad rovnicí máme více volného prostoru kvůli prázdnému řádku. Ačkoli máme okamžitě za rovnicí povel `\par`, prázdný řádek pod rovnicí v tomto případě nevzniká. □

3.5. Boxy

Box je základním stavebním kamenem v tiskovém materiálu. Boxem může být skupina znaků, řádek v odstavci, položka v tabulce, celá tabulka, záhlaví nebo celá strana. Podobné boxům jsou též další elementy sazby: sazba jednotlivého znaku a linky. Tyto objekty se ovšem v mnohých ohledech od boxů liší, a proto je nebudeme nazývat termínem box.

Box, stejně jako sazba znaku a linky, je v TEXu interpretován jako pravoúhlý dvourozměrný útvar, který se umisťuje do sazby s ohledem na tzv. *účaří* (baseline). Účařím je pomyslná základní vodorovná linka každého řádku. Každý box (rovněž sazba znaku nebo linky) má referenční bod a je určen třemi rozměry. Od referenčního bodu nahoru se měří *výška* (height), dolů *hloubka* (depth) a doprava *šířka* (width). Referenčním bodem prochází účaří.

Vlastní kresby znaků nemusí mít nic společného s rozměry boxu. Například při sazbě skloněného písma obvykle kresby znaků přečnívají přes hranice, určené jejich

výškou, hloubkou a šířkou. TEX samotný se zabývá jen uvedenými třemi rozměrovými parametry každého znaku a vůbec ho nezajímá, jak budou v `dvi` ovladači znaky vykresleny. Rozměrové údaje každého znaku načítá z metriky `tfm`. Bývá obvyklé, že se tyto rozměrové údaje co nejvíce blíží rozměrům kresby znaků, zvláště jejich šířky. Podle šířek totiž staví TEX jednotlivé znaky vedle sebe, což je pro sazbu textu snad to nejdůležitější.

Chcete-li zjistit, jaké rozměry mají jednotlivé znaky nebo boxy, použijte třeba tento kód:

```
169 \setbox0=\hbox{testovaný text}
170 \message{výška: \the\ht0, hloubka: \the\dp0, šířka: \the\wd0}
```

Chcete-li si nakreslit „obvod“ boxu, abyste viděli jeho rozměry graficky společně s textem, pak můžete použít třeba:

```
171 \def\obvod #1{\vbox{\hrule \hbox{\vrule #1\vrule}\hrule}}
172 \obvod{testovaný text}
```

□

V horizontálním seznamu se jednotlivé znaky, linky a boxy usazují vedle sebe podle společného účaří. Výjimky umístění směrem nahoru a dolů je možné deklarovat primitivem `\raise` nebo `\lower`. Vedle sebe se jednotlivé boxy, linky a znaky vkládají bez mezer, pokud samozřejmě není vložen nějaký výplněk. Užitečná je představa tzv. *aktuálního bodu sazby*. Při vložení dalšího elementu sazby (box, linka nebo sazba znaku) se referenční bod tohoto elementu kryje s aktuálním bodem sazby a aktuální bod sazby se následně posune o šířku elementu směrem doprava.

Boxy mohou mít některé rozměry záporné. Například box o záporné šířce −20 pt můžeme vytvořit pomocí `\hbox{\kern-20pt}` nebo `\hbox to-20pt{}`. Pokud vložíme takový box do horizontálního seznamu, pak se nám aktuální bod sazby posune o 20 pt doleva. Dodejme, že `\hbox` nemůže mít zápornou výšku a hloubku a `\vbox` nemůže mít zápornou šířku. Výjimka z tohoto pravidla může nastat při „ručním“ ukládání hodnot do registrů `\ht`, `\dp` a `\wd`.

Principiálně lze vytvořit metriku fontu, kde budou i základní rozměry znaků záporné. V této souvislosti ovšem upozorňuji, že sazba zprava doleva v orientálních jazycích je většinou řešena trochu jinak. Používá se poněkud upravený TEX s názvem TEX-XET.

Konečně uvedeme pravidla usazování tiskových elementů ve vertikálním seznamu. V tomto seznamu se jednotlivé boxy a linky usazují s ohledem na požadavek, aby referenční body byly přesně pod sebou. Výjimky je možné deklarovat pomocí primitivu `\moveleft` a `\moveright`. Při usazování boxů pod sebou se většinou vkládá mezířádkový výplněk, aby například vzdálenosti jednotlivých účaří boxů byly pokud možno ekvidistantní. Podrobněji viz sekci 3.7. □

V dalším textu až do konce této sekce se budeme zabývat operacemi, které jsou v činnosti při vzniku boxu. Říkáme tomu *kompletace boxu*. Pod tímto pojmem rozumíme proces, kdy se vertikální nebo horizontální seznam tiskového materiálu uzavírá do boxu. Při kompletaci boxu se odehrávají tyto události:

- Vypočítá se celková výška, hloubka a šířka nového boxu.
- Pružné výplňky uvnitř boxu získají definitivní rozměr.
- Linky s neurčitými rozměry získají v boxu definitivní rozměr.
- Vypočítá se tzv. „hodnota badness“ boxu.
- Je-li badness vyšší než požadovaná, vypíše se varování.

V následujícím textu rozebereme každou zmíněnou událost podrobněji. Než se do toho pustíme, je třeba upozornit na to, že objekty, které při kompletaci získají definitivní rozměr, se mohou zpětně vrátit k neurčitému rozměru. K takovým situacím dochází při použití primitivu `\unhbox`, resp. `\unvbox`, které vlastně provádějí inverzní proces ke kompletaci. Výsledkem činnosti těchto primitivů je horizontální, resp. vertikální seznam, který byl dříve kompletován do boxu.

• **Horizontální box.** Pro jednoduchost začneme popisem kompletace horizontálního boxu (`\hbox`). Do boxu tedy vstupuje horizontální seznam. Celková výška nového boxu je určena maximem výšek jednotlivých boxů, linek a znaků uvnitř seznamu. Nejsou-li tam takové elementy, případně všechny mají výšku zápornou, je výška nového boxu nula. Analogicky se počítá celková hloubka boxu. Samozřejmě boxy, které jsou v seznamu posunuty nahoru nebo dolů pomocí `\raise` nebo `\lower`, se při výpočtu maxima výšky elementů v horizontálním seznamu prezentují vzdáleností horního okraje posunutého boxu od účaří. To odpovídá přirozenému požadavku, aby výška kompletovaného boxu byla počítána podle „nejvýše vyčnívajícího elementu“ v seznamu. Analogicky se počítá hloubka kompletovaného boxu.

Jakmile je stanovena výška a hloubka kompletovaného boxu, lze určit rozměry pro linky typu `\vrule`. Všechny tyto linky s neurčitou výškou budou mít výšku kompletovaného boxu, všechny linky s neurčitou hloubkou budou mít hloubku kompletovaného boxu a všechny linky s neurčitou šířkou budou mít šířku rovnou implicitní hodnotě 0,4 pt. □

Nyní se budeme zabývat vyhodnocením šířky horizontálního boxu. TEX nejprve počítá tzv. *přirozenou šířku boxu*. Budeme ji označovat znakem w. Hodnotu w má šířka teoretického boxu, který je sestaven z daného horizontálního seznamu a ve kterém jsou všechny pružné výplňky typu $\langle glue \rangle$ počítány jako pevné pouze se základní velikostí výplňku.

Není-li stanoven požadavek na jinou šířku boxu než přirozenou, pak je kompletace boxu hotova. V tomto případě se totiž hodnota badness bere rovna nule, pružné

výplňky získaly hodnotu své základní velikosti a šířka kompletovaného boxu je rovna přirozené šířce boxu.

Složitější situace nastává v případě, kdy je stanoven požadavek na jinou šířku boxu, než je přirozená šířka. Takový požadavek je možné formulovat explicitně v ⟨*box specification*⟩ pomocí klíčových slov `spread` nebo `to`. Dále může požadavek na jinou šířku boxu vyplynout z vestavěných algoritmů:

- Při řádkovém zlomu odstavce mají řádky obvykle šířku `\hsize`.
- V `\halign` (tabulky) mají boxy položek šířku nejširší položky.

Než se pustíme do pokud možno přesného, ale suchého výkladu algoritmu na určení velikosti pružných výplňků, tak aby byl splněn požadavek na šířku boxu, učiníme dva kroky. Za prvé odkážeme čtenáře zpětně na stranu 77 v sekci 3.3, aby si zopakoval terminologii, používanou v souvislosti s pružnými výplňky typu ⟨*glue*⟩. Za druhé ilustrujeme problematiku na příkladě, z něhož snad vyplyne základní myšlenka, která vedla autora TEXu k zavedení pružných výplňků, a tedy k algoritmu, který popíšeme níže. Vyzkoušíme kód:

```
\tt \hsize=30pc
\hbox to\hsize{A\hskip 1cm plus0.5cm   B\hskip 2mm plus3cm  C}
\hbox to\hsize{D\hskip 1cm plus0.5fil  E\hskip 2mm plus3fil F}
\hbox to\hsize{G\hskip 1cm plus0.5fill H\hskip 2mm plus3fil I}
\hbox to1cm   {J\hskip 1cm plus0.5fill K\hskip 2mm plus3cm  L}
```

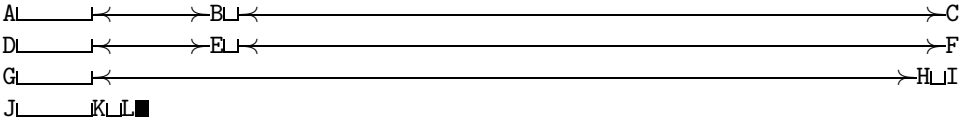
Na terminálu dostáváme toto varování:

```
Underfull \hbox (badness 3009) detected at line 174
\tentt A B C
Overfull \hbox (26.69035pt too wide) detected at line 177
\tentt J K L|
```

a na výstupu obdržíme:

```
A              B                                                C
D              E                                                F
G                                                             H I
J     K L■
```

Abychom to lépe prokoukli, vyznačíme si v místě pružného výplňku jeho základní velikost pomocí vaničky a jeho pružný rozměr pomocí šipky. Máme tedy na každém řádku první vaničku širokou 1 cm a druhou vaničku širokou 2 mm.



V každém řádku vystupují ve výplňcích dvě hodnoty roztažení. V prvním řádku se jedná o hodnoty 0,5 cm a 3 cm (nultého řádu). Poměr těchto rozměrů je 1:6. Proto také délky šipek na prvním řádku mají poměr 1:6 a těmito šipkami je vyplněn prostor boxu tak, aby celková šířka boxu byla `\hsize`. V druhém řádku je poměr první hodnoty roztažení ku druhé hodnotě rovněž 1:6. Rozdíl je pouze v tom, že tyto hodnoty jsou prvního řádu (`fil`). Délky šipek jsou zcela stejné. Ve třetím řádku má hodnota roztažení prvního výplňku řád 2 (`fill`) a hodnota druhého výplňku pouze řád 1 (`fil`). Proto je roztažení v druhém výplňku zcela potlačeno a pracuje jen roztažení vyššího řádu v prvním výplňku. Konečně v posledním řádku nelze požadavek na šířku boxu 1 cm splnit, protože přirozená šířka boxu je větší a ve výplňcích nefigurují žádné hodnoty stažení (`minus`). TEX tedy ohlásí `Overfull \hbox` a do sazby dokreslí černý obdélníček pomocí `\vrule width\overfullrule`, aby sazeč nepřehlédl havárii.

V prvním řádku vyšly nakonec velikosti šipek 3,1103 krát větší než odpovídá rozměrům 0,5 cm a 3 cm udaným v hodnotách roztažení ve výplňcích. Označme tento „koeficient roztažení“ písmenem k a spočítejme tzv. hodnotu *badness* podle vzorce:

$$b = 100\,k^3 = 3009$$

Protože je b větší než hodnota registru `\hbadness` (v plainu je nastavena na 1000), obdržíme na terminálu varování `Underfull \hbox (badness 3009)`. Hodnota badness (přeložili bychom jako chybovost, ale překládat nebudeme) vystihuje populárně řečeno „stupeň násilí, který byl na pružných výplňcích vykonán, aby se dosáhlo požadované šířky“. Pokud považujeme hodnoty roztažení nultého řádu ve výplňcích za maximální přípustnou mez a nastavíme `\hbadness=100`, budeme na terminálu vidět všechny případy, kdy došlo k překročení této meze. Roztahují-li se výplňky typu `fil(l(l))`, je hodnota badness nulová. □

Nyní přikročíme k obecnému popisu algoritmu na určení velikosti pružných výplňků, aby měl box požadovanou šířku. Algoritmus se může zdát poněkud nepřehledný, ovšem jednodušší formulaci se mi nepodařilo najít. Přitom v případě, že hodnoty roztažení či stažení výplňků jsou záporné, nám asi intuitivní představa nepomůže. Ve většině případů si ale s intuitivním názorem vystačíme. Pusťme se nyní do algoritmu:

1. Je-li kompletovaný horizontální seznam prázdný, vytvoří se prázdný box s požadovanou šířkou a badness je nulová. V takovém případě algoritmus končí.

2. Pokud je požadovaná šířka boxu w_g a přirozená šířka boxu w, pak se nejprve vypočítá tzv. hodnota *spread* podle vzorce $s = w_g - w$. Je-li dána hodnota s pomocí klíčového slova `spread`, nemusí se nic počítat.

3. Je-li $s > 0$, materiál v boxu bude potřeba roztáhnout. V takovém případě si TEX u jednotlivých výplňků všímá pouze hodnot roztažení a ignoruje hodnoty stažení. Je-li $s < 0$, materiál v boxu bude potřeba stlačit. TEX si všímá pouze hodnot stažení a ignoruje hodnoty roztažení výplňků. Pro stručnost zavedeme pojem hodnoty *deformace* výplňku, což při $s > 0$ bude znamenat hodnotu roztažení a při $s < 0$ hodnotu stažení výplňku. Je-li $s = 0$, není co dělat a algoritmus končí.

4. Nechť f_i pro $i = 0, 1, 2, 3$ označuje součty hodnot deformací i-tého řádu všech výplňků. Například máme v seznamu pět výplňků. Jejich hodnoty deformace nechť jsou postupně `1fill`, `0.2fil`, `1fil`, `-1fill` a `0.5mm`. Pak je $f_0 = 0{,}5\,\mathrm{mm}$, $f_1 = 1{,}2$ a $f_2 = f_3 = 0$. Všimneme si, že hodnota f_0 má rozměrovou jednotku, zatímco ostatní hodnoty jsou bezrozměrné.

5. Nechť r je takový řád, pro který je $f_r \neq 0$, a přitom všechny vyšší řády mají $f_i = 0$. Veškeré deformace se budou provádět jen s výplňky, které mají hodnotu deformace řádu r. Všechny ostatní výplňky budou mít pouze základní velikost bez deformace. V našem příkladě s pěti výplňky se provedou deformace na úrovni řádu 1, takže se týkají druhého a třetího výplňku. Pokud je $f_i = 0$ pro všechna i, pak se neprovedou změny na žádném výplňku a dále se algoritmus chová, jako by $r = -1$.

6. K základní velikosti jednotlivých výplňků, které mají hodnotu deformace h řádu r, se přičte rozměr $(s/f_r)\,h$. Tím se dosáhne zvětšení celkové šířky boxu o s (nebo zmenšení o $-s$), což bylo cílem celého algoritmu.

7. Pokud je $r \geq 1$, pak je hodnota badness nulová. Jedná se o situaci, kdy se deformují výplňky na úrovni některého řádu typu `fil(l(l))`. Při $r = 0$ a $f_0 > 0$ se hodnota badness počítá podle vzorce:

$$b = \min\left(100\,\left|\frac{s}{f_0}\right|^3, 10\,000\right)$$

Výjimkou je situace, kdy je $b > 100$ a $s < 0$, tj. snaha o stažení materiálu v boxu o více, než dovoluje maximální povolené stažení. Pak je hodnota badness rovna ∞ a dojde ke stažení s novou hodnotou spread $s_1 = -f_0$. TEX ohlásí `Overfull \hbox (`s_1 `- `s `too wide)`. Znamená to, že zabudovaný algoritmus nedovolí větší stažení, než odpovídá maximálnímu povolenému stažení. Na druhé straně, roztažení materiálu v boxu je možné ad absurdum a poznáme to jen podle varovného hlášení `Underfull \hbox` na terminálu, které se objeví vždy, když je badness v intervalu $\langle 100, 10\,000\rangle$ a současně je badness > `\hbadness`.

8. Při $r \leq 0$ a $f_0 \leq 0$ je badness rovno 10 000, resp. ∞, podle toho, zda $s > 0$, resp. $s < 0$. V případě $s < 0$ TEX ohlásí `Overfull \hbox (`$-f_0 - s$ `too wide)` a box nechá při $f_0 = 0$ s přirozenou šířkou. Při $f_0 < 0$ bude box dokonce o $-f_0$ širší.

9. Pokud se nepovedlo roztáhnout materiál v boxu na požadovanou šířku, tj. při `Overfull` nebo při $r = -1$, zůstává šířka obsahu boxu rozdílná od požadované šířky boxu. Šířka boxu se přesto nastaví na požadovanou. Obsah boxu je usazen tak, že se kryje referenční bod boxu s referenčním bodem prvního elementu uvnitř boxu. Zhruba řečeno, „konec“ horizontálního seznamu uvnitř boxu v tomto případě nemusí odpovídat „pravé hraně“ boxu. Při `Overfull` se navíc na konec obsahu boxu připojí *slimák* tvaru `\vrule width\overfullrule`.

10. Tisk zprávy `Overfull \hbox` je možno potlačit kladnou hodnotou `\hfuzz`. Přesahuje-li obsah boxu nejvýše o `\hfuzz`, přesněji je-li $-s - f_0 \leq$ `\hfuzz`, pak se hlášení neobjeví a „slimák“ se nepřipojí. □

Připojíme vysvětlení, jak číst zprávy TEXu v souboru `log`, které se týkají informací o boxech v souvislosti s nastavením pružných výplňků. Jedná se o zprávy, které se v `log`u vyskytnou po varovných hlášeních, případně po `\showlists` nebo `\showbox`. Zajímavá je pro nás fráze „`glue set` k“, která se vypíše hned vedle rozměrů boxu. Například:

```
\hbox(6.94444+0.0)x0.0, glue set - 31.66673fil
```

Hodnota k znamená koeficient s/f_r, který je použit pro výpočet velikostí výplňků v kroku 6 našeho algoritmu. Stojí-li za hodnotou k slovo `fil`, resp. `fill`, resp. `filll`, pak víme, že r je rovno 1, resp. 2, resp. 3. Není-li za hodnotou k žádné slovo, je $r = 0$. Je-li před hodnotou k znaménko minus oddělené od číselného vyjádření koeficientu mezerou, pak se provádí stažení boxu ($s < 0$), jinak se provádí roztažení ($s > 0$). Pozor, pokud je znaménko minus přímo navázáno na číselné vyjádření koeficientu, pak to znamená, že je samotné f_r záporné. Čtenář si může rozmyslet, jakým způsobem jsme dosáhli tohoto výpisu: `glue set - -2.0fil`. □

• **Vertikální box.** Pusťme se nyní do kompletace vertikálních boxů. Situace je velmi podobná, ovšem v některých ohledech se algoritmy liší. Nejprve budeme sestavovat `\vbox` a až později `\vtop` a `\vcenter`.

Šířka `\vbox`u se spočítá jako maximum šířek jednotlivých boxů a linek uvnitř vertikálního seznamu. U boxů posunutých v horizontálním směru pomocí `\moveright` nebo `\moveleft` se při výpočtu maxima k jejich šířkám přičítají, resp. odečítají, velikosti jejich posunu doprava, resp. doleva. To je v souladu s představou, že šířka `\vbox`u se bude počítat podle „pravého okraje nejvíce vystrčeného boxu směrem doprava“. Znamená to tedy, že boxy posunuté doprava jsou vždy uvnitř obvodu výsledného `\vbox`u. Naproti tomu boxy posunuté doleva vyčnívají z obvodu výsledného boxu o hodnotu posunu.

Není-li v kompletovaném vertikálním seznamu žádný box ani linka, případně maximum šířek podle výše zmíněného algoritmu vychází záporné, položí se šířka \vboxu rovna nule.

Jakmile je stanovena šířka kompletovaného boxu, lze určit rozměry pro linky typu \hrule. Všechny tyto linky s neurčitou šířkou budou mít šířku kompletovaného boxu. Mají-li tyto linky neurčitou výšku, pak je stanovena na implicitní hodnotu 0,4 pt a neurčitá hloubka se nahradí hodnotou 0 pt.

K určení hloubky kompletovaného \vboxu se TEX podívá na poslední linku nebo box ve vertikálním seznamu. Pokud za touto linkou nebo boxem nenásleduje výplněk typu ⟨*glue*⟩ nebo \kern, je hloubka kompletovaného boxu rovna hloubce této poslední linky nebo boxu v seznamu. Při vyhodnocování této podmínky se ignorují všechny ostatní objekty vertikálního seznamu (\penalty, \mark apod.). Pokud podmínka není splněna, případně v kompletovaném seznamu není ani jedna linka či box, je hloubka kompletovaného boxu 0 pt.

Hloubka boxu d vyhodnocená výše zmíněným způsobem se může dále změnit podle hodnoty \boxmaxdepth. Je-li $d >$ \boxmaxdepth, pak je hloubka \vboxu s konečnou platností stanovena na \boxmaxdepth. V plainu je nastaveno \boxmaxdepth=\maxdimen, takže se nikdy hloubka boxu tímto způsobem neupravuje. Pouze ve výstupní rutině je \boxmaxdepth=\maxdepth.

Přirozená výška \vboxu se stanoví jako součet všech výšek a hloubek boxů a linek v kompletovaném seznamu plus součet všech velikostí výplňků typu \kern a základních velikostí výplňků typu ⟨*glue*⟩. Od této hodnoty se odečte hloubka boxu d, vypočítaná podle algoritmu zmíněného výše. Je tedy vždy zaručeno, aby přirozená výška plus hloubka kompletovaného boxu odpovídala „celkové výšce vertikálního seznamu“, který je kompletován. Této hodnotě říkáme *celková výška* boxu.

Není-li stanovena požadovaná výška \vboxu pomocí klíčového slova `spread` nebo `to`, případně při sestavování položek u \valign, pak výška \vboxu je shodná s přirozenou výškou boxu a badness je rovna nule.

Je-li stanovena požadovaná výška \vboxu, pak se provede analogický algoritmus na určení velikosti pružných výplňků a výpočtu hodnoty \badness, jaký byl uveden výše pro \hbox. Můžete si tento algoritmus přečíst znova, pouze některá slova v textu nahraďte alternativami:

\hbox	⟶	\vbox
šířka	⟶	výška
horizontální seznam	⟶	vertikální seznam
\hbadness	⟶	\vbadness
\hfuzz	⟶	\vfuzz
`too wide`	⟶	`too high`

Další rozdílností mezi algoritmem pro `\hbox` a `\vbox` je skutečnost, že se při `Overfull` nikdy nepřipojuje žádný slimák. Konečně je potřeba přeformulovat jednu větu v pravidle 9: Obsah boxu se usadí tak, že „horní okraj“ kompletovaného boxu se kryje s „horním okrajem“ prvního elementu uvnitř boxu. Dost špatně se to formalizuje, ale je myslím zřejmé, co jsem chtěl říci. □

Nyní kompletujeme `\vtop` a `\vcenter` a tím budeme mít tuto sekci o boxech kompletní. Nejprve uvedeme hrubou ideu. Protože účaří `\vbox`u se (obvykle) kryje s účařím posledního řádku, budou mít dva `\vbox`y vedle sebe ve stejné výšce poslední řádky. My bychom někdy chtěli, aby byly ve stejné výšce řádky první. Například při sazbě do dvou nestejně zaplněných sloupců, které klademe vedle sebe. V takovém případě se hodí `\vtop`, což je box, jehož účaří se (obvykle) kryje s účařím prvního řádku. Jeho výška je tedy shodná s výškou prvního řádku a hloubka zahrnuje zbytek vertikálního materiálu. V matematické sazbě zase může být užitečné sázet některé objekty na matematickou osu (například matice). V takovém případě použijeme `\vcenter`.

`\vtop` i `\vcenter` se nejprve kompletují jako `\vbox` se vším všudy, tj. včetně určení velikosti pružných výplňků a stanovení hodnoty badness. V závěrečné fázi se přehodnotí výška a hloubka boxu tak, aby (1) součet výšky a hloubky zůstal zachován. (2) při `\vtop` je výška rovna výšce prvního boxu nebo linky v seznamu. Začíná-li seznam něčím jiným, je výška nulová. (3) při `\vcenter` bude výška boxu nad matematickou osou rovna hloubce boxu pod matematickou osou. Rozdíl skutečné výšky od hloubky boxu (vztaženy k účaří) bude tedy roven dvojnásobku vzdálenosti matematické osy od účaří. Tato vzdálenost je určena hodnotou `\fontdimen22` fontu rodiny 2 (viz stranu 181). Při přehodnocení výšky a hloubky boxu pro `\vtop` a `\vcenter` se ignoruje hodnota `\boxmaxdepth`. □

3.6. Mezery v horizontálním seznamu

Mezera v horizontálním seznamu způsobí posun aktuálního bodu sazby doprava o hodnotu mezery. Je-li hodnota mezery záporná, pak se aktuální bod sazby posune doleva, takže následující element může překrývat element předchozí, nebo dokonce může stát vlevo od předchozího elementu.

Mezery jsou dvojího druhu: *Kern* je pevná mezera, ve které (obvykle) není možno provést řádkový zlom a ⟨*glue*⟩ může být pružná mezera, ve které (obvykle) je možno provést řádkový zlom. Mezery typu kern se do horizontálního seznamu vkládají za těchto okolností:

- Implicitně, podle tabulky kerningových párů použitého fontu
- Explicitně, vložením mezery povelem `\kern` nebo `\/`.

Každá dvojice znaků ze společného fontu může mít v tabulce kerningových párů fontu nějakou hodnotu, která se mezi znaky vloží v podobě *implicitního kernu* automaticky. Klasický příklad „VA“ zahrnuje obvykle záporný kern mezi V a A. Můžete si ověřit:

```
178 VA\showlists
```

zapíše do `log` souboru:

```
### horizontal mode entered at line 178
\hbox(0.0+0.0)x20.0
\tenrm V
\kern-1.11113
\tenrm A
spacefactor 999
```

Nejdříve vidíme box odstavcové zarážky, pak písmeno V, pak implicitní kern a konečně A. Dále ještě následuje slovo `spacefactor`, kterým se budeme zabývat v této sekci později.

Ukážeme si makro, které zjistí hodnotu implicitního kernu. Po použití makra `\showkern AB` dostaneme na terminálu zprávu `-1.11113pt`.

```
179 \newdimen\kernamount
180 \def\showkern #1#2{\setbox0=\hbox{#1#2}\kernamount=\wd0
181   \setbox1=\hbox{#1}\setbox2=\hbox{#2}%
182   \advance\kernamount by-\wd1 \advance\kernamount by-\wd2
183   \message{\the\kernamount}}
```

Poznamenejme, že ve fontu jsou nejen údaje o implicitních kernech, ale též o ligaturách. Dvojice nebo větší skupina znaků se může automaticky proměnit v jediný znak ve fontu. Vyzkoušejte si: `---\showlists`. Podrobněji o ligaturách, viz stranu 305 v sekci 7.3.

Pokud se mezi vložením dvojice znaků do horizontálního seznamu vykoná jakýkoli jiný povel hlavního procesoru, implicitní kern se nevloží a ligatura se nevytvoří. Například při `A{}V`, `A\relax V`, `A\spacefactor=500V` se implicitní kern nevloží. Při `f{i}` se ligatura primárně nevytvoří. Projde-li ale seznam druhým průchodem řádkového zlomu, pak se i `f{i}` může spojit v ligaturu (srovnej poznámku o ligaturách na straně 220). Pišme proto raději: „`f\/i`“. □

Mezera typu ⟨*glue*⟩ se do horizontálního seznamu vloží za těchto okolností:

- Mezislovní mezera, povel $\boxed{\sqcup}_{10}$.
- Explicitní mezera, povel `\␣`.

- Mezera vyjádřená povelem `\hskip`.
- Mezera vložená pomocí primitivů `\hss`, `\hfil`, `\hfill` a `\hfilneg`.

Víme, že mezera typu ⟨*glue*⟩ obsahuje tři komponenty: základní velikost, hodnotu roztažení a stažení. Nyní se budeme zabývat tím, jak tyto komponenty v jednotlivých případech vypadají.

Nejjednodušší to je v případě použití `\hskip`. Tam je přesně řečeno, jak jednotlivé komponenty mezery typu ⟨*glue*⟩ vypadají. Stejně tak zkratky `\hss`, `\hfil` a další mají jednoznačný význam. Zkracují pouze konkrétní zápis pro `\hskip` a všechny jsou jednotlivě vyloženy v části B. □

Poněkud více práce nám dá výklad algoritmu pro výpočet mezislovních mezer vyprodukovaných pomocí $\boxed{\sqcup}_{10}$. Hodnoty pro tyto mezery se berou z parametrů `\fontdimen` zrovna použitého fontu. Každý textový font v TEXu musí mít aspoň sedm těchto parametrů. Zde je jejich přehled:

- `\fontdimen1` — sklon písma.
- `\fontdimen2` — základní velikost mezislovní mezery.
- `\fontdimen3` — hodnota roztažení mezislovní mezery.
- `\fontdimen4` — hodnota stažení mezislovní mezery.
- `\fontdimen5` — výška písmene x.
- `\fontdimen6` — stupeň kuželky, tj. velikost písma (např. 10 pt).
- `\fontdimen7` — dodatečná velikost mezislovní mezery.

Přístup k těmto parametrům fontu realizujeme v makrech tak, že za číslo parametru připojíme název fontu. Například `\fontdimen2\tenbf` znamená základní velikost mezislovní mezery pro font zavedený pod názvem `\tenbf`. Nebo třeba `\fontdimen6\font` je velikost zrovna použitého fontu. Pro mezislovní mezery jsou důležité hodnoty `\fontdimen` 2, 3, 4 a 7. □

Abychom vyložili způsob výpočtu hodnoty mezislovních mezer, je nutné se chvíli zabývat registrem `\spacefactor`. Jeho hodnota ovlivní dodatečnou deformaci mezislovní mezery. Naopak, hodnota tohoto registru se průběžně mění podle tzv. `\sfcode` jednotlivých znaků, které jsou vkládány do horizontálního seznamu.

Každý znak má svůj `\sfcode`, což je celé nezáporné číslo. Při inicializaci v iniTEXu je výchozí `\sfcode` každého znaku 1000 s výjimkou velkých písmen `A`–`Z`, která mají `\sfcode` 999. Tento kód je možno měnit pomocí primitivu `\sfcode`. Například `\sfcode‘.=3000`. Viz makra plainu `\frenchspacing` a `\nonfrenchspacing`.

TEX při vkládání jednotlivých elementů do horizontálního seznamu průběžně mění hodnotu pracovního registru `\spacefactor` takto: Na začátku horizontálního seznamu má registr `\spacefactor` hodnotu 1000. To označuje (jak uvidíme později) žádnou deformaci mezery. Po vložení znaku přijme tento registr hodnotu

\sfcode právě vloženého znaku. Toto pravidlo má výjimku, kterou uvedeme za chvíli. Při vložení boxu nebo linky se \spacefactor vrací na hodnotu 1000. Při vložení jiného elementu (mezera, kern, penalta) se registr nemění. Pomocí přiřazení \spacefactor=⟨*number*⟩ je možno pro daný okamžik nastavit hodnotu registru manuálně. TEX se ovšem brání nastavit tomuto registru nulovou nebo zápornou hodnotu.

Nyní vyjmenujeme výjimky při změnách registru \spacefactor podle hodnoty \sfcode právě vloženého znaku. Označme \sfcode vloženého znaku písmenem s a \spacefactor písmenem f. Je-li $s = 0$, f zůstane nezměněno. Je-li $f < 1000 < s$, nastaví se $f := 1000$. Jinak se nastaví $f := s$. Lidově řečeno, hodnota f po vložení znaku nemůže prudce změnit svou hodnotu z menší než tisíc na větší než tisíc. Teprve po dvou znacích s kódem $s > 1000$ se hodnota f přizpůsobí.

Při vložení mezislovní mezery povelem ␣10 se vypočítá velikost této mezery z \fontdimen 2, 3, 4 a 7 a současné hodnoty f registru \spacefactor. Uvažujme nejprve $f = 1000$. V tomto případě se vloží mezera, kterou bychom mohli zapsat pomocí:

```
184 \hskip\fontdimen2\font plus\fontdimen3\font minus\fontdimen4\font
```

Při $f \neq 1000$ se navíc mění hodnoty roztažení a stažení. Hodnota roztažení je násobena koeficientem $f/1000$ a hodnota stažení koeficientem $1000/f$. Základní velikost mezery zůstává pro $f < 2000$ konstantní. Při $f \geq 2000$ je navíc k základní velikosti mezery přičtena hodnota \fontdimen7. □

Uvedeme si příklad. Nechť mají (pro jednoduchost) všechny parametry \fontdimen hodnotu 10 pt. Při \spacefactor=1000 se vloží mezera 10 pt plus 10 pt minus 10 pt. Při \spacefactor=600 se vloží mezera 10 pt plus 6 pt minus 16,666 pt. Při \spacefactor=1500 vznikne mezera 10 pt plus 15 pt minus 6,666 pt. Konečně při \spacefactor=3000 se mění skokem i základní velikost a máme 20 pt plus 30 pt minus 3,333 pt.

Plain nastavuje \sfcode tečky, otazníku a vykřičníku na 3000 a čárky na 1250. Dále parametry \fontdimen mají pro font cmr10 rozměry 3,33333 pt (základní velikost), 1,66666 pt (hodnota roztažení), 1,11111 pt (hodnota stažení) a 1,11111 pt (dodatečná mezera). Vidíme tedy, že základní velikost mezery za tečkou, vykřičníkem a otazníkem je zvětšena o 1,11111 pt a navíc se mezera ochotněji roztahuje a méně ochotně stahuje. Mezera za čárkou nemění svou základní velikost, ale co se týče ochoty ke stahování a roztahování, chování je podobné, jako u tečky. Tyto hodnoty \sfcode odpovídají tradicím americké sazby, kde se za větou vkládala vždy větší mezera. Pro českou sazbu je to naprosto nevyhovující, a proto voláme makro \frenchspacing (většinou prostřednictvím makra \chyph), které nastavuje kódy pro interpunkci na 1000. Ještě lepší nastavení ukážeme v závěru sekce.

Při hodnotách \sfcode podle americké sazby vyjde najevo, proč jsou \sfcode velkých písmen 999. Tato hodnota je „skoro“ 1000, takže rozdíly ve velikostech mezer nejsou vidět. Ale při zkratkách (například D. E. Knuth) nezvedne zapsaná tečka \spacefactor hned na 3000, protože algoritmus nedovolí prudkou změnu přes hranici 1000. Za tečkou ve zkratce tedy máme \spacefactor roven 1000, a vysází se proto normální a nikoli prodloužená mezera. □

Jednotlivým \fontdimen lze sice přiřadit makrem jinou hodnotu, než je výchozí údaj z metriky fontu, ale toto přiřazení je globální. Proto se změna těchto parametrů pro lokální účely nedoporučuje. Místo toho použijeme registry typu ⟨*glue*⟩ s názvem \spaceskip a \xspaceskip. Tyto registry mají přednost před \fontdimen, pokud mají aspoň jednu komponentu nenulovou: Je-li $f < 2000$ a \spaceskip je nenulová, vloží se tato mezera s pronásobením hodnot roztažení, resp. stažení, koeficientem $f/1000$, resp. $1000/f$. Při $f \geq 2000$ a nenulové \xspaceskip se použije tato mezera.

Registry \spaceskip a \xspaceskip jsou použity v makrech pro sazbu na praporek. Tehdy totiž chceme, aby hodnoty stažení a roztažení mezislovních mezer byly nulové. Viz makro \raggedright. □

• **Příklady.** V pravidlech pro českou sazbu se doporučuje naopak vkládat za tečku a čárku zúžený výplněk. Pokud má například běžný mezislovní výplněk velikost 3,3 pt, pak zúžený výplněk má velikost 2 pt. Zúžený výplněk také vkládáme před fyzikální jednotky (zde třeba před jednotku pt) a na mnoho dalších míst. Proč existuje toto doporučení? Tečka nebo čárka samotná nevyplňuje opticky celé své místo a následující mezera se subjektivně jeví jako větší, což narušuje plynulost sazby. Staří mistři sazeči proto zužovali mezery za tečkami a čárkami. V tomto odstavci je ukázáno, jak takové zúžení vypadá. Pokud bychom chtěli jít ve šlépějích našich sazečských předků, volme (například pro font `csr10`) hodnoty:

```
\fontdimen7\tenrm=-1.11111pt
\chyph \sfcode`.=2000 \sfcode`,=2000
\newcount\num \num=`\A
\loop \sfcode\num=1000 \ifnum\num<`\Z \advance\num by1 \repeat
```

Při těchto hodnotách budeme mít mezery za tečkami a čárkami zhruba dvoubodové, přitom mezery se nebudou ochotně stlačovat (už jsou tak dost malé), ale budou se ochotně roztahovat. To je přesně to, co staří sazeči měli na mysli. □

Explicitní mezera vložená pomocí \␣ je shodná s mezislovní mezerou, jako by \spacefactor byl roven 1000. Vložení této mezery přitom nemění skutečnou hodnotu registru \spacefactor. □

Na závěr si ukážeme makro, které nám umožní vytvářet „prostrkávanou sazbu“. Tento způsob vyznačování se dnes

již moc nepoužívá. Podstatně vhodnější je vyznačovat kurzívou nebo polotučným řezem. Přesto se může stát, že takové makro budeme k něčemu potřebovat.

```
\let\lqq=^^fe \let\rqq=^^ff \def\pomlka{---}
\newdimen\proklad \proklad=2pt \newdimen\mm
\def\prostrkej#1{\bgroup\let\pp=\relax\let\next=\strk \strk#1^^X}
\def\strk{\afterassignment\osetriznak \let\znak= }
\def\osetriznak{%
  \if \noexpand\znak^^X\let\next=\konec
  \else \if\noexpand\znak\space \space\kern\proklad\let\pp=\relax
        \else \ifcat \noexpand\znak A\prokladznaku
              \else \ifcat \noexpand\znak .\prokladznaku
                    \else \znak % makra, hranice skupin apod.
  \fi \fi \fi \fi \next}
\def\prokladznaku{\setbox0=\hbox{\pp\znak}%
    \setbox1=\hbox{\pp}\setbox2=\hbox{\znak}%
    \mm=\wd0 \advance\mm-\wd1 \advance\mm-\wd2
    \advance\mm\proklad
    \ifhmode \kern\mm \fi \znak
    \global\let\pp=\znak }
\def\konec{\kern\proklad \egroup}
```

Uživatel napíše `\prostrkej{zvýrazněný text}` a po zpracování dostane na výstupu z v ý r a z n ě n ý t e x t . Makro postupně odebírá pomocí `\let\znak` token za tokenem (rozpozná i mezeru) a v `\osetriznak` zjistí, zda jde o token kategorie 11 nebo 12. Pokud ano, provede `\prokladznaku`, tj. sazbu `\kern\mm` následovanou znakem. Přitom `\mm` je veliké jako `\proklad`, ovšem upravené o hodnotu implicitního kernu s předchozím znakem. Uživatel může měnit hodnotu registru `\proklad`, která je implicitně nastavena na 2 pt. Při prokladu třeba 10 pt bude mít v e l m i p r o s t r k a n ý t e x t .

Makro jsme se snažili uvést v co nejjednodušší podobě, a proto má své slabiny. Ne vše je v parametru makra `\prostrkej` povoleno. Přepínače fontů a přechody do skupin fungují. Například `\prostrkej{cosi {\bf tučně}}` bude bez problémů. Nelze ovšem použít žádný povel hlavního procesoru ani makro, které by mělo parametr. Třeba `\prostrkej{\uv{text}}` havaruje. Proto jsou v úvodu zavedeny sekvence `\lqq` a `\rqq`, aby se uvozovky daly psát aspoň jako `\prostrkej{\lqq text\rqq}`. Z podobných důvodů použijeme v prostrkané sazbě místo přímého zápisu pomlčky makro `\pomlka`.

Použití makra zcela potlačuje dělení slov. Pokud bychom chtěli používat prostrkávanou sazbu i s dělením slov a často, bude nutné si k tomu účelu připravit virtuální font. □

3.7. Mezery ve vertikálním seznamu

Mezery mezi boxy ve vertikálním seznamu měříme od spodní hrany prvního boxu po horní hranu druhého. Obvykle je v sazbě požadováno, aby jednotlivá účaří boxů měla ve vertikálním seznamu jednotnou vzdálenost (tj. abychom měli stejnoměrné řádkování). V ideálním případě jsou řádky na každé straně umístěny do zcela stejné sítě účaří, které říkáme *řádková osnova*. Typografové též hovoří o *řádkovém rejstříku*. Protože jednotlivé boxy mají různé výšky a hloubky, znamená to, že pro zachování požadavku na dodržení řádkového rejstříku bude potřeba mezi boxy vkládat různě velké mezery.

Mezi každé dva boxy ve vertikálním seznamu je automaticky vložena mezera typu ⟨*glue*⟩, jejíž velikost je (A) počítána z hodnoty `\baselineskip`, nebo (B) je přímo rovna hodnotě `\lineskip`. Případ (A) způsobí vložení mezery, jejíž velikost je závislá na hloubce prvního boxu a výšce druhého tak, aby vzdálenost účaří byla rovna základní velikosti z `\baselineskip`. Případ (B) vkládá mezi boxy pevně definovanou mezeru, což při různých výškách a hloubkách boxů vede k nestejnoměrnému řádkování.

Mezi alternativami (A) a (B) rozhoduje algoritmus na automatické vložení mezery prostřednictvím registru `\lineskiplimit` takto: Nejprve se spočítá teoretická velikost mezery, jako by se jednalo o případ (A). Je-li tato velikost větší nebo rovna `\lineskiplimit`, pak se použije případ (A), jinak se použije případ (B).

Nechť například `\lineskiplimit=0pt`, `\baselineskip=12pt` a `\lineskip=1pt`. Pokud se boxy při dodržení vzdálenosti účaří 12 pt „nedotknou“, pak bude vložena mezera tak, aby vzdálenost účaří byla skutečně 12 pt. Pokud by se ale měly překrývat, začne pracovat alternativa (B) a ve skutečnosti se překrývat nebudou. Naopak, ještě se mezi ně vloží nepatrná mezera velikosti 1 pt a bude „rozhozeno“ řádkování. Přesně toto nastavení odpovídá výchozím hodnotám zmíněných tří registrů v plainu. □

Jestliže trváme na zachování řádkování i za cenu toho, že se nám jednotlivé boxy budou (při výjimečných velikostech) překrývat, volme pro `\lineskiplimit` dostatečnou zápornou hodnotu. Například `\lineskiplimit=-\maxdimen` nám zaručí, že nikdy nebude použita alternativa (B) a bude tedy zachováno řádkování za jakýchkoli okolností.

Na tomto místě bude užitečné připomenout, že $\mathcal{CS}$-fonty ve velikosti 10 pt nebyly vhodné pro standardní nastavení `\baselineskip=12pt` a `\lineskiplimit=0pt`. Často docházelo k rozhození řádkování. Akcentované verzálky měly totiž nastavenu výšku boxu o 1,2 pt větší, než by odpovídalo jejich kresbě. V takovém případě bylo nutné nastavit záporný `\lineskiplimit`. Teprve ve verzi $\mathcal{CS}$-fontů z října 1996 je tato chyba opravena.

Nastavení záporné hodnoty \lineskiplimit (i pro bezchybné fonty) je vždy určitým kompromisem mezi požadavkem na stejnoměrné řádkování a požadavkem na to, aby se kresby písmen z jednotlivých řádků nepřekrývaly. Zde je vidět nevýhoda TEXu, který nelokalizuje v jednotlivých řádcích přesně místo s nejvyšším, resp. nejnižším, elementem, ale přiřadí celému boxu rozměry nejvyššího a nejnižšího elementu. Velmi pravděpodobně se ale na řádcích pod sebou nejvíce vystrčené elementy nesetkají. Pokud ovšem v matematickém článku použije autor uvnitř textu „větší“ vzorec (např. $\int_0^{t^2} f(x)\,\mathrm{d}x$), pak je práce alternativy (B) přímo nutností. □

V tabulkách, ve kterých sázíme mezi sloupečky vertikální linky, je potřeba volat makro \offinterlineskip, které je definováno takto:

```
\def\offinterlineskip{\baselineskip=-1000pt
  \lineskip=0pt \lineskiplimit=\maxdimen}
```

V tomto případě naopak bude vždy pracovat alternativa (B) a díky nulové hodnotě \lineskip se jednotlivé řádky budou „opírat“ jeden o druhý. To je pro navazování jednotlivých vertikálních linek z řádků žádoucí. Aby měly jednotlivé řádky stejně vzdálené účaří, je potřeba v makru tabulky vložit do každého řádku neviditelnou podpěru konstantní výšky a hloubky, přičemž tyto hodnoty budou větší nejvýše rovny největší možné výšce, resp. hloubce, řádku bez této podpěry. □

Registry \baselineskip a \lineskip jsou typu ⟨*glue*⟩ a \lineskiplimit je typu ⟨*dimen*⟩. Znamená to, že mezi jednotlivými boxy ve vertikálním seznamu mohou pracovat též hodnoty stažení a roztažení. Při rozhodování mezi alternativou (A) a (B) tyto hodnoty nemají vliv, ovšem po výpočtu základní velikosti ⟨*glue*⟩ podle (A), resp. (B), se připojí případná hodnota stažení a roztažení podle \baselineskip, resp. \lineskip.

Uvedeme si neužitečný, ale ilustrativní příklad. Nechť je:

```
\baselineskip=20pt plus 50pt \lineskip=-10pt minus 5pt
\lineskiplimit=0pt
\def\X{\hbox{\vrule height10pt depth10pt}}% výška a hloubka 10
\def\Y{\hbox{\vrule height11pt depth0pt}}% výška 11, hloubka 0
\X \X \Y \Y
\showboxbreadth=20 \showlists
```

V souboru `log` máme tento výsledek:

```
\hbox(10.0+10.0)x0.4
\glue(\baselineskip) 0.0 plus 50.0
\hbox(10.0+10.0)x0.4
\glue(\lineskip) -10.0 minus 5.0
\hbox(11.0+0.0)x0.4
```

```
\glue(\baselineskip) 9.0 plus 50.0
\hbox(11.0+0.0)x0.4
```

Vidíme tedy, že mezi prvním a druhým boxem (oba jsou `\X`) je vložena mezera počítaná z `\baselineskip` podle (A). Mezera má hodnotu 0 pt plus 50 pt. Základní vzdálenost účaří prvních dvou boxů je skutečně 20 pt. Dále mezi `\X` a `\Y` je vložena mezera podle `\lineskip`, protože při požadavku na zachování vzdálenosti účaří 20 pt nám vychází vzdálenost boxů −1 pt a to je méně, než dovoluje `\lineskiplimit`. Konečně mezi posledními boxy (oba jsou `\Y`) je podle alternativy (A) vypočítána mezera 9 pt plus 50 pt z `\baselineskip`. □

Mezerám, které se vloží mezi boxy automaticky podle výše uvedeného algoritmu, budeme v dalším textu říkat *meziřádkové mezery*. Je-li mezi boxy vloženo libovolné množství jiného vertikálního materiálu (s výjimkou linky), je přesto vložena meziřádková mezera. Při výpočtu její velikosti se jiný vertikální materiál mezi boxy nebere v úvahu a meziřádková mezera je vložena těsně před nově vložený box. Tato vlastnost je vnitřně implementována tak, že po vložení boxu se uloží do registru `\prevdepth` hloubka tohoto boxu. Pak se vkládají další elementy, až přijde na řadu znovu box. U něj se změří jeho výška, vezme se hloubka z `\prevdepth` a na základě těchto údajů se vloží meziřádková mezera. Podrobněji, viz heslo `\prevdepth` v části B.

Například změníme na řádku 213 vzdálenost mezi boxy takto:

```
215 \X \kern2pt \vskip8pt \X \nobreak\vskip10pt \Y \kern-20pt \Y
```

a v souboru `log` si můžeme přečíst, že hodnoty meziřádkových mezer, které se přidávají k explicitně vyjádřeným mezerám, zůstaly nezměněny:

```
\hbox(10.0+10.0)x0.4
\kern 2.0
\glue 8.0
\glue(\baselineskip) 0.0 plus 50.0
\hbox(10.0+10.0)x0.4
\penalty 10000
\glue 10.0
\glue(\lineskip) -10.0 minus 5.0
\hbox(11.0+0.0)x0.4
\kern -20.0
\glue(\baselineskip) 9.0 plus 50.0
\hbox(11.0+0.0)x0.4
```

Tato vlastnost nám umožní dodržet rovnoměrné řádkování i v případě, že vkládáme explicitní vertikální mezery. Přitom se nemusíme starat o výšky a hloubky boxů. Například při řádkování 12 pt můžeme „vynechat prázdný řádek“ pomocí

`\vskip12pt`. Aktuální výšku a hloubku řádků koriguje meziřádková mezera, vložená těsně před další řádek. Proto bude při použití tohoto `\vskip` a při činnosti alternativy (A) vzdálenost mezi účařími skutečně 24 pt.

Jiná situace nastává při vložení linky. Nalézá-li se mezi dvěma sousedícími boxy ve vertikálním seznamu aspoň jedna linka, je algoritmus vkládání meziřádkové mezery zcela potlačen. Jednoduše můžeme říci, že za linkou nebude nikdy meziřádková mezera. Linka se přitom nechová jako box, takže meziřádková mezera není ani těsně před linkou. Zkusíme si znovu náš příklad, tentokrát zapsaný takto:

```
216 \X \hrule\kern2pt\hrule\vskip8pt \X \vskip10pt\hrule \Y \Y
```

V souboru `log` zjistíme, že skutečně dvě ze tří meziřádkových mezer jsou potlačeny:

```
\hbox(10.0+10.0)x0.4
\rule(0.4+0.0)x*
\kern 2.0
\rule(0.4+0.0)x*
\glue 8.0
\hbox(10.0+10.0)x0.4
\glue 10.0
\rule(0.4+0.0)x*
\hbox(11.0+0.0)x0.4
\glue(\baselineskip) 9.0 plus 50.0
\hbox(11.0+0.0)x0.4
```

Pokud chceme vložit linku a zachovat přitom řádkování, můžeme vykreslit linku pomocí `\leaders` třeba takto:

```
217 První řádek.
218 \vskip6pt \leaders\hrule\vskip.4pt \vskip5.6pt
219 Druhý řádek.
```

Protože `\leaders` je typu ⟨*glue*⟩, a nikoli typu linka, bude skutečně před druhý řádek vložena správná meziřádková mezera. Uvedená konstrukce má ovšem několik nevýhod: Poloha linky není jednoznačná vzhledem k řádkové osnově (řádkovému rejstříku), ale závisí na hloubce předchozího boxu. Také při stránkovém zlomu nám může linka zcela zmizet. Lepší řešení, kdy linka bude například ležet přesně na účaří vynechaného řádku a nemizí ve stránkovém zlomu, vede přes vložení horizontálního boxu s `\leaders`:

```
220 První řádek.
221 \par \noindent \hrulefill\null\par
222 Druhý řádek.
```

Kdybychom nevložili těsně před \par prázdný box \null, rutiny pro ukončení odstavce by zlikvidovaly \hrulefill, což nechceme. Potřebujeme-li linku posunout jinam, než na účaří, máme více možností. Jednotlivé řádky v ukázce znázorňují různá řešení stejného problému:

```
223 \par\noindent\leaders\vrule height4.4pt depth-4pt\hfill\null\par
224 \vskip-4pt \noindent \hrulefill\null\vskip4pt
225 \par \line{\raise4pt\line{\hrulefill}}
```

□

• **Mezera podle \topskip.** Nyní se pustíme do výkladu algoritmů, které vkládají další mezery do vertikálního seznamu. Pro vložení počáteční mezery na stránce je použit registr \topskip. Algoritmus se stará o to, aby na každé stránce byla zahájena řádková osnova ve stejné výšce vzhledem k hornímu okraji boxu stránky. Výpočet velikosti počáteční mezery závisí na výšce prvního boxu nebo linky na stránce tak, aby účaří prvního boxu nebo linky bylo vzdáleno od horního okraje o základní velikost z \topskip. Toto pravidlo neplatí, je-li výška prvního boxu nebo linky větší než základní velikost z \topskip. V takovém případě se vloží mezera 0 pt. V obou případech se dále přidá hodnota stažení nebo roztažení z \topskip beze změny.

Například, je-li \topskip=10pt (hodnota z plainu), pak při výšce prvního boxu 8 pt se před něj vloží z \topskip mezera velikosti 2 pt. Při výšce prvního boxu 11 pt se ovšem vloží mezera 0 pt.

Mezera z \topskip se vkládá automaticky vždy před první box nebo linku na stránce. To souvisí s algoritmem plnění strany a stránkového zlomu, který je vyložen v sekci 6.6. □

Všimněme si, že mezi mezerou z \baselineskip a \topskip najdeme rysy podobné, ovšem též rozdílné: V obou případech závisí základní hodnota mezery nikoli jen na základní hodnotě registru, ale svou roli hraje výška vkládaného boxu (případně u \baselineskip ještě hloubka předchozího). Na druhé straně nenajdeme alternativu typu \topskiplimit. Efekt záporného \lineskiplimit řešíme tak, že volíme \topskip dostatečně velké a potom při usazení výsledného stránkového boxu ve výstupní rutině se postaráme o kompenzování volného prostoru.

Nejdůležitější odlišností je jiné chování linek. Zatímco meziřádkové mezery se nikdy před linky nepřidávají, u mezery z \topskip může tato mezera předcházet před linkou podle stejného pravidla, jako předchází před boxem. □

Věnujme nyní pozornost makru z plainu \vglue, které vloží mezeru stejně jako \vskip, ale tato mezera nikdy „nezmizí“ ve stránkovém zlomu. Knuth pro tyto účely vložil neviditelnou linku s nulovou výškou, za kterou připojil \nobreak a požadovaný \vskip. Tím má zaručeno, že mezera nikdy nepodlehne stránkovému zlomu. Protože ale linka způsobí potlačení meziřádkové mezery, pracuje makro \vglue

s parametrem \prevdepth. Po vložení linky se restauruje hodnota \prevdepth z doby před vložením linky. Toto je způsob, jak znovu obnovit možnost vložení meziřádkové mezery, ačkoli byla vložena linka. Makro vypadá následovně:

```
\def\vglue{\afterassignment\vgl@\skip0=}
\def\vgl@{\par \dimen0=\prevdepth \hrule height0pt
  \nobreak\vskip\skip0 \prevdepth=\dimen0 }
```

Pokud je \vglue použito jako první element na stránce, je před neviditelnou linku vloženo \topskip. Fyzická mezera pak je o \topskip větší, než si přeje uživatel. Proto je v plainu ještě makro \topglue, které pomocí \vglue-\topskip tento rozdíl kompenzuje. Je potřeba si ale uvědomit, že ani \topglue neumožní kontrolovat polohu následující řádkové osnovy na stránce. Umístění prvního účaří totiž bude záviset na výšce prvního boxu. V tomto případě totiž před prvním boxem nepracuje ani \topskip, ani \baselineskip.

Jak tuto nevýhodu odstranit? Definujeme si makro \skiplines, které nepodléhá „mizení“ ve stránkovém zlomu, a přesto vloží do vertikálního seznamu tolik místa, kolik řádků si přeje uživatel. Vše je počítáno na řádky. Tj. například \skiplines2 vynechá 2 řádky a \skiplines{12} jich vynechá 12.

```
\def\skiplines#1{\par\hbox{}\nobreak\vskip-\baselineskip
      \vskip#1\baselineskip}
```

Skutečně, takto implementovaná mezera nebude mizet ve stránkovém zlomu díky vloženému \hbox{} a bude přesně dodržovat řádkový rejstřík. V případě výskytu takového makra na začátku stránky se účaří prázdného boxu umístí podle \topskip a dále se vynechá požadovaná mezera. Před prvním dalším boxem bude pracovat \baselineskip.

Mezera z \topskip se vkládá jedině do hlavního vertikálního seznamu, protože má smysl jen při stránkovém zlomu. Ve vnitřním vertikálním seznamu se ale může objevit analogická mezera ze \splittopskip, která se vkládá před zbytek vertikálního materiálu po provedeném \vsplit. Podrobněji viz část B. □

• **Mezera podle \parskip.** Posledním typem automaticky vložené mezery do vertikálního seznamu je mezera mezi odstavci z registru \parskip. Pokud je založen odstavec v zatím prázdném vertikálním seznamu (přesněji: v seznamu není žádná linka ani box), tato mezera se nevloží. Jinak se v okamžiku přechodu z vertikálního do odstavcového módu vloží do vertikálního seznamu mezera, která je přesně rovna registru \parskip.

Plain nastavuje \parskip na 0 pt plus 1 pt, tj. mezi odstavci dovoluje případné roztažení. Když je pak ve výstupní rutině kompletována strana, roztáhne se na výšku podle \vsize, tj. případně se roztáhnou mezery z \parskip. Tím je sice

zaručeno, že poslední řádky z každé strany budou mít společné účaří, ovšem rozhodí se řádkový rejstřík. Knuth nepovažoval ve formátu plain požadavek na řádkový rejstřík za hlavní kritérium, protože při sazbě spousty matematických vzorců je na každé straně dost nestejnoměrného bílého místa a dodržování řádkového rejstříku příliš nevynikne. □

• **Příklady.** Pokud nesázíme matematiku, měli bychom se o dodržení řádkového rejstříku postarat. V následujících ukázkách se zaměříme právě na tuto problematiku. Ve všech ukázkách budeme uvažovat `\baselineskip=12pt`. Pokud čtenář pracuje s jiným řádkováním, snadno pozná, které hodnoty v makrech bude muset upravit.

Nejprve odstraníme nežádoucí pružnost ze všech výplňků. K tomu účelu je potřeba přehodnotit všechny registry, které obsahují rozměry výplňků pro vertikální seznam. Především musí být `\parskip=0pt` a dále třeba uživatel používá makro `\bigskip` pro vložení prázdného řádku a `\medskip` pro vložení půlky řádku. V tom případě je potřeba psát:

```
231 \bigskipamount=12pt \medskipamount=6pt
232 \raggedbottom
```

Pokyn `\raggedbottom` zařídí, že se výstupní rutina nebude snažit roztáhnout stránku tak, aby účaří posledních řádků byla vždy na stejném místě přesně podle `\vsize`. Nemá k tomu totiž nástroje, protože jsme zrušili pružnost z `\parskip` a dalších výplňků. Pokud bude probíhat stránkový zlom „stejnoměrně“, pak bude mít každá strana stejný počet řádků a požadavek na spodní řádek ve stejné výšce je automaticky splněn (problémy ovšem existují, podrobněji viz sekci 6.6). Makro `\raggedbottom` vkládá do registru `\topskip` hodnotu roztažení, což dává algoritmu stránkového zlomu určitý manévrovací prostor. Přitom výstupní rutina tuto pružnost nakonec nepoužije. □

Dále se zaměříme na titulky, které usadíme třeba tak, že nad titulkem vynecháme 8 pt a pod ním 4 pt. Text se nám tedy znovu vrátí do řádkového rejstříku. Pokud bychom definovali například:

```
233 \def\titulek #1\par{\vskip8pt
234    \noindent{\bf #1}\par \nobreak\vskip4pt\relax}
```

pak to bude bezvadně fungovat všude tam, kde je titulek uvnitř sloupce textu. Jakmile se nám ovšem titulek dostane nahoru na stránku, ztratí se mezera `\vskip8pt` a zcela nám to rozhodí řádkování. Lepší řešení tedy je využít naše makro z řádku 229:

```
235 \def\titulek #1\par{\skiplines{.7}
236    \noindent{\bf #1}\par \nobreak\vskip.3\baselineskip}
```

Pokud používáme v titulcích větší písmo, je potřeba nastavit v makru dostatečně zápornou hodnotu `\lineskiplimit`. Takové nastavení stačí provést jen lokálně. □

V závěrečném příkladě si ukážeme makro na vkládání nestejně vysokých obrázků do sazby, ve které chceme dodržet řádkový rejstřík. Uživatel napíše například `\vycentruj\vbox{...}` a příslušný `\vbox` libovolné velikosti se usadí do řádkové osnovy tak, aby se nahoře i dole vynechalo stejné místo velikosti maximálně půl řádku. Následující řádky pod obrázkem budou znovu zapadat do řádkové osnovy.

```
237 \def\vycentruj{\afterassignment\uvnitrboxu \setbox0=}
238 \def\uvnitrboxu{\aftergroup\usadbox}
239 \def\usadbox{\par \dimen0=\ht0 \advance\dimen0 by \dp0
240    \divide \dimen0 by\baselineskip
241    \multiply \dimen0 by\baselineskip
242    \advance\dimen0 by\baselineskip
243    \vbox to0pt{\kern-8.5pt \vbox to\dimen0{\vfil\box0\vfil}\vss}
244    \nobreak \advance\dimen0 by-2\baselineskip
245    \vskip \dimen0 \hbox{}}
```

Na prvních dvou řádcích ukázky je zařízeno, aby se nejprve usazovaný box uložil do registru `\box0` a teprve potom se provedlo naše makro. Makro `\uvnitrboxu` se totiž provede okamžitě po vstupu do načítaného boxu za závorkou „{“. V té době ještě není provedeno přiřazení do registru `\box0`, a proto vše zpozdíme ještě prostřednictvím `\aftergroup`.

V `\dimen0` máme nejprve součet výšky a hloubky usazovaného boxu. Pak tento údaj projde výpočtem „modulo `\baselineskip`“, tj. je roven nejmenšímu násobku `\baselineskip`, pro který je nový `\dimen0` větší než původní. Je-li například `\baselineskip=12pt` a máme výšku měřeného boxu 20 pt, pak bude nový `\dimen0` roven 24 pt. Pokud ale je výška měřeného boxu přesně 24 pt, skočí nám hodnota nového `\dimen0` na 36 pt.

V řádku 243 usadíme box s nulovou výškou i hloubkou do výstupního vertikálního seznamu. Z tohoto boxu bude nahoru o 8,5 pt „přečnívat“ teoretický box s výškou `\dimen0`, uvnitř něhož bude centrován vlastní `\box0`, který chtěl uživatel centrovat. Ono Pišvejcovo číslo 8,5 pt odpovídá výšce `\strut` pro sazbu s řádkováním 12 pt. Máme-li jiné řádkování, doporučuji tento údaj upravit, případně počítat nějak podle vnějších souvislostí. Pro jednoduchost jsme zde napsali pevnou konstantu.

Na posledních dvou řádcích naší ukázky vynecháváme volné místo v násobcích `\baselineskip`. Nejprve jsme zmenšili `\dimen0` o dva `\baselineskip` a takovou velikost mezery vložíme pod náš `\vbox`. Mezera je „nezlomitelná“ díky `\nobreak`. Nakonec celou konstrukci uzavřeme prázdným boxem `\hbox`. Pod ním vznikne při vložení dalšího řádku uživatelem meziřádková mezera podle `\baselineskip` a v této mezeře se už může provést stránkový zlom. □

4. Tvorba tabulek

4.1. Opakovací výplňky typu \leaders

Slovo \leaders bychom mohli přeložit jako „vedení“, s významem vést oči v obsahu knihy od názvu kapitoly k číslu strany. Typografové k tomu obvykle používají tečky, ale \leaders umí stanovený prostor opakovaně vyplňovat čímkoli.

\leaders se chová jako \hskip v horizontálním seznamu nebo \vskip ve vertikálním seznamu. Vytvoří tedy pružný výplněk. Na rozdíl od netisknoucích pružných výplňků vytvořených pomocí \hskip a \vskip, primitiv \leaders do místa výplňku obvykle něco tiskne. Povel \leaders má tyto parametry:

```
1 \leaders ⟨box or rule⟩ ⟨glue specification⟩
```

kde ⟨*box or rule*⟩ je konstrukce boxu podle pravidla ⟨*box*⟩ nebo linka \hrule nebo \vrule s ⟨*rule specification*⟩ (viz syntaktická pravidla v části B). Je možné použít oba primitivy pro linky bez závislosti na tom, zda se vytváří horizontální nebo vertikální seznam. Parametr ⟨*box or rule*⟩ určuje, co se bude (třeba opakovaně) ve výplňku sázet. Dále ⟨*glue specification*⟩ udává charakter výplňku ⟨*glue*⟩. Ve vertikálním módu můžeme použít třeba \vskip, \vss a v horizontálním módu \hskip, \hfil a další. Tato specifikace určuje chování \leaders, pokud se na něj díváme jako na pružný výplněk.

```
2 \leaders \hbox{abc}\hfil % chová se jako \hfil a sází abcabc...
3 \leaders \hrule\hskip5cm % linka 0.4pt vysoká a 5cm široká
4 \leaders \vrule\hskip5cm % výška, hloubka podle vnějšího boxu
5 \leaders \vrule\hfil     % nyní pruží všechny tři rozměry
6 \leaders \vrule height3pt depth0pt \hskip10pt  % totéž jako:
7   \vrule height3pt depth0pt width10pt
8 \leaders \hbox{abc}\vfil % jen ve vert. módu, pod sebou boxy abc
```

Vyložíme nejprve činnost primitivu \leaders, je-li ⟨*box or rule*⟩ skutečně linkou, tj. je-li použit primitiv \hrule nebo \vrule. V horizontálním módu se ignoruje šířka linky a místo ní se použije šířka podle výsledného rozměru pružného výplňku. Uvažujme například:

```
9  \hbox to1cm {\leaders \hrule \hfil}
10 \hbox to1cm {\leaders \vrule height0.4pt \hfil}
11 \hbox to1cm {\vbox to0.4pt{}\leaders \vrule \hfil}
```

Ve všech třech případech dostáváme stejný výsledek. Box s nulovou hloubkou vysoký 0,4 pt a široký 1 cm. Celý box je černý, takže vidíme vodorovnou linku délky 1 cm a tloušťky 0,4 pt. V prvním případě (na řádku 9) plyne výška boxu z implicitní výšky \hrule, ve druhém případě jsme ji napsali explicitně a ve třetím vyplynula výška \vrule z nejvyššího elementu v boxu, kterým je prázdný \vbox.

Ve vertikálním módu se ignoruje výška a hloubka linky. Hloubka bude nulová a výška bude rovna výšce pružného výplňku, který \leaders vyplňuje. Například:

```
12 \vbox to1mm{\leaders \hrule height10cm depth10cm width10cm\vfil}
```

vytvoří box s výškou 1 mm, hloubkou 0 mm a šířkou 10 cm. Celý je vyplněn černou barvou. Poznamenejme, že i jiné barvy jsou možné, je-li použit vhodný \special pro konkrétní tiskové zařízení (například PostScriptové zařízení).

Je-li výsledný rozměr pružného výplňku z \leaders záporný, nekreslí se nic. □

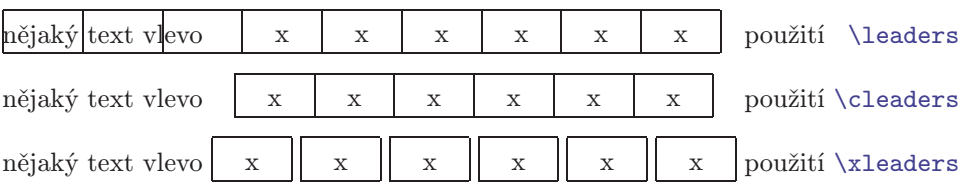
Například v záhlaví této knížky je použito:

```
13 \leaders\hrule height3pt depth-2.6pt \hfil
```

což vyplňuje prostor vedle textu záhlaví čarou. Celková tloušťka této čáry je tedy 3 pt − 2,6 pt = 0,4 pt a je na řádku ve výšce 2,6 pt od účaří. □

Uvažujme nyní případ, kdy ⟨*box or rule*⟩ je skutečně ⟨*box*⟩. Pak se mezera, odpovídající definitivní velikosti pružného výplňku určeného ⟨*glue specification*⟩, vyplní opakovaně daným boxem. Zhruba řečeno, daný box se použije tolikrát, aby se jeho kopiemi položenými vedle sebe celá mezera vyplnila. Tušíme, že to naprosto přesně nejde, protože boxem šířky w můžeme opakovaně vyplnit jen mezeru o velikosti celého násobku w, ale mezera, daná ⟨*glue specification*⟩ může být jakákoli.

TEX řeší uvedené dilema třemi různými způsoby podle toho, zda se použije primitiv \leaders, nebo jeho alternativy \cleaders a \xleaders. Všechny tři primitivy mají stejnou syntaxi zápisu a liší se jen v algoritmu umísťování opakovaných boxů do stanovené mezery. Nejprve ilustrujeme chování těchto primitivů na obrázku:



Tento obrázek je výstupem tohoto kódu (až na rámečky):

```
\hbox to\hsize{nějaký text vlevo%
    \leaders\hbox to30pt{\hfil x\hfil}\hfil použití " \leaders"}
\hbox to\hsize{nějaký text vlevo%
   \cleaders\hbox to30pt{\hfil x\hfil}\hfil použití "\cleaders"}
\hbox to\hsize{nějaký text vlevo%
   \xleaders\hbox to30pt{\hfil x\hfil}\hfil použití "\xleaders"}
% uvozovky "..." jsou zde hranicemi verbatim prostředí.
```

Opakovaný box má ve všech případech šířku 30 pt a má uprostřed písmeno „x“. Popíšeme nyní všechny tři způsoby umístění boxů:

Při použití `\leaders` se TEX podívá na levý okraj vnějšího boxu, v němž je `\leaders` použit. Odtud začne pomyslně opakovat stanovený box až do chvíle, kdy je pokryt celý text vlevo od místa, kde začíná `\leaders`. Pak naváže a začne skutečně sázet boxy až do chvíle, kdy je vyplněna požadovaná mezera tak, že se další box už nevejde.

Je-li použit `\cleaders`, TEX vloží do požadované mezery maximální počet boxů, všechny jsou těsně u sebe. Vlevo i vpravo od této skupiny boxů TEX doplní stejnou mezeru k vyplnění případného nevyčerpaného zbytku.

Při použití `\xleaders` TEX vloží rovněž maximální počet boxů, ovšem před první, za poslední a mezi každé dva vloží stejnou mezeru. Velikosti těchto mezer vyplní v součtu nevyčerpaný zbytek. □

První alternativa, tj. použití `\leaders`, se hodí pro vkládání teček do obsahů. Pokud třeba máme

```
\tocline{⟨název kapitoly⟩}{⟨číslo strany⟩}
```

pak můžeme definovat

```
\def\tocline #1#2{%
    \line{#1 \leaders\hbox to1em{\hfil.\hfil}\hfil\ #2}}
```

Protože TEX měří usazení boxů s tečkami od levého okraje vnějšího boxu, budou všechny tečky na jednotlivých řádcích v obsahu přesně pod sebou. Nezáleží na tom, jak dlouhý je text s názvem kapitol. □

Při opakování boxů ve vertikálním módu TEX pracuje s celkovou výškou opakovaného boxu (tj. výška plus hloubka). S tímto rozměrem pracuje analogicky, jako v horizontálním módu pracuje s šířku opakovaného boxu. Boxy klade pod sebe.

Není-li výsledná mezera z ⟨*glue specification*⟩ v `\leaders` kladná, TEX nevysází žádný box. Není-li rozměr opakovaného boxu kladný, TEX rovněž netiskne žádný

box. „Rozměr“ zde znamená šířku v případě horizontálního módu a celkovou výšku ve vertikálním módu.

V místě pro zápis boxu můžeme použít primitiv \box. TEX pak opakuje box z daného registru. Je ovšem třeba upozornit na to, že po takovém použití boxu v \leaders je hodnota registru ztracena. Box je sice v rámci jednoho \leaders opakován, ale nakonec je registr vyprázdněn. Pokud budeme obsah boxu potřebovat i nadále, musíme použít namísto \box primitiv \copy. □

• **Příklad.** Popíšeme sazbu obsahu v této knížce. Všimněme si, že tečky jsou tam střídány tak, že když na ně položíme pravítko pod úhlem 63,5°, máme tečky na jeho hraně pěkně v jedné řadě.

Uvažujme, že obsah sestavujeme z pracovního souboru `tbn.toc`, kde jednotlivé řádky vypadají třeba takto:

```
\partline {A}{Algoritmy}{9}
\chaptocline {1}{Vstupní části \TeX u}{10}
\subtocline {1.1}{Koncept vstupní brány \TeX u}{10}
\subtocline {1.2}{Input procesor}{12}
\subtocline {1.3}{Token procesor}{19}
\chaptocline {2}{Expand procesor}{31}
\subtocline {2.1}{Definování maker}{31}
⟨atd...⟩
```

Vidíme, že pro kapitolu máme řádek začínající sekvencí \chaptocline a pro sekci je použita sekvence \subtocline. Pro sazbu takových teček používáme \leaders s opakovaným boxem:

```
\hbox to12pt{\kern\dimen0.\hss}
```

kde pro \dimen0 postupně střídáme hodnoty 3 pt a 9 pt. Pro vystřídání hodnot je použito makro \cvak. Celá věc vypadá tedy takto:

```
\def\cvak{\ifdim\dimen0<6pt \dimen0=9pt \else \dimen0=3pt \fi}
\def\puntiky{\leaders\hbox to12pt{\kern\dimen0.\hss}\hfil}
\def\chaptocline #1#2#3{\medskip
   \line{#1.\enskip #2\hfil #3}\medskip}
\def\subtocline #1#2#3{%
   \line{\qquad #1\quad #2\puntiky\hbox to2em{\hfil#3}}\cvak}
\def\partline #1#2#3{\cvak\bigskip
   \line{\bf Část #1: #2 \puntiky\ #3}\cvak}
```

Údaje o stranách kapitol nejsou v řádkové osnově. Nelze tedy v těchto řádcích umístit tečky tak, aby v obou směrech navazovaly. Proto jsem se při návrhu typografie

dokumentu rozhodl, že tam žádné tečky nebudou. Jelikož mezi posledním řádkem sekce nějaké kapitoly a dalším řádkem sekce následující kapitoly jsou vynechány dohromady dva řádky (půl plus jedna plus půl), stačí v tomto místě uplatnit jedno `\cvak`. Dále se provede vždy jedno `\cvak` za každým řádkem sekce. □

V plainu jsou pomocí `\leaders` a `\cleaders` definována makra `\hrulefill` a `\dotfill` (viz část B). □

4.2. Tabulky s pevnou šířkou sloupců

Pokud dopředu víme, jak široké budou jednotlivé sloupce tabulky, nemusíme použít primitiv `\halign`, o němž bude řeč v další kapitole. Využijeme možnost psát `\hbox to`⟨*dimen*⟩. Tím nastavíme šířky jednotlivých sloupců.

Myšlenku hned ilustrujeme na příkladě. Každý semestr za mnou přijdou kolegové, že potřebují vytisknout seznamy studentů, aby si do seznamů mohli dělat v průběhu jednotlivých týdnů v semestru poznámky. Jména studentů jsou výstupem jisté databáze. Předpokládejme, že jsou jména rozdělena po jednotlivých studijních skupinách. V jedné skupině je maximálně 25 studentů. Vytiskneme pro každou skupinu pětadvacetiřádkový formulář, kde v každém řádku bude napsáno jméno jednoho studenta. Je-li studentů méně než 25, budou zbylé řádky prázdné, ale přihrádky a číslo nakreslíme.

Jeden řádek bude vypadat třeba takto:

1.	Mravenec Ferdinand									

První sloupec obsahuje pořadové číslo, druhý jméno studenta, další políčko je pro individuální využití učitele. Pak za dvojitou čarou následuje patnáct přihrádek, protože semestr má obvykle patnáct týdnů. Čtenář v knize vidí jen osm přihrádek, protože na šířku sazby knihy by všech patnáct přihrádek vypadalo moc stísněně.

Najednou tiskneme seznamy se stovkami až tisíci studenty a je vhodné, aby formuláře pro jednotlivé studijní skupiny vypadaly stejně. To znamená, že šířky přihrádek by měly být stanoveny jednotně. Kdybychom použili `\halign`, museli bychom do TEXu načíst celou databázi, aby TEX rozhodl podle „nejdelšího“ jména, jak širokou volit přihrádku pro jména. To je neekonomické a dokonce neužitečné. Zatímco výjimečně nějaký zahraniční student z Dálného východu má jméno složeno z pěti nebo více slov, většina studentů má jméno obvyklé délky. Kvůli těm výjimkám ze zahraničí by seznamy všech ostatních studijních skupin v přihrádce pro jméno zbytečně plýtvaly místem. Je proto vhodnější toho Muhameda Abu Ali Admal atd. nechat vysázet přes okraj přihrádky.

Volíme tedy pevnou šířku přihrádek. Například pro pořadové číslo nám bude stačit 1 cm, protože se tam pro každou skupinu vyskytuje nejvýše číslo dvojciferné. Dále pro jméno jsme stanovili šířku přihrádky 5 cm. Další políčko určené pro potřebu učitele bude mít 2 cm a pak ostatních 15 přihrádek vyplní stejnoměrně zbytek.

Pro začátek a pro jednoduchost předpokládejme, že vstupní soubor z databáze jmen už u každého studenta obsahuje TEXovskou sekvenci a oddělovač, tedy údaje vypadají takto:

```
41 \p Mravenec Ferdinand*
42 \p Novák Jan*
43 ⟨atd...⟩
```

Vytvoříme makro `\p`, které bude sázet jeden řádek formuláře:

```
44 \offinterlineskip
45 \newcount\num \newcount\tempnum
46 \parindent=0pt \parskip=0pt
47 \def\\#1 #2|{\hbox to#1{\enspace #2\hss}\cara}
48 \def\cara{\kern-0.4pt\vrule}
49 \def\radek #1|#2|{\vrule height4.5mm depth1.5mm
50    \\10mm \hfill #1\enspace|\\50mm #2|\\19mm |\\1mm |%
51    \zbytek \par \hrule}
52 \def\zbytek{\tempnum=0
53    \loop \advance\tempnum by1 \hfill\cara
54    \ifnum\tempnum<15 \repeat}
55 \def\p #1*{\advance\num1 \radek\the\num.|#1|}
```

Zde jsme si na řádku 47 definovali zkratku, abychom nemuseli pořád dokola psát `\hbox to`⟨*dimen*⟩. Makro `\radek` vytvoří celý jeden řádek formuláře a vezme dva parametry: první, číslo, vytiskne do první přihrádky a druhý, jméno studenta, do druhé. Linka `\vrule` se stanovenou výškou a hloubkou (řádek 49) určuje celkovou výšku každého řádku formuláře. Linka zde funguje jako podpěra. Předpokládá se, že žádný text se jménem nebude mít nikdy výšku nebo hloubku větší, než má tato linka. Tato linka také zahajuje odstavcový mód. Na následujícím řádku v kódu (50) pak definujeme celou strukturu řádku formuláře. Sazba probíhá v odstavcovém módu. Pomocí makra `\zbytek` vložíme patnáct vertikálních čar stejně daleko od sebe (`\hfill` vytlačí `\parfillskip` z konce řádku odstavce). Pak uzavřeme odstavec. Konečně nakreslíme vodorovnou čáru pod řádkem (`\hrule`).

Nesmíme zapomenout přidat čáru nad prvním řádkem formuláře. Dále po načtení seznamu studentů jedné skupiny se musíme postarat o to, aby byl zbytek řádků do dvaceti pěti doplněn prázdnými kolonkami. Proto ještě definujeme makro `\dodelej`, které dotiskne zbytek do dvaceti pěti řádků:

```
\def\dodelej{\loop \advance\num by1
   \ifnum\num<26 {\radek\the\num.||}\repeat \vfil\break}
```

Všimněme si, že jsme zde sazbu řádku makrem `\radek` uzavřeli do skupiny. Máme tam totiž ještě jeden `\loop` a je potřeba, aby se vnitřní cyklus s vnějším nehádal.

Nyní makro trošku vylepšíme, abychom nemuseli v databázi před každé jméno psát `\p` a na konec nemuseli připojovat hvězdičku. Nechť se pomocí `\obeylines` stane konec vstupního řádku `^^M` aktivní. Znak `^^M` bude sloužit jako separátor:

```
{\obeylines\gdef\p #1^^M{\advance\num1 \radek\the\num.|#1|}}
```

Nyní stačí načíst soubor z databáze při aktivním `^^M` a při `\everypar={\p}`. □

Uvedený příklad rozvedeme na ukázce maker pro zpracování skutečného výstupu z databáze, která se používá na naší fakultě. Tvůrci výstupních procedur z databáze předpokládali, že uživatel použije jednoduchou řádkovou tiskárnu a na TEX vůbec nepomysleli. O to budeme mít v makru více práce správně načíst výstupní sestavu z databáze, která vypadá takto:

```
  Komponenta studium                Datum a čas: 23.09.96 15:17

    Ročník:   1   Skupina:     21

   Příjmení           Jméno             Rodné číslo
   -----------------  ----------------  -----------
   Mravenec           Ferdinand         1234567890
   Erben              Karel Jaromír     0987654321
   ⟨atd., další studenti⟩
   ⟨prázdné řádky⟩
                                            Strana 1     ze 18
^^L

  Komponenta studium                Datum a čas: 23.09.96 15:17

    Ročník:   1   Skupina:    22

   Příjmení           Jméno             Rodné číslo
   ------------------ ----------------- -----------
   Novák              Jan               3333333333
   ⟨další studenti a prázdné řádky⟩
                                            Strana 2     ze 18
^^L
⟨další skupiny stejně⟩
```

Makro, které dokáže takovou databázi načíst a vytisknout tabulky se seznamy studentů podle právě popsaného formátu, může vypadat následovně:

```
\nopagenumbers \offinterlineskip
\newcount\num \newcount\tempnum
\special{landscape}  \hsize=\vsize \voffset=-.5cm
\parindent=0pt  \parskip=0pt
\def\\#1 #2|{\leavevmode\hbox to#1{\enspace #2\hss}\cara}
\def\cara{\kern-0.4pt\vrule}
\def\radek #1|#2|{\vrule height4.5mm depth1.5mm
   \\10mm \hfill #1\enspace|\\50mm #2|\\19mm |\\1mm |%
   \zbytek{}\par \hrule}
\def\zbytek #1{\tempnum=0
   \loop \advance\tempnum by1
     \hfill\hbox to0pt{\hss#1\hss}\hfill\cara
   \ifnum\tempnum<15 \repeat\hbox{}}
\def\dodelej{\loop \advance\num by1 %
   \ifnum\num<26 {\radek\the\num.||}\repeat \vfil\break \num=0}
\def\setcatnumbers#1{\tempnum=`0 % kategorie pro číslice
    \loop \catcode\tempnum=#1
       \ifnum\tempnum<`9 \advance\tempnum by1 \repeat}
\obeylines
\def\p #1^^M{\advance\num1 \radek\the\num.|#1|}%
\def\demask Komponenta #1Ročník: #2 Skupina: #3^^M{\everypar={}%
   \vrule height5mm depth2mm width0mm %
   {\let\cara=\relax % Sazba záhlaví tabulky; nepotřebujeme čáry
    \\80mm \bf \vrule width0pt depth4pt  Skupina #3|% podpěra
    \zbytek{\the\tempnum}}\par\hrule                % čísla týdnů
   \setcatnumbers{9} \maskpata}
\def\maskpata #1-^^M#2Strana#3^^M{\everypar={\p}#2\everypar={}%
   \dodelej \everypar={\demask}\setcatnumbers{12}}
\everypar={\demask}
\input database.txt
\end
```

Některé části makra si ještě zaslouží naši pozornost. Na řádku 85 jsme použili `\special{landscape}`, což dává do `dvi` pokyn pro ovladač `dvips`, aby se formulář tiskl na stranu otočen o 90°. Proto budeme mít pro jednotlivé řádky dost místa a můžeme nastavit `\hsize=\vsize`.

Makro `\zbytek` jsme rozšířili, aby obsahovalo jeden parametr, který bude umístěn uprostřed každé přihrádky pro týden semestru. Použití `\zbytek{\the\tempnum}` na řádku 107 nám tedy vysází čísla jednotlivých týdnů, což potřebujeme do záhlaví.

Od řádku 101 nastavujeme znak `^^M` (konec řádku) jako aktivní a má význam `\par`. Konec řádku bude totiž oddělovačem pro parametry našich dalších maker. Makro `\demask` se spustí z `\everypar`, sotva se TEX dostane k prvnímu slovu v databázi, kterým je slovo `Komponenta`. Vše až po slovo `Ročník` přeskočíme a do `#3` ukládáme číslo skupiny, které budeme sázet v záhlaví. Po sazbě záhlaví potřebujeme z následujících vstupních údajů „oddělit zrno od plev“, což provede makro `\maskpata`. Do parametru `#1` se schovají zbytečnosti ze vstupního záhlaví včetně všech podtrhovacích znaků „`---`“ až po poslední. To jsou plevy. V parametru `#2` máme zrno, které je navíc načteno v okamžiku, kdy číslice `0` až `9` mají kategorii 9 (ignoruj), takže ignorujeme údaje o rodných číslech. Konečně v parametru `#3` jsou zase plevy. Při `\everypar={\p}` pak zpracujeme jednotlivé řádky „zrna“ způsobem, který jsme popsali výše. Nakonec vracíme číslicím původní kategorii 12, abychom mohli přečíst ze záhlaví další strany číslo skupiny.

Všimneme si ještě, že výstup z databáze klade mezi jednotlivé stránky určené pro levnou řádkovou tiskárnu znak `^^L` (konec strany). To nás v plainu vůbec nevyvádí z míry, protože tento znak zde má aktivní kategorii a má význam `\par`. □

• **Makro plainu emulující vlastnosti psacího stroje.** Ve zbylé části této sekce se budeme zabývat makry plainu pro tzv. `\tabalign`, což umožňuje nastavovat a posléze používat sloupečky s pevnou šířkou. Využití makra není příliš časté. Snad je to proto, že až příliš připomíná vlastnosti psacího stroje. Víme, že příprava sazby a používání psacího stroje jsou dvě zcela odlišné věci.

Nejprve vyložíme vlastnosti makra na uživatelské úrovni a teprve v další části se pustíme do rozboru maker plainu, kterými jsou tyto vlastnosti implementované.

V prostředí `\tabalign` uživatel používá tzv. *tabulátory*, které označuje symbolem „`&`“. Jedná se o neviditelné značky, jejichž poloha v řádku určuje umístění levých okrajů jednotlivých sloupců.

Řádek řízený prostředím `\tabalign` se zapisuje ve formátu `\+`⟨*obsah řádku*⟩`\cr`. V ⟨*obsahu řádku*⟩ se mohou vyskytovat znaky „`&`“, které mohou mít jeden z těchto významů:

- Nastavení polohy tabulátoru podle polohy aktuálního bodu sazby.
- Využití polohy tabulátoru pro umístění následujícího textu.

Znaky „`&`“ v ⟨*obsahu řádku*⟩ počítáme zleva „první, druhý, třetí“ atd. Přitom n-tý znak „`&`“ reprezentuje n-tý tabulátor. Není-li n-tý tabulátor nastaven, přítomnost n-tého znaku „`&`“ jej nastavuje. Je-li n-tý tabulátor nastaven, nenastavuje se znovu, ale začátek textu za n-tým znakem „`&`“ bude (po ignorování vstupních mezer) umístěn v poloze tabulátoru. Na počátku není nastaven žádný tabulátor. Příklad:

```
\+ Tady nastavuji první:& tu druhý:& a píšu další text \cr
\+ text od kraje & od prvního & od druhého, nastavuji třetí:&\cr
\+ &&& text od třetího tabulátoru\cr
```

Tento experiment nám dává následující výsledek:

Tady nastavuji první:tu druhý:a píšu další text
text od kraje od prvníhood druhého, nastavuji třetí:
text od třetího tabulátoru

V ukázce si můžeme všimnout dvou vlastností: (1) Text podle tabulátorů se může beztrestně překrývat. (2) Text sázený z ⟨*obsahu řádku*⟩ může beztrestně vyčnívat ze zrcadla sazby daného šířkou `\hsize`.

Pomocí makra `\cleartabs` použitém v ⟨*obsahu řádku*⟩ můžeme mazat nastavení všech tabulátorů vpravo od zápisu `\cleartabs`. Jinými slovy, všechny následující znaky „`&`“ použité za `\cleartabs` v ⟨*obsahu řádku*⟩ polohu tabulátoru znovu nastavují a nepoužívají. Navážeme-li na náš předchozí příklad:

```
\+ další text od kraje & od prvního a znovu
   nastavuji druhý:\cleartabs & a nastavuji ještě třetí:&\cr
```

Pomocí makra `\settabs`, které používáme mimo ⟨*obsah řádku*⟩, lze vymazat polohy všech tabulátorů a nastavit je znova. Použije-li se toto makro ve formátu

```
\settabs\+ ⟨obsah řádku⟩\cr
```

pak znaky „`&`“ uvnitř ⟨*obsahu řádku*⟩ nastavují tabulátory, ale výsledný text se netiskne. Můžeme tedy nastavit tabulátory podle nějakého abstraktního textu, o kterém víme, že reprezentuje například nejširší zápis v jednotlivých sloupečcích.

Existuje ještě jeden způsob použití makra `\settabs` pro nastavení tabulátorů. Při zápise `\settabs`⟨*number*⟩`\columns` TEX nastaví $n = \langle number \rangle$ tabulátorů tak, že poslední (n-tý) je přesně na pravém okraji zrcadla určeného podle `\hsize` a všechny ostatní jsou rovnoměrně daleko od sebe a rozdělují šířku zrcadla na n stejně širokých sloupečků. Například po:

```
\settabs 3\columns
\+ Od kraje & Od prvního & Od druhého \cr
% "& Od třetího" neukážeme, protože by vyčníval ven ze zrcadla.
```

dostáváme:

Od kraje Od prvního Od druhého

Zarovnání textu nalevo v každém sloupečku je zařízeno pomocí `\hss` připojovaném vpravo k textu. Proto můžeme ve sloupci text centrovat třeba takto:

```
\settabs 3\columns
\+\hss první sloupec&\hss druhý&\hss a třetí&\cr
```

Kdybychom těsně před `\cr` nenapsali znak „`&`“, nebude TEX k textu žádné `\hss` připojovat. Text posledního sloupce těsně před `\cr` totiž není nijak upravován. □

Nyní si popíšeme činnost makra plainu, které nám uvedené prostředí `\tabalign` nabízí. Zjistíme, že se makro nakonec opírá o primitiv `\halign`, ovšem není využita jeho vlastnost sestavování sloupečků podle nejširšího textu.

Makro budeme komentovat po částech. Nejprve uvedeme deklarace:

```
\newif\ifus@ \newif\if@cr
\newbox\tabs \newbox\tabsyet \newbox\tabsdone
```

Pomocí `\ifus@` se ptáme na to, zda si uživatel přeje tisknout ⟨*obsah řádku*⟩. Víme, že při `\settabs\+`⟨*obsah řádku*⟩`\cr` podléhá sestavení ⟨*obsahu řádku*⟩ stejným pravidlům, pouze výsledek nakonec nevytiskneme. V takovém případě bude `\ifus@` dávat „false“. Dále podle `\if@cr` makro zjišťuje, zda je přečtena poslední položka, za kterou už následuje `\cr`. Tuto položku makro zpracuje odlišným způsobem.

V boxu `\tabs` je seznam boxů, jejichž šířky určují šířky jednotlivých sloupců v opačném pořadí, než ve kterém počítáme sloupce. Šířka posledního boxu v `\tabs` tedy určuje šířku prvního sloupce. Například po `\settabs 3\columns` máme v `\tabs` tři boxy, každý má šířku rovnou třetině `\hsize`.

Na začátku sestavování ⟨*obsahu řádku*⟩ vložíme boxy z `\tabs` do `\tabsyet`. Dále v průběhu sestavování ⟨*obsahu řádku*⟩ budeme jednotlivé boxy odpovídající zpracovávaným položkám přerovnávat z `\tabsyet` do `\tabsdone`. Při této činnosti můžeme některé boxy přidat (nové tabulátory), některé mazat (`\cleartabs`) a posléze třeba zase přidávat. Na konci řádku (při `\cr`) vrátíme (případně) nové údaje o tabulátorech do `\tabs`.

Nyní následují makra `\cleartabs` a `\settabs` a začátek makra `\+`.

```
\def\cleartabs{\global\setbox\tabsyet=\null \setbox\tabs=\null}
\def\settabs{\setbox\tabs=\null \futurelet\next\sett@b}
\let\+=\relax % in case this file is being read in twice
\def\sett@b{\ifx\next\+
    \def\nxt{\afterassignment\s@tt@b\let\nxt}%
  \else\let\nxt=\s@tcols\fi \let\next=\relax \nxt}
\def\s@tt@b{\let\nxt\relax \us@false \m@ketabbox}
```

```
\def\tabalign{\us@true \m@ketabbox} % non-\outer version of \+
\outer\def\+{\tabalign}
\def\s@tcols#1\columns{\count@=#1 \dimen@=\hsize
  \loop \ifnum\count@>\z@ \@nother \repeat}
\def\@nother{\dimen@ii=\dimen@ \divide\dimen@ii by\count@
  \setbox\tabs=\hbox{\hbox to\dimen@ii{}\unhbox\tabs}%
  \advance\dimen@ by-\dimen@ii \advance\count@ by-1 }
```

Vidíme, že `\cleartabs` maže boxy v `\tabsyet`, ale nikoli v `\tabsdone`. To znamená, že už zpracované položky budou mít tabulátory zachovány, ale následující budou nově nastaveny. Makro `\settabs` nuluje všechny tabulátory (v `\tabs`) a dále se pomocí `\futurelet` ptá, zda následuje token `\+`. Pokud ano, provede se zpracování řádku (`\m@ketabbox`) s parametrem „netiskni výsledek“ (`\us@false`). Pokud ne, makro se chová jako `\s@tcols`, tj. čte ⟨*number*⟩ před separátorem `\columns` a vkládá jej do `\count@`. V `\@nother` se postupně v cyklu uloží do `\tabs` prázdné boxy o šířce `\hsize`/⟨*number*⟩. Promyslete si, proč.

Důležitou vlastností makra `\+` je, že je definováno prefixem `\outer`. To zamezí chybě typu `\+`⟨*obsah řádku*⟩ `\+` (tj. zapomenuté `\cr`). Na druhé straně s tím máme v makru trochu problémy a musíme se s tím vyrovnat. Následující rozbor makra je tedy možno chápat též jako ukázku různých nebezpečí, číhajících při definování jakéhokoli makra prefixem `\outer`.

V těle definice makra `\setb@x` je použit token `\+`, a proto v tu chvíli nesmí být typu `\outer`. Z toho důvodu je před tím řečeno `\let\+=\relax`, abychom zrušili možný původní význam tokenu `\+`. To může být užitečné při opakovaném čtení souboru `plain.tex`.

V makru `\settabs` se může pomocí `\futurelet` ztotožnit význam `\next` s tokenem `\+`. Pak ovšem má `\next` typ `\outer` a už nikdy by nebylo možné jej použít v těle definice. Přitom slůvko `\next` používáme v tělech definic rádi. Proto na řádku 132 dáváme tokenu `\next` význam `\relax`.

Vlastní makro `\+` má verzi bez `\outer` s názvem `\tabalign`. Tato verze se dá použít v našich vlastních definicích. Makro `\tabalign` expanduje nakonec na `\m@ketabbox` při `\us@true`, tj. bude se tisknout výsledný ⟨*obsah řádku*⟩.

Konečně uvedeme makra, která zpracovávají ⟨*obsah řádku*⟩.

```
\def\m@ketabbox{\begingroup
  \global\setbox\tabsyet=\copy\tabs
  \global\setbox\tabsdone=\null
  \def\cr{\@crtrue\crcr\egroup\egroup
    \ifus@ \unvbox0\lastbox \fi\endgroup
    \setbox\tabs\hbox{\unhbox\tabsyet\unhbox\tabsdone}}%
```

```
147    \setbox0=\vbox\bgroup\@crfalse
148      \ialign\bgroup&\t@bbox##\t@bb@x\crcr}
149  \def\t@bbox{\setbox0=\hbox\bgroup}
150  \def\t@bb@x{\if@cr\egroup % now \box0 holds the column
151    \else\hss\egroup
152       \global\setbox\tabsyet=\hbox{\unhbox\tabsyet
153            \global\setbox1=\lastbox}% now \box1 holds its size
154       \ifvoid1 \global\setbox1=\hbox to\wd0{}%
155       \else \setbox0=\hbox to\wd1{\unhbox0}\fi
156       \global\setbox\tabsdone=\hbox{\box0\unhbox\tabsdone}
157    \fi \box0}
```

Nejprve v `\m@ketabbox` otevřeme skupinu a v rámci ní nastavíme výchozí hodnoty pro „přerovnávací řady“ vzorových boxů `\tabsyet` a `\tabsdone`. Předefinujeme `\cr`, takže všude v souvislosti s `\halign` budeme používat náhradní `\crcr`. Do definice `\cr` se podíváme až nakonec. Nyní začneme číst makro od řádku 147. Tam otvíráme `\vbox` a do něj (jak uvidíme později) postupně vložíme výsledek jednořádkového `\halign`.

Makro `\ialign` je ekvivalent k `\halign` s nastavením nulového `\tabskip`. Zde máme v deklaraci řádku jediné schéma položky s příznakem návratu (terminologie k `\halign`, viz následující sekci). Při zahájení položky se expanduje `\t@bbox`, tj. provede se „`\setbox0=\hbox{`“. Jakmile narazíme při zpracování položky na znak „`&`“, box 0 se uzavře (viz `\t@bb@x`). Máme tedy v boxu 0 právě načtenou položku. Pokud je položka ukončena znakem „`&`“, dává `\if@cr` hodnotu „false“, neboť jsme zatím nedosáhli konce řádku pomocí `\cr`. Povíme si, co se v tomto případě odehrává (řádky 151 až 156):

Předně položka se uzavře s připojením `\hss` na konec textu položky. Pak se do boxu 1 vloží z `\tabsyet` box, který by měl svou šířkou udávat šířku právě zpracovávané položky. Přitom se z `\tabsyet` tento box zruší. Pokud se odtud nepodařilo získat informaci o šířce položky (`\ifvoid1`), budeme tabulátor nastavovat. Proto nastavíme v boxu 1 šířku právě načtené položky (`\wd0`). Pokud informace o tabulátoru existuje (`\else`), nebude se tabulátor nastavovat, ale upraví se šířka boxu 0 s načtenou položkou na požadovanou šířku `\wd1`. Nakonec se do `\tabsdone` uloží (případně nový) box udávající šířku položky, neboli polohu tabulátoru. Ven z makra ke zpracování v položce `\halign` vystupuje vždy box 0.

Ukončí-li uživatel řádek pomocí `\cr`, toto makro expanduje na `\@crtrue` následované `\crcr` (tj. skutečným ukončením řádku v `\halign`). Nyní už v závěrečné části schématu položky není žádná rozsáhlá činnost, pouze se text z boxu 0 dostává do sazby položky, jak jí rozumí primitiv `\halign`.

První `\egroup` na řádku 144 uzavírá `\halign`. Tento primitiv už nemá co na práci. Jeden řádek „tabulky“ s rozměry položek tak, jak je připravilo makro, vloží do

vertikálního seznamu. Druhé \egroup uzavírá \vbox otevřený na řádku 147. Tento box tedy nyní vstupuje do pracovního boxu 0. V tomto boxu máme jediný řádek odpovídající ⟨*obsahu řádku*⟩, který jsme právě zpracovali. Má-li být tento řádek tištěn (\ifus@), vrátíme jej do vnějšího vertikálního seznamu pomocí \unvbox0. Zde stojí za pozornost trik s \lastbox z řádku 145. Tento trik je rozebrán u hesla \unvbox v části B.

Nakonec na řádku 146 obnovíme stav tabulátorů udržovaný v \tabs. Nejprve vkládáme nevyčerpané tabulátory z \tabsyet a za nimi použité a případně nově zpracované tabulátory z \tabsdone. □

4.3. Tabulky pomocí \halign

Algoritmy, které souvisejí s primitivem \halign, jsou poměrně komplikované. Začneme proto zlehka, jednoduchým příkladem:

```
158 \tabskip=2em plus 5em
159 \halign{{\bf Položka:} #\hfil & \hfil#\hfil \quad konec\cr
160   první & toto bude na střed \cr \noalign{\hrule\smallskip}
161   druhá & další text \cr
162   třetí \cr}
```

Tento kód vyprodukuje následující tabulku:

Položka: první	toto bude na střed	konec
Položka: druhá	další text	konec
Položka: třetí		

\halign má (zhruba) tuto strukturu zápisu:

```
163 \halign{⟨deklarace řádku⟩ \cr
164   ⟨položky tabulky prvního řádku⟩ \cr
165   ⟨položky tabulky druhého řádku⟩ \cr
166   ...
167   ⟨položky tabulky posledního řádku⟩ \cr}
```

Pojmem *tělo tabulky* označujeme vše, co je zapsáno mezi závorkami {...} za primitivem \halign. Tělo tabulky se dělí na dvě části: *deklaraci řádku* ukončenou prvním \cr a *datovou část* tabulky. V datové části jsou *položky* tabulky, které jsou od sebe odděleny tokenem kategorie 4 (ASCII kód nehraje roli, obvykle se používá &$_4$). Jednotlivé řádky jsou v datové části odděleny od sebe sekvencí \cr. Za sekvencí \cr může být primitiv \noalign a za ním pak {⟨*vertikální seznam*⟩}. Pokud se tak stane, TEX zatím neotvírá další položku, ale akceptuje ⟨*vertikální seznam*⟩, který

uloží pod právě ukončený řádek. Pokud je `\noalign` hned za prvním `\cr`, které ukončuje deklaraci řádku, seznam se uloží nad tabulku před první řádek tabulky.

Primitiv `\cr` musí být uveden i za posledním řádkem tabulky, jinak se dočkáme chyby. Pokud si konstrukci `\halign` píšeme sami, pak toto omezení není problémem, protože si na to dáme pozor. Pokud ale vytváříme makro pro uživatele, který jednou použije `\cr` na konci své tabulky a podruhé ne, pak mohou nastat problémy. Proto existuje primitiv `\crcr`, který je po sekvenci `\cr` (a případně za konstrukcí `\noalign`) ignorován, zatímco všude jinde se chová jako `\cr`. V makrech tedy doporučujeme psát pro uzavření konstrukce tabulky `\crcr`}.

Deklarace řádku se člení na *schémata položek*, která jsou od sebe oddělena tokenem kategorie 4 (`&`$_4$). Každé schéma položky *musí* obsahovat právě jeden token kategorie 6 (obvykle se používá `#`$_6$). Samozřejmě, pokud je schéma součástí těla nějaké definice, znak `#`$_6$ musí být zdvojený. S tímto problémem jsme se už setkali v sekci o definování maker 2.1.

Při čtení obsahu jedné položky TEX otevře `\hbox` a tím též vstoupí do nové úrovně skupiny. V tomto boxu začne nejprve zpracovávat tokeny uvedené v odpovídajícím schématu položky až po znak `#`$_6$ mimo něj. Přitom provádí expanzi maker. V našem příkladě se třeba `\bf` expanduje na `\fam\bffam\tenbf`. Poté přejde TEX do datové části a čte (a expanduje) tokeny odtud až po token uzavírající položku (`&`$_4$ nebo `\cr`). Pak se vrátí k druhé části schématu položky za znakem `#`$_6$ a dokončí zpracování položky. Tím vytvoří horizontální seznam a uzavře `\hbox` položky. Takto postupně prochází všechny položky v datové části `\halign`. Pro každou z nich vytvoří speciální `\hbox`.

Poznámka k mezerám. Při zahájení čtení položky z datové části jsou ignorovány všechny mezery až po první token různý od mezery. Další (případné) mezery v datové části už jsou významné. V našem příkladě na řádku 160 je před slovem `toto` ve druhé položce mezera ignorována, ale za slovem `střed` mezera zůstává. Rovněž v deklaraci řádku se ignoruje mezera za znakem `&`$_6$, takže například na řádku 159 je mezera za tímto znakem nevýznamná.

Je-li v datové části v rámci jednoho řádku méně položek, než je schémat položek v deklaraci řádku, boxy zbývajících položek zůstanou prázdné — tj. nebudou obsahovat ani materiál ze schématu položek. Viz řádek 162 v naší ukázce, kde chybí druhá položka v datové části, a tudíž se nevyprodukuje ani slovo „konec“, společné všem ostatním řádkům.

Pokud je v datové části na jednom řádku více položek, než je schémat položek v deklaraci řádku, dočkáme se chybového hlášení `Extra alignment tab has been changed to \cr`. V deklaraci řádku ale můžeme definovat jednoduchou smyčku, která pracuje takto: Po vyčerpání schémat položek se TEX vrací na schéma s tzv. *příznakem návratu* a od něj znovu postupuje až k poslednímu schématu.

Pak se případně znovu vrací na totéž schéma s příznakem návratu, atd. Schéma s příznakem návratu se značí znakem kategorie 4 na jeho začátku. Syntakticky to znamená, že znak `&`$_4$ se vyskytuje těsně za otevírací závorkou `{`$_1$ konstrukce `\halign` (tj. hned první schéma má příznak návratu) nebo se sejdou dva znaky `&`$_4$ vedle sebe. To neznačí prázdné schéma, nýbrž se tím praví, že následující schéma má příznak návratu. V deklaraci řádku může mít příznak návratu nejvýše jedno schéma, ke kterému se pak cyklus vrací. Například:

```
\halign{1:# & 2:# && 3:# & 4:# \cr
  a & b & c & d & e & f & g & h \cr
  i \cr
  j & k & l & m & n \cr}
```

dává na výstupu:

1:a 2:b 3:c 4:d 3:e 4:f 3:g 4:h
1:i
1:j 2:k 3:l 4:m 3:n □

Nyní přejdeme k výkladu nejdůležitější vlastnosti primitivu `\halign`. Tabulka je sestavována ve dvou průchodech. V prvním průchodu se vytvoří boxy jednotlivých položek způsobem popsaným výše, přičemž se na tyto boxy neklade žádný požadavek na stažení nebo roztažení. Pak TEX spočítá maximální šířku takových boxů v každém sloupci položek. Pro i-tý sloupec budeme značit toto maximum znakem w_i.

V druhém průchodu sestavuje TEX jednotlivé řádky tabulky tak, že do vnějšího vertikálního seznamu nejprve zavede ⟨*vertikální materiál*⟩ sestavený z případného `\noalign`, který byl hned za prvním `\cr`. Dále vloží první řádek tabulky jako `\hbox` obsahující: „t_0, box první položky upravený na šířku w_1, dále t_1, pak box druhé položky upravený na šířku w_2, t_2 atd. až po box poslední položky upravený na šířku w_n a konečně t_n“. Znaky t_0 až t_n zde znamenají výplňky typu ⟨*glue*⟩ odpovídající hodnotě `\tabskip` před tabulkou, mezi položkami a za tabulkou. Podrobněji o nich viz níže. Tím je první řádek tabulky zaveden do vnějšího vertikálního seznamu. Pak se vloží do tohoto seznamu případně ⟨*vertikální materiál*⟩ z `\noalign` uvedeného za prvním řádkem, pak se analogicky vloží `\hbox` s druhým řádkem tabulky apod. Protože jsou t_0 až t_n a w_1 až w_n v každém řádku stejné, budou jednotlivé položky lícovat pod sebou.

Při novém stanovení šířky boxu položky na šířku w_i je potlačeno hlášení `Underfull hbox`. Může se tedy stát, že TEX například vytvoří *bez varování* položku s obludnými mezislovními mezerami. Obvykle ale bývají v boxech položek pružné výplňky typu ⟨*glue*⟩, které jsou často specifikovány ve schématu položky a kterými programátor maker řídí způsob umístění položky. Například na řádku 159 je řečeno, že na konci každé položky z prvního sloupce bude `\hfil`, takže položky budou zarovnány

na levou zarážku. V druhém sloupci je proměnlivý text obehnán výplňkem `\hfil` z obou stran, takže tento text se bude centrovat, zatímco opakovaný text „konec“ bude usazen přesně pod sebou. □

• **Registr `\tabskip`.** Jak jsou stanoveny hodnoty mezisloupcových výplňků typu ⟨*glue*⟩ označované výše jako t_0 až t_n? Tyto hodnoty se získají z hodnoty registru `\tabskip` v době čtení deklarace řádku. V našem příkladě na řádku 159 je tato hodnota rovna `2em plus 5em`, to znamená, že t_0, t_1 i t_2 budou mít tuto hodnotu. Primitiv `\halign` může před otevírací závorkou mít určenu šířku tabulky klíčovým slovem `to` nebo `spread`, jak to známe ze specifikací boxů. Pokud je tato specifikace uvedena, pak začne pracovat roztahovatelnost nebo stahovatelnost příslušných mezisloupcových výplňků t_i tak, aby například při `\halign to10cm{...}` byla celková šířka každého řádku (a tím i celé tabulky) 10 cm.

Zbývá otázka, jak zařídit, aby se jednotlivé hodnoty t_0 až t_n od sebe lišily. K tomu si budeme muset upřesnit algoritmus čtení deklarace řádku v okamžiku, kdy TEX zahajuje zpracování `\halign`. Při tomto čtení se *neprovádí expanze* a jednotlivé řady tokenů odpovídající jednotlivým schématům položek si TEX uchovává ve vyhrazené paměti pro použití při pozdějším čtení datové části. Při čtení deklarace řádku si TEX speciálním způsobem všímá pouze těchto tokenů: `\cr` pro konec čtení deklarace řádku, `&` pro oddělení jednotlivých schémat položek (token kategorie 4), `#` pro místo, kam vložit proměnlivý text položky (token kategorie 6), `\tabskip` pro změnu registru `\tabskip` mezi jednotlivými položkami a konečně `\span`, pokud je nutno následující token v tuto chvíli expandovat. Za tokenem `\tabskip` výjimečně probíhá expanze tak dlouho, až hlavní procesor získá celou hodnotu podle syntaktického pravidla ⟨*glue*⟩, kterou přiřadí do registru. Celá část tvaru `\tabskip`⟨*equals*⟩⟨*glue*⟩ je ze schématu položky vyhozena jako zbytečná. Nicméně vnitřní registr `\tabskip` už byl změněn a o to šlo. Hodnota t_0 je pak rovna hodnotě `\tabskip` v okamžiku vstupu do konstrukce `\halign` a hodnota t_i je rovna hodnotě `\tabskip` v okamžiku ukončení čtení i-tého schématu položky. Uvedeme si příklad:

```
172 \tabskip=0pt % to je sice implicitní, ale jistota je jistota.
173         % (pro tyto účely lze použít zkratku v plainu: \ialign)
174 \halign to\hsize{\tabskip=0pt plus1fil       #\hfil
175                  &\tabskip=0pt          \hfil #\hfil \cr
176     toto bude  & a toto bude zase\cr
177     vlevo      & vpravo\cr
178     na stránce & na stránce\cr}
```

Uvedený kód nám dá:

toto bude	a toto bude zase
vlevo	vpravo
na stránce	na stránce

V tomto příkladě je t_0=0pt, t_1=0pt plus1fil a t_2=0pt. Poznamenejme, že po sestavení tabulky se hodnota \tabskip vrací do hodnoty t_0, protože všechny ostatní změny tohoto registru proběhly lokálně v rámci skupiny \halign{...}. □

Uvedeme si několik příkladů, které jsou zajímavým důsledkem skutečnosti, že v době čtení deklarace řádku se neprovádí expanze. Chceme zařadit pod sebe desetinná čísla, aby byla zarovnána podle desetinné čárky:

```
179 \halign{\catcode`\,=4\hfil #,&#\hfil &\quad --- #\hfil \cr
180   127,34    & tady je text, kde čárka nevadí \cr
181     3,1415  & vidíme, že čísla mají čárku pod sebou \cr}
```

A na výstupu skutečně máme:

```
127,34    — tady je text, kde čárka nevadí
  3,1415  — vidíme, že čísla mají čárku pod sebou
```

Ukážeme si, jak to funguje. Při čtení deklarace řádku se do prvního schématu uloží

182 catcode ` 12 , = 12 4 12 hfil # 6 , 12

Čárka má ve schématu kategorii 12, protože se zatím neprovedlo přiřazení \catcode. K tomuto přiřazení dojde, až TEX otevře \hbox první položky a zpracuje uvedenou sadu tokenů ze schématu. Proto uživatelova čárka v datové oblasti má stejnou funkci, jako separátor mezi položkami. První položka má tedy proměnlivý text „127“ a druhá „34“. Další čárky uvedené ve třetí položce (a případně i ve druhé) už mají zase kategorii 12, protože se uzavřením boxu první položky ukončila skupina.

Stejný výsledek bez změny kategorií můžeme dosáhnout též následujícím trikem:

```
183 \def\cislo#1,{#1,&}
184 \halign{\hfil\cislo#&#\hfil &\quad --- #\hfil \cr
185   127,34    & tady je text, kde čárka nevadí \cr
186     3,1415  & vidíme, že čísla mají čárku pod sebou \cr}
```

Makro \cislo bude expandovat až při sestavování položky a nikoli v době načítání deklarace řádku. □

V dalším příkladě budeme ve schématu položky znovu přepínat kategorie a umožníme uživateli psát v rovnicích pouze znak =, který bude zarovnán pod sebou. Nepotřebujeme používat LATEXovské konstrukce typu &=&. Příklad následuje i s ukázkou, jak to dopadne, ovšem už bez vysvětlování. Všimněte si, že je zde použita smyčka v deklaraci řádku (příznak návratu má první schéma), takže uživatel může psát vedle sebe libovolné množství rovnic (pokud se mu ovšem vejdou na řádek).

```
\let\eq== \thickmuskip=10mu % aby kolem bin.rel. bylo více místa
$$\vbox{\halign{&\catcode`\==4 \hfil$#$&$\{}=#$\hfil\qquad\cr
      z^2 =  x^2+y^2      & a+b    = c+d+e          & \sigma\cr
\sum_{i\eq0}^\infty x_i=T & \alpha = \beta = \gamma & \tau \cr}}
$$
```

$$\begin{aligned} z^2 &= x^2+y^2 & a+b &= c+d+e & \sigma \\ \textstyle\sum_{i=0}^{\infty} x_i &= T & \alpha &= \beta = \gamma & \tau \end{aligned}$$

Vidíme, že každá legrace něco stojí. Nemusíme sice psát `&=&`, zato ale v případě potřeby symbolu `=` v jiném smyslu, než pro zarovnání (viz index sumy), musíme použít náhradní symbol. Zde jsme zavedli symbol `\eq`. Kdybychom použili řešení bez změny kategorie (podobně jako v příkladě se zarovnáváním podle desetinné čárky), pak můžeme psát i `\sum_{i=0}`. Zase ale budeme mít problémy se sloupcem `\sigma` a `\tau`. □

• **Příklad na automatické generování schémat položek.** V tomto příkladě ukážeme makra, která umožní jisté pohodlí při zadávání matic. Uživatel napíše:

```
$$\matice{cc|r}{1&p& -1\\ p-1&2& 0} \sim
  \matice{cc|c}{1&p& -1\\ 0&(p-2)(p+1)& 1-p}. $$
```

a na výstupu dostane:

$$\left(\begin{array}{cc|c} 1 & p & -1 \\ p-1 & 2 & 0 \end{array}\right) \sim \left(\begin{array}{cc|c} 1 & p & -1 \\ 0 & (p-2)(p+1) & 1-p \end{array}\right).$$

Vidíme, že uživatel stanoví v jednoduché hlavičce vlastnosti sloupců (`c`: centruj, `r`: zarovnej doprava a `|`: nakresli svislou čáru). To známe z LaTeXu. Naším úkolem bude přečíst tyto údaje a vygenerovat odpovídající schéma pro `\halign`. Navrhneme proto následující makro:

```
\newtoks\schemata \newdimen\tbs
\def\addschema#1{\schemata\expandafter{\the\schemata#1}}
\def\addseparator{\ifx\params\empty \else \addschema&\fi}
\def\vemznak#1#2;{\def\params{#2}\csname P:#1\endcsname}
\expandafter\def\csname P:c\endcsname{%
   \addschema{\hfil$##$\hfil}\addseparator}
\expandafter\def\csname P:r\endcsname{%
   \addschema{\hfil$##$}\addseparator}
\expandafter\def\csname P:|\endcsname{%
   \addschema{\vrule\hskip2\tbs}}
\def\genschemata{\ifx\params\empty
   \else\expandafter\vemznak\params;\expandafter\genschemata\fi}
\tbs=.4em  \let\\=\cr
\def\matice#1#2{\left(\vcenter{\offinterlineskip
```

```
208     \schemata={\tabskip=2\tbs \strut}
209     \def\params{#1}\genschemata \tabskip=\tbs
210     \halign{\span\the\schemata \tabskip=\tbs \cr #2\crcr}}\right)}
```

⟨*Deklaraci řádku*⟩ pro \halign postupně vytvoříme v proměnné \schemata. Přitom uživatelovy parametry máme v makru \params. Pomocné makro \addschema přidá do proměnné \schemata co potřebujeme a \addseparator přidá oddělovač &$_4$. Makro \vemznak z řádku 197 vezme z parametrů \params další znak a expanduje na P:⟨*znak*⟩. Právě tato makra udělají potřebnou práci; například P:c přidá do proměnné \schemata text: „\hfil$#$\hfil&“.

Makro \genschemata opakovaně bere token z \params a přidává odpovídající úseky do proměnné \schemata. Na řádku 210 je obsah proměnné \schemata použit primitivem \halign. Výchozí hodnota této proměnné (řádek 208) obsahuje vertikální podpěru (\strut), protože jednotlivé řádky budou na sebe těsně navazovat. Změnou definice \strut můžeme změnit řádkování v matici. Konečně změnou údaje \tbs (řádek 206) měníme poloviční vzdálenost mezi sloupci. Registr \tabskip je totiž na okrajích matice roven \tbs a mezi sloupci má hodnotu 2\tbs. □

• **Výjimky z pravidel daných schématy položek.** V datové části potřebujeme občas vložit položku, která nepodléhá formátování podle schématu položky. V jednoduchých případech stačí, když do místa položky vložíme výplněk typu ⟨*glue*⟩, který spolupracuje s výplňky, použitými ve schématu položky. Například, pokud ve schématu položky máme zarovnávání nalevo implementováno jako #\hfil, pak položka \hfil A bude výjimečně centrovaná a položka \hfill A nám odcestuje doprava.

Mocnějším nástrojem je primitiv \omit, který musí po expanzi proměnlivé části položky přicházet (po ignorování mezer) jako první. Pokud je tomu tak, pak se box položky vyplní pouze materiálem z datové části a odpovídající schéma položky je ignorováno.

Nejmocnějším nástrojem je primitiv \span. S ním jsme se už setkali při výkladu způsobu čtení schémat položek na straně 132. Jeho výskyt v datové části \halign má ale docela jiný (a pro nás podstatně zajímavější) význam. Sekvence \span zde může nahradit libovolný separátor typu &. Tiskový materiál pro sousední položky spojené primitivem \span pak má společný sdílený box. V případě, že není použit \omit, obsahuje materiál společného boxu nejen datovou část položek, ale i schémata příslušných položek. Bývá obvyklé spojit \span s \omit. Například:

```
211  .. & \omit \span \omit \span \omit Tento text obsazuje prostor
212   tří za sebou jdoucích položek, jsou ignorována schémata& ..\cr
```

Místo \omit\span...\omit lze použít stručnější zápis pomocí makra plainu \multispan (viz část B).

TEX může mít jisté potíže vložit materiál položky vytvořený pomocí `\span` do struktury boxů zarovnaných pod sebou o šířkách w_1 až w_n. Předpokládejme například, že materiál ze `\span` nahrazuje položky druhého až čtvrtého sloupce, tedy nahrazuje boxy o šířkách w_2, w_3 a w_4. Dále nechť g_2 a g_3 jsou základní rozměry (tj. bez roztažení a stlačení) z `\tabskip` brané z hodnot t_2 a t_3. Spočítejme si šířku místa, kam by měl být vložen box s materiálem ze `\span` v našem příkladě (položky 2 až 4):

$$W = w_2 + g_2 + w_3 + g_3 + w_4$$

Pokud přirozená šířka boxu s materiálem z našeho `\span` je menší nebo rovna hodnotě W, pak se do výsledného řádku tabulky zařadí tento box, ale přepočítán na šířku W (pracují pružné výplňky uvnitř boxu). Tento box nahrazuje v řádku tabulky obvyklý materiál typu: „box w_2, t_2, box w_3, t_3 a box w_4“. Pokud dále dojde ke stlačení či roztažení výplňků `\tabskip` v místě t_2 a t_3 (protože je použit `\halign to` nebo `\halign spread`), pak se dodatečně upraví nová šířka našeho boxu ze `\span` a ke slovu se znovu dostanou pružné výplňky uvnitř tohoto boxu.

Výše popsané chování je v souladu s tím, co od konstrukcí `\span` můžeme očekávat. Problémy ovšem nastávají, pokud přirozená šířka materiálu ze `\span` je *větší* než rozměr W. Pak totiž dojde k nejhoršímu a TEX zvětší nějakou hodnotu w_i tak, aby rozměr W byl roven přirozené šířce materiálu ze `\span`. Zjistit, na kterém sloupci bude toto ukrutné násilí provedeno, je možné prostudováním poměrně složitého algoritmu z TEXbooku ze strany 245. Většinou nás to ale neuklidní, protože ten sloupec obvykle za nic nemůže. TEX provádí popsanou změnu hodnoty w_i dříve, než začne vkládat první řádek tabulky do výsledného vertikálního seznamu. Tím není narušen základní požadavek, aby položky jednotlivých sloupců lícovaly pod sebou. S problémem se ještě setkáme níže v našich ukázkách. □

• **Čtení, expanze a zpracování položek v datové části.** Než TEX začne expandovat první část schématu položky, vyžádá si z datové části položky (po přeskočení mezer) jeden token od expand procesoru. Z tohoto tokenu se může vyklubat (po expanzi) `\omit` nebo `\noalign`. V takovém případě víme, že se TEX bude chovat jinak. Pokud se tak nestalo, TEX začne expandovat a v hlavním procesoru provádět první část schématu, pak naváže na již expandovaný token z datové části a dále se pustí do čtení a expanze případného dalšího obsahu položky.

Jakmile se při zpracování datové části položky objeví token kategorie 4 nebo `\cr` nebo `\span`, TEX *okamžitě* nahradí tento token druhou částí schématu. K této události může dojít na různých úrovních zpracování vstupní informace. Například pokud se makro v datové části položky expanduje na token $\boxed{\&}_4$, pak TEX přejde k druhé části schématu po expanzi tohoto makra. Pokud je ale načítán text parametru makra a v tomto textu se vyskytne $\boxed{\&}_4$, pak je už v průběhu načítání parametru provedena záměna za druhou část schématu a čtení parametru pokračuje

z druhé části schématu. Do parametru tedy token $\boxed{\&}_4$ nevstupuje. Tuto skutečnost ilustrujeme na příkladě:

```
\def\a#1\zarazka{\message{#1}} \let\zarazka=\relax
\halign{\a X#Y\zarazka & #Z\zarazka\cr
  abc& de\cr
  fgh& i\a jkl\cr
  mno& \a pqr\cr
  stu& xyz\cr}
```

První sloupec je bezproblémový. Tam se při čtení parametru makra `\a` (zapsaného v první části schématu) plynule přejde na datovou část položky a po dosažení tokenu $\boxed{\&}_4$ zpět na druhou část schématu. Takže do parametru `#1` vstupuje například z prvního řádku text „`XabcY`“. Druhý sloupec je podstatně zajímavější. V prvním řádku druhého sloupce se ještě neděje nic neobvyklého a `\zarazka` ve schématu pracuje jako `\relax`. V druhém řádku druhého sloupce se při načítání parametru makra `\a` promění token `\cr` v druhou část schématu, takže do `#1` vstupuje „`jklZ`“. Ve třetím řádku druhého sloupce nám to havaruje. Je to tím, že první token z datové části (v tomto případě se jedná o makro `\a`) je expandován dřív, než vůbec začne zpracování položky. V té chvíli ještě není `\cr` ani $\boxed{\&}_4$ nijak interpretováno. Do parametru `#1` se tedy dostává `pqr\cr stu& xyz\cr}` a obdržíme chybu `Argument of \a has an extra }`. Abychom tento problém odstranili, definujme makro `\a` jinak:

```
\def\a{\relax\A} \def\A#1\zarazka{\message{#1}}
```

□

Při načítání parametru makra v datové části položky se při výskytu `&` nebo `\cr` provede přechod do závěrečné části schématu položky jen za podmínky, že se nezpracovává vnořená skupina. Smysl této dodatečné podmínky osvětlí příklad:

```
$$\matrix{\pmatrix{1&2\cr 3&4}\cr \pmatrix{5\cr 6}}$$
%                    ^  ^   ^                  ^
% Označené "&" a "\cr" neukončí položku vnější \matrix, protože
% uvnitř skupiny je potlačeno ukončení čtení položky.
```

Vidíme, že díky potlačení konce položky uvnitř skupiny je možno jako text parametru makra `\pmatrix` zavést „`1&2\cr 3&4`“. Lze tedy psát vnořené tabulky typu `\halign` i prostřednictvím maker, které si do parametru načtou datovou část všech položek současně. V našem příkladě je takovým makrem `\pmatrix` z plainu. □

• **Tabulky s linkami `\vrule`, `\hrule`.** V závěru této sekce se zaměříme na způsoby, jakými se do tabulek `\halign` zavádějí linky. Uvidíme, že věc není na první pohled vůbec jednoduchá. Dovednost vytvářet takové tabulky je prvním stupněm umění makrojazyka TEXu. Vrcholným uměním pak je dovednost vytvářet makra tvořící rozhraní mezi primitivem `\halign` a uživatelem, který nechce nebo nemůže

přemýšlet v kategoriích „box, kern, glue“. Příkladem takového umění je makro pro prostředí `tabular` v LaTeXu nebo ještě propracovanější makra pro tabulky od Michaela Spivaka.

Nejprve uvedeme pravidla, podle kterých TeX zachází s linkami `\hrule` a `\vrule` při sestavování tabulky. Pokud se v boxech položek vyskytovaly v první hladině (tj. nikoli uvnitř dalších boxů) linky `\vrule` s neurčitou výškou nebo hloubkou, jsou tyto údaje dopočítány na výšku a hloubku řádku tabulky jako celku.

Pokud se vyskytovaly ve ⟨*vertikálním materiálu*⟩ z `\noalign` linky `\hrule` s neurčitou šířkou, pak po kompletaci tabulky získají tyto linky definitivní šířku rovnou šířce tabulky a dalšímu případnému prodlužování do šířky vnějšího `\vbox`u nepodléhají. □

Svislé linky se většinou do tabulek zařazují jako `\vrule` uvnitř schémat položek. Vodorovné linky se do tabulek obvykle vkládají pomocí `\noalign{\hrule}`. To nám ovšem zcela rozhodí řádkování — viz náš první příklad na straně 129. Vkládat do `\noalign` ještě další mezery (jako ve zmíněném prvním příkladě, řádek 160) není vhodné řešení, protože bychom neměli napojeny vertikální linky nad sebou vytvořené v jednotlivých položkách pomocí `\vrule`. Z tohoto důvodu je bezpodmínečně nutné použít při sazbě tabulek s čárami makro `\offinterlineskip`, které nastavuje `\baselineskip` na nulu a dále je nutno vypodložit každý řádek tabulky neviditelnou podpěrou typu `\strut`. Takovou podpěru zařadíme do schématu některé položky. Bez této podpěry bychom měli řádky tabulky (1) velmi nízké, (2) byly by přilepeny jeden na druhý a (3) nebyly by stejně vysoké. □

Tabulka vytvořená pomocí `\halign` může být zlomena „v půli“ stránkovým zlomem. Je to z toho důvodu, že výsledek práce `\halign` jsou jednotlivé řádky vkládané do vertikálního seznamu, přičemž tyto řádky mají mezi sebou ⟨*glue*⟩ z `\baselineskip` nebo z `\lineskip`. V těchto místech může dojít ke stránkovému zlomu. Abychom tomu zabránili, dáme obvykle celou tabulku do `\vbox`u.

Jestliže vkládáme za každý řádek `\noalign{\hrule}`, tabulka se nikdy nezlomí. V takovém případě totiž mezi řádky chybí zmíněný výplněk ⟨*glue*⟩, ve kterém by se mohla tabulka zlomit. Chceme-li dovolit stránkový zlom, pišme třeba `\noalign{\hrule\penalty50}`. Pokud dovolíme stránkový zlom uvnitř tabulky, máme vodorovnou čáru obvykle za posledním řádkem předchozí strany, ale už ne nad prvním řádkem strany následující. A už vůbec se nám na následující straně neopakuje hlavička. Řešení tohoto problému uvádíme na jiném místě této knihy (viz sekci příkladů výstupních rutin 6.9, strana 268). □

Půjdeme rovnou na věc a ukážeme si sazbu tabulky s jednoduchými i dvojitými linkami.

Měsíc	Prodej zboží	
	A	B
Leden	865	16
Únor	917	8
Březen	1036	18

Tato tabulka byla vytvořena takto:

```
\bgroup\offinterlineskip
\def\vvrule{\vrule \kern1.4pt \vrule}
\def\\{\crcr\omit\vvrule\hrulefill\vvrule&
            \multispan2\hrulefill\vvrule\cr}
\def\vkern{\multispan3\vrule height1.2pt \hfil \vrule}
\def\strut{\vrule height11pt depth 5pt width0pt }
\halign {\vvrule\quad#\hfil\quad\vvrule &\quad\hfil#\quad\vrule
       &\quad\hfil#\quad\vvrule\strut \cr  % konec deklarace
\noalign{\hrule} \vkern\\
& % první položka (vlevo od "Prodej zboží") je prázdná
\multispan2\hfil\quad\bf Prodej zboží\quad\hfil \vvrule\strut\cr
\hfil\vbox to0pt{\vss\hbox{\bf Měsíc}\vss}%
        &\multispan2\hrulefill \vvrule\cr
% Vlastní data --------------------
           & \bf A\hfil & \bf B\hfil \\ \vkern\\
% -------------------------------
    Leden  &         865&    16\\
    Únor   &         917&     8\\
    Březen &        1036&    18\\
\vkern\cr\noalign{\hrule}}\egroup
```

Makro `\vvrule` je použito pro dvojitou vertikální linku, která je v každém řádku tabulky použita vlevo, vpravo a mezi prvním a druhým sloupcem. Za pozornost stojí makro `\\`, které ukončí řádek a vytvoří pod ním horizontální linku. Tato linka není vedena přes celou šířku tabulky — to by stačilo použít `\cr\noalign{\hrule}`. Místo toho je horizontální linka přerušena v místě dvojitých vertikálních čar. Proto je definována jako samostatný řádek tabulky, ve kterém je vodorovná čára vykreslena po úsecích jako `\hrulefill`. Všimneme si, že tento řádek tabulky má výšku pouze 0,4 pt, takže makra `\vvrule`, která jsou v něm použita, vykreslí jen miniaturní čtverečky. Makro `\vkern` použijeme pro mezeru ve dvojitých horizontálních linkách. Zde požadujeme, aby se nakreslily kratičké vertikální linky na levém a pravém okraji tabulky a nic víc. Konečně `\strut` vytváří neviditelnou podpěru v každém řádku a volbou výšky a hloubky této podpěry regulujeme velikosti všech řádků tabulky.

Vlastní tabulka se skládá ze tří sloupců. Ke schématům položek jednotlivých sloupců asi není co dodat. Poněkud komplikovanější je ovšem sazba záhlaví, protože zde máme méně standardní strukturu, která není přímo implementována do `\halign`. Záhlaví sestává ze tří řádků, ačkoli čtenář vysázené tabulky vidí řádky dva. V prvním řádku je sloupec „Měsíc“ prázdný a text „Prodej zboží“ je zaveden jako `\span` přes dva následující sloupce. Protože v použitém `\multispan` je skryto i `\omit`, je potřeba postarat se ručně o koncové `\vvrule\strut`, viz řádek 234 v kódu. Druhý řádek tabulky má pouze výšku vodorovné linky (tj. 0,4 pt) a obsahuje v první položce text „Měsíc“ centrovaný vertikálně na střed tohoto řádku. Další dvě položky tohoto řádku jsou vyplněny jako `\span` obsahující natahovací vodorovnou linku `\hrulefill`. Tento trik už známe z makra `\\`. Na třetím řádku tabulky je znovu přeskočena oblast „Měsíc“ a pokračuje záhlaví A a B. Pak už následují data zapsaná běžným způsobem.

Zkusíme nyní natáhnout tabulku na šířku sazby, takže pišme `\halign to\hsize`. Hodnoty t_0 a t_n necháme nulové, zatímco ostatní hodnoty `\tabskip` mezi jednotlivými sloupci budou natahovací. Pokud to uděláme bez další úpravy, dočkáme se nepříjemného výsledku:

Měsíc	**Prodej zboží**	
	A	**B**
Leden	865	16
Únor	917	8
Březen	1036	18

Vidíme, že některé čáry kreslené pomocí `\hrulefill` nám nezasahují do prostoru, ve kterém pruží `\tabskip`. Dále šíře prvního sloupce zůstala stejná, zatímco ostatní sloupce se vizuálně protáhly. Čtenář si sám rozmyslí, proč se tak stalo.

Abychom překonali tento problém, musíme vnitřní koncepci naší tabulky zcela přepracovat. Místo tří sloupců budeme pracovat s osmi sloupci podle následujícího náčrtu:

V prvním řádku tohoto náčrtu je vyznačeno číslo sloupce, ve druhém je zakreslen charakteristický obsah odpovídajícího sloupce a ve třetím řádku jsou uvedeny velikosti `\tabskip` mezi jednotlivými sloupci. Vidíme, že vlastní data budeme zapisovat do druhého, pátého a sedmého sloupce. V prvním, čtvrtém a osmém sloupci bude kreslena dvojitá vertikální čára a v šestém sloupci čára jednoduchá. Ve třetím sloupci nebude nikdy kresleno nic — tento sloupec bude použit jako záchytný

bod pro horizontální linky \hrulefill. Tyto linky budou totiž v místě „Měsíc“ implementovány jako \span přes 1. až 3. položku a v místě „Prodej zboží“ budou kresleny pomocí \span přes 4. až 8. položku. Takto definované linky budou obsahovat všechny pružné části tabulky, pocházející z mezisloupcových \tabskip a budou se tedy správným způsobem natahovat. Makro pro naši tabulku nyní vypadá takto:

```
\bgroup\offinterlineskip
\def\vvrule{\vrule \kern1.4pt \vrule}
\def\\{\crcr\multispan3\vvrule\hrulefill&
            \multispan5\vvrule\hrulefill\vvrule\cr}
\def\vkern{\multispan8\vrule height1.2pt \hfil \vrule}
\def\strut{\vrule height11pt depth 5pt width0pt }
\halign to\hsize{\tabskip=0pt plus 1fil\vvrule#&\quad#\hfil\quad
   &\tabskip=0pt#&\vvrule#\tabskip=0pt plus1fil&\quad\hfil#\quad
   &\vrule#&\quad\hfil#\quad &\vvrule#\strut\tabskip=0pt\cr
\noalign{\hrule} \vkern\\
&&&&\multispan3\hfil\quad\bf Prodej zboží\quad\hfil &\cr
&\hfil\vbox to0pt{\vss\hbox{\bf Měsíc}\vss}%
        &&\multispan5\vvrule\hrulefill\vvrule\cr
% Vlastní data --------------------
&           &&& \bf A\hfil && \bf B\hfil &\\ \vkern\\
% --------------------------------
&    Leden  &&&         865&&    16&\\
&    Únor   &&&         917&&     8&\\
&    Březen &&&        1036&&    18&\\
\vkern\cr\noalign{\hrule}}\egroup
```

Měsíc	Prodej zboží	
	A	B
Leden	865	16
Únor	917	8
Březen	1036	18

Myslím, že teprve v této ukázce si čtenář povšiml další záludnosti. Nadpis **B** není uprostřed sloupce! Je to tím, že přirozená šířka položky „Prodej zboží“ je větší, než součet přirozených šířek sloupců A a B a hodnot \tabskip mezi nimi. Přirozená šířka \tabskip je totiž 0 pt. Jak jsme uvedli na straně 136, TEX v takových situacích pozmění nějakou hodnotu w_i. V našem případě změnil hodnotu w_7 odpovídající šířce sloupce B. Zatímco v ukázce na straně 139 nám to vůbec nevadilo a dokonce nám tato změna šířky sloupce B připadala přirozená, zde to bije do očí, protože prostor v tabulce je opticky větší. Abychom překonali tento poslední problém, použijme na řádku 254 makro plainu \hidewidth následujícím způsobem:

```
264 &\multispan3\hfil\hidewidth\bf Prodej zboží\hidewidth\hfil&\cr
```

Tím bude mít náš `\span` přirozenou šířku zápornou a problém odpadne. Jiným řešením je vložit `\tabskip` s dostačující přirozenou šířkou, aby šířka `\span` byla menší než napočítaná hodnota W. V naší ukázce stačí volit `\tabskip=20pt plus1fil`. □

V TEXu je možné také vkládat více konstrukcí typu `\halign` do sebe. Jinak řečeno: každá položka tabulky může obsahovat libovolné množství dalších tabulek. Aby toho nebylo dost, existuje ještě primitiv `\valign`, který se chová stejně, jako `\halign` na vstupu, ale vyrábí transponovanou tabulku. Jednotlivé položky se vkládají do `\vbox`ů, `\cr` ukončuje sloupce a `\vbox`y položek se přepočítají na maximální výšku v každém řádku. Hloubky těchto boxů jsou nulové, protože se nejprve provede posun účaří na spodní část každého boxu (to je jediný algoritmus, který nemá v případě `\halign` obdobu). Konečně `\valign` vrací hotové sloupce do horizontálního seznamu oddělené od sebe materiálem z `\noalign`. Přitom položky ve sloupcích jsou od sebe odděleny výplňky typu ⟨*glue*⟩ podle `\tabskip`.

V posledním příkladě této sekce ukážeme použití `\valign`. Zůstaneme u naší oblíbené tabulky a přidáme jí ještě jeden poněkud větší řádek. Každá položka tohoto řádku bude obsahovat kratší vodorovnou linku a nad ní i pod ní bude samostatný odstavec. Odstavce budou mít centrované řádky, ba co víc, horní odstavce budou centrovány podél vodorovné osy jako celek a spodní odstavce budou mít na vodorovné pomyslné lince společný první řádek. Přitom chceme tuto konstrukci udělat nezávislou na množství textu, který se v odstavcích může vyskytnout. K tomu se hodí `\valign`. Tabulka:

Měsíc	**Prodej zboží**	
	A	**B**
Leden	865	16
Únor	917	8
Březen	1036	18
Shrnutí **Závěr**	V případě zboží A vidíme poměrně rychlý nárust obratu zvláště v březnu. Zdá se, že to bude tím, že se v tuto dobu oteplilo. **Ponechat a dále sledovat vývoj.**	Pokud jde o zboží B, je to mizerné. **Zrušit výrobu**

byla vytvořena stejným způsobem, jako je uvedeno na řádcích 244–263, jen přibyla nová definice pro formátování jednotlivých odstavců:

```
\def\setparams{\parindent=0pt \hsize=.27\hsize
  \baselineskip=12pt \lineskiplimit=-10pt
  \leftskip=0pt plus3em \rightskip=0pt plus3em
  \parfillskip=0pt}
\let\hi=\hidewidth
```

a místo řádku 263 je napsáno:

```
\hbox{\setparams \valign{\vfil\noindent\strut#\strut\vfil
                    &\hrule\bf\noindent\strut#\strut\vfil\cr
      \bf Shrnutí & Závěr\cr
       V případě zboží A vidíme poměrně rychlý
          nárust obratu zvláště v březnu. Zdá se, že
          to bude tím, že se v tuto dobu oteplilo.&
          Ponechat a dále sledovat vývoj.\cr
       Pokud jde o zboží B, je to mizerné.&
          Zrušit výrobu\cr }% konec \valign
  \global\setbox3=\lastbox \global\setbox2=\lastbox
  \global\setbox1=\lastbox}%
& \hi\box1\hi &&& \hi\box2\hi && \hi\box3\hi &\\
\vkern\cr\noalign{\hrule}}\egroup % konec \halign a celé skupiny
```

Zde se nejprve nanečisto vyrobí \hbox se sloupečky tak, jak je vytvořil \valign. Pak se tyto sloupečky posbírají do boxů \box1 až \box3 a tyto boxy se umístí do položek hlavní tabulky \halign. Kolem každé položky jsme vložili \hi, což jsme definovali jako zkratku za \hidewidth. Tím máme zaručeno, že tyto položky budou mít zápornou přirozenou šířku a neovlivní výpočet maxima šířky boxu ve sloupci. Kdybychom to neudělali, číselné údaje, které jsme sázeli na pravou zarážku ve sloupcích A a B, by byly posunuty doprava podle pravé hrany odstavce. Nám se víc líbí, že jsou uprostřed sloupce, i když na pravou zarážku. □

5. Matematická sazba

5.1. Matematický seznam

I při výkladu matematické sazby se budeme držet pravidla „psaní naruby“, tj. nejprve vyložíme, jak věci fungují, a teprve později ukážeme použití některých maker a metody psaní obtížnějších vzorečků (v sekci 5.5). Pokud čtenář potřebuje vysázet třeba:

$$\frac{1}{\pi}\int_0^{\pi}\cos(n\varphi - z\sin\varphi)\,\mathrm{d}\varphi = \sum_{p=0}^{\infty}\left(\frac{z}{2}\right)^{n+2p}\frac{(-1)^p}{p!\,(n+p)!}$$

a neví, že může psát například:

```
\let\phi=\varphi % místo znaku ϕ budu používat znak φ
$$ {1\over\pi} \int_0^\pi \cos(n\phi - z\sin\phi)\,{\rm d}\phi =
   \sum_{p=0}^\infty
        \left(z\over2\right)^{n+2p} {(-1)^p\over p!\,(n+p)!} $$
```

pak je třeba takového čtenáře upozornit na to, že existuje mnoho příruček pro začátečníky (například [3], [6], [8]). Z nich se tyto základy naučí. V mnohých případech se schopnost TEXu pracovat s matematickou sazbou využívá i jinde než jen při tvorbě matematických vzorečků. Proto zvládnutí této problematiky patří k „malé násobilce“ každého TEXisty. □

V matematickém módu vytváří hlavní procesor *matematický seznam*, který na rozdíl od horizontálního nebo vertikálního seznamu neobsahuje jednoznačné informace, na kterém místě má být co vysázeno. Skutečně, třeba po kompletaci `\hbox`u získají i pružné výplňky v horizontálním seznamu definitivní velikost. Takový seznam lze rovnou zapisovat do výstupu v `dvi`, protože všechny rozměry jsou známy. Na druhé straně matematický seznam zaznamenává logickou strukturu matematické sazby (tj. například co je indexem čeho), místo aby se staral o konečné umístění elementů sazby (tj. například o kolik posunout níže box se sazbou indexu).

Jakmile TEX ukončí matematický mód (dosáhne ukončovací značky `$` nebo `$$`), provede nový průchod nad právě vytvořeným matematickým seznamem a konvertuje jej do horizontálního seznamu. Vyjadřovací prostředky jsou pak už jen `\hbox`, `\vbox`, o kolik nahoru nebo dolu, `\kern` a ⟨*glue*⟩. Ztrácí se informace o logické struktuře sazby, ale seznam je použitelný pro výstup do `dvi`. Pomocí `\showlists` můžeme do `log`u vypsat obsah jednotlivých seznamů. Napíšeme-li třeba

```
\showboxbreadth=20
$ 1 + x_i^{1-y} \showlists $ \showlists
```

pak první `\showlists` ukáže ještě matematický seznam s logickou strukturou, zatímco druhý `\showlists` nám už ukáže výsledný horizontální seznam, kde je použito termínů jako `\vbox shifted 2.78741`, nebo `\glue(\medmuskip) 2.22217 plus 1.11108 minus 2.22217`.

V dalším výkladu se nejprve zaměříme na popis matematického seznamu a způsob jeho vytváření. To bude obsahem této sekce. Teprve v následující sekci 5.2 naznačíme metody konverze tohoto seznamu do „skutečné sazby“ reprezentované horizontálním seznamem. □

Matematický seznam může obsahovat tyto elementy:

- Atom — základní nositel strukturované informace.
- Značky typu `\left`, `\right`, `\scriptstyle` atd. pro pozdější konverzi.
- Prvky z horizontálního seznamu (linky, boxy, penalty, `\discretionary`).
- Pružné výplňky (`\hskip`, `\hfil`, `\hss`, ..., `\mskip`, `\nonscript`).
- Pevné výplňky (`\kern`, `\mkern`).
- Značky jako v horiz./vert. seznamu. (`\write`, `\mark`, `\insert`, `\vadjust`).

Hlavní pozornost věnujeme nyní *atomům*, protože vložit linku do seznamu pomocí `\vrule`, nebo box pomocí `\hbox`, `\vbox` jsme se naučili už dříve. Každý atom sestává ze tří částí:

- *Základ* (nucleus) — objekt, ke kterému jsou připojovány indexy a exponenty.
- *Exponent* (superscript).
- *Index* (subscript).

Každá z těchto tří částí může obsahovat definitivní sazbu jednoho matematického znaku nebo může obsahovat matematický seznam. Právě druhá možnost umožňuje mít seznam atomů například v indexu, přičemž každý atom může mít další indexy a exponenty. Úroveň vnoření není omezena. Každá z částí atomu může být též prázdná, například:

```
$ x^2         % atom má základ x, exponent 2 a index je prázdný
  x_i^{a+b^2} % základ x, index i a exponent je matem. seznam
              % s atomy se základy a, +, b, poslední s expon. 2
  {}^a_b  $   % prázdný základ, exponent a, index b
```

Rozlišujeme atomy několika typů. Typ atomu určuje při pozdější transformaci do horizontálního seznamu mezerování mezi atomy v řadě vedle sebe a způsob umístění indexů a exponentů. Některé speciální typy atomů se sázejí specifickým způsobem

(například odmocniny). Následuje seznam všech typů atomů. V závorkách jsou uvedeny příklady.

0. Ord — běžné matematické značky (název proměnné x, y).
1. Op — velké matematické operátory ($\sum, \int, \lim$).
2. Bin — binární operátory mezi Ord ($+, -, \times$).
3. Rel — relace, kolem je z obou stran větší mezera ($=, <, \geq$).
4. Open — otevírací závorky ($\langle, \{$).
5. Close — zavírací závorky ($\rangle, \}$).
6. Punct — interpunkce (čárka, je uvedena ve všech předchozích ukázkách).
7. Inner — výsledek sazby s `\left`, `\right`.
8. Over — atom bude sázen s čarou nad ($\overline{x+y}$).
9. Under — atom bude sázen s čarou pod ($\underline{x+y}$).
10. Acc — atom bude opatřen specifikovaným akcentem ($\widetilde{xyz} + \hat{a}$).
11. Rad — atom bude sázen jako odmocnina ($\sqrt{x}$).
12. Vcent — sazba jako `\vbox` centrovaná na matematickou osu (`\vcenter`).

Atomy typu 0–6 jsou obvykle vytvářeny automaticky při činnosti povelů pro sazbu znaku. Pro další typy atomů jsou v TEXu rezervovány speciální primitivy. Nyní se zaměříme na povely pro sazbu znaku. Například, napíšeme-li:

```
11 $ 1-x^2 $
```

pak povel `1`$_{12}$ vytvoří atom typu Ord. Dále při činnosti povelu `-`$_{12}$ vzniká atom typu Bin a povel `x`$_{11}$ vytvoří atom typu Ord. Konečně povel `^`$_{7}$ zahájí tvorbu exponentu tohoto atomu, a `2`$_{12}$ říká, že exponent bude sázen jako dvojka. V dalším textu se zaměříme na otázku, jak TEX poznal, že se při `-`$_{12}$ má vytvořit atom typu Bin, zatímco při `1`$_{12}$ a `x`$_{11}$ vytváří atom typu Ord.

Každý povel hlavního procesoru, který vytváří nějaký znak sazby, nese v případě horizontálního módu jedinou informaci: pozici ve fontu, ze které se má právě nastaveným fontem znak vysázet. Pokud se jedná o sazbu znaku, je pozice ve fontu určena ASCII hodnotou tohoto znaku. V matematickém módu je věc podstatně komplikovanější.

Každý povel pro sazbu znaku v matematickém módu nese tři informace:

- třída matematického objektu (číslo v rozsahu 0 až 7, tři bity),
- rodina matematického fontu (číslo v rozsahu 0 až 15, čtyři bity),
- pozice znaku ve fontu (číslo v rozsahu 0 až 255, osm bitů).

Uvedené údaje lze seskupit do jediného patnáctibitového čísla, kde první tři bity zleva značí třídu, další čtyři rodinu a posledních osm pozici. Sazba každého znaku v matematickém módu je určena tímto patnáctibitovým číslem. Nastavení textového fontu nemá v matematickém módu žádný vliv.

Třída matematického objektu udává typ vytvářeného atomu podle předchozí tabulky typů atomů. Například, je-li při sazbě znaku použita třída 2, vytvoří se atom typu Bin. Třída 6 vytváří atom typu Punct.

Třída 7 je výjimečná. Má-li povel pro sazbu znaku tuto třídu, vytváří se atom typu Ord a TEX je informován o tom, že údaj o rodině matematického fontu není definitivní, ale jen implicitní. TEX tento údaj bude ignorovat a použije místo něj hodnotu z `\fam`, pokud v době činnosti povelu je registr `\fam` v intervalu $\langle 0, 15\rangle$. Při každém vstupu do matematického módu je tento registr nastaven na hodnotu -1, takže pokud tento registr v matematickém módu nezměníme, použije se implicitní informace.

Například při sazbě písmene je použita třída 7. Uvažujme:

```
12 $ f(x)\,{\fam=0 d}x $
```

Písmena f a x mají třídu 7 a budou se sázet podle implicitní rodiny fontů. Ta má v tomto případě číslo 1 a výsledek bude sázen matematickou kurzívou (proč, to ukážeme až v sekci 5.3). Písmeno „d“ bude ale sázeno podle rodiny fontů 0 (antikva), protože je tak nastaven registr `\fam`. Uživatel většinou použije makro `\rm`:

```
13 $ f(x)\,{\rm d}x $
```

Toto makro je definováno tak, že nastaví `\fam=0`, takže efekt je stejný jako v předchozí ukázce.

Protože jsme problematiku fontů v matematické sazbě odsunuli do sekce 5.3, nebudeme zde rozepisovat význam druhého údaje o rodině fontů. V tomto okamžiku se pouze spokojíme s tím, že sazba každého znaku v matematickém seznamu nese údaj o rodině fontů (číslo v rozsahu 0 až 15) a tento údaj později při konverzi na horizontální seznam rozhoduje o tom, jaký konkrétní font se pro sazbu znaku má použít. □

Primitiv pro sazbu znaku v matematickém módu vypadá takto:

```
14 \mathchar ⟨15-bit number⟩
```

kde ⟨*15-bit number*⟩ nese potřebné informace pro sazbu znaku, jak jsme uvedli výše. Číslo ⟨*15-bit number*⟩ zapisujeme obvykle v hexadecimálním tvaru, abychom mohli snadno odečíst jednotlivé údaje. Například:

```
15 \mathchar"2200
```

znamená, že se vysází znak třídy 2 (vytvoří se atom typu Bin), rodina fontu je 2 a pozice znaku ve fontu je 0. Jak později ukážeme, tento povel vysází v plainu binární operátor „minus“, protože ve fontu `cmsy10` je na pozici 0 tento znak.

Povel `\mathchar` se skoro nikdy nepoužívá. Místo toho se pomocí `\mathcode` přiřadí každému ASCII znaku nějaký patnáctibitový kód, určující sazbu matematického znaku, nebo se deklaruje uživatelská sekvence použitím `\mathchardef`. Plain například definuje:

```
16 \mathcode‘\-="2200
17 \mathchardef\sum="1350
```

což znamená, že při povelu $\boxed{-}_{12}$ v matematickém módu (na tom, zda je kategorie 11 nebo 12 nezáleží) se provede sazba znaku podle kódu `"2200` a vysází se tedy matematické minus. Proto v našem příkladě z řádku 11 na straně 146 se znak „-“ chová jako `\mathchar"2200` a TEX nejen ví, jakého typu má být vložený atom, ale vysází nakonec znaménko minus z fontu `cmsy10` z pozice 0, což nemá žádnou souvislost s právě nastaveným textovým fontem ani s ASCII hodnotou znaku minus. Když uživatel napíše v matematickém módu `\sum`, TEX vytvoří velký operátor (typ Op, třída 1), použije font `cmex10` (to souvisí s rodinou fontu 3 a sekcí 5.3) a znak vezme z pozice `"50` (hexadecimálně). Vznikne tím symbol sumy ($\sum$). Všechny matematické znaky, deklarované v plainu, jsou souhrnně uvedeny v sekci 5.4.

Před inicializací formátu (v iniTEXu) jsou každému ASCII znaku přiřazeny výchozí matematické kódy takto: `\mathcode`⟨*ASCII*⟩=⟨*ASCII*⟩, tj. třída nula, rodina nula a pozice ve fontu rovna ASCII hodnotě. Výjimku tvoří všechna malá i velká písmena anglické abecedy a číslice. Písmena mají v iniTEXu nastaven matematický kód na `"7100`+ASCII a číslice mají `"7000`+ASCII. Tyto hodnoty nejsou v plainu měněny. Proto sazba písmene v matematickém módu vytvoří atom typu Ord (třída 7) a použije se implicitní rodina 1, kterou lze změnit nastavením registru `\fam`. Podobně pro číslice, kde má implicitní rodina číslo 0. □

V další části této sekce vyjmenujeme všechny povely (na primitivní úrovni), které mají co do činění s tvorbou atomů v matematickém seznamu.

• **Sazba znaku.** O této problematice jsme právě mluvili, takže jen stručně. Sazba znaku se realizuje buď primitivem `\mathchar`, nebo sekvencí deklarovanou z `\mathchardef`, nebo zápisem znaku samotného. V posledním případě se ASCII hodnota znaku převede na matematický kód podle deklarace z `\mathcode`. TEX vytvoří atom takového typu, který odpovídá třídě dané v matematickém kódu a znak bude vložen do základu atomu. Exponent a index zůstávají (zatím) prázdné. □

• **Skupiny.** Při posloupnosti `{`⟨*matematický materiál*⟩`}` TEX vytvoří atom typu Ord. Výsledkem sazby ⟨*matematického materiálu*⟩ je nějaký matematický seznam,

který TEX vloží do vytvořeného atomu jako základ. Exponent s indexem ponechá (zatím) prázdný. Závorky mají dvojí význam, neboť navíc otevírají skupinu. Všechna případná přiřazení jsou tedy ve skupině lokální. Závorky lze nahradit zástupnou sekvencí deklarovanou pomocí `\let` (např. `\bgroup` a `\egroup`). Je-li výsledkem sazby ⟨*matematického materiálu*⟩ jediný atom typu Ord, je tento atom považován za výsledek sazby celé skupiny (tj. nevzniká atom typu Ord, jehož základem by byl atom typu Ord.). Příklad:

```
18 $ {a+b} % vytvoří se atom typu Ord, základem je seznam a+b
19   {c} $ % totéž co samotné c, tj. atom typu Ord se základem c
```

Pro zápis sazby s použitím skupiny nebo sazby samotného znaku TEXbook zavádí syntaktické pravidlo ⟨*math field*⟩. To je buď povel pro sazbu znaku nebo posloupnost `{`⟨*matematický materiál*⟩`}`. Za *výsledek sazby* ⟨*math field*⟩ považujeme buď sazbu jednotlivého znaku nebo matematický seznam, který je výsledkem sazby ⟨*matematického materiálu*⟩. □

• **Sazba exponentu** se realizuje povelem `^`$_7$ ⟨*math field*⟩ (na ASCII hodnotě tokenu `^`$_7$ nezáleží). Pokud je posledním elementem sazby atom s prázdným exponentem, naplní TEX tento exponent výsledkem sazby ⟨*math field*⟩. Pokud není exponent prázdný, TEX ohlásí chybu `Double superscript`. Konečně, pokud není posledním elementem sazby atom, TEX vytvoří atom typu Ord s prázdným základem a naplní exponent výsledkem sazby ⟨*math field*⟩. Například:

```
20 $ ^2       % vytvoří se atom typu Ord s exponentem 2
21   x^{a+b}  % exponentem je matematický seznam a+b
22   \vrule^3 % za \vrule se vytvoří atom typu Ord s exp. 3
23   y^{c}  $ % jako y^c, exponent je jenom c, ne atom typu Ord
```
□

• **Sazba indexu** se zapisuje pomocí `_`$_8$ ⟨*math field*⟩. Pro index platí analogicky totéž, co bylo před chvílí řečeno pro exponent. Důsledek: je jedno, v jakém pořadí použijeme matematické konstruktory:

```
24 $ a_i^2 \hbox{ je totéž jako } a^2_i $
```
□

• **Sazba boxu.** Při zápisu `\hbox{`⟨*horizontální materiál*⟩`}` TEX vytvoří atom typu Ord a do jeho základu vloží `\hbox` s horizontálním seznamem, který je výsledkem sazby ⟨*horizontálního materiálu*⟩. Exponent a index atomu ponechá (zatím) prázdný. Analogicky si počíná se zápisem `\vbox{`⟨*vertikální materiál*⟩`}`.

• **Explicitně daný typ atomu.** Pomocí primitivů `\mathord`, `\mathop`, `\mathbin`, `\mathrel`, `\mathopen`, `\mathclose`, `\mathpunct`, `\mathinner`, `\overline`, resp. `\underline` (následovaných ⟨*math field*⟩) lze vytvořit atom daného typu (Ord, Op, Bin, Rel, Open, Close, Punct, Inner, Over, resp. Under), přičemž základem je výsledek sazby ⟨*math field*⟩ a exponent s indexem je (zatím) prázdný.

Například pomocí „`\mathord,`“ můžeme sázet v matematickém módu desetinnou čárku. Při nastavení z plainu totiž nelze použít čárku přímo, neboť čárka má nastavenu třídu 6 a vytváří tedy atom typu Punct. Důsledkem toho je, že za takovou čárkou je v zápise desetinného čísla nežádoucí mezera. Buď změníme `\mathcode` čárky (tím nám ale přestane fungovat čárka ve významu vyjmenovávání údajů), nebo použijeme `\mathord`, nebo (což je nejčastější řešení) píšeme třeba `3{,}1415`. Proč to funguje, už víme z odstavce o skupinách (strana 148).

Dalším příkladem použití těchto primitivů může být definice matematického operátoru `\lim`:

```
\def\lim{\mathop{\rm lim}}
```

Písmena „lim“ jsou tedy sázena románským písmem (`\rm`) a celek vystupuje jako atom typu Op. To ovlivní umístění indexů. Uživatel může tedy psát:

```
$$ \sum_{i=1}^\infty    % Index a exponent bude nad a pod sumou
   \lim_{x\to\infty} $$ % Index bude rovněž pod slůvkem "lim"
```

• **Závorky proměnlivé výšky.** TEX umožňuje vysázet závorky, jejichž výška je závislá na výšce matematického seznamu, který je závorkami obklopen. Tato vlastnost se používá nejen pro sazbu matic, ale též pro sazbu závorek, které uzavírají „větší vzorce“. Je směšné, pokud je například vzorec s velkým operátorem integrálu nebo sumy uzavřen do mrňavých závorek. Proto sázíme například:

$$\left(\sum_{i=1}^{\infty} x_i\right) \times \Phi$$

pomocí:

```
$$ \left(\sum_{i=1}^\infty x_i \right) \times \Phi $$
```

Za primitivy `\left` a `\right` musí být element, se kterým TEX pracuje ve významu tzv. *delimiteru*. V naší ukázce se jednalo o kulaté závorky. Delimiterem může být libovolný ASCII znak, který má přiděleno nezáporné `\delcode`. Tento primitiv (podobně jako třeba `\catcode`, `\lccode`, `\uccode`, `\mathcode`) přiděluje danému ASCII znaku číslo a zapisuje se ve tvaru:

```
\delcode ⟨ASCII⟩ = ⟨24-bit number⟩
```

Přidělené 24-bitové číslo obsahuje tyto údaje o delimiteru: Rodina fontu (4 bity) a pozice (8 bitů) pro znak základní velikosti a dále rodina fontu (4 bity) a pozice (8 bitů) pro znak první větší velikosti. Bude-li TEX potřebovat ještě větší velikost, najde už další informace ve fontu. Podrobněji o sazbě závorek s proměnlivou velikostí, viz následující sekci 5.2.

Například po deklaraci `\delcode‘\(="028300` je možno použít otevírací závorku ve významu delimiteru, přičemž základní velikost závorky se najde ve fontu rodiny 0 na pozici "28 (hexadecimálně) a první větší velikost je ve fontu rodiny 3 na pozici 0. Všechny delimitery, jak je nastavuje plain, najde čtenář na straně 185.

Delimiter lze zapsat také primitivem `\delimiter`, za nímž následuje 27 bitové číslo. První tři bity navíc označují třídu matematického objektu jako u `\mathchar`. Ostatní bity mají stejný význam jako u `\delcode`. Údaj o třídě se použije jen tehdy, když je primitiv `\delimiter` použit pro sazbu obyčejného matematického znaku:

```
30 \left\delimiter"4028300 % je totéž jako \left(, protože
31                        % v plainu je \delcode‘\(="028300
32 \delimiter"4028300      % je totéž jako \mathchar"4028 a to je
33 % totéž jako pouhá (, protože v plainu je \mathcode‘\(="4028.
```

Konstrukce `\left`⟨*delimiter*⟩ ⟨*matematický materiál*⟩ `\right`⟨*delimiter*⟩ otevírá a zavírá skupinu podobně jako `{` ⟨*matematický materiál*⟩ `}`. Ke každému `\left` musí na stejné úrovni skupiny stát `\right`. Výsledkem konstrukce je atom typu Inner, v jehož základu je nejprve značka typu `\left`, dále výsledek sazby ⟨*matematického materiálu*⟩ a konečně značka typu `\right`. V místě značek vytvoří TEX závorky odpovídající velikosti až při konverzi na horizontální seznam.

Uvedeme jednu záludnost. Použijeme-li například:

```
34 $$ \left|^* \sum_{i=1}^\infty \right|^* $$
```

dostáváme:

$$\left|{}^* \sum_{i=1}^{\infty} \right|^*$$

Exponenty jsou v nestejné výšce. To je proto, že za značkou typu `\left` TEX při zpracování exponentu vytvořil prázdný atom typu Ord, zatímco exponent uvedený za `\right` patří celému atomu typu Inner a ten má jinou výšku. Pokud bychom chtěli skutečně mít hvězdičky ve stejné výši, musíme použít makro plainu `\Biggl` a `\Biggr` takto:

```
35 $$ \Biggl|^* \sum_{i=1}^\infty \Biggr|^* $$
```

□

• **Zlomky** se zapisují ve tvaru „`{`⟨*čitatel*⟩ `\over` ⟨*jmenovatel*⟩`}`“. ⟨*Čitatel*⟩ i ⟨*jmenovatel*⟩ je libovolný ⟨*matematický materiál*⟩. Na primitiv `\over` tedy pohlížíme jako na separátor, který odděluje sazbu čitatele od jmenovatele. Závorky vymezují zlomek jako celek a navíc otevírají skupinu. Oddělovač `\over` a zavírací závorka se musí vyskytovat v základní hladině této skupiny. Místo závorek lze

pro vymezení zlomku použít hranice matematického módu nebo deklarace závorek proměnlivé výšky (`\left`, `\right`).

LaTeX definuje makro `\frac`:

```
\def\frac #1#2{{#1\over #2}}
```

takže ve většině případů je potřeba použít o dvě složené závorky více. Zkuste srovnat, co se píše příjemněji:

```
$a+b\over c+d$ nebo $\frac{a+b}{c+d}$
```

Vedle primitivu `\over` lze ve stejném syntaktickém významu použít další primitivy, které navíc definují vzhled zlomkové čáry a připojují kolem případně závorky proměnlivé výšky.

```
{a \over b}       % zlomková čára implicitní výšky
{a \atop b}       % nulová výška zlomkové čáry, tj. není vidět
{a \above 2pt b} % zlomková čára výšky 2pt
{a \overwithdelims() b}      % jako \left(a \over b\right)
{a \atopwithdelims() b}      % jako \left(a \atop b\right)
{a \abovewithdelims[] 2pt b} % jako \left(a \above2pt b\right]
```

Zlomek jako celek je vždy vložen do základu atomu typu Ord. Základ v tomto případě obsahuje dva matematické seznamy: čitatel a jmenovatel. Dále zde může být obsažena informace o `\delcode` použitých závorek (pracuje-li varianta `\...withdelims`) nebo o tloušťce zlomkové čáry (pro variantu `\above...`). □

• **Matematické akcenty.** Atom typu Acc vzniká použitím primitivu `\mathaccent`, za nímž následuje údaj o sazbě akcentu v ⟨*15-bit number*⟩. Toto číslo má stejný význam jako třeba v `\mathchar`. Dále následuje ⟨*math field*⟩. TeX vloží do základu atomu typu Acc výsledek sazby ⟨*math field*⟩. Exponent a index ponechá (zatím) prázdný. Atom typu Acc nese kromě tří standardních částí ještě jeden údaj — o sazbě akcentu. TeX tam vloží uvedené ⟨*15-bit number*⟩. Údaj o třídě je v tomto čísle ignorován. Způsob sazby akcentu je vysvětlen v části B u hesla `\skewchar` a `\mathaccent`. □

• **Odmocnina.** Povelem `\radical` ⟨*27-bit number*⟩ ⟨*math field*⟩ se vytvoří atom typu Rad a do jeho základu TeX vloží výsledek sazby ⟨*math field*⟩. Atom Rad nese kromě tří standardních částí ještě údaj typu `\delimiter` ve tvaru ⟨*24-bit number*⟩. Ze vstupního ⟨*27-bit number*⟩ je tedy ignorován údaj o třídě a zůstává informace o rodině fontu a o pozici jednak pro základní velikost a dále pro první větší velikost. Na těchto pozicích bývají obvykle znaky pro odmocninu (√). Způsob sazby odmocniny je uveden v sekci 5.2. □

• **Centrovaný box.** Například při tvorbě matic máme v matematické sazbě rozsáhlé boxy, které je potřeba usadit na matematickou osu. To dělá primitiv `\vcenter`, který se chová stejně jako `\vbox`. TEX vytvoří atom typu Vcent a do jeho základu uloží příslušný `\vbox`. Exponent a index zůstávají (jako obvykle) zatím prázdné. Centrování boxu provede TEX až při konverzi do horizontálního seznamu. □

• **Pohled do souboru `log`.** Na závěr této sekce se podívejme do souboru `log`, co nám TEX poví o vytvořeném matematickém seznamu. Použijeme-li například v ukázce z řádku 34 primitiv `\showlists` ještě před ukončovací `$$`, můžeme v `log`u číst:

```
\mathinner
.\left"26A30C
.\mathord
.^\fam2 ^^C
.\mathop
..\fam3 P
.^\fam2 1
._\mathord
.._.\fam1 i
._\mathrel
.._.\fam0 =
._\mathord
.._.\fam0 1
.\right"26A30C
^\fam2 ^^C
```

Vidíme, že je vytvořen atom typu Inner (`\mathinner`) který ve svém základě (řádky začínající tečkami) má matematický seznam a v exponentu (poslední řádek začínající znakem „`^`“) má hvězdičku sázenou fontem rodiny 2 (`\fam2`) z pozice 3 (`^^C`). Vnitřní seznam je zahájen značkou `\left` s údajem typu `\delcode`. Dále pokračuje atom typu Ord (`\mathord`), který má prázdný základ a v exponentu je hvězdička. Pak vidíme atom typu Op (`\mathop`), který má v základu symbol sumy z fontu rodiny 3 z pozice stejné jako ASCII velkého P. V exponentu má symbol nekonečna z fontu rodiny 2 z pozice 49 (to je ASCII hodnota znaku „`1`“). V indexu (řádky začínají „`._`“) je matematický seznam se třemi atomy: postupně Ord, Rel a Ord. Konečně za atomem typu Op následuje značka `\right`. □

5.2. Konverze z matematického do horizontálního seznamu

Po ukončení sestavování matematického seznamu je tento seznam konvertován do horizontálního, ve kterém už je přesně řečeno, jaké fonty jsou použity, jaké rozměry mají jednotlivé elementy sazby a jaké jsou mezi nimi mezery. Postupně probereme jednotlivé algoritmy, které s touto konverzí souvisejí.

• **Matematické styly.** Při konverzi TEX přepíná mezi tzv. *styly* matematické sazby. Podle stylu je volena velikost použitého fontu a způsob usazování dalších indexů a exponentů. Rozlišujeme tyto styly:

Označení	*Primitiv*	*Vysvětlení*
D	`\displaystyle`	Základní velikost při sazbě rovnic
T	`\textstyle`	Základní velikost v textu odstavce
S	`\scriptstyle`	Index a exponent první úrovně
SS	`\scriptscriptstyle`	Index a exponent druhé a další úrovně
D'		Redukované D, např. $\sqrt{D}$
T'		Redukované T
S'		Redukované S, indexy první úrovně
SS'		Redukované SS, indexy další úrovně

Ve stylech D, D', T a T' volí TEX základní velikost fontu (tj. jako v obyčejném textu). Ve stylech S a S' je použita zmenšená velikost pro indexy a exponenty první úrovně (třeba tahle). Nejmenší velikost fontu (třeba tuto) TEX používá pro indexy a exponenty druhé a další úrovně (styl SS a SS'). Skutečnost, že TEX neumí přepínat do ještě menší velikosti fontu (třeba na úrovni indexu indexu indexu) nás nemusí mrzet, protože ani v dobách ruční sazby se to nepoužívalo. Menší písmo se už opravdu nedá číst. I ruční sazeči se snažili takto hluboce vnořeným indexům vyhnout, nebo použili písmo stejně velké, jako na druhé úrovni indexů.

TEX ve struktuře matematického seznamu automaticky přepíná mezi jednotlivými styly. Ve vnitřním matematickém módu je sazba zahájena ve stylu T a v display matematickém módu se startuje stylem D. K přepínání mezi styly dochází při přechodu od sazby základu každého atomu k sazbě indexů a exponentů:

$$D^{S}_{S'},\quad T^{S}_{S'},\quad S^{SS}_{SS'},\quad SS^{SS}_{SS'},\quad D'^{S'}_{S'},\quad T'^{S'}_{S'},\quad S'^{SS'}_{SS'},\quad SS'^{SS'}_{SS'}$$

První vzorec znamená, že TEX ve stylu D při sazbě exponentu přechází do stylu S a při sazbě indexu do stylu S'. Analogicky čteme další vzorce. Ke změně stylů dochází také při sazbě zlomků:

$$D:\frac{T}{T'},\quad D':\frac{T'}{T'},\quad T:\frac{S}{S'},\quad T':\frac{S'}{S'},\quad S,SS:\frac{SS}{SS'},\quad S',SS':\frac{SS'}{SS'}$$

Tyto vzorce čteme rovněž názorným způsobem. První údaj například říká, že ve stylu D je čitatel sázen stylem T a jmenovatel stylem T'.

Z uvedeného algoritmu okamžitě vidíme, proč po `$1\over2$` dostáváme „maličký“ zlomek $\frac{1}{2}$, zatímco při `$$1\over2$$` je čitatel i jmenovatel sázen základní velikostí písma. □

Redukované (čárkované) styly se používají všude tam, kde si je TEX vědom, že nad sestavovanou částí vzorečku je ještě nějaký objekt sazby. Jedná se o tyto případy: index, jmenovatel zlomku (je nad ním zlomková čára), dále základy v atomech typu Over (nad vzorečkem je čára), Acc (nad vzorečkem je akcent) a Rad (nad vzorečkem je čára z odmocniny).

V redukovaných stylech jsou případné exponenty s ohledem na stísněnost prostoru sázeny poněkud níže, než ve stejné situaci u základních stylů. Srovnáme sazbu exponentu ve vzorci e^2 v následujícím řádku:

$$e^2, e^2, \sqrt{e^2}$$

První vzorec byl sázen ve stylu D, druhý ve stylu T a poslední (kde je exponent usazen nejníže) byl sázen ve stylu D'. □

Pokud nám nevyhovuje automatické přepínání mezi styly, můžeme kdekoli v matematické sazbě použít primitivy `\displaystyle`, `\textstyle`, `\scriptstyle` a `\scriptscriptstyle`. Těmito primitivy můžeme přepnout do požadovaného stylu „ručně“. Po vyzkoušení:

```
1) $1\over2$ \qquad
2) $\textstyle 1\over2$  \qquad
3) $\textstyle 1\over\textstyle2$ \qquad
4) $\displaystyle {1\over2} $  \qquad
5) $\scriptstyle {1\over2} $
```

dostaneme:

1) $\frac{1}{2}$ 2) $\frac{1}{2}$ 3) $\frac{1}{2}$ 4) $\displaystyle\frac{1}{2}$ 5) $\frac{1}{2}$

a zjistíme, že je použitelná akorát první a čtvrtá možnost. Druhá je úplně špatně, protože zlomek je tvaru T/S' a třetí má kolem zlomkové čáry málo místa. Vidíme tedy, že sazba zlomkové čáry ve stylu D vede ke vhodným mezerám nad a pod čarou, zatímco ve stylu T s ručně upravenými styly čitatele a jmenovatele se dočkáme zklamání. Pátá možnost vyžaduje od čtenáře použití lupy, protože zlomek byl sázen jako SS/SS'. □

Pokud vytváříme makra pro matematickou sazbu, hodilo by se nám, kdybychom věděli, ve kterém stylu se právě nacházíme, a podle toho pracovali například s různými rozměry. K tomu nám slouží primitiv `\mathchoice`, který má čtyři parametry. První parametr obsahuje sazbu ⟨*matematického materiálu*⟩ pro případ, že bude zpracováván ve stylu D, druhý pro případ stylu T, třetí pro případ stylu S a poslední pro případ stylu SS. Mezi redukovanými styly a jejich neredukovanými protějšky se nerozlišuje.

Primitiv `\mathchoice` fakticky vloží do matematického seznamu výsledek sazby pro všechny čtyři alternativy. Teprve při konverzi na horizontální seznam TEX použije jen tu alternativu, která odpovídá právě aktuálnímu stylu. Toto „zpoždění“ není z pohledu programátora maker podstatné a na `\mathchoice` se dívá jako na účelný nástroj pro „větvení výpočtu“ podle zrovna aktuálního stylu.

Uvedeme si příklad. Chceme vyrobit makro `\ctverecek`, které bude sázet čtvereček v matematické sazbě jako binární operátor. Protože bude uživatel používat `\ctverecek` i na úrovni indexů a třeba i indexů indexů, potřebujeme, aby makro kreslilo čtvereček různě velký podle jednotlivých situací. Zde je řešení:

```
\def\ctvr #1{\mathbin{% bude to binární operátor
   \mkern 2mu
   \vbox{\hrule\hbox to#1{\vrule height#1\hss\vrule}\hrule}}
   \mkern 2mu }
\def\ctverecek{\mathchoice {\ctvr{5pt}}   {\ctvr{5pt}}
                           {\ctvr{3.5pt}} {\ctvr{2.5pt}}}
% A nyní to vyzkoušíme:
$$ \sum_{i_{\ctverecek}+j\ctverecek k} A \ctverecek B $$
```

a dostáváme

$$\sum_{i\square+j\square k} A \square B$$

Kdo tvrdí, že to není čtvereček, ale obdélníček, má pravdu (viz též poznámku na straně 91 za kódem 151 až 153). V ukázce jsou mimo to dvě věci, které si zaslouží pozornost. Před a za boxem jsme vložili mezeru `\mkern 2mu`, jejíž velikost, jak za chvíli ukážeme, rovněž závisí na zrovna aktivním stylu. Bohužel, pro konstrukci čtverečku se nám tato vlastnost nehodí, protože v době, kdy je v činnosti `\vrule height...`, se hlavní procesor nalézá v horizontálním seznamu a tam žádné `\mkern` nelze použít. Proto jsme použili `\mathchoice`.

Druhou věcí je jistá uživatelská nepřítulnost našeho makra. Uživatel nemůže použít přímo `i_\ctverecek`, ale musí psát `\i_{\ctverecek}`. Je to proto, že po expanzi našeho makra není sazba připravena ve formátu ⟨*math field*⟩. Pokud chceme, aby se `\ctverecek` samotný choval jako binární operátor, pak nelze v makru použít

skupinu, protože by vznikl pouze atom typu Ord a nikoli Bin. Je ovšem pravda, že v ukázce je použit `\ctverecek` ve dvou různých významech. Jednou jako binární operátor a jednou jako samotný symbol. Pokud si bude uživatel těchto dvou významů neustále vědom, chybu neudělá.

Jiné řešení úkolu vede přes vytvoření tří fontů různě velkých, každý by obsahoval čtvereček. Tyto fonty by bylo potřeba zavést do TEXu jako rodinu matematických fontů a definovat `\ctverecek` pomocí `\mathchardef`. Potom by uživatel mohl psát též zkrácené `i_\ctverecek`. □

• **Mezery v matematické sazbě.** Především připomeneme, že token $\sqcup_{10}$ je při sestavování matematického seznamu zcela ignorován. Pokud tedy chceme mít mezi jednotlivými objekty v matematické sazbě mezery, můžeme použít `\␣` nebo `\kern`, `\hskip`, `\hfil` apod., nebo konečně `\mkern`, `\mskip`. S posledními dvěma primitivy jsme se při sestavování horizontálního seznamu nesetkali, proto se s nimi nyní seznámíme blíže.

Je ovšem potřeba upozornit na to, že než uživatel začne do matematického seznamu vkládat mezery, měl by si být přesně vědom, co dělá. Měl by tedy znát automatické mezerování TEXu v matematickém seznamu, o němž si povíme za chvíli. Teprve pokud uživateli toto automatické mezerování nevyhovuje, měl by začít vkládat další mezery.

Primitivy `\kern`, `\hskip` apod. vytvářejí v matematickém seznamu definitivní hodnoty mezer už v době jeho sestavování. Na druhé straně velikost jednotky `mu`, která se jako jediná smí použít v primitivech `\mskip` a `\mkern`, se vyhodnotí až při konverzi do horizontálního seznamu. Platí toto pravidlo:

$$\texttt{18 mu} = \langle quad \rangle$$

kde $\langle quad \rangle$ je velikost použitého písma. V sekci 3.6 na straně 104 jsme uvedli, že velikost písma čte TEX z parametru `\fontdimen6` fontu. V případě jednotky `mu` se jedná o font, který TEX používá v daném matematickém stylu. Proto jednotka `mu` závisí na stylu, ve kterém je použita. Například `18mu` při nastavení fontů z plainu znamená ve stylu D a T deset bodů, zatímco ve stylu S zhruba osm bodů a ve stylu SS sedm bodů. Podrobněji se o konkrétních fontech v matematické sazbě zmíníme v následující sekci.

V plainu je definováno makro `\quad` takto:

```
57 \def\quad{\hskip 1em\relax}
```

Co to udělá v textové a co v matematické sazbě? V textové sazbě se použije údaj z `\fontdimen6` právě nastaveného textového fontu. V matematické sazbě se rovněž použije údaj z textového fontu. To není příliš logické, protože matematická sazba

je na nastavení textového fontu (až na použití jednotky `em` nebo použití povelu `\␣`) zcela nezávislá. Makro `\quad` navíc vytvoří stejnou mezeru ve stylu T, ale třeba také v indexu. Pokud si to nepřejeme, můžeme definovat makro `\quad` poněkud důsledněji:

```
58 \def\quad{\ifmmode\mskip 18mu\else\hskip 1em\fi \relax}
```

Nyní bude `\quad` dávat v T mezeru 10 pt a v indexech bude odpovídajícím způsobem menší. □

• **Automatické mezerování.** TEX se snaží při sazbě matematiky dodržet „vhodné“ horizontální vzdálenosti mezi objekty, které mají v matematické sazbě jistý konkrétní význam. Autor TEXu si stanovil tyto požadavky:

1. Kolem relací $(=, \leq)$ má být z obou stran větší mezera.
2. Kolem binárních operací $(+, \times)$ má být z obou stran střední mezera.
3. Za čárkou má být malá mezera.
4. Kolem velkých operátorů $(\sum, \int)$ má být malá mezera.
5. Kolem vzorců se závorkami `\left`, `\right` mají být malé mezery.

Do TEXu jsou implementovány tři primitivní registry typu ⟨*muglue*⟩: `\thickmuskip` pro hodnotu „větší mezery“, `\medmuskip` pro hodnotu „střední mezery“ a `\thinmuskip` pro hodnotu „malé mezery“. V plainu jsou naplněny těmito hodnotami:

```
59 % Skutečné nastavení                  % Velikost při ⟨quad⟩=10pt
60 \thickmuskip= 5mu plus 5mu            % 2.7pt plus 2.7pt
61 \medmuskip  = 4mu plus 2mu minus 4mu  % 2.22pt plus 1.11pt...
62 \thinmuskip = 3mu                     % 1.66pt
```

Při procházení jednotlivými typy atomů v matematickém seznamu je v činnosti algoritmus, který mezi každé dva atomy specifikovaného typu může vložit mezeru podle jednoho ze tří uvedených registrů. Mezery jsou vkládány podle tohoto schématu:

1. Ord, Op, Close, Inner [3]	**Rel**	[3]	Ord, Op, Open, Inner
2. Ord, Close, Inner [2]	**Bin**	[2]	Ord, Op, Open, Inner
3.	**Punct**	[1]	Ord, Op, Rel, Open, Close, Punct, Inner
4. Ord, Close, Inner [1]	**Op**	[1]	Ord, Op
5. Ord, Op, Close [1]	**Inner**	[1]	Ord, Open, Punct, Inner

Znak [3] zde značí `\thickmuskip` (velká mezera), znak [2] znamená `\medmuskip` (střední mezera) a znak [1] je `\thinmuskip` (malá mezera). Například první řádek v tabulce čteme takto: Pokud se v sazbě vyskytne atom typu Rel a před ním je některý z atomů typu Ord, Op, Close nebo Inner, vloží TEX před atom typu

Rel \thickmuskip. Pokud za atomem typu Rel následuje některý z atomů typu Ord, Op, Open nebo Inner, vloží TEX také za atom typu Rel mezeru velikosti \thickmuskip. Analogicky čteme ostatní řádky. Čísla řádků korespondují s původním záměrem, který jsme uvedli výše.

Atomy typu Over, Under, Acc, Rad a Vcent se chovají při mezerování stejně, jako atom typu Ord.

Pokud se v sazbě vyskytne dvojice následujících atomů, pro kterou není v tabulce uveden údaj o mezeře (například Ord-Ord), není mezi ně vkládána žádná mezera. Výjimku z tohoto pravidla tvoří atom typu Bin, který má snahu měnit svůj typ, aby byl kompatibilní se svým sousedem. Přesněji: Je-li atom b typu Bin a je prvním atomem v seznamu nebo předchozí atom je typu Bin, Op, Rel, Open nebo Punct, je změněn typ atomu b na Ord. Jestliže za atomem b typu Bin následuje atom typu Rel, Close nebo Punct, je rovněž typ atomu b změněn na Ord.

Příklad: `$-1$` bude sázeno jako Ord-Ord, ačkoli znak „-“ původně vytváří atom typu Bin. Typ tohoto atomu je změněn na Ord, protože je prvním atomem v seznamu. Jiný příklad: `$+++$` udělá mezery kolem prostředního plus, které jediné se chová jako atom typu Bin. První plus je změněno z typu Bin na Ord, protože nepředchází nic a třetí plus je změněno na Ord, protože předchází Bin.

Pokud je mezi dvěma atomy jiný materiál, TEX si ho při rozhodování o vložení mezery nevšímá. Případnou mezeru pak vloží za tento materiál. Například při `$A = \hskip20pt\vrule B$` je mezi \vrule a B vložena mezera \thickmuskip, protože dvojice „$= B$“ je dvojicí typu Rel-Ord.

Automaticky počítané mezery nejsou vkládány ve stylech S, SS, S', SS'. Výjimku tvoří pravidlo o mezerování kolem atomů typu Op (řádek 4 v našem schématu). Podle tohoto pravidla jsou mezery vkládány za všech okolností. □

Automatické mezerování si procvičíme na příkladě. Uvažujme formuli:

$$M_{x+y} = \{=, +\}$$

Zde jsou po řadě atomy Ord (v indexu Ord, Bin, Ord), dále Rel, Open, Rel, Punct, Bin, Close. Index $x+y$ je podle naposledy zmíněného pravidla sázen bez mezer. Poslední atom typu Bin v seznamu změní svůj typ na Ord, protože před ním předchází typ Punct. Dostáváme tedy toto mezerování:

$$M_{x\,[0]\,+\,[0]\,y}\,[3] = [3]\,\{\,[0] = [0]\,,\,[1] + [0]\,\}$$

V této notaci symbol [0] znamená žádná mezera, [1] je mezera z \thinmuskip a [3] je mezera z \thickmuskip. □

• **Sazba atomů, obecná pravidla.** Každá neprázdná část atomu (základ, exponent, index) se při konverzi převede na box, obsahující příslušný horizontální seznam. Výjimku tvoří jednoznakové základy, které obvykle nejsou vkládány do boxu, ale objeví se v horizontálním seznamu jako samotné znaky.

Příklad. Sazba `${a+b=c}$` je trochu odlišná od `$a+b=c$`. V prvním případě je na hlavní úrovni matematického seznamu atom typu Ord, jehož základem je seznam „$a + b = c$“. Tento seznam tedy vstoupí do boxu. Důsledkem toho je skutečnost, že hodnoty pružnosti z `\medmuskip` (kolem operátoru „+“) a `\thickmuskip` (kolem operátoru „=“) nebudou mít ve vnějším horizontálním seznamu žádný vliv. Ve druhém příkladě vzniká ve výstupním horizontálním seznamu pět elementů pro sazbu jednotlivých znaků. Mezi nimi jsou mezery z příslušných `\thickmuskip` a `\medmuskip`. Pružnosti z těchto mezer nyní mohou pracovat ve vnějším horizontálním seznamu, například při sestavování výsledných řádků odstavce.

Další rozdíl uvedeného příkladu: první způsob zápisu vede na jeden box, který se při sestavování odstavce do řádků nikdy nerozlomí. Druhý způsob zápisu umožňuje řádkový zlom za operátorem „+“ nebo za relací „=“. To souvisí s následujícím pravidlem:

Těsně za každým atomem typu Bin je vložena `\penalty` o hodnotě podle registru `\binoppenalty`. Těsně za každým atomem typu Rel je připojena `\penalty` o hodnotě `\relpenalty`. Plain nastavuje

```
\binoppenalty=700
\relpenalty=500
```

takže za těmito matematickými objekty je dovolen řádkový zlom s jistou penalizací. Tyto `\penalty` jsou jediná implicitně vkládaná místa, kde je povolen zlom, protože v mezerách typu ⟨*glue*⟩ je v matematické sazbě zlom zakázán.

Uvedené pravidlo nám podle tradic české sazby příliš nevyhovuje. Můžeme proto nastavit uvedené registry na hodnotu 10 000 nebo udělat následující trik. Nastaví-li se `\mathcode` nějakého znaku na `"8000`, chová se v matematickém módu jako aktivní znak. Přitom mimo matematický mód se tento znak chová jako běžný znak a nemusí být aktivní. Této vlastnosti využijeme a deklarujeme znak „+“ jako aktivní v matematickém módu. Bude expandován na `\discretionary`. Tím dosáhneme efektu opakování znaku „+“ i na následujícím řádku v případě, že se má provést řádkový zlom za tímto operátorem. To je přesně to, co ruční sazeči v matematických knihách vždycky dělali.

```
{\catcode`\+=13    % definice aktivního +
\expandafter }\expandafter \def\noexpand+{%
                           \mathplus\discretionary{}{+}{}}
\mathchardef\mathplus=\mathcode`\+  % sazba znaku + v mat. módu
```

```
69 \mathcode`\+="8000  % nastavení + jako aktivní v mat. módu
70                    % jinde zůstává neaktivní
71 \binoppenalty=10000 % aby se to nelámalo jinak
```

Pokud nyní napíšeme třeba `$A+B+C+D$`, dostáváme v místě zlomu řádku $A+B+C+$ $+\,D$, takže se znak „+“ přepíše na začátek dalšího řádku. Protože je při automatickém mezerování vkládáno `\medmuskip` až za ostatní tiskový materiál (v našem případě až za `\discretionary`), je správné mezerování nejen před znakem „+“ na konci řádku, ale též za prvním znakem „+“ na řádku následujícím. Je ovšem potřeba dát pozor na unární „+“. Například $+\infty$ by se mohlo rozdělit jako $+$ $+\infty$, což není vůbec žádoucí. Pišme tedy `$\mathplus\infty$`.

Jestliže čtenář nemůže přečíst trik s `\expandafter` na řádku 66, připomínáme, že jsme podobnou věc řešili na stránce 46. Jde o to, abychom mohli v těle definice pracovat s neaktivním znakem „`+`“. S parametry v `\discretionary` TEX pracuje jen v horizontálním módu, takže tam zůstává znak plus neaktivní.

Podobně jako znak „+“ v ukázce bychom předefinovali znaky „$-,=$“ a případně další. Protože parametr `\discretionary` nesmí obsahovat přechod do matematického módu, použijeme trik `\newbox\minusbox \setbox\minusbox=\hbox{$-$}` a dále v definici symbolu „-“ píšeme `\discretionary{}{\copy\minusbox}{}`. □

• **Kerny a ligatury v matematické sazbě.** Za každý znak v matematické sazbě připojí TEX jeho italickou korekci, pokud je nenulová. Výjimkou je případ tzv. *textového znaku v matematickém módu*, který je definován takto: je to znak z atomu typu Ord, pro který současně platí (1) atom je jednoznakový a bez exponentu a indexu, (2) za tímto atomem bezprostředně následuje jednoznakový atom typu Ord, Op, Bin, Rel, Open, Close nebo Punct, (3) znaky z obou atomů jsou sázeny stejným fontem. Jsou-li splněny tyto podmínky textového znaku, a současně je sazba prováděna fontem s nenulovou mezislovní mezerou (nenulovým `\fontdimen2`), nepřipojí se italická korekce. Matematická kurzíva z fontu `cmmi10` má ale mezislovní mezeru nulovou, takže za znaky z tohoto fontu se italická korekce připojuje vždy. Italická korekce znaku navíc ovlivní umístění případného indexu a exponentu (viz níže).

Jsou-li splněny podmínky pro textový znak v matematickém módu, připojuje se za tento znak také implicitní kern podle tabulky kerningových párů ve fontu. Pokud je pro dvojici znaků ve fontu odkaz na ligaturu, promění se dvojice atomů v jediný atom typu Ord obsahující sazbu ligatury. Vyzkoušíme si `${\rm fi}_j$`. Zde jsou za sebou dva atomy typu Ord, které jsou převedeny na jediný znak ligatury: „fi“. Při sazbě proměnných v matematické sazbě se používá obvykle font pro matematickou kurzívu `cmmi10` a v něm žádné takové ligatury nejsou. Proto bude `$fi$` sázeno způsobem, který obvykle v matematice čteme jako „f krát i“, tj. fi.

Vyzkoušíme si ještě `$V,$\showlists`. V souboru `log` shledáme, že za písmenem „V“ byly vloženy dva kerny. Nejprve `\kern2.22223`, což je italická korekce

znaku „V“. Dále je vložen `\kern-1.66667`, což je implicitní kern mezi znaky „V“ a čárkou ve fontu `cmmi10`. □

Následující příklad patří mezi tzv. „špinavé triky“. TEX nedává programátorovi maker základní prostředky na prozkoumání vlastnosti fontů. Nicméně například na straně 103 jsme ukázali trik, jak zjistit z fontu informace o implicitních kerningových párech. Zjistit, zda dvojice znaků dává ve fontu ligaturu nebo ne, je úkol poněkud obtížnější. TEX totiž nedisponuje primitivem typu `\lastchar`, který by se choval v horizontálním seznamu podobně jako `\lastbox` a pomocí něhož bychom po vložení dvou znaků do pracovního `\hbox`u snadno poznali, zda se tento znak proměnil v ligaturu. Algoritmy matematické sazby nám ale umožňují pomocí prostředků makrojazyka zjistit, zda dvojice znaků vede na ligaturu nebo ne. Řešení je následující:

```
\newif\ifligature
\def\testligature #1#2{\setbox0=\hbox{%
   \thickmuskip=1000mu \textfont0=\the\font
   $\mathchar`#1 \mathrel\mathchar`#2$}%
   \ifdim\wd0>500pt \ligaturefalse \else \ligaturetrue \fi}
```

Po použití `\testligature fi` bude mít `\ifligature` hodnotu „true“, pokud znaky f a i vedou na ligaturu. Jak makro pracuje? Do pracovního boxu 0 jsme vložili matematickou sazbu skládající se ze dvou atomů. První atom je typu Ord a v základu má první písmeno. Druhý atom je typu Rel a v základu má druhé písmeno. Registr `\thickmuskip` jsme nastavili na velmi vysokou hodnotu. Pomocí `\textfont0` jsme zařídili, aby sazba proběhla právě aktuálním textovým fontem. Pokud dva znaky splynou v ligaturu, neuplatní se mezera mezi Ord a Rel a výsledný box 0 bude mít šířku určitě menší, než 500 pt. Pokud ale nesplynou v ligaturu, pak mezi znaky bude mezera 1000 mu $\doteq$ 555 pt a box bude širší než 500 pt. Pak už jen stačí změřit šířku boxu. □

• **Umístění exponentu a indexu.** Pro všechny typy atomů (s výjimkou atomu typu Op) je jejich exponent připojen bez mezery vpravo od základu a je posunut nahoru. Index je rovněž vpravo od základu a je posunut dolů. Použité rozměry pro vertikální umístění exponentu a indexu přečte TEX z údajů `\fontdimen` matematického fontu. O těchto parametrech se podrobněji zmíníme až v další sekci.

Pokud je základ jednoznakový, je exponent posunut o velikost italické korekce základu doprava, zatímco index je připojen k základu bez italické korekce. Důsledek: Má-li základ kladnou italickou korekci, nejsou exponent a index přesně nad sebou, ale exponent je více vpravo než index.

Má-li atom neprázdný exponent nebo index (sázený vpravo), je za ním připojena mezera velikosti `\scriptspace`, která má stejnou velikost pro všechny matematické styly. □

• **Atom typu Op (velký operátor).** Je-li sázen jednoznakový základ atomu typu Op, je vložen do boxu, který je posunut ve vertikálním směru tak, že horizontální osa znaku splývá s *matematickou osou*. Matematická osa sazby prochází středem vodorovných linek v symbolech „+, −“. Její polohu se TEX dozví z použitého matematického fontu. Všimneme si, že například operátor „$\sum$“ je posunut přes řádek mírně dolů, aby jeho osa splynula s osou znaků „+“ a „−“.

Při výpočtu horizontální osy znaku nerozhoduje jen výška znaku, ale pracuje se s celkovou výškou, tedy se součtem výška + hloubka. Například po podrobnějším průzkumu znaku „$\sum$“ z fontu `cmex10` zjistíme, že má nulovou výšku a celý jeho vertikální rozměr je zanesen jako hloubka. Pokud bychom jej tedy nesázeli jako atom typu Op, ale třeba pomocí `$\mathord\sum$`, dostaneme hrozivý výsledek: „$\sum$“. □

Každý znak v matematickém fontu může mít tzv. *následníka* nebo i více následníků. Tyto informace jsou uloženy ve fontu. Podrobněji se jim budeme věnovat v sekci 5.3 o matematických fontech. Probíhá-li sazba atomu typu Op ve stylu D nebo D' a atom má jednoznakový základ a tento znak má ve fontu následníka, bude sázen tento následník. Jinak bude sázen přímo symbol, který je v základu atomu uveden. Důsledek: Tento algoritmus dovoluje implementovat sazbu dvou variant velikostí velkých operátorů. Malá varianta se sází ve stylu T a T', zatímco velká ve stylu D a D'.

Například ve fontu `cmex10` jsou dvě velikosti symbolů $\sum, \prod, \coprod, \int, \bigcap, \bigcup, \bigsqcup, \bigvee, \bigwedge, \bigodot, \bigotimes, \bigoplus, \biguplus$. Větší velikost je vždy následníkem menší alternativy. Proto třeba při `$\sum$` dostáváme malou sumu a při `$$\sum$$` velkou:

$$\texttt{\textstyle} : \sum, \quad \texttt{\displaystyle} : \sum$$

Exponent a index jsou u atomů typu Op sázeny takto: Ve stylu D nebo D' je exponent a index sázen nad a pod základ, zatímco v ostatních stylech jsou sázeny vpravo (jako u jiných typů atomů). Proto například `$\sum_0^\infty$` vede k $\sum_0^\infty$, zatímco `$$\sum_0^\infty$$` vytvoří velkou sumu s indexy nad a pod.

Výjimky z uvedeného pravidla lze zařídit vložením primitivů `\limits` nebo `\nolimits` mezi základ a konstruktory pro index nebo exponent. Primitiv musí stát před prvním konstruktorem. Použití `\limits` si vynucuje sazbu indexu a exponentu nad a pod základem bez závislosti na stylu. Primitiv `\nolimits` si naopak vynucuje sazbu exponentu a indexu vedle základu, rovněž bez závislosti na stylu. Například pomocí:

```
$$ \sum_0^\infty            \quad % Velká suma, indexy nad a pod
   \sum\nolimits_0^\infty   \quad % Velká suma, indexy vedle
 \textstyle \sum_0^\infty   \quad % Malá suma, indexy vedle
   \sum\limits_0^\infty       $$ % Malá suma, indexy nad a pod
```

vytvoříme všechny čtyři varianty pro sazbu sumy s indexy:

$$\sum_0^\infty \quad \sum\nolimits_0^\infty \quad \sum\nolimits_0^\infty \quad \sum\limits_0^\infty$$

Sejde-li se těsně vedle sebe více primitivů `\limits` a `\nolimits`, platí ten poslední. Tuto vlastnost je možné dobře ilustrovat na makru plainu `\int`, které je definováno takto:

```
81 \mathchardef\intop="1352 \def\int{\intop\nolimits}
```

Vidíme, že makro `\int` bude klást indexy vždy vedle základu nezávisle na stylu. Pokud uživatel chce sázet nad a pod, napíše:

```
82 \int\limits_0^\infty
```

a po expanzi makra se sejde `\nolimits\limits`. Platí přitom ten poslední primitiv (`\limits`), který napsal uživatel.

Má-li být index a exponent vysázen nahoru a dolů, pak jsou společně se základem umístěny na společnou vertikální osu. Pokud ale má jednoznakový základ nenulovou italickou korekci, je exponent vychýlen doprava od osy o polovinu této korekce a index je o stejnou velikost posunut doleva. Této vlastnosti si všimneme například u symbolu $\int$. □

• **Závorky proměnlivé výšky.** Tyto závorky budou sázeny na matematickou osu. Proto se TEX nejprve musí vyrovnat s problémem, kdy vzorec, který má být závorkami obklopován, není centrován podle matematické osy. TEX tedy vloží obklopovaný výraz do boxu těchto vlastností: (1) Box je centrován na matematickou osu, tj. jeho velikost nad matematickou osou je shodná s velikostí pod ní. (2) Celý výraz se do boxu vejde, aniž by s výrazem bylo posunuto nahoru nebo dolů. (3) Box má minimální rozměry, pro které jsou splněny podmínky (1) a (2). Znamená to tedy, že horní nebo dolní okraj výrazu se kryje s okrajem boxu. Pokud byl výraz už původně centrován, kryjí se horní i dolní okraje současně.

Nechť v je celková výška boxu, vytvořeného výše popsaným způsobem. Výška závorky z musí splňovat:

$$z \ \geq \max(vf/1000, v - l)$$

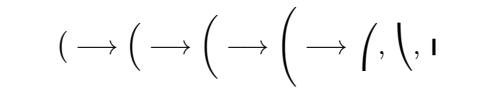
kde f je hodnota `\delimiterfactor` a l je velikost z `\delimitershortfall`. Plain nastavuje tyto hodnoty na:

```
83 \delimiterfactor=901
84 \delimitershortfall=5pt
```

TEX postupně prochází jednotlivé velikosti závorek ve fontech. Nejprve změří závorku v základní velikosti (podle údaje typu delimiter), dále vyzkouší závorku první větší velikosti a pokud má tato závorka ve fontu následníka, pokračuje ve zkoumání následníka a případně následníka následníka atd. Nakonec použije takovou závorku, jejíž výška jako první v řadě zkoumaných závorek vyhovuje uvedené nerovnosti.

{Poslední následník ve fontu může mít odkaz na tři nebo čtyři následníky, což jsou segmenty, ze kterých pak TEX dokáže vyrobit závorku libovolné velikosti: začátek závorky, konec a segment pro opakované vyplnění. Segment pro vyplnění je nad sebou sázen třeba víckrát tak dlouho, až je závorka dostatečně vysoká. Navíc může existovat odkaz na čtvrtý segment, který se použije uprostřed těla závorky. Například tento odstavec je obklopen závorkami. Každou z nich TEX sestavil ze čtyř různých segmentů (včetně prostředního).}

Pro ilustraci, ve fontu `cmex10` najdeme pro kulatou závorku „(“ (což je první větší velikost kulaté závorky) tyto následníky:



Pokud se TEXu nepodaří najít dostatečně velkou závorku a poslední následník neobsahuje odkaz na segmenty, TEX vysází posledního následníka.

Výsledné závorky TEX centruje na matematickou osu.

Pokud je sázena závorka typu *prázdný delimiter*, tj. `\delcode` je rovno nule, TEX vloží prázdný `\hbox` šířky `\nulldelimiterspace`. Tento registr je v plainu nastaven na 1,2 pt. □

Trasujme algoritmy TEXu na tomto příkladě:

```
85 $$ \left( \vrule height10pt depth0pt \right. $$
```

Protože je v matematickém fontu v plainu matematická osa 2,5 pt nad účařím, bude výška výrazu 10 pt − 2,5 pt = 7,5 pt. Pod osou je jen 2,5 pt, takže TEX považuje tento výraz, jako by měl pod osou rovněž 7,5 pt a celková výška tedy je 15 pt. Výška závorky z (při nastavení fontů a registrů z plainu) splňuje podmínku:

$$z \ \geq \max\left(15 \times \frac{901}{1000}\,\text{pt}, 15\,\text{pt} - 5\,\text{pt}\right) = \max\left(13{,}515\,\text{pt}, 10\,\text{pt}\right) = 13{,}515\,\text{pt}$$

Malá varianta závorky z fontu `cmr10` nevyhovuje. První větší závorka z fontu `cmex10` má celkovou výšku 12 pt a její následník 18 pt. TEX tedy použije závorku o výšce 18 pt, která nakonec bude nepatrně vyšší, než celková výška výrazu. Na pravé straně výrazu (`\right`) bude pouze prázdný box o šířce 1,2 pt. Pokud bychom nechali

v naší ukázce neurčenou hloubku linky (nechť se hloubka linky dopočte podle celkové hloubky boxu), zjistíme, že linka bude nakonec centrována na matematickou osu a má celkovou výšku 15 pt. TEX totiž pro obklopovaný výraz vytvořil box, který je centrován na matematickou osu. V našem příkladě má tento box výšku 10 pt a hloubku 5 pt. □

• **Zlomky.** Zlomková čára (linka) je kladena na matematickou osu a její šířka odpovídá širšímu z výrazů ⟨*čitatel*⟩, ⟨*jmenovatel*⟩. Tyto výrazy jsou centrovány podle společné vertikální osy.

Při zlomcích typu `\over`, (zlomková čára má implicitní výšku) `\atop` (zlomková čára má nulovou výšku) a `\above` (zlomková čára má výšku určenou uživatelem) není použito obklopovacích závorek. V tomto případě je místo těchto závorek vložen z obou stran zlomku prázdný box šířky `\nulldelimiterspace` jako při prázdném delimiteru. Tím získáme kolem zlomku trochu místa.

U zlomků typu `\...withdelims` jsou kolem zlomku konstruovány závorky proměnlivé výšky, jak bylo uvedeno v předchozím odstavci. □

• **Atomy Rad, Acc, Over, Under.** Zabývejme se sazbou odmocniny u atomů typu Rad. Pro symbol odmocniny po levé straně výrazu je vybrán z fontu znak s větší celkovou výškou, než je celková výška odmocňovaného výrazu (základu atomu). Výběr probíhá z řady následníků ve fontu. Algoritmus je podobný jako při závorkách proměnlivé velikosti. Rozdíl je ale v tom, že neprobíhá centrování na matematickou osu a vybraný znak musí být vždy větší, než celková výška výrazu. Nepracují tedy registry `\delimiterfactor` a `\delimitershortfall`.

Ve fontu `cmex10` je řada následníků pro znak odmocniny ukončena segmenty, ze kterých TEX vytvoří v případě potřeby znak odmocniny libovolné velikosti:

$$\surd \longrightarrow \surd \longrightarrow \surd \longrightarrow \surd \longrightarrow \left\lceil , \left| , \right\lceil \right.$$

Nad odmocňovaný výraz je umístěna čára, která je stejně dlouhá jako je šířka výrazu. Výška této čáry (tj. její tloušťka) je rovna výšce boxu pro znak odmocniny, který byl pro danou příležitost vybrán. Tvůrce fontu tedy pro každý znak odmocniny určuje tloušťku čáry, která bude použita. Celkovou výšku znaku typu „$\surd$“ vloží do hloubky boxu pro znak a výšku tohoto boxu volí podle toho, jak vysokou chce mít čáru nad odmocňovaným výrazem.

Pro usazení případného čísla odmocniny (například $\sqrt[3]{2}$) nejsou v TEXu implementovány žádné algoritmy. Tuto věc řeší například v plainu makro `\root` (viz část B). □

Matematický akcent pomocí \mathaccent (atom typu Acc) je hledán ve fontu v řadě následníků tak, aby byl široký nejvýše jako základ atomu. Použije se největší s touto vlastností. O usazení akcentu nad základ viz heslo \skewchar v části B. O použití primitivu \mathaccent v plainu viz stranu 183. □

To nejjednodušší nakonec. Atomy typu Over/Under se sázejí tak, že se nad/pod základ připojí linka stejné šířky, jakou má základ. Například napíšeme:

```
86 $\underline{ab_i}\times\overline{a+b}^2$
```

a dostaneme: $\underline{ab_i} \times \overline{a+b}^2$. O tloušťce připojovaných linek viz hesla \underline a \overline v části B. □

• **Hlavní skupina matematického módu.** Celá výroba matematického seznamu probíhá uvnitř skupiny. Přesněji: Při vstupu do matematického módu TEX otevře skupinu, takže všechna přiřazení uvnitř tohoto módu jsou lokální. Pak zařadí na vstup expand procesoru obsah registru \everymath (při vnitřním matematickém módu) nebo \everydisplay (při display módu). Potom čte další obsah vstupního textu. Přitom vytváří matematický seznam. Po dosažení koncové značky matematického módu TEX překontroluje, zda je ve stejné úrovni skupiny jako při vstupu do tohoto módu. Pokud to neplatí, TEX křičí `Missing } inserted`. Dále TEX provede nový průchod nad matematickým seznamem a konvertuje jej do horizontálního seznamu.

V případě vnitřního matematického módu TEX ještě vloží kolem vytvořeného horizontálního seznamu z obou stran mezeru podle \mathsurround. V případě display módu se výsledek umístí do sazby způsobem vyloženým v sekci 5.6. Nakonec TEX uzavře skupinu, kterou otevřel na začátku při vstupu do matematického módu. □

5.3. Fonty v matematické sazbě

V horizontálním módu jsme měli s fonty snadné pořízení. Stačilo zavést font pomocí primitivu \font a tím byl současně deklarován přepínač. Tento přepínač jsme mohli vesele používat k přepínání aktuálního fontu. Aktuální font určuje sazbu znaků v horizontálním módu, ale nemá skoro žádný vliv na sazbu v matematickém módu. Pro matematiku je potřeba deklarovat rodiny fontů a s nimi v sazbě pracovat.

Rodina fontů je celé číslo v intervalu $\langle 0, 15\rangle$. Každému tomuto číslu je možné přiřadit trojici fontů, které byly dříve zavedeny primitivem \font. První font trojice (\textfont) určuje font základní (textové) velikosti, který TEX použije v matematických stylech D, D', T nebo T'. Druhý font z trojice (\scriptfont) je obvykle fontem s menšími znaky pro sazbu v indexové velikosti a TEX jej použije ve stylech S a S'. Konečně třetí font (\scriptscriptfont) obsahuje obvykle nejdrobnější

znaky pro sazbu indexů druhé úrovně a TEX jej použije ve stylech SS a SS'. Deklarace rodiny fontů se provádí stejnojmennými primitivy. Například rodina fontů 0 je v csplainu deklarována takto:

```
% Zavedení fontů do TeXu a deklarace přepínačů pro horiz. mód:
\font\tenrm=csr10
\font\sevenrm=csr7
\font\fiverm=csr5
% Deklarace rodiny 0 pro matematický mód:
\textfont0=\tenrm           % 10 pt pro základní velikost
\scriptfont0=\sevenrm       %  7 pt pro indexovou velikost
\scriptscriptfont0=\fiverm  %  5 pt pro indexy druhé úrovně
```

Nechť je třeba v matematickém seznamu řečeno, že se má vysázet třeba znak „X“ v rodině 0. TEX použije znak „X“ z fontu `\tenrm` (tj. `csr10`) v případě, že se jedná o sazbu ve stylu D, D', T' nebo T'. Stejný znak „x“, ovšem z fontu `\sevenrm` (tj. `csr7`), se použije při sazbě ve stylu S nebo S'. Konečně ve stylu SS a SS' se použije znak „x“ z fontu `\fiverm` (tj. `csr5`). □

• **Základní rodiny matematických fontů.** Aby byl TEX „matematicky gramotný“, musí mít bezpodmínečně deklarovány rodiny číslo 2 a 3. Z fontů těchto rodin čerpá TEX prostřednictvím parametrů `\fontdimen` veškeré znalosti o tom, jak sestavovat matematickou sazbu. Například jak vysoko posunout exponent nebo kolik místa udělat kolem zlomkové čáry. Pouze rodiny 2 a 3 mají tuto významnou roli.

V rodině 2 jsou soustředěny parametry, které jsou rozdílné pro každou ze tří velikostí fontu (například vzdálenost matematické osy od účaří). V rodině 3 jsou pak parametry, které nezávisejí na velikosti fontu (například tloušťka zlomkové čáry bude ve všech matematických stylech stejná). Seznam všech parametrů `\fontdimen`, se kterými TEX pracuje v rodinách 2 a 3, je uveden na konci této sekce.

Proč přidělil autor TEXu tuto významnou funkci právě rodinám 2 a 3? To zřejmě vyplynulo z koncepce, která je použita v plainu při deklaraci rodin matematických fontů s čísly 0 až 3:

- Rodina 0: Antikva; běžně v textu a méně v matematice ($\cos, \lim$).
- Rodina 1: Kurzíva; sazba matematických proměnných (a, x, M, α).
- Rodina 2: Značky v matematice, binární operace a relace ($\oplus, \leq$).
- Rodina 3: Operátory a „zvětšující se“ objekty pomocí následníků ($\sum, \{, \Big\{$).

Pokud v sázecím systému chceme sázet matematiku, potřebujeme speciální font se značkami. Tento font před použitím v TEXu musí odborník na fonty opatřit požadovanými parametry `\fontdimen`. My pak můžeme deklarovat rodinu 2. Dále budeme potřebovat speciální font pro využití případných TEXovských efektů (závorky

proměnlivé velikosti, velké operátory dvojí velikosti). Tento font rovněž odborník opatří požadovanými `\fontdimen` a odkazy typu „následník“. My jeho práci využijeme při deklaraci rodiny 3. Antikva a kurzíva (rodiny 0 a 1) se může použít přímo z textových variant fontů. Není proto nutné tyto fonty opatřovat speciálními `\fontdimen`.

V plainu jsou použity fonty Computer Modern, které samozřejmě v případě matematických fontů požadované parametry `\fontdimen` mají. Rodiny 0 až 3 jsou v plainu deklarovány takto:

	`\textfont`	`\scriptfont`	`\scriptscriptfont`
• Rodina 0:	`cmr10` (`\tenrm`)	`cmr7` (`\sevenrm`)	`cmr5` (`\fiverm`)
• Rodina 1:	`cmmi10` (`\teni`)	`cmmi7` (`\seveni`)	`cmmi5` (`\fivei`)
• Rodina 2:	`cmsy10` (`\tensy`)	`cmsy7` (`\sevensy`)	`cmsy5` (`\fivesy`)
• Rodina 3:	`cmex10` (`\tenex`)	`cmex10` (`\tenex`)	`cmex10` (`\tenex`)

V závorce uvádíme přepínač, který byl u každého fontu deklarován při zavedení primitivem `\font`. Přepínač je tedy možné použít při sazbě v horizontálním módu.

V uvedené tabulce si všimneme dvou zvláštností. Za prvé, pro rodinu 1 není použita běžná textová kurzíva (`cmti*`), ale speciální matematická. Kresby znaků jsou nepatrně širší a kerningové a ligační tabulky jsou rozdílné. Vidíme, že Knuth udělal v případě matematické sazby hodně pečlivé práce.

Druhá zvláštnost. V rodině 3 máme ve všech třech variantách zaveden jediný font. To většinou stačí, protože velké operátory se v matematických vzorcích používají obvykle jen v základní velikosti. Větší varianty závorek a dalších znaků proměnlivé velikosti začne TEX hledat až po vyčerpání možností použít základní verzi v dané velikosti a ve „větších“ matematických stylech. Srozumitelněji tuto vlastnost ilustrujeme na příkladě. Uvažujme:

```
95 $$ \scriptscriptstyle \left( ⟨vzorec⟩ \right. $$
```

Protože je `\delcode‘\(="028300`, TEX nejprve vyzkouší závorku z pozice `"28` (hexadecimálně) z rodiny 0. Nejprve vezme závorku z fontu odpovídajícímu aktuálnímu stylu (tj. `cmr5`). Nevyhovuje-li závorka odtud, TEX vezme další závorku z fontu „většího“ stylu (tj. `cmr7`) znovu z pozice `"28`. Pak je na řadě font `cmr10`, stále stejná pozice. Pokud nevyhovuje ani tato velikost závorky, TEX teprve nyní přejde podle údaje z `\delcode` na rodinu 3, pozici 0. Začíná fontem této rodiny podle aktuálního stylu. Stačí tedy, aby zde pokračovala řada následníků závorek, které jsou větší než závorka v základní textové velikosti.

Pokud sázíme vzorečky, kde jsou velké operátory i v indexech, například:

$$A_{\sum,0,\infty}(x) = \sum_{i=0}^{\infty} x_i$$

tak to dopadne ucházejícím způsobem, protože jsme sazbu provedli ve stylu D (display) a tam zapracuje dvojí velikost symbolu sumy. Správnější by ale bylo v indexech používat pro tento symbol místo `\sum` sekvenci `\Sigma`. Pak dostaneme tento výsledek:

$$A_{\Sigma,0,\infty}(x) = \sum_{i=0}^{\infty} x_i$$

Pokud nutně potřebujeme sázet velké operátory v indexech, můžeme rozšířit zavedení fontů z plainu takto:

```
96 \font\sevenex=cmex10 scaled 700 \font\fiveex=cmex10 scaled 500
97 \scriptfont3=\sevenex  \scriptscriptfont3=\fiveex
```

Toto rozšíření způsobí v jistých nepatrných drobnostech rozdílnou sazbu, než byla původně. Například tloušťka zlomkové čáry bude v indexové velikosti menší než v základní. Jak se s těmito drobnostmi vyrovnat, to si povíme na konci této sekce. □

Plain deklaruje další rodiny fontů, které odpovídají použitým textovým fontům:

Rodina	`\textfont`	`\scriptfont`	`\scriptscriptfont`
• **4.** (kurzíva)	`cmti10` (`\tenit`)	—	—
• **5.** (skloněný)	`cmsl10` (`\tensl`)	—	—
• **6.** (tučný)	`cmbx10` (`\tenbf`)	`cmbx7` (`\sevenbf`)	`cmbx5` (`\fivebf`)
• **7.** (strojopis)	`cmtt10` (`\tentt`)	—	—

Vidíme, že většina těchto rodin je neúplná. Pokud bychom chtěli sázet index například strojopisem (`cmtt*`), TEX ohlásí chybu, že v rodině 7 není deklarován `\scriptfont`. To nás asi příliš netrápí. Kdybychom přesto takový index potřebovali, umíme už doplnit deklaraci příslušné rodiny.

V csplainu jsou v rodinách 0, 4, 5, 6, 7 místo CM fontů s názvy souborů `cm*` použity $\mathcal{CS}$-fonty s analogickými názvy `cs*`. Protože se tyto fonty v pozicích menších než 128 zcela shodují s fonty CM, není v matematické sazbě žádný rozdíl. Matematické fonty rodin 1, 2 a 3 zůstávají v csplainu zcela nezměněny.

Pro čísla rodin 0 až 3 jsou deklarace zapisovány přímo (třeba zápisem `\textfont3`). U dalších rodin se většinou používá alokační makro `\newfam`. Také plain alokuje ostatní rodiny tímto makrem:

```
98 \newfam\itfam % textová kurzíva, rodina 4
99 \newfam\slfam % skloněná antikva, rodina 5
```

```
100 \newfam\bffam % polotučný řez, rodina 6
101 \newfam\ttfam % strojopis, rodina 7
```

I my budeme používat toto alokační makro a nebudeme se dále zajímat, jaké skutečné číslo rodina má. Samozřejmě to můžeme dělat jen potud, pokud nevyčerpáme maximální počet šestnácti rodin. Při každém zpracování matematického seznamu TEX dokáže pracovat současně s nejvýše 16 rodinami fontů. Samozřejmě, při zpracování dalšího matematického seznamu ve stejném dokumentu může být těchto 16 rodin nastaveno jinak. □

• **Výběr fontu pro sazbu znaku.** Nyní si zopakujeme, jak je postupně zpracováván vstupní povel pro sazbu znaku v matematickém módu. Cestu od vstupního povelu ke znaku ve fontu můžeme naznačit takto:

$$\langle \mathit{ASCII} \rangle \text{ nebo } \texttt{\sekvence} \xrightarrow{(1)} \text{rodina, pozice} \xrightarrow{(2)} \text{font, pozice.}$$

Transformace (1) se provádí při sestavování matematického seznamu (sekce 5.1). Údaje o rodině a pozici TEX získá z deklarace `\mathcode` (u $\langle \mathit{ASCII} \rangle$) nebo z `\mathchardef` (u `\sekvence`). Transformace (2) se děje až v okamžiku konverze z matematického do horizontálního seznamu (sekce 5.2). Údaje o fontu TEX získá podle deklarace rodiny (`\textfont` až `\scriptscriptfont`) a závisejí na právě zpracovávaném matematickém stylu.

Při transformaci (1) může být údaj o rodině přečten z hodnoty registru `\fam`. To se stane právě tehdy, když je hodnota `\fam` nezáporná a zároveň třída znaku je rovna 7. Připomeneme, které znaky mají v plainu třídu 7 :

- Písmena anglické abecedy (implicitní rodina 1)
- Číslice 0123456789 (implicitní rodina 0)
- Znaky velké řecké abecedy (implicitní rodina 0)

Podrobněji jsme o třídách mluvili na straně 146. □

Pokud chceme přepínat mezi fonty v matematickém módu, nelze vůbec použít přepínače textových fontů. Také je chybou psát:

```
102 $$ abc \textfont1=\tenbf xyz $$
```

Po vyzkoušení tohoto příkladu zjistíme, že všech šest písmen (abcxyz) je sázeno ve fontu `\tenbf`. Možná někoho překvapí, proč to postihlo i písmena abc. Vysvětlíme si to. Přiřazení `\textfont1=\tenbf` nemá při sestavování matematického seznamu žádný vliv. Všechny znaky budou mít implicitní rodinu 1. V druhém průchodu při konverzi matematického seznamu na horizontální se pak tato rodina transformuje podle `\textfont1` na `\tenbf`.

Pokud chceme přepínat sazbu písmen anglické abecedy nebo číslic (přesněji sazbu znaků s třídou 7), musíme v matematickém módu nastavit registr `\fam` na nezápornou hodnotu. Při každém startu matematického módu je tento registr nastaven na -1, takže bez jeho změny se použijí implicitní rodiny. Například zápis:

```
$ abc + {\rm abc} + xyz \bf + xyz_{012} $
```

způsobí sazbu: $abc + \mathrm{abc} + xyz + \mathbf{xyz_{012}}$. Makro `\rm` totiž nastavuje `\fam` na hodnotu 0 (viz část B), takže druhá skupina „abc“ bude sázena v rodině fontů 0. Dále první skupina xyz bude mít implicitní rodinu 1, protože je znovu `\fam=-1`. Konečně `\bf` nastavuje `\fam` na hodnotu `\bffam` (neboli na hodnotu 6) a poslední skupina xyz včetně indexu bude tučně. Znak „+“ bude vždy sázen z rodiny 0, protože jeho třída není rovna 7 (`\mathcode` znaku „+“ je `"202B`). □

Nastavení aktuálního fontu pro horizontální mód (říkáme zkráceně textový font) nemá podle uvedených pravidel v matematické sazbě žádný vliv. Existují ovšem tyto výjimky:

- Např. při `\hbox{`⟨*text*⟩`}` uvnitř matematiky je ⟨*text*⟩ sázen textovým fontem.
- Velikost pružné mezery při povelu `\␣` je určena textovým fontem.
- Velikost jednotek `em` a `ex` závisí na textovém fontu. □

• **PostScriptové fonty a matematická sazba.** Představme si, že máme například skupinu PostScriptových fontů, které bychom rádi použili v textové sazbě. Přitom balík neobsahuje vhodný matematický font s odpovídajícími `\fontdimen`, takže není možné jej přímo použít v matematické sazbě. Proto bychom rádi ponechali matematickou sazbu v Computer Modern, ale textovou sazbu bychom nastavili na PostScriptový font. Takový kompromis se velmi často v TEXu používá, protože běžné PostScriptové fonty skutečně nelze bez pracných úprav přímo použít v matematické sazbě se všemi vymoženostmi, které TEX s matematickými fonty dokáže.

Při řešení vytyčeného úkolu nám velmi pomůže, že fonty pro matematickou sazbu jsou na textových nezávislé. Například použijeme font Times-Roman se jménem souboru `ptmr8z`. Budeme jej kombinovat s variantami Bold (`ptmb8z`), Italic (`ptmri8z`) a pro strojopis použijeme Courier (`pcrr8u`). Pak stačí v plainu zavést nové fonty:

```
\font\tenrm=ptmr8z at 10pt  % Times-Roman
\font\tenbf=ptmb8z at 10pt  % Times-Bold
\font\tenit=ptmri8z at 10pt % Times-Italic
\font\tentt=pcrr8u at 10pt  % Courier
\tenrm
```

a můžeme používat přepínače \bf, \it, \rm a \tt pro sazbu novými fonty v horizontálním módu. V matematickém módu zůstává sazba fontem Computer Modern. Proč to funguje? V plainu bylo například řečeno:

```
109 \textfont0=\tenrm
```

a tím se do rodiny 0 zavedl font, který v té době byl reprezentován sekvencí \tenrm. Byl to font cmr10 (v csplainu csr10). Pokud byla *později* na řádku 104 předefinována sekvence \tenrm, na nastavení rodiny 0 to už nemá vliv. Takže matematická sazba zůstává v Computer Modern.

Uvedeme několik důsledků a problémů používání dvojího fontu. Uživatel by si měl být vědom, že pokud napíše třeba:

```
110 Číslo $-2$ musím dát mezi dolary, ale číslo 2 nemusím.
```

pak se sazba dvojky mezi dolary provede v Computer Modern, zatímco sazba dvojky mimo dolary bude v PostScriptovém fontu. Toto míchání různých rodin není asi příliš vhodné. Pokud tedy nastavíme rodinu 0 na nový PostSriptový font, můžeme mít zdánlivě po problému. Skutečně, nyní máme i dvojku mezi dolary v PostScriptovém fontu a znak minus je brán z rodiny 2, kde jsou matematické symboly z původního Computer Modern. Ale problémy pokračují. Nyní už nemůžeme psát třeba \Sigma, protože ta je deklarovaná v plainu jako \mathchardef\Sigma="7006, takže znak „Σ“ je implicitně sázen z rodiny 0. Tento znak přitom v Computer Modern Roman je, ale nenajdeme ho v Times Roman. Museli bychom zavést PostScriptový font Symbol, tam vybrat použité znaky a předefinovat pomocí \mathcode a \mathchardef všechny potřebné matematické znaky. Nikdy nebudeme zcela spokojeni, protože nelze kombinovat původní \fontdimen pro matematickou sazbu společně se znaky fontu Symbol. I kdybychom nastavili nová potřebná \fontdimen, stále se bez Computer Modern neobejdeme, protože ve fontu Symbol nejsou „natahovací“ závorky a podobné TEXovské speciality. □

• **Přepínání velikosti matematické sazby.** V dalším příkladě budeme řešit jiný úkol. Zůstáváme pro jednoduchost při fontech Computer Modern, ale budeme potřebovat například pro nadpisy nebo poznámky pod čarou jinou velikost fontu. Přitom i tam budeme používat matematickou sazbu.

Vytvoříme si makro \footnotefonts, které nastaví jednak textové a jednak matematické fonty do velikosti 8 bodů. Indexy první úrovně budou mít 6 bodů a druhé úrovně 5 bodů. Při definici poznámky pod čarou pak jen napíšeme:

```
111 \def\footnote#1{...{\footnotefonts \baselineskip=10pt #1}...}
```

a budeme mít fonty nastaveny do požadované velikosti. Přitom budou fungovat přepínače `\bf`, `\rm`, `\it`, které budou pracovat s osmibodovými fonty. Při tvorbě makra `\footnotefonts` budeme pro začátek opisovat z dodatku E v TEXbooku:

```
\def\footnotefonts{\def\rm{\fam0\eigrm}% 8 bodové písmo
 \textfont0=\eigrm \scriptfont0=\sixrm \scriptscriptfont0=\fiverm
 \textfont1=\eigi  \scriptfont1=\sixi  \scriptscriptfont1=\fivei
 \textfont2=\eigsy \scriptfont2=\sixsy \scriptscriptfont2=\fivesy
 \textfont3=\eigex \scriptfont3=\eigex \scriptscriptfont3=\eigex
 \textfont\bffam=\eigbf \scriptfont\bffam=\sixbf
 \scriptscriptfont\bffam=\fivebf
 \def\bf{\fam\bffam\eigbf} \let\it=\eigit \let\tt=\eigtt \rm}
```

Zde jsme pro jednoduchost předpokládali, že v matematické sazbě budeme používat pouze základní rodiny 0 až 3 a rodinu `\bf`. Přepínání `\it` a `\tt` v matematice v poznámkách pod čarou dělat nebudeme. Přepínač `\sl` vůbec nebudeme používat. Povely `\textfont` až `\scriptscriptfont` deklarují rodiny lokálně, takže po opuštění skupiny v makru `\footnote` se sazba vrací do desetibodového písma včetně matematiky.

Před prvním použitím makra `\footnotefonts` musíme ještě zavést do TEXu fonty `csr*` (`\eigrm`, `\sixrm`), `cmmi*` (`\eigi`, `\sixi`), `cmsy*` (`\eigsy`, `\sixsy`), dále `cmex10 at8pt` (`\eigex`), `csbx*` (`\eigbf`, `\sixbf`) a konečně `csti8` (`\eigit`) a `cstt8` (`\eigtt`). Varianty `\five...` už máme zavedeny v plainu. Vidíme tedy, že ještě musíme jedenáctkrát použít primitiv `\font` a zavést metriky fontů odpovídající velikosti. Pokud bychom potřebovali použít fonty pro další velikosti (kapitoly, sekce), pak by byl kód zbytečně únavný, rozsáhlý a nudný. Proto v dalším příkladě ukážeme jiné řešení. □

Budeme pracovat s identifikátory fontů ve tvaru ⟨*fam-t*⟩⟨*size*⟩, přičemž ⟨*fam-t*⟩ je textové označení rodiny, například `rm`, `mi`, `sy`. Údaj ⟨*size*⟩ označuje velikost fontu vyjádřenou číselně. Identifikátory fontů tedy budou vypadat třeba takto: `rm10`, `mi12`. Nejprve vytvoříme makra, která tyto identifikátory obsluhují:

```
\def\setsizes #1 #2 #3 {%
  \def\sizeT{#1}\def\sizeS{#2}\def\sizeSS{#3}}
\def\loadfonts #1 #2#3#4{%  ⟨fam-t⟩ {⟨file1⟩}{⟨file2⟩}{⟨file3⟩}
  \expandafter \font\csname#1\sizeT\endcsname=#2
  \expandafter \font\csname#1\sizeS\endcsname=#3
  \expandafter \font\csname#1\sizeSS\endcsname=#4 }
\def\setfamf #1=#2 #3#4{%  ⟨fam-n⟩=⟨fam-t⟩ ⟨command⟩⟨size⟩
  \expandafter \ifx\csname#2#4\endcsname \relax
  \else #3#1=\csname#2#4\endcsname \fi}
\def\setonefam #1=#2 {% ⟨fam-n⟩=⟨fam-t⟩
  \setfamf#1=#2 \textfont\sizeT \setfamf#1=#2 \scriptfont\sizeS
```

```
131   \setfamf#1=#2 \scriptscriptfont\sizeSS
132   \expandafter \def\csname#2\endcsname{\fam#1%
133      \csname#2\sizeT\endcsname}}  % např. \def\rm{\fam0\rm10}
134 \def\setallfams{\setonefam0=rm \setonefam1=mi
135   \setonefam2=sy \setonefam3=ex \setonefam\bffam=bf }
```

Smysl těchto maker vyplyne ze způsobu jejich použití. Nejprve se pomocí `\setsizes` nastaví trojice velikostí, se kterou se bude pracovat, a pak se pomocí `\loadfonts` postupně zavádějí trojice fontů v dané velikosti. Například pro velikosti nadpisů kapitol použijeme hodnoty 14, 12, a 10 (po řadě základní velikost, indexová velikost první a druhé úrovně). Fonty pro nadpisy kapitol zavedeme takto:

```
136 \setsizes 14 12 10 % pro \chapfonts
137 \loadfonts rm {csr12 scaled\magstep1}  {csr12}         {csr10}
138 \loadfonts mi {cmmi12 scaled\magstep1} {cmmi12}        {cmmi10}
139 \loadfonts sy {cmsy10 scaled\magstep2} {cmsy10 at12pt} {cmsy10}
140 \loadfonts ex {cmex10 scaled\magstep2}
141    {cmex10 scaled\magstep2} {cmex10 scaled\magstep2}
142 \loadfonts bf {csbx12 scaled\magstep1} {csbx12} {csbx10}
143 \def\chapfonts {\setsizes 14 12 10 \setallfams \bf}
```

Vidíme, že jsme snadno a rychle nejen zavedli fonty, ale též jsme *jedním řádkem* definovali makro `\chapfonts`, které je analogické makru `\footnotefonts` z předchozí ukázky. Podobně zavedeme třeba fonty ve velikosti sekcí:

```
144 \setsizes 12 10 8 % pro \secfonts
145 \loadfonts rm {csr12}         {csr10}         {csr8}
146 \loadfonts mi {cmmi12}        {cmmi10}        {cmmi8}
147 \loadfonts sy {cmsy10 at12pt} {cmsy10}        {cmsy8}
148 \loadfonts ex {cmex10 at12pt} {cmex10 at12pt} {cmex10 at12pt}
149 \loadfonts bf {csbx12}        {csbx10}        {csbx8}
150 \def\secfonts {\setsizes 12 10 8 \setallfams \bf}
```

Můžete namítnout, že je neefektivní zavádět stejné fonty několikrát za sebou, dokonce pod jiným názvem. Například font `csr10` byl v plainu zaveden jako `\tenrm` a my jej nyní ještě dvakrát zavádíme jako `rm10`. To ovšem vůbec nevadí, protože TEX se k tomuto problému staví chytře. Pokud je už font zaveden, vůbec znovu neotevírá soubor s fontem. Pouze bude přiřazen k už zavedenému fontu nový identifikátor.

Nyní stačí naše makro vyzkoušet:

```
151 \def\chap #1 \par{\bigskip{\chapfonts \noindent #1\par}
152    \nobreak\bigskip ...}
153 \def\sec #1 \par{\medskip{\secfonts \noindent #1\par}
```

```
    \nobreak\medskip ...}

\chap 0 funkci $f(t)=\int_0^t e^{x^2}\,{\rm d}x \times\Gamma(t)$

\sec Základní vlastnosti funkce
     $f(t) = \int_0^t e^{x^2}\,{\rm d}x \times \Gamma(t)$

Vztahem $$ f(t) = \int_0^t e^{x^2}\,{\rm d}x \times \Gamma(t) $$
definujeme funkci, kterou budeme nazývat ...
```

Kdyby nás tento způsob práce s `\expandafter` a `\csname...\endcsname` bavil více a chtěli bychom makro zdokonalovat, asi bychom postupně dospěli k něčemu podobnému, jako je Mittelbachovo NFSS (New font selection scheme). Ovšem při práci na sazbě konkrétního dokumentu nepotřebujeme tak širokou univerzálnost makra. V plainu je vhodné hledat rozumný kompromis mezi univerzálností a přitom ještě dostatečnou přehledností makra. O NFSS se rozhodně nedá tvrdit, že do něj nahlédneme a okamžitě víme, co to dělá. □

• **Matematická sazba v polotučném řezu.** Při prohlížení výsledku z předchozí ukázky nás může napadnout, že matematická sazba má sice odpovídající velikost, ale v souvislosti s okolním textem sázeným v nadpisech polotučným řezem jsou znaky z matematiky příliš hubené. Doplníme si proto naše makro o sekvenci `\boldmath`, která například pro nadpisy kapitol zařídí, že bude i matematika sázena polotučně.

Po podrobnějším průzkumu matematických fontů z Computer Modern zjistíme, že tam jsou varianty pro polotučnou matematickou kurzívu (`cmmib10`) a pro matematické symboly (`cmbsy10`). Tyto fonty využijeme a zavedeme je společně s fonty pro kapitolu (při `\setsizes 14 12 10`) takto:

```
\loadfonts mib {cmmib10 at14.4pt} {cmmib10 at12pt} {cmmib10}
\loadfonts syb {cmbsy10 at14.4pt} {cmbsy10 at12pt} {cmbsy10}
```

Nyní můžeme makro `\boldmath` definovat tímto způsobem:

```
\def\boldmath{\setonefam0=bf \setonefam1=mib \setonefam2=syb }
```

Makro tedy deklaruje jinak rodinu 0 (místo `rm` bude použito `bf`) a dále rodiny 1 a 2. Rodinu 3 ponecháme nezměněnu, protože k ní nemáme polotučnou variantu fontu `cmex10`. Vzhled sazby i tak bude uspokojivý. Můžete si třeba vyzkoušet:

```
\def\chap #1 \par{\bigskip{\chapfonts\boldmath \noindent #1\par}
    \nobreak\bigskip ...}
```

Pokud bychom chtěli, aby makro `\boldmath` fungovalo i v jiných velikostech, stačí pro tyto velikosti zavést skupiny fontů pomocí `\loadfonts mib` a `\loadfonts syb`. □

• **Deklarace dalších rodin fontů.** Uvedeme si nyní příklad, ve kterém zavedeme do TEXu další skupinu matematických fontů, která obohatí náš sortiment znaků v matematické sazbě. Až dosud jsme v ukázkách nahrazovali rodiny 0 až 3 jinými alternativami fontů stejných vlastností, abychom dosáhli většího nebo polotučného řezu. Nyní deklarujeme *vedle* těchto základních rodin další rodiny a budeme v jednom vzorečku kombinovat znaky ze základních rodin i z rodin nových.

V článku *Dvojité hranaté závorky v matematice* z TEX-bulletinu 1/93 jsem tuto problematiku naznačil. V následujícím příkladě ji zde znovu zopakuji.

Uvažujme fonty z balíku `bbold` pro sazbu symbolů s dvojitou vertikální kresbou. Balík obsahuje font `bbold10`, v němž jsou velká a malá písmena abecedy latinské i řecké (𝔸, 𝔹, 𝕒, 𝕓, 𝕬, 𝕭, ...) a dále některé speciální znaky (𝕤, 𝕥, 𝕜, 𝕔, 𝕕, 𝕞, ≤, ≥, 𝟘, 𝟙, 𝟚, 𝟛, ...). Tento font je ještě ve variantách `bbold12` a `bbold5-9`. Dále je v balíku font `cspex10`, který obsahuje některé méně běžné velké operátory (⨁, ⨂). Operátory jsou prostřednictvím následníků ve fontu seskupeny do dvojic: malá a velká varianta. Stačí tedy deklarovat znak třídy 1 (typ Op), který se odvolává na malou variantu, a TEX použije malou nebo velkou variantu podle kontextu (`\textstyle`, `\displaystyle`). Ve fontu `cspex10` jsou dále prostřednictvím následníků implementovány zvětšující se závorky. Jejich základní velikost najdeme v `bbold` a vypadají takto: ⟦, ⟧.

Pro nové fonty zavedeme dvě rodiny `\bbfam` a `\bebfam`. První obsahuje font `bbold` ve třech různých velikostech. Druhá rodina bude obsahovat font `cspex10` v jediné velikosti, podobně jako font `cmex10` v rodině 3. Fonty zavedeme takto:

```
\newfam\bbfam \newfam\bebfam
\font\bbtext=bbold10
\font\bbscript=bbold7
\font\bbscriptscript=bbold5
\font\bbex=cspex10

\textfont\bbfam=\bbtext \scriptfont\bbfam=\bbscript
   \scriptscriptfont\bbfam=\bbscriptscript
\textfont\bebfam=\bbex  \scriptfont\bebfam=\bbex
   \scriptscriptfont\bebfam=\bbex
```

Pomocí maker z předchozího příkladu (řádky 120 až 135) můžeme zavést fonty elegantněji takto:

```
\setsizes 10 7 5
\loadfonts bbsy {bbold10} {bbold7}  {bbold5}
\loadfonts bbex {cspex10} {cspex10} {cspex10}
\newfam\bbfam   \setonefam\bbfam=bbsy
\newfam\bebfam  \setonefam\bebfam=bbex
```

Nyní přichází vlastní práce. Musíme deklarovat nové řídicí sekvence, které budou využívat přítomnost nových rodin fontů. Vytiskneme si proto pomocí Knuthova makra `testfont` tabulky použitých fontů (o tomto makru je podrobnější zmínka na straně 288). Podle těchto tabulek začneme jednotlivým pozicím znaků přiřazovat řídicí sekvence. Protože pracujeme s šestnáctkovými zápisy kódů matematických znaků, je výhodné použít makro, které převádí číselný údaj o rodině fontu na šestnáctkovou číslici:

```
\def\sixt#1{\ifcase#10\or1\or2\or3\or4\or5\or6\or7\or8\or9\or
    A\or B\or C\or D\or E\or F\fi}
```

Nejprve definujeme sekvence `\[` a `\]` pro dvojité hranaté závorky pružné velikosti:

```
\def\[{\delimiter"4\sixt\bbfam5B\sixt\bebfam02 }
\def\]{\delimiter"5\sixt\bbfam5D\sixt\bebfam03 }
```

Základní velikost otevírací závorky se bere z rodiny `\bbfam` z pozice `5B` a první větší velikost se nalézá v rodině `\bebfam` na pozici `02`. Pokud není závorka použita za `\left` nebo `\right`, bude mít třídu 4 (typ atomu Open). Podobně čteme deklaraci druhé závorky. Všimněme si, že v definicích je na konci záměrně mezera, aby se nám při použití `\[1` neslilo číslo pro `\delimiter` s číslovkou, zapsanou uživatelem.

Dále si definujeme makro `\bbmathcodes`, které přepíná standardní nastavení `\mathcode` některých znaků pro alternativy ve fontu `bbold`. Podobně bude makro měnit významy některých řídicích sekvencí deklarovaných jako `\matchardef`.

```
\def\bbmathcodes{%
    \mathcode`\*="2\sixt\bbfam2A   \mathcode`+ ="2\sixt\bbfam2B
    \mathcode`\(="4\sixt\bbfam28   \mathcode`\)="5\sixt\bbfam29
    ⟨... atd...⟩
    \mathchardef\alpha="0\sixt\bbfam0B
    \mathchardef\beta ="0\sixt\bbfam0C
    ⟨... atd...⟩ }
```

Obsah makra bude odpovídat požadavkům, které znaky a sekvence budeme chtít použít. Vycházíme z tabulky fontu `bbold` a inspirovat se můžeme pohledem do tabulek symbolů z plainu, které jsou uvedeny v následující sekci 5.4. ⟨*ASCII*⟩ kódy většiny znaků píšeme tak, jak jsme zvyklí: `` `\* ``, `` `\- ``, ale nemůžeme si to dovolit

v případě `‘\+`. Plain totiž používá sekvenci `\+` jako makro pro prostředí `\tabalign` a dává jí příznak `\outer`.

Mimo makro `\bbmathcodes` můžeme definovat zcela nové řídicí sekvence pro velké operátory z fontu `cspex10`. Například po:

```
194 \mathchardef\squarcap="1\sixt\bebfam46
195 \mathchardef\squarcapcup="1\sixt\bebfam48
196 \mathchardef\parallelism="1\sixt\bebfam4A
197 \mathchardef\interleaving="1\sixt\bebfam4C
198 \mathchardef\Dijkstra="1\sixt\bebfam4E
199 \mathchardef\circlevee="1\sixt\bebfam50
200 \mathchardef\circlewedge="1\sixt\bebfam52
```

můžeme používat třeba sekvenci `\circlevee` ve významu operátoru $\bigcirc\!\!\!\vee$.

Konečně definujeme pro naši novou skupinu fontů přepínač `\bb` takto:

```
201 \def\bb{\fam\bbfam \bbmathcodes \bbtext} % klasické zavedení
202 \def\bb{\bbsy \bbmathcodes} % při \loadfonts, řádky 178--182
```

V matematickém módu pracuje `\fam\bbfam` a `\bbmathcodes`. Nejprve se nastavuje explicitní rodina na `\bbfam`, takže všechny znaky s třídou 7 budou sázeny fonty této rodiny. Například `$\bb A$` vytvoří $\mathbb{A}$, protože znak `A` má třídu 7. Makro `\bbmathcodes` přepíná lokálně kódy dalších znaků a řídicí sekvence. Proto při `$$ \alpha + {\bb \alpha + \beta} = \beta $$` dostáváme

$$\alpha + \mathbb{\alpha + \beta} = \beta$$

V horizontálním módu je zase významné, že makro `\bb` obsahuje sekvenci `\bbtext`, která přepíná aktuální textový font. □

• **Struktura informací v matematických fontech.** Nebudeme uvádět tabulky fontů, protože tabulky si dokáže čtenář vytisknout sám. Připomeneme, že stačí spustit `tex testfont`. TEX se pak zeptá na jméno fontu a vytiskne tabulku fontu nebo vzorek textu. Zaměříme se tedy jen na informace, které z těchto tabulek nelze přímo vyčíst.

Už dříve jsme slíbili vysvětlit pojem *následník* znaku ve fontu. Zatím jsme říkali, že každý znak může obsahovat údaje o jednom nebo více následnících. To není zcela přesné. Uvedeme to nyní na pravou míru.

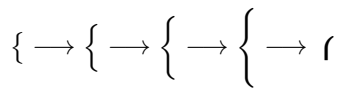
V METAFONTových zdrojových textech fontu je možné deklarovat posloupnost znaků pomocí příkazu `charlist`. To je vlastně posloupnost následníků, o které

jsme mluvili výše. Ovšem tato posloupnost je konečná a každý znak v ní má jediného následníka. Kromě toho každý znak může obsahovat čtyři údaje o dalších znacích pomocí METAFONTového příkazu `extensible`. Jakmile při vyhledávání závorky TEX narazí na znak, který má deklaraci `extensible`, není tento znak použit a místo něj TEX sestaví závorku ze segmentů z uvedené čtveřice podle `extensible`.

Lépe to pochopíme, pokud se podíváme například do souboru `bigdel.mf`, na jehož začátku jsou všechny deklarace pro závorky typu delimiter fontu `cmex10`. Například posloupnost následníků pro složenou závorku je deklarována takto:

```
203 charlist oct"010": oct"156": oct"032": oct"050": oct"070";
```

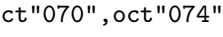
což deklaruje řadu těchto následníků:



Poslední znak v této posloupnosti má deklaraci `extensible` tvaru:

```
204 extensible oct"070": oct"070",oct"074",oct"072",oct"076";
```

TEX v takovém případě znak „⎛“ nesází, ale vytvoří závorku složenou ze segmentů, uvedených v `extensible`:



První údaj znamená horní okraj závorky, druhý střed, třetí spodní okraj závorky a poslední znak je použit pro opakované vyplnění vlastního těla závorky. Každý z uvedených údajů, kromě posledního, může být prázdný (v METAFONTu se to označí kódem nula). Například kulaté závorky nemají střed. Nebo znaky „‖“ mají jen znak pro opakované vyplnění.

Poznamenejme, že informace o následnících ve fontu jsou v plainu použity pouze ve fontu `cmex10` s velkými závorkami a operátory. □

Pro matematickou sazbu jsou nejdůležitější parametry `\fontdimen` zavedené do rodiny 2 a 3. Hodnoty těchto parametrů lze zjistit například takto:

```
205 \newcount\num
206 \def\body{\global\advance\num by1
207   \the\num & \the\fontdimen\num\textfont2,&
208   \the\fontdimen\num\scriptfont2,&
209   \the\fontdimen\num\scriptscriptfont2\cr
210   \ifnum \num<22 \expandafter\body \fi }
211 {\bf Rodina 2:}\medskip
```

```
212 \halign{fontdimen \hfil#:&&\quad \hfil#\cr \body}
```

Významy parametrů `\fontdimen` 1 až 7 jsme už uvedli na straně 104 a nebudeme je zde opakovat. Jsou podstatné především pro sazbu v horizontálním módu, ale matematické fonty je mají také. Mezi nimi připomeneme jen `\fontdimen6` fontu rodiny 2, který určuje velikost hodnoty ⟨*quad*⟩ pro výpočet rozměru jednotky `mu`. Další `\fontdimen` fontu rodiny 2 mají tyto významy:

8. Posun čitatele nahoru vzhledem k ose v D, D'.
9. Posun čitatele nahoru vzhledem k ose v ostatních stylech.
10. Jako 9, ale při nulové výšce zlomkové čáry.
11. Posun jmenovatele dolů vzhledem k ose v D, D'.
12. Posun jmenovatele dolů vzhledem k ose v ostatních stylech.
13. Posun exponentu v D nahoru.
14. Posun exponentu v T, S nebo SS nahoru.
15. Posun exponentu v D', T', S' nebo SS' nahoru.
16. Posun indexu dolů, je-li prázdný exponent.
17. Posun indexu dolů, není-li prázdný exponent.
18. Pro výpočet minima posunutí exponentu nahoru.
19. Pro výpočet minima posunutí indexu dolů.
20. Minimální velikost závorky proměnlivé velikosti v D, D'.
21. Minimální velikost závorky proměnlivé velikosti v ostatních stylech.
22. Vzdálenost matematické osy od účaří směrem nahoru.

Je potřeba upozornit na to, že tato tabulka si neklade nárok na příliš velkou přesnost. Uvedli jsme ji pouze jako ilustraci. Naprosto přesně se s těmito údaji může čtenář seznámit v TEXbooku v dodatku G. Všechny parametry čte TEX z takového fontu rodiny 2, který zrovna odpovídá sázenému stylu. Například při stylu S je použito těchto 22 parametrů z fontu `\sevensy` (uvažujeme nastavení `\scriptfont2=\sevensy`). TEX tedy vlastně v matematické sazbě pracuje dohromady s $22 \times 3 = 66$ parametry rodiny 2.

V rodině 3 TEX pracuje s těmito parametry:

8. Implicitní výška vodorovných čar (zlomky, `\overline`, `\underline`).
9–13. Určují mezery kolem velkých operátorů.

Ve fontu rodiny 3 tedy TEX čte 13 parametrů `\fontdimen`. Prvních sedm v tomto případě asi nikdy nepoužije. Pokud bychom potřebovali některé parametry měnit, můžeme třeba psát:

```
213 \fontdimen8\scriptfont3 = \fontdimen8\textfont3
214 \fontdimen8\scriptscriptfont3 = \fontdimen8\textfont3
```

Uvedené přiřazení zaručí stejnou tloušťku zlomkové čáry ve všech stylech. To má smysl pouze tehdy, když do rodiny 3 zavádíme fonty různě velké a rozdílnost zlomkových čar nám vadí. Ve skutečnosti nám asi vadit nebude, protože se jedná o rozdíly pouhým okem téměř neviditelné. □

5.4. Symboly matematické sazby definované v plainu

Nejprve uvedeme `\mathcode` všech ASCII znaků. Písmena anglické abecedy `a`–`z`, `A`–`Z` mají `\mathcode` rovno `"7100` + ⟨*ASCII*⟩, tj. třída 7, rodina 1 a pozice podle ⟨*ASCII*⟩. Číslice `0123456789` mají kód `"7000` + ⟨*ASCII*⟩, tj. třída 7, rodina 0 a pozice podle ⟨*ASCII*⟩. Toto nastavení je implicitní z iniTEXu. Plain dále nastavuje `\mathcode` podle následující tabulky.

`^^@`	`"2201`	$\cdot$	`^^Q`	`"321B`	$\supset$	`"`	`"0022`	”	`=`	`"303D`	$=$
`^^A`	`"3223`	$\downarrow$	`^^R`	`"225C`	$\cap$	`#`	`"0023`	#	`>`	`"313E`	$>$
`^^B`	`"010B`	α	`^^S`	`"225B`	$\cup$	`$`	`"0024`	$	`?`	`"503F`	$?$
`^^C`	`"010C`	β	`^^T`	`"0238`	$\forall$	`%`	`"0025`	%	`@`	`"0040`	$@$
`^^D`	`"225E`	$\wedge$	`^^U`	`"0239`	$\exists$	`&`	`"0026`	&	`[`	`"405B`	$[$
`^^E`	`"023A`	$\neg$	`^^V`	`"220A`	$\otimes$	`’`	`"8000`		`\`	`"026E`	$\backslash$
`^^F`	`"3232`	$\in$	`^^W`	`"3224`	$\leftrightarrow$	`(`	`"4028`	(	`]`	`"505D`	$]$
`^^G`	`"0119`	π	`^^X`	`"3220`	$\leftarrow$	`)`	`"5029`	)	`^`	`"005E`	$\hat{}$
`^^H`	`"0115`	λ	`^^Y`	`"3221`	$\rightarrow$	`*`	`"2203`	$*$	`_`	`"8000`	
`^^I`	`"010D`	γ	`^^Z`	`"8000`		`+`	`"202B`	$+$	`‘`	`"0060`	‘
`^^J`	`"010E`	δ	`^^[`	`"2205`	$\diamond$	`,`	`"613B`	,	`{`	`"4266`	$\{$
`^^K`	`"3222`	$\uparrow$	`^^\`	`"3214`	$\leq$	`-`	`"2200`	$-$	`|`	`"026A`	$\vert$
`^^L`	`"2206`	$\pm$	`^^]`	`"3215`	$\geq$	`.`	`"013A`	.	`}`	`"5267`	$\}$
`^^M`	`"2208`	$\oplus$	`^^^`	`"3211`	$\equiv$	`/`	`"013D`	/	`~`	`"007E`	$\tilde{}$
`^^N`	`"0231`	∞	`^^_`	`"225F`	$\vee$	`:`	`"303A`	:	`^^?`	`"1273`	$\int$
`^^O`	`"0140`	∂	␣	`"8000`		`;`	`"603B`	;			
`^^P`	`"321A`	$\subset$	`!`	`"5021`	!	`<`	`"313C`	$<$			

Vidíme, že jsou nastaveny i kódy znaků, které v běžných editorech nevytvoříme: `^^@` až `^^_`. Tyto kódy se v praxi v matematické sazbě příliš nevyužívají. Místo toho používáme alternativní řídicí sekvence, jejichž tabulku uvedeme později.

Znaky s kódem $c \geq 128$ nemají `\mathcode` měněno, tj. je rovno přímo kódu c. To se projeví třeba při této školácké chybě: `$v~mat. módu píšu text$`. Na výstupu máme: $v\ mat.módupíšutext$. Vidíme tedy, že písmena anglické abecedy mají rodinu 1 a jsou sázena kurzívou (jako matematické proměnné), zatímco akcentovaná písmena naší abecedy mají rodinu 0. Jsou tedy sázena antikvou a správně nemají v matematickém módu co pohledávat.

Znak „'“ má \mathcode roven "8000, tj. chová se v matematickém módu aktivně. Je definován jako ^\prime (viz heslo ’$_{12}$ v části B). Dalším „matematicky aktivním“ znakem je „␣“. Token ␣$_{10}$ je v matematickém módu zcela ignorován, takže „aktivita“ mezery by přicházela v úvahu při změně její kategorie třeba na 12. Pak ale aktivní mezera expanduje na ␣$_{10}$ (jako při \obeyspaces), takže se zase nic neděje.

Při změně kategorie znaku „_“ na 12 se bude tento znak v matematickém módu chovat jako _, což kreslí podtržítko široké 0,3 em pomocí \hrule. Konečně aktivní ^^Z se chová jako \ne, tj. vytiskne se „$\neq$“. □

Následuje seznam všech matematických akcentů definovaných pomocí primitivu \mathaccent.

```
\def\acute{\mathaccent"7013 }      á
\def\bar{\mathaccent"7016 }        ā
\def\breve{\mathaccent"7015 }      ă
\def\check{\mathaccent"7014 }      ǎ
\def\ddot{\mathaccent"707F }       ä
\def\dot{\mathaccent"705F }        ȧ
\def\grave{\mathaccent"7012 }      à
\def\hat{\mathaccent"705E }        â
\def\vec{\mathaccent"017E }        a⃗
\def\tilde{\mathaccent"707E }      ã
\def\widetilde{\mathaccent"0365 }  ã
\def\widehat{\mathaccent"0362 }    â
```

Vpravo od definic vidíme aplikování příslušné řídicí sekvence na písmeno „a“. Například „\bar a“ dá $\bar a$.

Pouze poslední dva akcenty ("0365 a "0362) mají ve fontu následníky a mají tedy tendenci se roztahovat. Největší velikost ale není příliš krkolomná; schová pod sebe zhruba tři proměnné:

$$\widehat{abc}, \widetilde{abc}, \text{ ale } \widehat{abcde}, \widetilde{abcde} \text{ už nevystačí.}$$

Další symboly, které se chovají jako matematické akcenty, už nejsou definovány pomocí \mathaccent, ale jinými prostředky. Například:

```
$$ \overrightarrow{abc} \quad \overleftarrow{ABC} \quad
   \overbrace{xyz} \quad \underbrace{XYZ} $$
```

vytvoří šipky a svorky libovolné délky podle šířky výrazu, ke kterému jsou připojeny:

$$\overrightarrow{abc} \quad \overleftarrow{ABC} \quad \overbrace{xyz} \quad \underbrace{XYZ}$$

Definice uvedených maker vypadá v plainu takto:

```
\def\overrightarrow#1{\vbox{\m@th\ialign{##\crcr
      \rightarrowfill\crcr\noalign{\kern-1pt\nointerlineskip}
      $\hfil\displaystyle{#1}\hfil$\crcr}}}
\def\overleftarrow#1{\vbox{\m@th\ialign{##\crcr
      \leftarrowfill\crcr\noalign{\kern-1pt\nointerlineskip}
      $\hfil\displaystyle{#1}\hfil$\crcr}}}
\def\overbrace#1{\mathop{\vbox{
      \m@th\ialign{##\crcr\noalign{\kern3pt}
      \downbracefill\crcr\noalign{\kern3pt\nointerlineskip}
      $\hfil\displaystyle{#1}\hfil$\crcr}}}\limits}
\def\underbrace#1{\mathop{\vtop{\m@th\ialign{##\crcr
      $\hfil\displaystyle{#1}\hfil$
      \crcr\noalign{\kern3pt\nointerlineskip}
      \upbracefill\crcr\noalign{\kern3pt}}}}\limits}
\def\rightarrowfill{$\m@th\smash-\mkern-6mu%
  \cleaders\hbox{$\mkern-2mu\smash-\mkern-2mu$}\hfill
  \mkern-6mu\mathord\rightarrow$}
\def\leftarrowfill{$\m@th\mathord\leftarrow\mkern-6mu%
  \cleaders\hbox{$\mkern-2mu\smash-\mkern-2mu$}\hfill
  \mkern-6mu\smash-$}
\mathchardef\braceld="37A \mathchardef\bracerd="37B %znaky: ⎧, ⎫
\mathchardef\bracelu="37C \mathchardef\braceru="37D %znaky: ⎩, ⎭
\def\downbracefill{$\m@th \setbox0=\hbox{$\braceld$}%
  \braceld\leaders\vrule height\ht0 depth0pt \hfill\braceru
  \bracelu\leaders\vrule height\ht0 depth0pt \hfill\bracerd$}
\def\upbracefill{$\m@th \setbox0=\hbox{$\braceld$}%
  \bracelu\leaders\vrule height\ht0 depth0pt \hfill\bracerd
  \braceld\leaders\vrule height\ht0 depth0pt \hfill\braceru$}
\def\m@th{\mathsurround=0pt }
```

Vidíme, že `\overrightarrow` a `\overleftarrow` jsou definovány jako `\vbox`y v matematickém režimu, tj. budou se chovat jako atomy typu Ord. Vlastní `\vbox` obsahuje `\ialign` (tj. `\halign` s nulovým `\tabskip`) se dvěma řádky. Nejprve je šipka a potom (bohužel vždy ve stylu D) je sazba vzorečku, nad kterým má být šipka. Natahovací šipka `\rightarrowfill` je definována primitivem `\cleaders`, který opakuje znak minus. Makro `\smash` před tímto znakem nuluje výšku a hloubku výsledné konstrukce (o tomto makru viz část B).

Podobně pracují i makra `\overbrace` a `\underbrace`. Jen natahovací podoba závorky (např. `\downbracefill`) je propracovanější. Makro změří výšku znaku

\braceld. To bude výška pružné čáry, která je vytvořena pomocí \leaders. Celá závorka je poskládaná ze šesti segmentů: ⎧ \leaders ⎪ \leaders ⎩. □

Nyní uvedeme tabulku \delcode znaků. Tyto kódy pracují při použití znaku za primitivy \left a \right.

(	"028300	(	] "05D303	]	/ "02F30E /
)	"029301	)	< "26830A	⟨	\ "26E30F \
[	"05B302	[	> "26930B	⟩	| "26A30C |

Kromě toho znak „.“ má \delcode=0 (neviditelný delimiter) a ostatní znaky mají \delcode=-1 (není možné je v konstrukcích \left a \right použít).

Za \left a \right lze též použít některé řídicí sekvence, protože jsou definovány jako \delimiter. Zde je jejich abecední seznam:

```
\def\arrowvert{\delimiter"26A33C }
\def\Arrowvert{\delimiter"26B33D }
\def\backslash{\delimiter"26E30F }
\def\bracevert{\delimiter"77C33E }
\def\downarrow{\delimiter"3223379 }
\def\Downarrow{\delimiter"322B37F }
\def\langle{\delimiter"426830A }
\def\lbrace{\delimiter"4266308 }
\def\lceil{\delimiter"4264306 }
\def\lfloor{\delimiter"4262304 }
\def\lgroup{\delimiter"462833A }
\def\lmoustache{\delimiter"437A340 }
\def\rangle{\delimiter"526930B }
\def\rbrace{\delimiter"5267309 }
\def\rceil{\delimiter"5265307 }
\def\rfloor{\delimiter"5263305 }
\def\rgroup{\delimiter"562933B }
\def\rmoustache{\delimiter"537B341 }
\def\uparrow{\delimiter"3222378 }
\def\Uparrow{\delimiter"322A37E }
\def\updownarrow{\delimiter"326C33F }
\def\Updownarrow{\delimiter"326D377 }
\def\vert{\delimiter"26A30C }
\def\Vert{\delimiter"26B30D }
\let\}=\rbrace  \let\{=\lbrace  \let\|=\Vert
```

Vpravo od každé definice vidíme tři symboly. První odpovídá znaku sázenému bez použití primitivů \left, \right. Tento znak je určen prvními čtyřmi hexadecimálními číslicemi kódu. Pak následuje znak první větší velikosti, který je určen

posledními třemi číslicemi v kódu. Nakonec je vykreslen znak, který je výsledkem použití makra `\Big` na danou řídicí sekvenci. Obvykle se jedná o prvního následníka ve fontu.

Pokud má znak první větší velikosti následníky deklarované METAFONTovým příkazem `extensible` a nikoli pomocí `charlist`, není takový znak v algoritmech `\left`, `\right` vykreslen. Místo něj je okamžitě vykreslena závorka poskládaná ze segmentů, které jsou příkazem `extensible` deklarovány. Tohoto jevu si můžeme všimnout u `\arrowvert`, `\lmoustache` a některých dalších sekvencí. Podrobněji o následnících viz stranu 179. □

V další části sekce uvedeme tabulku, která bude nejrozsáhlejší. V abecedním pořadí seřadíme všechny řídicí sekvence, které plain deklaruje pomocí `\mathchardef`. Nejprve vidíme řídicí sekvenci, potom její matematický kód v hexadecimálním tvaru (třída, rodina, pozice) a nakonec je vytištěn symbol, který pro daný kód odpovídá implicitnímu nastavení fontů z plainu.

`\aleph`	`"0240`	$\aleph$	`\chi`	`"011F`	χ
`\alpha`	`"010B`	α	`\circ`	`"220E`	$\circ$
`\amalg`	`"2271`	$\amalg$	`\clubsuit`	`"027C`	$\clubsuit$
`\approx`	`"3219`	$\approx$	`\colon`	`"603A`	$:$
`\ast`	`"2203`	$*$	`\coprod`	`"1360`	$\coprod$
`\asymp`	`"3210`	$\asymp$	`\cup`	`"225B`	$\cup$
`\beta`	`"010C`	β	`\dagger`	`"2279`	$\dagger$
`\bigcap`	`"1354`	$\bigcap$	`\dashv`	`"3261`	$\dashv$
`\bigcirc`	`"220D`	$\bigcirc$	`\ddagger`	`"227A`	$\ddagger$
`\bigcup`	`"1353`	$\bigcup$	`\Delta`	`"7001`	Δ
`\bigodot`	`"134A`	$\bigodot$	`\delta`	`"010E`	δ
`\bigoplus`	`"134C`	$\bigoplus$	`\diamond`	`"2205`	$\diamond$
`\bigotimes`	`"134E`	$\bigotimes$	`\diamondsuit`	`"027D`	$\diamondsuit$
`\bigsqcup`	`"1346`	$\bigsqcup$	`\div`	`"2204`	$\div$
`\bigtriangledown`	`"2235`	$\bigtriangledown$	`\ell`	`"0160`	ℓ
`\bigtriangleup`	`"2234`	$\bigtriangleup$	`\emptyset`	`"023B`	$\emptyset$
`\biguplus`	`"1355`	$\biguplus$	`\epsilon`	`"010F`	ϵ
`\bigvee`	`"1357`	$\bigvee$	`\equiv`	`"3211`	$\equiv$
`\bigwedge`	`"1356`	$\bigwedge$	`\eta`	`"0111`	η
`\bot`	`"023F`	$\bot$	`\exists`	`"0239`	$\exists$
`\braceld`	`"037A`	$\braceld$	`\flat`	`"015B`	$\flat$
`\bracelu`	`"037C`	$\bracelu$	`\forall`	`"0238`	$\forall$
`\bracerd`	`"037B`	$\bracerd$	`\frown`	`"315F`	$\frown$
`\braceru`	`"037D`	$\braceru$	`\Gamma`	`"7000`	Γ
`\bullet`	`"220F`	$\bullet$	`\gamma`	`"010D`	γ
`\cap`	`"225C`	$\cap$	`\geq`	`"3215`	$\geq$
`\cdot`	`"2201`	$\cdot$	`\gg`	`"321D`	$\gg$
`\cdotp`	`"6201`	$\cdotp$	`\heartsuit`	`"027E`	$\heartsuit$

\Im	"023D	$\Im$	\Pi	"7005	Π
\imath	"017B	$\imath$	\pi	"0119	π
\in	"3232	$\in$	\pm	"2206	$\pm$
\infty	"0231	∞	\prec	"321E	$\prec$
\intop	"1352	$\int$	\preceq	"3216	$\preceq$
\iota	"0113	ι	\prime	"0230	$\prime$
\jmath	"017C	$\jmath$	\prod	"1351	$\prod$
\kappa	"0114	κ	\propto	"322F	$\propto$
\Lambda	"7003	Λ	\Psi	"7009	Ψ
\lambda	"0115	λ	\psi	"0120	ψ
\ldotp	"613A	$.$	\Re	"023C	$\Re$
\Leftarrow	"3228	$\Leftarrow$	\rho	"011A	ρ
\leftarrow	"3220	$\leftarrow$	\rhook	"312D	$\rhook$
\leftharpoondown	"3129	$\leftharpoondown$	\Rightarrow	"3229	$\Rightarrow$
\leftharpoonup	"3128	$\leftharpoonup$	\rightarrow	"3221	$\rightarrow$
\Leftrightarrow	"322C	$\Leftrightarrow$	\rightharpoondown	"312B	$\rightharpoondown$
\leftrightarrow	"3224	$\leftrightarrow$	\rightharpoonup	"312A	$\rightharpoonup$
\leq	"3214	$\leq$	\searrow	"3226	$\searrow$
\lhook	"312C	$\lhook$	\setminus	"226E	$\setminus$
\ll	"321C	$\ll$	\sharp	"015D	$\sharp$
\mapstochar	"3237	$\mapstochar$	\Sigma	"7006	Σ
\mid	"326A	$\mid$	\sigma	"011B	σ
\mp	"2207	$\mp$	\sim	"3218	$\sim$
\mu	"0116	μ	\simeq	"3227	$\simeq$
\nabla	"0272	∇	\smallint	"1273	$\smallint$
\natural	"015C	$\natural$	\smile	"315E	$\smile$
\nearrow	"3225	$\nearrow$	\spadesuit	"027F	$\spadesuit$
\neg	"023A	$\neg$	\sqcap	"2275	$\sqcap$
\ni	"3233	$\ni$	\sqcup	"2274	$\sqcup$
\not	"3236	$\not$	\sqsubseteq	"3276	$\sqsubseteq$
\nu	"0117	ν	\sqsupseteq	"3277	$\sqsupseteq$
\nwarrow	"322D	$\nwarrow$	\star	"213F	$\star$
\odot	"220C	$\odot$	\subset	"321A	$\subset$
\ointop	"1348	$\oint$	\subseteq	"3212	$\subseteq$
\Omega	"700A	Ω	\succ	"321F	$\succ$
\omega	"0121	ω	\succeq	"3217	$\succeq$
\ominus	"2209	$\ominus$	\sum	"1350	$\sum$
\oplus	"2208	$\oplus$	\supset	"321B	$\supset$
\oslash	"220B	$\oslash$	\supseteq	"3213	$\supseteq$
\otimes	"220A	$\otimes$	\swarrow	"322E	$\swarrow$
\parallel	"326B	$\parallel$	\tau	"011C	τ
\partial	"0140	∂	\Theta	"7002	Θ
\perp	"323F	$\perp$	\theta	"0112	θ
\Phi	"7008	Φ	\times	"2202	$\times$
\phi	"011E	ϕ	\top	"023E	$\top$

\triangle	"0234	$\triangle$	\varsigma	"0126	ς
\triangleleft	"212F	$\triangleleft$	\vartheta	"0123	ϑ
\triangleright	"212E	$\triangleright$	\vdash	"3260	$\vdash$
\uplus	"225D	$\uplus$	\vee	"225F	$\vee$
\Upsilon	"7007	Υ	\wedge	"225E	$\wedge$
\upsilon	"011D	υ	\wp	"017D	$\wp$
\varepsilon	"0122	ε	\wr	"226F	$\wr$
\varphi	"0127	φ	\Xi	"7004	Ξ
\varpi	"0124	ϖ	\xi	"0118	ξ
\varrho	"0125	ϱ	\zeta	"0110	ζ

Některé řídicí sekvence mají své alternativy definované pomocí \let nebo \def:

```
\let\ge=\geq                                  ≥
\let\gets=\leftarrow                          ←
\def\int{\intop\nolimits}                     ∫
\let\land=\wedge                              ∧
\let\le=\leq                                  ≤
\let\lnot=\neg                                ¬
\let\lor=\vee                                 ∨
\def\oint{\ointop\nolimits}                   ∮
\let\owns=\ni                                 ∋
\def\surd{{\mathchar"1270}}                   √
\let\to=\rightarrow                           →
```

Další symboly jsou v plainu sestaveny z dílčích segmentů:

```
\def\neq{\not=}     \let\ne=\neq              ≠
\def\cong{\mathrel{\mathpalette\@vereq\sim}}  ≅
\def\hbar{{\mathchar'26\mkern-9muh}}          ℏ
\def\hookleftarrow{\leftarrow\joinrel\rhook}  ↩
\def\hookrightarrow{\lhook\joinrel\rightarrow}  ↪
\def\notin{\mathrel{\mathpalette\c@ncel\in}}  ∉
\def\rightleftharpoons{%                      ⇌
  \mathrel{\mathpalette\rlh@{}}}
\def\doteq{\buildrel\textstyle.\over=}        ≐
\def\angle{{\vbox{\ialign{$\m@th\scriptstyle##$\crcr
      \not\mathrel{\mkern14mu}\crcr           ∠
      \noalign{\nointerlineskip}\mkern2.5mu
      \leaders\hrule height.34pt\hfill\mkern2.5mu\crcr}}}}
\def\bowtie{\mathrel\triangleright
      \joinrel\mathrel\triangleleft}          ⋈
\def\models{\mathrel|\joinrel=}               ⊨
\def\Longrightarrow{\Relbar\joinrel\Rightarrow}  ⟹
\def\longrightarrow{\relbar\joinrel\rightarrow}  ⟶
```

```
\def\Longleftarrow{\Leftarrow\joinrel\Relbar}      ⟸
\def\longleftarrow{\leftarrow\joinrel\relbar}      ⟵
\def\mapsto{\mapstochar\rightarrow}                ↦
\def\longmapsto{\mapstochar\longrightarrow}        ⟼
\def\longleftrightarrow{\leftarrow                 ⟷
       \joinrel\rightarrow}
\def\Longleftrightarrow{\Leftarrow                 ⟺
       \joinrel\Rightarrow}
\def\iff{\;\Longleftrightarrow\;}

% Pomocná makra:
\def\@vereq#1#2{\lower.5pt\vbox{%
  \lineskiplimit=\maxdimen\lineskip=-.5pt
  \ialign{$\m@th#1\hfil##\hfil$\crcr#2\crcr=\crcr}}}
\def\c@ncel#1#2{\m@th
  \ooalign{$\hfil#1\mkern1mu/\hfil$\crcr$#1#2$}}
\def\rlh@#1{\vcenter{\m@th\hbox{\ooalign{\raise2pt\hbox{%
        $#1\rightharpoonup$}\crcr$#1\leftharpoondown$}}}}
\def\mathpalette#1#2{\mathchoice{#1\displaystyle{#2}}%
  {#1\textstyle{#2}}{#1\scriptstyle{#2}}%
  {#1\scriptscriptstyle{#2}}}
\def\buildrel#1\over#2{\mathrel{\mathop{\kern0pt#2}\limits^{#1}}}
\def\joinrel{\mathrel{\mkern-3mu}}
\def\relbar{\mathrel{\smash-}}
\def\Relbar{\mathrel=}
\def\m@th{\mathsurround=0pt }
```

Mezi důležité symboly při sestavování složených značek patří „škrtátko“ `\not`. Znak má nulovou šířku a vpravo od tohoto pomyslného boxu v polovině šířky většiny znaků pro binární relace je kresba šikmého lomítka „/“. Sekvence `\not` má třídu Rel, takže následné Rel bude sázeno bez mezery, ovšem vlevo a vpravo od této dvojice Rel může být větší mezera, jako kolem jednoduchého atomu typu Rel. Proto se dá psát například `\not=`, `\not\equiv`, `\not\Rightarrow` a dostáváme: $\not=$, $\not\equiv$, $\not\Rightarrow$. □

Nakonec shrneme víceznakové symboly, používané v matematice. Většinou jde o názvy funkcí či operátorů, které se sázejí v antikvě.

```
\def\arccos{\mathop{\rm arccos}\nolimits}          arccos
\def\arcsin{\mathop{\rm arcsin}\nolimits}          arcsin
\def\arctan{\mathop{\rm arctan}\nolimits}          arctan
\def\arg{\mathop{\rm arg}\nolimits}                arg
\def\cosh{\mathop{\rm cosh}\nolimits}              cosh
\def\cos{\mathop{\rm cos}\nolimits}                cos
\def\coth{\mathop{\rm coth}\nolimits}              coth
\def\cot{\mathop{\rm cot}\nolimits}                cot
```

```
\def\csc{\mathop{\rm csc}\nolimits}
\def\deg{\mathop{\rm deg}\nolimits}
\def\det{\mathop{\rm det}}
\def\dim{\mathop{\rm dim}\nolimits}
\def\exp{\mathop{\rm exp}\nolimits}
\def\gcd{\mathop{\rm gcd}}
\def\hom{\mathop{\rm hom}\nolimits}
\def\inf{\mathop{\rm inf}}
\def\ker{\mathop{\rm ker}\nolimits}
\def\lg{\mathop{\rm lg}\nolimits}
\def\liminf{\mathop{\rm lim\,inf}}
\def\limsup{\mathop{\rm lim\,sup}}
\def\lim{\mathop{\rm lim}}
\def\ln{\mathop{\rm ln}\nolimits}
\def\log{\mathop{\rm log}\nolimits}
\def\max{\mathop{\rm max}}
\def\min{\mathop{\rm min}}
\def\bmod{\nonscript\mskip-\medmuskip\mkern5mu
    \mathbin{\rm mod}\penalty900
    \mkern5mu\nonscript\mskip-\medmuskip}
\def\pmod#1{\penalty0
    \mkern18mu({\rm mod}\,\,#1)}
\def\Pr{\mathop{\rm Pr}}
\def\sec{\mathop{\rm sec}\nolimits}
\def\sinh{\mathop{\rm sinh}\nolimits}
\def\sin{\mathop{\rm sin}\nolimits}
\def\sup{\mathop{\rm sup}}
\def\tanh{\mathop{\rm tanh}\nolimits}
\def\tan{\mathop{\rm tan}\nolimits}
\def\ldots{\mathinner{\ldotp\ldotp\ldotp}}
\def\cdots{\mathinner{\cdotp\cdotp\cdotp}}
\def\vdots{\vbox{\baselineskip=4pt
    \lineskiplimit=0pt \kern6pt
    \hbox{.}\hbox{.}\hbox{.}}}
\def\ddots{\mathinner{\mkern1mu
    \raise7pt\vbox{\kern7pt\hbox{.}}\mkern2mu
    \raise4pt\hbox{.}\mkern2mu
    \raise1pt\hbox{.}\mkern1mu}}
```

csc
deg
det
dim
exp
gcd
hom
inf
ker
lg
lim inf
lim sup
lim
ln
log
max
min

mod

$x = y \pmod 2$

Pr
sec
sinh
sin
sup
tanh
tan
…
⋯
⋮
⋱

5.5. Tipy, triky a zvyky v matematické sazbě

V úvodu této sekce připomeneme některá užitečná makra plainu, která se používají v matematické sazbě a která jsme neuvedli v předchozí sekci mezi výčtem symbolů.

- `\,`, `\>`, `\;`, `\!`, `\quad` a `\qquad` vytvoří různě velké mezery.
- `\mit` a `\cal` přepínají do fontů rodiny 1 a 2 (0123456789 a $\mathcal{ABCDEF}\ldots$).
- `\big`, `\Big`, `\bigg`, `\Bigg` vytvářejí velké závorky typu Ord.
- Totéž, ale typ Open: `\bigl`,..., Close: `\bigr`, ... a Rel: `\bigm`, ...
- `\strut`, `\mathstrut` pro vytvoření neviditelné podpěry.
- `\phantom`, `\vphantom`, `\hphantom`, vytvářejí neviditelné výrazy.
- `\smash` vysází výraz v boxu s nulovou výškou.
- `\matrix`, `\pmatrix`, `\bordermatrix` pro matice a `\cases` pro sazbu alternativ.

Pozastavíme se u některých maker podrobněji a ukážeme si jejich využití. Nebudeme ale uvádět definice maker, protože jsou zahrnuty do části B. □

Menší mezera (`\,`), střední mezera (`\>`) a větší mezera (`\;`) jsou makra, která dávají mezeru podle registrů `\thinmuskip`, `\medmuskip` a `\thickmuskip`. Jak už víme, tyto registry určují automatické mezerování mezi různými typy atomů. Konečně `\!` dává zápornou mezeru stejné velikosti jako malá mezera `\,`. Nejčastěji se používá malá kladná a záporná mezera:

```
\int f(x)\,{\rm d}x  % \, před dx                    ∫ f(x) dx
74\,{\rm mm}         % \, před jednotkou              74 mm
n! \, (n+1)          % někdy se hodí před závorkou    n! (n+1)
\sqrt 2 \, z         % \, za odmocninou               √2 z
a^2 \!/\! \sin(x+1)  % \! výjimečně kolem znaku /    a^2/sin(x+1)
\Gamma_{\! 2}        % \! někdy dle tvaru písmene     Γ_2
% Doporučujeme definovat:
\def\mm{\,{\rm mm}}  % pro jednotku; můžu psát: 74\mm
\def\d{{\rm d}}      % pro dx, můžu psát: {\d x\over \d y}
                     % nebo: \int f(x)\,\d x
```

Dále už každý musí zvážit vložení mezery podle svého citu. Nesmíme ale zapomínat na to, že TEX klade mezi některé atomy mezery automaticky.

Makra `\quad` a `\qquad` dávají mezery velikosti 1 em a 2 em. Tato makra fungují i v horizontálním módu, zatímco ostatní zmíněná makra pracují s jednotkou „mu“ a jsou tedy použitelná jen v matematickém režimu. Velmi často se ovšem hodí používat makro `\,` ve významu malé mezery i v horizontálním módu. Třeba v tomto odstavci jsem psal „`... velikosti 1\,em a 2\,em`“ a přechodem do matematického módu jsem se neobtěžoval. Je tedy užitečné definovat:

```
\def\,{\ifmmode\mskip\thinmuskip\else\leavevmode\thinspace\fi}
```

Pak můžeme sekvenci `\,` používat i v horizontálním módu. □

Makra `\big`, `\Big`, `\bigg`, `\Bigg` mají jeden parametr, který musí být podle pravidla ⟨*delimiter*⟩. Makra vytvoří odpovídající velikost delimiteru použitím dvojice primitivů `\left` a `\right`. Vše se odehrává uvnitř samostatného boxu, takže není nutné k makrům použít párová makra na „druhé straně vzorce“.

Ilustrujeme jednotlivé velikosti závorek na delimiteru `\Vert`:

- `\big\Vert` je první větší velikost k „‖“: „$\big\Vert$“.
- `\Big\Vert` dává druhou větší velikost: „$\Big\Vert$“.
- `\bigg\Vert` je velikost, která schová zlomek: „$\bigg\Vert$“.
- `\Bigg\Vert` je ještě větší, než `\bigg\Vert`: „$\Bigg\Vert$“.

Uvedená makra vkládají výsledek do atomu typu Ord. Další alternativy `\bigl`, `\Bigl`, `\biggl` a `\Biggl` vkládají analogický výsledek do atomu typu Open (otevírací závorka). Podobně alternativy `\bigr`, `\Bigr`, `\biggr` a `\Biggr` vytvářejí atom typu Close (zavírací závorka) a `\bigm`, `\Bigm`, `\biggm`, `\Biggm` dávají atom typu Rel (binární relace). Podle toho TEX rozhodne o mezerování kolem závorek.

Bývá slušností v případě vzorce s více vnořenými závorkami sázet každou další úroveň o stupeň větší velikostí, aby se v tom čtenář vyznal. Například:

```
$$ \bigl(f(x)-g(x)\bigr)
\Bigl((x-y)+\bigl(f(x)-f(y)\bigr)\bigl(g(x)-g(y)\bigr)\Bigr) $$
```

dává:

$$\bigl(f(x)-g(x)\bigr)\Bigl((x-y)+\bigl(f(x)-f(y)\bigr)\bigl(g(x)-g(y)\bigr)\Bigr)$$

Makra `\big...` využijeme i tehdy, kdy potřebujeme rozdělit vzorec na dva řádky, přičemž na jednom řádku velká závorka začíná a na druhém končí. V takovém případě nelze použít dvojici `\left`, `\right`, protože měřený vzorec TEX vkládá do boxu a ten už na jednotlivé řádky nerozdělíme. □

Makro `\strut` vloží do horizontálního nebo matematického seznamu neviditelnou podpěru (`\vrule`) výšky 8,5 pt a hloubky 3,5 pt. S touto podpěrou jsme se už setkali a používáme ji pro udržování stejných odstupů mezi řádky tam, kde nelze využít algoritmů z `\baselineskip`. Knuth doporučuje použít tuto podpěru například ve složených zlomcích:

```
$$ \let\ds=\displaystyle
   a_0 + {1\over\ds a_1 + {\strut 1\over\ds a_2 +
           {\strut 1\over\ds a_3 }}} $$
```

což dává

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3}}} \qquad a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3}}}$$

Druhý zlomek byl pro ilustraci vysázen bez použití \strut, aby bylo vidět, že jedničky se nepříjemně přibližují zlomkové čáře.

Makro \mathstrut na rozdíl od makra \strut dává trochu menší výsledek, který je navíc závislý na matematickém stylu. Do seznamu vkládá box nulové šířky. Výška s hloubkou tohoto boxu odpovídá výšce a hloubce závorky v aktuálním stylu. Například ve stylu D a T je výška 7,5 pt a hloubka 2,5 pt, tedy z obou stran o jeden bod menší než \strut. V „menších“ stylech bude \mathstrut ještě menší. □

Makro \phantom čte vzoreček jako argument. Vzoreček je vysázen jen pomyslně a na jeho místě bude prázdný box o velikosti vzorečku. Vzoreček samotný vysázen nebude, ale umístění okolní sazby bude stejné, jako by tam ten vzoreček byl. Makro \vphantom dává analogický box, ovšem navíc nuluje jeho šířku (respektuje se jen výška a hloubka měřeného vzorečku). Konečně \hphantom dává prázdný box s nulovou výškou a hloubkou, jehož šířka je rovna šířce měřeného vzorečku.

Makro využijeme všude tam, kde se nám může zdát matematická sazba z důvodu různě vyplněných indexů a exponentů okatě nesouměrná. Vyzkoušíme:

```
1) $ \sqrt{\log x}                  \sqrt{\log_2 x} $,
2) $ \sqrt{\log_{\vphantom{2}} x}   \sqrt{\log_2 x} $
```

a dostaneme: 1) $\sqrt{\log x}\sqrt{\log_2 x}$, 2) $\sqrt{\log x}\sqrt{\log_2 x}$. Vidíme, že v prvním případě jsou odmocniny různě veliké, což nevypadá pěkně. Ve druhém případě jsme ve vzorečku s „log“ vytvořili prázdný index, aby výška výrazu pod odmocninou byla stejná jako v případě s „$\log_2$“. Tím dostáváme stejně vysoké odmocniny, což bylo naším cílem.

Makra typu \phantom lze použít i mimo matematický mód. □

Makro \matrix v sobě spojuje vlastnosti primitivů \halign a \vcenter. Používáme je pro sazbu matic. Makro \pmatrix navíc připojuje kulaté závorky, takže nemusíme psát \left(\matrix{...} \right). Například:

```
$$ \left[ \matrix{a&b&c\cr d&e&f\cr} \right] \cdot
   \pmatrix {x_1\cr x_2\cr x_3} $$
```

dává tento výsledek:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Makro `\bordermatrix` umožní „vystrčit“ řádek nad matici a sloupec vedle matice ohraničené kulatou závorkou. Napíšeme-li třeba:

```
$$ M = \bordermatrix{&s_1&s_2\cr r_1&0&x-y\cr r_2&1&x\cr} $$
```

pak obdržíme:

$$M = \bordermatrix{ & s_1 & s_2 \cr r_1 & 0 & x-y \cr r_2 & 1 & x \cr }$$

Konečně makro `\cases` je zkratkou za `\left\{ \vcenter{\halign{...}}\right.` a používá se pro sazbu alternativ. Například pomocí:

```
$$ f(x)=\cases{1 & pro $x\in\bf Q$\cr x-1 & pro ostatní $x$} $$
```

vysázíme:

$$f(x) = \begin{cases} 1 & \text{pro } x \in \mathbf{Q} \\ x-1 & \text{pro ostatní } x \end{cases}$$

Všimneme si, že první sloupec v tabulce je v matematickém módu, zatímco druhý v horizontálním. □

Nyní se zaměříme na další užitečné triky a zvyky. Chceme v textu odstavce použít znak „=“ mezi dvěma slovy. Třeba srovnáváme *náklady = výdaje*. Kdybychom psali `{\it náklady\/}~=~{\it výdaje}`, budeme mít kolem rovnítka mezislovní mezeru. Může se stát, že se nám víc líbí trochu menší mezera, kterou TEX klade kolem matematické relace. Kdybychom psali:

```
$\it náklady = výdaje$ nebo
{\it náklady\/}$=${\it výdaje\/}
```

tak v obou případech jsme se dopustili chyby. V prvním případě budou písmena „á“ a „ý“ sázena antikvou, zatímco všechna ostatní písmena kurzívou. Je to z toho důvodu, že písmena české abecedy nemají nastavenu `\mathcode` a nemají tedy třídu 7. V druhém případě zase nemáme kolem rovnítka žádnou mezeru, protože

nikde v matematickém seznamu nefigurují dvojice atomů Rel-Ord a Ord-Rel. Nejobvyklejší řešení je vnutit do matematického seznamu prázdné atomy typu Ord:

```
{\it náklady\/${}={}$výdaje\/}   nebo,
{\it náklady\/${{}={}}$výdaje\/} když nechceme zlom za =, nebo
\hbox{\it náklady\/${}={}$výdaje\/} když nedělíme nikde.
```

Často vkládáme do matematické sazby slovíčka, například „pro každé“, „existuje“, „pokud neplatí“. V display módu taková slovíčka výhradně dáváme do \hboxu a vnitřní matematický mód můžeme kvůli jejich sazbě občas opustit.

Při vstupu do \hboxu TEX nastaví sazbu podle aktuálního textového fontu. Jeho velikost v případě použití slovíčka v indexu je nevyhovující, například:

```
$$ \sum_{\hbox{pro všechna }i\in M} x_i $$  to je špatně
$$ \sum_{\hbox{\sevenrm pro všechna }i\in M} x_i $$ to je lepší
```

Abychom nemuseli pořád myslet na to, v kterém stylu je slovíčko použito, napíšeme makro \mathbox, které se dá snadno použít ve všech stylech, přičemž velikost písma se přizpůsobí. Makro se samozřejmě chová jako \hbox. Můžeme tedy psát:

```
$$ \mathbox{Platí: } \sum_{\mathbox{pro všechna }i\in M} ... $$
```

Při návrhu makra využijeme primitivu \mathchoice:

```
\def\mathbox #1{{\mathchoice{\mB\textfont{#1}}{\mB\textfont{#1}}
  {\mB\scriptfont{#1}}{\mB\scriptscriptfont{#1}}}}
\def\mB #1#2{\hbox{\ifnum\fam<0 \fam=0 \fi \the#1\fam #2}}
```

Makro \mathbox tedy podle čtyř různých situací v \mathchoice předá pomocnému makru \mB údaj o použité velikosti fontu a vlastní text. Pomocné makro otevře box, dále zjistí, zda je explicitně nastaven registr \fam. Pokud není, nastaví jej na implicitní hodnotu: sazba antikvou. Pak pomocí primitivu \the následovaném slovem \textfont až \scriptscriptfont přepne aktuální textový font na požadovaný a vysází obsah parametru #2.

Představme si, že vytváříme makro pro sazbu symbolu $^1/_2$ a budeme chtít, aby makro pracovalo v horizontálním i matematickém módu. Vlastní sazba symbolu proběhne v matematickém módu. Sazbu tedy schováme do \hboxu a tam otevřeme „svůj“ matematický mód. Makro může vypadat takto:

```
\def\jednapol{\leavevmode\hbox{\mathsurround=0pt
   $^1\mkern-3mu/\mkern-3mu_2$}}
```

Nyní ovšem nastane problém s užitím makra v indexu, kde výsledná sazba nebude mít správnou velikost. Mohli bychom zase použít `\mathchoice`, ale v tomto případě se bez něj obejdeme a raději budeme testovat, z jakého módu je makro voláno:

```
\def\jednapol{\ifmmode \let\konec=\relax \bgroup\else
  \def\konec{$}\leavevmode\hbox\bgroup\mathsurround=0pt $ \fi
  ^1\mkern-3mu/\mkern-3mu_2 \konec \egroup }
```

Nyní už jen bude potřeba každému uživateli zdůraznit, aby za použitím makra `\jednapol` napsal `{}`, pokud za tímto symbolem chce mít mezeru. Jestliže nás takové poučování uživatelů nebaví, definujme:

```
\def\1/2{\jednapol}
```

a řekněme uživatelům, ať používají sekvenci `\1/2`. Zde jsme definovali makro `\1` s povinným separátorem `/2`. Tím samozřejmě zabráníme možnosti definovat `\1/4`. Proto bude asi lepší použít `\def\1/#1{\jednalomeno{#1}}` a pomocné makro `\jednalomeno` si už čtenář jistě napíše sám. □

V závěrečné části této sekce ukážeme, jak se sestavují kompozitní znaky z elementů, které máme k dispozici například v plainu a které byly shrnuty v sekci 5.4.

Velmi jednoduché a přitom efektní kompozity můžeme vytvořit vložením záporné mezery mezi elementy. Potřebujeme například psát $\langle\langle X, Y \rangle\rangle$. K tomu stačí definovat:

```
\def\llangle{\langle\!\langle} \def\rrangle{\rangle\!\rangle}
```

Tyto dvojité závorky se navenek (pro automatické mezerování) chovají jako jediný atom typu Open nebo Close. Například mezera vlevo od `\llangle` je počítána jako mezera vlevo od Open, protože první element kompozitu je typu Open. Mezera vpravo od `\llangle` je také počítána jako mezera vpravo od Open, protože poslední element kompozitu je Open. Mezi jednotlivými elementy není jiná mezera než `\!`, protože dvojice Open-Open nemá mezi sebou automatickou mezeru. Na podobné myšlence je založena tvorba kompozitů typu Rel, jak vidíme na příkladech na straně 188.

Takto definované závorky budou zmenšovat svoji velikost ve stylech S a SS, protože se v těchto stylech zmenšují jednotlivé elementy kompozitu samostatně a mezera `\!` má velikost rovněž závislou na stylu. Podobně můžeme definovat třeba `\llbrack` a `\rrbrack`.

$$\left\langle\!\!\left\langle \sum_{\langle\!\langle X,Y \rangle\!\rangle} [\![A, B]\!] \right\rangle\!\!\right\rangle$$

Velké závorky v tomto vzorci nemohly být vytvořeny prostým `\left\llangle` a `\right\rrangle`. Pro závorky pružné velikosti je potřeba definovat další makra:

```
424 \def\LLangle{\left\langle \!\!\left\langle}
425 \def\RRangle{\right\rangle\!\!\right\rangle}
```

Toto ale nebude fungovat ve spolupráci s makry „`\big...`“. Pokud skutečně potřebujeme tyto závorky používat i v souvislosti třeba s `\biggl`, bude asi nutné definovat jednotlivé velikosti závorek samostatně. Například definujeme sekvenci `\biggLLangle` a další. □

Chceme-li sestavovat kompozity například pomocí `\halign`, budeme mít trochu více práce. Uvedeme příklad. Chceme vytvořit znaky ⦾ a ⦿ pomocí jednotlivých elementů ◯, ∘ a ●. Konstrukce:

```
426 \ooalign{$\bigcirc$\cr \hfil$\circ$\hfil}
```

sice dává první z požadovaných znaků, ale tento znak se (1) nechová jako atom typu Bin a (2) nebude se zmenšovat ve stylech S a SS. Tušíme, že budeme muset využít vlastností primitivu `\mathchoice`. O tento primitiv se opírá makro plainu `\mathpalette` (viz část B). Budeme tedy definovat:

```
427 \def\bcirc{\mathbin{\mathpalette\makebcirc{\circ}}}
428 \def\bbullet{\mathbin{\mathpalette\makebcirc{\bullet}}}
429 \def\msurr{\mathsurround=0pt}
430 \def\makebcirc#1#2{% #1 je ⟨style primitive⟩ a #2 \circ nebo \bullet
431    \ooalign{$#1\bigcirc\msurr$\cr \hfil$#1#2\msurr$\hfil}}
```

Obrat s `\msurr` je v makrech plainu častý. Tam se používá stejné makro `\m@th`. Jde o to, že v dokumentu může být nastaveno nenulové `\mathsurround`, zatímco my potřebujeme mít pro lokální účely jistotu, že kolem sazby z `$...$` nevznikne žádná mezera. □

5.6. Display mód

Nejprve uvedeme primitivy `\eqno` a `\leqno`, které umožňují vedle centrované rovnice umístit na okraj značku. Bohužel, většina LaTeXových uživatelů tyto primitivy nezná. Není se čemu divit, protože vesměs všechny LaTeXové příručky tyto důležité primitivy ignorují.

Existují tři možné zápisy display módu:

```
$$⟨formule⟩$$  % ⟨formule⟩ bude centrovaná doprostřed
$$⟨formule⟩\eqno⟨značka⟩$$  % ⟨formule⟩ uprostřed, ⟨značka⟩ vpravo
$$⟨formule⟩\leqno⟨značka⟩$$ % ⟨formule⟩ uprostřed, ⟨značka⟩ vlevo
```

Například:

```
$$ x^2 + y^2 = z^2 \eqno [{\rm Py}_1] $$
```

dává:

$$x^2 + y^2 = z^2 \eqno [\mathrm{Py}_1]$$

TEX v případě výskytu \eqno nebo \leqno zpracuje zvlášť ⟨formuli⟩ v display módu a zvlášť ⟨značku⟩ ve vnitřním matematickém módu. Kolem ⟨značky⟩ nejsou ale připojeny mezery z \mathsurround. Výsledkem zpracování jsou dva boxy: jeden s ⟨formulí⟩ a jeden se ⟨značkou⟩. Box s ⟨formulí⟩ je v řádku centrován bez ohledu na to, zda je či není přítomna ⟨značka⟩ a zda je přítomna vpravo nebo vlevo. Výjimkou z tohoto pravidla je pouze situace, kdy je ⟨formule⟩ tak „široká“, že by její obsah mohl kolidovat se ⟨značkou⟩. O tom si ale povíme podrobněji za chvíli. Box se ⟨značkou⟩ je umístěn do stejného řádku, v jakém je ⟨formule⟩ (výjimky uvedeme rovněž později). Při \eqno se pravý okraj boxu se ⟨značkou⟩ kryje s pravým okrajem řádku a při \leqno se kryje levý okraj boxu s levým okrajem řádku.

Řádek s rovnicí a případnou ⟨značkou⟩ má obvykle šířku \hsize a není posunut do strany. Výjimka z tohoto pravidla může vyplynout při použití nenulového \hangindent nebo při \parshape. Při těchto úpravách tvaru odstavce TEX vynechává pro rovnici tři řádky v odstavci. Šířka a případné posunutí samotného řádku s rovnicí jsou brány z údaje \parshape nebo \hangindent o druhém ze tří vynechaných řádků. Ostatní dva (jakoby) řádky nad a pod rovnicí zůstávají fyzicky nevyplněny. Místo nich TEX (obvykle) vkládá vertikální mezeru \abovedisplayskip nad rovnici a \belowdisplayskip pod rovnici. □

Uvedený popis usazení rovnice ve vnějším vertikálním seznamu není zcela přesný. Proto se jej nyní pokusíme trochu formalizovat. TEX totiž při umisťování rovnice pracuje s řadou primitivních registrů, se kterými se při popisu algoritmu blíže seznámíme. Algoritmus usazení rovnice rozdělíme do několika kroků.

1. krok. Přerušení horizontálního seznamu. Při startu display módu (když hlavní procesor obdrží vstupní dvojici $$) TEX musí být v odstavcovém módu (viz sekci 3.4, strana 93). TEX přeruší sestavování stávajícího horizontálního seznamu. Tato činnost se podobá práci povelu \par na konci horizontálního seznamu, tj. TEX připojí do posledního řádku \parfillskip a sestaví řádky odstavce, které zařadí do vnějšího vertikálního seznamu. Podrobněji o sestavení odstavce viz sekci 6.4 a heslo \par v části B.

2. krok. Otevření skupiny a nastavení parametrů. Po otevření základní skupiny display módu TEX nastaví tyto parametry:

- `\displaywidth` — šířka řádku s rovnicí,
- `\displayindent` — údaj o posunutí řádku s rovnicí doprava,
- `\predisplaysize` — údaj o pravém okraji posledního řádku v odstavci.

Šířka `\displaywidth` je obvykle `\hsize`. Při nenulovém `\hangindent` se ale jedná o šířku řádku s číslem $n+2$, kde n značí počet již sestavených řádků v odstavci. Podobně při `\parshape`. Posunutí `\displayindent` je obvykle rovno 0 pt. Ovšem při nenulovém `\hangindent` nebo při `\parshape` může TEX pracovat s nenulovým posunutím rovněž podle řádku s číslem $n+2$.

Konečně `\predisplaysize` je šířka textu v posledním řádku odstavce plus 2 em (aktuálního fontu). Šířku textu TEX zjistí pohledem do posledního řádku a ignorováním všech výplňků typu ⟨*glue*⟩ a `\kern` zprava, až narazí na první box nebo písmeno. Velikost `\predisplaysize` pro odstavec, který právě čtete, je znázorněna čarou pod odstavcem.

——————————— = `\predisplaysize`
——— = 2 em

Je-li poslední řádek odstavce posunut vpravo nebo vlevo z důvodu použití nenulového `\hangindent` nebo při povelu `\parshape`, je hodnota posunu do `\predisplaysize` také započtena.

Ve výjimečných případech může mít `\predisplaysize` hodnotu $-$`\maxdimen` nebo `\maxdimen`. První případ nastane při prázdném řádku nad rovnicí, tj. třeba při `\noindent $$`. K druhému případu dojde, pokud mezislovní mezery v posledním řádku „pruží“, tj. když nemá `\parfillskip` nekonečnou pružnost. TEX se totiž brání prozradit rozměr sazby, pokud v pružných mezerách pracují hodnoty roztažení a stažení. Ačkoli jsou při těchto výpočtech zaokrouhlovací chyby pod hranicí vlnové délky světla, přesto tam nějaké chyby jsou a bohužel jsou implementačně závislé. Kdyby TEX vracel výsledky těchto výpočtů do makrojazyka, mohla by existovat implementačně závislá makra. To Knuth nepřipustil.

3. krok. Vlastní sestavení ⟨*formule*⟩**.** TEX nyní expanduje obsah registru `\everydisplay` a dále pokračuje ve čtení dalších povelů pro sestavování matematického seznamu. V tuto dobu můžeme svými makry zjistit hodnoty načtených parametrů podle kroku 2, ba dokonce je upravit k obrazu svému. Jejich hodnoty TEX použije až v dalších krocích. Většinou tyto změny neděláme.

TEX sestavuje matematický seznam ⟨*formule*⟩ a po dosažení koncového znaku (`$$` nebo `\eqno` nebo `\leqno`) konvertuje tento seznam na horizontální. V případě

výskytu `\eqno` nebo `\leqno` TEX navíc otevře vnitřní matematický mód, expanduje `\everymath` a sestaví matematický seznam. Ten konvertuje a definitivní sazbu ⟨*značky*⟩ vloží do boxu.

4. krok. Umístění ⟨*formule*⟩ v řádku. Uvažujme nejprve, že není přítomná ⟨*značka*⟩. Je-li šířka ⟨*formule*⟩ menší než `\displaywidth`, ⟨*formule*⟩ bude v řádku centrována ve své přirozené šířce. Jinak mohou pracovat hodnoty stažení v pružných výplních ve ⟨*formuli*⟩, aby její šířka byla `\displaywidth`. Nepovede-li se to, obdržíme `Overfull \hbox`.

Předpokládejme nyní, že je přítomna ⟨*značka*⟩. Je-li při centrování ⟨*formule*⟩ v řádku šířky `\displaywidth` vzdálenost okraje ⟨*formule*⟩ od ⟨*značky*⟩ větší než šířka ⟨*značky*⟩, bude skutečně ⟨*formule*⟩ v řádku centrována. Je tedy nutné, aby vzdálenost okraje centrované ⟨*formule*⟩ od okraje řádku byla větší než dvojnásobek šířky ⟨*značky*⟩. Jinak se ⟨*formule*⟩ umístí do volného místa M mezi okrajem řádku na jedné straně a ⟨*značkou*⟩ na straně druhé (je jedno, zda je ⟨*značka*⟩ vpravo nebo vlevo). V tomto případě existují dvě možnosti: Začíná-li ⟨*formule*⟩ mezerou typu ⟨*glue*⟩, je taková ⟨*formule*⟩ v místě M umístěna vlevo nebo vpravo, vždy na protější stranu než je ⟨*značka*⟩. Nezačíná-li ⟨*formule*⟩ mezerou typu ⟨*glue*⟩, je v místě M umístěna doprostřed.

Ilustrujme si popsané chování na příkladě. Nechť ⟨*formule*⟩ má šířku pouze slova „⟨*formule*⟩“ a ⟨*značka*⟩ zabírá nejdříve pouze velikost slova „⟨*značka*⟩“ a podruhé čtvrtinu `\hsize`. Pak situace vypadá následovně (střídáme `\eqno` a `\leqno`):

⟨*formule*⟩ ⟨*značka*⟩

⟨*značka*⟩ ⟨*formule*⟩

⟨*formule*⟩ |..... ⟨*značka*⟩ .....|

|..... ⟨*značka*⟩ .....| ⟨*formule*⟩

⟨*glue*⟩⟨*formule*⟩ |..... ⟨*značka*⟩ .....|

|..... ⟨*značka*⟩ .....| ⟨*glue*⟩⟨*formule*⟩

Šířka ⟨*formule*⟩ musí být při výskytu ⟨*značky*⟩ vždy menší nebo rovna hodnotě w:

$$w = \texttt{\textbackslash displaywidth} - \text{šířka } \langle\textit{značky}\rangle - \langle\textit{quad}\rangle \qquad \text{(R)}$$

kde ⟨*quad*⟩ je čteno z fontu `\textfont2`. Pokud je přirozená šířka ⟨*formule*⟩ větší než w, pracují hodnoty stažení v pružných výplních ⟨*formule*⟩, aby její šířka byla právě rovna w. Pokud se to nepovede, obdržíme `Overfull \hbox`. Do algoritmu pro

umístění ⟨*formule*⟩ a ⟨*značky*⟩ na řádku, jak jsme před chvílí popsali i ilustrovali, vstupuje už případně upravená ⟨*formule*⟩ na šířku w.

5. krok. Mezery nad a pod rovnicí. Načrtněme si pracovně řádek s ⟨*formulí*⟩ a ⟨*značkou*⟩ a posuňme jej doprava od pomyslného okraje odstavce o hodnotu `\displayindent`. Pokud je v řádku ⟨*značka*⟩ vlevo nebo pokud vzdálenost okraje odstavce od levého okraje ⟨*formule*⟩ je menší než `\predisplaysize`, říkáme, že předchozí řádek odstavce *zasahuje do rovnice*.

Jestliže předchozí řádek odstavce zasahuje do rovnice, je nad rovnicí použita mezera z `\abovedisplayskip` a pod rovnicí z `\belowdisplayskip`. Nezasahuje-li předchozí řádek do rovnice, je nad rovnicí použita mezera `\abovedisplayshortskip` a pod rovnicí `\belowdisplayshortskip`.

6. krok. Konečné umístění materiálu. Do vertikálního seznamu, kam TEX naposledy vložil box posledního řádku odstavce v kroku 1, nyní přibude postupně tento materiál:

- Penalta hodnoty `\predisplaypenalty`.
- Mezera nad rovnicí podle kroku 5.
- Řádek s ⟨*formulí*⟩ (podle kroku 4) posunutý vpravo o `\displayindent`.
- Penalta hodnoty `\postdisplaypenalty`.
- Mezera pod rovnicí podle kroku 5.

Jednotlivé boxy ve vertikálním seznamu podléhají ještě mezi řádkovým mezerám, jak bylo řečeno v sekci 3.7.

Z uvedeného pravidla existuje jedna výjimka. Pokud má ⟨*značka*⟩ nulovou šířku, ovšem existuje, pak obsadí samostatný řádek a nesdílí řádek s rovnicí. Přesněji: je-li ⟨*značka*⟩ s nulovou šířkou použita vlevo (například `\leqno\rlap{...}$$`), pak místo mezery nad rovnicí se do seznamu zařadí řádek, mající vlevo ⟨*značku*⟩ a pod ním `\penalty10000`, a pod ním navazuje řádek s ⟨*formulí*⟩. Je-li ⟨*značka*⟩ s nulovou šířkou použita vpravo (například `\eqno\llap{...}$$`), je za řádek s ⟨*formulí*⟩ připojena `\penalty10000`, dále řádek se ⟨*značkou*⟩ vpravo a pod ním je vložena penalta z `\postdisplaypenalty`, která seznam uzavírá. Ilustrujme si oba výjimečné případy:

⟨*značka šířky 0pt*⟩ ⟨*formule*⟩

nebo

⟨*formule*⟩ ⟨*značka šířky 0pt*⟩

7. krok. Uzavření skupiny a navázání horizontálního seznamu. Po zařazení materiálu do vertikálního seznamu TEX uzavře skupinu display módu, která byla otevřena v kroku 2. Dále přičte k `\prevgraf` trojku (jako by byl odstavec obohacen o tři nové řádky) a založí prázdný horizontální seznam stejně jako při použití `\noindent`. Tím TEX přechází zpět do odstavcového módu. Následuje-li bezprostředně `\par`, je formátovaný horizontální seznam poslední části odstavce zcela prázdný. Proto se z něj do vertikálního seznamu nedostane ani prázdný řádek.

Pokud za ukončujícím znakem `$$` následuje mezera, je ignorována. Protože ukončující znak obvykle píšeme na konci řádku, kde token procesor vyrobí mezeru, je uvedená vlastnost užitečná. Nevzniká zde zavlečená mezera ve formě prvního výplňku typu $\langle glue \rangle$ v novém horizontálním seznamu. Ostatní mezery jsou už významné. □

Plain nastavuje registry, které rozhodují o usazení rovnice do vertikálního seznamu, na tyto hodnoty:

```
436 \abovedisplayskip=12pt plus 3pt minus 9pt
437 \abovedisplayshortskip=0pt plus 3pt
438 \belowdisplayskip=12pt plus 3pt minus 9pt
439 \belowdisplayshortskip=7pt plus 3pt minus 4pt
440 \predisplaypenalty=10000
441 % \postdisplaypenalty zůstává rovna 0
```

Vidíme, že pokud poslední řádek zasahuje do rovnice, je nad rovnicí i pod ní přidána mezera velikosti jednoho řádku a je opatřena dodatečnou pružností. Jestliže poslední řádek nezasahuje do rovnice, je nad rovnicí nulová mezera a pod ní mezera 7 pt.

Hodnota 10 000 pro `\predisplaypenalty` zaručí, že žádná rovnice nebude na začátku nové stránky. Výjimkou budou jen takové rovnice, které mají nad sebou prázdný odstavec (například při použití `$$` ve vertikálním módu). □

• **Příklady.** Chceme, aby všechny rovnice v textu byly shodně odsazeny od levého okraje o velikost odstavcové zarážky. Nechceme tedy, aby byly centrovány. Pokud napíšeme

```
442 $$ \hskip\parindent A+B = C \hskip\displaywidth minus 1fill $$
```

pak tato rovnice bude mít požadovanou vlastnost. Přirozená šířka $\langle formule \rangle$ se totiž určitě nevejde do požadovaného prostoru a bude tedy pracovat hodnota stažení `minus 1fill`.

Nyní tuto myšlenku zobecníme. Definujeme makro, které bude v `\everydisplay` pracovat za nás:

```
\everydisplay={\pridejmezery}
\def\pridejmezery #1$${\hskip\parindent\relax#1\koncovamezera$$}
\def\koncovamezera{\hskip \displaywidth minus 1fill}
```

Toto řešení skutečně funguje. Například při zápisu `$$ A+B = C $$` makro expanduje na text řádku 442. Ale makro zatím nefunguje v kombinaci s `\eqno`. Předefinujeme tedy `\eqno` tak, aby vzalo z konce zápisu token `\koncovamezera` a vložilo tento token před skutečné `\eqno`:

```
\let\orieqno=\eqno
\def\eqno #1\koncovamezera{\koncovamezera \orieqno #1}
```

Nyní už naše makro pracuje i s `\eqno`. Chceme, aby fungovalo též s `\leqno`? Pak ještě dodejme:

```
\let\orileqno=\leqno
\def\leqno #1\koncovamezera{\koncovamezera \orileqno
  \hbox to-\fontdimen6\textfont2{$#1$\hss}}
```

Makro `\leqno` nejen musí přemístit token `\koncovamezera`, ale navíc musí pro ⟨*značku*⟩ vytvořit box velikosti $-\langle quad\rangle$. Pracujeme totiž s ⟨*formulí*⟩ plné šířky w, která je do řádku usazena doprava, protože začíná mezerou typu ⟨*glue*⟩. Aby usazení souhlasilo s ostatními rovnicemi bez `\leqno`, musí platit $w =$ `\displaywidth`. Podle vztahu (R) ze strany 200 pak plyne, že musí být šířka ⟨*značky*⟩ rovna $-\langle quad\rangle$.

Nevýhodou uvedeného makra je skutečnost, že při přeplnění řádku s rovnicí se nedozvíme `Overfull \hbox`. Snad nám to zde nebude tolik vadit a jednotlivé rovnice si v textu před definitivním tiskem sami prohlédneme. Také nám usazení rovnic nebude fungovat při použití makra `\eqalignno`, o němž budeme mluvit v této sekci později. Bylo by potřeba ještě předefinovat toto makro. □

V dalším příkladě budeme měřit šířku posledního řádku v odstavci. Použijeme k tomu hodnoty registru `\predisplaysize`. V TEXu totiž neexistuje jiný způsob, jak efektivně tento rozměr měřit, než uvnitř display módu. Definujeme makro `\testlastline`, které pracuje jako `\par`, ovšem navíc uloží šířku posledního řádku do `\lastlinewidth`.

```
\newdimen\lastlinewidth    % šířka posledního řádku v odstavci
\def\testlastline{\ifhmode $$ \advance\predisplaysize by-2em
   \global\lastlinewidth=\predisplaysize
   \predisplaysize=\maxdimen
   \abovedisplayskip=-\baselineskip
   \belowdisplayskip=0pt $$\endgraf \fi}
```

Nejprve zmenšíme `\predisplaysize` o 2 em, abychom tam měli skutečnou šířku měřeného řádku. Tento údaj uložíme do `\lastlinewidth` globálně, abychom po opuštění skupiny display módu o něj nepřišli. Pak nastavíme lokálně další parametry zpracování prázdné rovnice: `\predisplaysize=\maxdimen` zaručí, že bude pracovat `\abovedisplayskip` a ne jeho varianta `\...shortskip`. Nakonec nastavíme `\above...` a `\below...` tak, aby vložení prázdné rovnice nevytvořilo mezi odstavci žádné nežádoucí místo. □

Pokud potřebujeme sázet více rovnic pod sebou (například soustavy rovnic), použijeme makro `\eqalign`, které expanduje na:

```
457 \vcenter{⟨zvětšení odstupů řádků⟩
458     \halign{⟨deklarace dvou sloupců s přechodem do matem. módu⟩\cr
459     ⟨parametr makra⟩\crcr}}
```

Například:

```
460 $$\eqalign{a^2+b^2 &= c^2\cr
461                   c &= \sqrt{a^2+b^2}\cr}$$
```

dává:

$$\begin{aligned} a^2+b^2 &= c^2 \\ c &= \sqrt{a^2+b^2} \end{aligned}$$

Pro tyto účely není vhodné použít dvakrát za sebou samostatné display módy. Jednak bychom měli bez nastavování vertikálních mezer mezi rovnicemi nevhodně velké místo, dále by rovnice mohly být odděleny stránkovým zlomem a nakonec, ale zdaleka ne v neposlední řadě: nepodaří se nám zarovnat pod sebe konkrétní místa rovnic. Například v uvedené ukázce zarovnáváme rovnítka.

Makro `\eqalign` využijeme i při manuálním rozdělování dlouhé rovnice na více řádků. Většinou chceme, aby relace nebo jiný významný symbol „pokračoval“ na dalších řádcích vždy pod sebou.

Vedle soustavy rovnic pomocí `\eqalign` můžeme také připojit značku použitím `\eqno` nebo `\leqno`. Ovšem jedinou značku pro celou soustavu. Nemůžeme tedy připojit ke každé rovnici soustavy speciální značku. Pokud máme tento požadavek, použijeme další vlastnost TEXu, kterou nyní rozebereme na primitivní úrovni. □

• **Primitiv `\halign` v display módu.** Pokud se jako první povel v display módu objeví (po případných přiřazeních) primitiv `\halign`, TEX nebude na hlavní úrovni display módu sestavovat žádný matematický seznam. Místo toho TEX vytvoří tabulku zcela stejným způsobem, jako by se `\halign` objevil ve vertikálním módu. Za tabulkou mohou pokračovat ještě povely pro přiřazení do registrů, ale už není

dovoleno použít povely pro vytváření matematického seznamu. Také není dovolen výskyt \eqno a \leqno. Zápis tohoto typu display módu vypadá takto:

```
$$ ⟨přiřazení⟩ \halign⟨specifikace boxu⟩{⟨tabulka⟩} ⟨přiřazení⟩ $$
```

Jak víme, výsledek sazby tabulky pomocí \halign je vertikální seznam jednotlivých řádků tabulky. Tento seznam se vloží do vnějšího vertikálního seznamu takto:

- Nejprve TEX vloží penaltu podle \predisplaypenalty.
- Dále je vložena mezera \abovedisplayskip.
- Pak jednotlivé řádky tabulky posunuté o případné \displayindent doprava.
- Potom penalta podle \postdisplaypenalty.
- Nakonec mezera \belowdisplayskip.

Na tuto alternativu display módu pohlížíme jako na přerušení odstavcového módu, aby bylo možno realizovat tabulku \halign. Dále pokračuje odstavcový mód. □

Ačkoli uvedená alternativa očividně nemá s matematikou moc společného, používá se zvláště v makrech pro sazbu soustav rovnic, kdy chceme, aby jednotlivé rovnice lícovaly například podle symbolu „=“ pod sebou a některé z nich měly značku vpravo nebo vlevo.

Makro \eqalignno expanduje na

```
⟨nastavení řádkování a dalšího provozu tabulky⟩
\halign to\displaywidth{⟨deklarace tří sloupců⟩\cr
    ⟨parametr makra⟩\crcr}
```

Přitom první dva sloupce jsou v ⟨*deklaraci*⟩ stejné jako u makra \eqalign. Nový třetí sloupec je vyhrazen pro značky jednotlivých rovnic na pravém okraji. Mezi začátkem a prvním sloupcem je \tabskip pružné, mezi prvním a druhým sloupcem je nulové, mezi druhým a třetím je pružné a mezi třetím a koncem tabulky je nulové. Proto budou první dva sloupce usazeny doprostřed sazby, zatímco poslední sloupec se značkami bude zcela na okraji.

Příklad použití makra:

```
$$\eqalignno{a^2+b^2 &= c^2             & \rm[Py_1]\cr
                   c &= \sqrt{a^2+b^2} & \rm[Py_2]\cr}$$
```

Na výstupu dostáváme:

$$a^2 + b^2 = c^2 \qquad [\mathrm{Py}_1]$$
$$c = \sqrt{a^2 + b^2} \qquad [\mathrm{Py}_2]$$

Mezi \eqalign a \eqalignno je několik zásadních rozdílů. Protože \eqalign schovává celou tabulku do \vcenter, nebudou rovnice nikdy odděleny stránkovým zlomem. Na druhé straně, řádky tabulky z \eqalignno vstupují samostatně do vertikálního seznamu a mohou tedy podléhat stránkovému zlomu. Nikdo nás v \eqalignno nenutí používat třetí sloupec se značkami. Můžeme tedy toto makro použít ve stejném smyslu jako \eqalign, navíc ale můžeme dovolit rozdělení na více stran.

Pokud naopak chceme použít značky, ale nechceme rozdělit soustavu do více stránek, můžeme nastavit registr plainu `\interdisplaylinepenalty` na 10 000. Nebo pišme celé \eqalignno do \vboxu takto:

```
$$\vbox{\eqalignno{⟨naše rovnice⟩}}$$
```

V tomto příkladě už nevyužíváme primitivní vlastnost TEXu sázet v display módu místo matematiky tabulku, protože tabulku sázíme v běžném vnitřním vertikálním módu. Display mód je zde použit jen proto, aby byly vloženy vhodné vertikální mezery nad a pod rovnicemi.

Dalším rozdílem mezi \eqalign a \eqalignno je možnost použít \eqalign i uvnitř matematické sazby a dát kolem třeba závorky pružné velikosti. To s \eqalignno dost dobře nejde, protože šířka tabulky vytvořené tímto makrem bude vždy rovna hodnotě \displaywidth.

V TEXbooku je velmi pěkný příklad na využití \noalign během práce makra \eqalignno. To zase nejde dělat s \eqalign. Třeba pomocí:

```
$$\eqalignno{a^2+b^2 &= c^2 \cr
\noalign{\hbox{což po přepočtu dává}}
                     c &= \sqrt{a^2+b^2} \cr}$$
```

vysázíme:

$$a^2 + b^2 = c^2$$

což po přepočtu dává

$$c = \sqrt{a^2 + b^2}$$

Zde je mezi rovnicemi text. Přesto rovnice pod sebou lícují podle rovnítek a jako celek centrují. Ano, i taková kouzla se dají v TEXu dělat.

Analogicky jako \eqalignno pracuje \leqalignno, kde pro změnu první sloupec je odsunut zcela vlevo a je rezervován pro sazbu levých značek k rovnicím. Dále v plainu existuje makro \displaylines, které by se rovněž mohlo hodit při sazbě rovnic. Další makra podobných vlastností si čtenář určitě dokáže udělat sám. □

• **Číslování rovnic.** V závěru sekce uvedeme příklad makra, které bude číslovat rovnice automaticky. Uživatel napíše třeba:

```
$$ x^2 + y^2 = z^2 \eqnum [rov1] $$
```

nebo:

```
$$\eqalignno{a^2+b^2 &= c^2 \cr
                     c &= \sqrt{a^2+b^2} & \eqnum[vypocet] \cr}$$
```

V textu bude uživatel odkazovat na vztah `\eqref[rov1]` nebo `\eqref[vypocet]`. TEX bude rovnice samostatně číslovat od jedničky a v místě `\eqref` bude tato čísla používat.

Řešení, které uvedeme, ukazuje obecné postupy pro realizaci křížových referencí. Je jedno, zda se jedná o čísla rovnic, čísla obrázků, tabulek apod.

Nejprve ukážeme „jednodušší“ variantu makra, kde předpokládáme, že všechny reference jsou *zpětné*. Tím myslíme, že uživatel nebude nikdy odkazovat pomocí `\eqref` na vztah, který uvede teprve později, ale vždy odkazuje na vztahy už dříve napsané. Pak stačí použít třeba takové makro:

```
\newcount\eqnumber  % číslo rovnice
\def\eqnum[#1]{\global\advance\eqnumber by1
   \expandafter\xdef\csname eq:#1\endcsname{\the\eqnumber}
   \ifinner\else \eqno \fi (\the\eqnumber)}
\def\eqref[#1]{\expandafter\ifx\csname eq:#1\endcsname \relax
   \errmessage{Error: Undefined eq. number [#1]}(??)%
   \else(\csname eq:#1\endcsname)\fi}
```

Při sazbě rovnice s `\eqnum` definujeme sekvenci `eq:`⟨*reference*⟩ jako skutečné číslo rovnice. Dále pomocí `\ifinner` poznáme, zda je sekvence `\eqnum` použita namísto `\eqno` (na hlavní úrovni display módu) nebo je použita samostatně (uvnitř `\eqalignno`). Makro `\eqref` pak prostě expanduje na `eq:`⟨*reference*⟩. Navíc testuje, zda je tato sekvence už definovaná. Pokud ne, ohlásíme chybu, že není na co odkazovat.

Jestliže si nepřejeme průběžné číslování v celém dokumentu, můžeme v makrech pro zahájení kapitoly, odstavce, úseku apod. nulovat registr `\eqnumber`. Rovněž můžeme definovat složitější číslování, kdy značky obsahují více čísel oddělených třeba tečkami. Například `(\the\chapnumber.\the\eqnumber)`. Formát značky, včetně okolních závorek, je zcela v našich rukou.

Pokud budeme používat nejen zpětné, ale i dopředné reference, musíme použít pomocný soubor. V prvním průchodu zpracování TEXem zapíšeme do tohoto souboru

vztah mezi referenční značkou a skutečným číslem rovnice. V druhém průchodu na začátku zpracování dokumentu pracovní soubor načteme. Tím si TEX „ujasní“ vztahy mezi referenčními značkami a čísly. Pak jen tyto „znalosti“ použije.

```
\newcount\eqnumber  % číslo rovnice
\newwrite\REF       % pracovní soubor
\def\eqnum[#1]{\global\advance\eqnumber by1
   \immediate\write\REF{\string\eqREF{#1}{\the\eqnumber}}
   \ifinner\else \eqno \fi (\the\eqnumber)}
\def\eqref[#1]{\expandafter\ifx\csname eq:#1\endcsname \relax
   \message{Warning: Undefined eq. number [#1]}(??)%
   \else(\csname eq:#1\endcsname)\fi}
\def\eqREF #1#2{\expandafter\ifx\csname eq:#1\endcsname \relax
      \expandafter\def \csname eq:#1\endcsname {#2}
   \else \errmessage{Error: Double eq. mark [#1]}\fi}
% Načtení souboru z předchozího běhu a otevření nového:
\softinput \jobname.ref
\immediate\openout\REF=\jobname.ref
```

Při sazbě rovnice budeme do souboru `\jobname.ref` ukládat informaci ve tvaru:

```
\eqREF{⟨referenční značka⟩}{⟨skutečné číslo rovnice⟩}
```

a pomocné makro `\eqREF` se při načtení souboru tyto údaje „naučí“, podobně jako to v předchozí ukázce dělalo přímo makro `\eqnum`. Navíc kontrolujeme, zda nebyla nějaká referenční značka použita dvakrát.

Pokud makro `\eqref` narazí na značku, která nemá přiřazeno číslo rovnice, přejdeme to tentokrát jen varováním. Taková věc se stane totiž vždy při prvním zpracování TEXem, kdy ještě soubor `\jobname.ref` není vytvořen.

Všechny zápisy do souboru děláme `\immediate`, tj. okamžitě. Nečekáme až na sestavení strany. V tomto případě to stačí, protože do souboru nezapisujeme informaci o čísle strany.

Makro `\softinput` čte soubor, jen když existuje. Makro uvádíme na jiném místě této knihy, na straně 288. □

6. Zalamování

6.1. Místa zlomu všeobecně

V TEXu rozlišujeme dva zalamovací algoritmy. Algoritmus, který rozděluje horizontální seznam do řádků (tzv. *řádkový zlom*), a dále algoritmus, který rozděluje hlavní vertikální seznam do stránek (tzv. *stránkový zlom*). Druhý jmenovaný má svou zjednodušenou obdobu i pro vnitřní vertikální seznam rozdělený prostřednictvím `\vsplit`.

V horizontálním i vertikálním seznamu existují místa, kde algoritmy pro řádkový nebo stránkový zlom nemohou seznam nikdy zlomit. Například seznam nelze zlomit mezi boxy, které těsně navazují, nebo mezi dvojicemi písmen, kde nebylo nalezeno místo pro dělení slova. Dále existují místa, kde tyto algoritmy mohou seznam zlomit s jistou penalizací, a konečně běžná místa zlomu, například mezery mezi slovy. V této sekci se zaměříme na výčet všech míst, ve kterých je možné v seznamu provést zlom.

Při provedení zlomu dochází v místě zlomu k případné úpravě materiálu. Například zlom v mezislovní mezeře způsobí, že tato mezera se neobjeví ani v právě ukončeném řádku, ani v řádku následujícím. Nebo při dělení slova je slovo rozděleno tak, že se přidá na konec první části slova do horizontálního seznamu spojovník (-). To je založeno na primitivu `\discretionary`, jak uvidíme později. □

Zlomem v konkrétním elementu seznamu rozumíme ukončení řádku (nebo strany) předchozím elementem a zahájení nového řádku (strany) některým z následujících elementů v seznamu. Pokud třeba řekneme, že TEX zlomil řádek v mezeře typu ⟨*glue*⟩, pak poslední element na řádku je ten, který předcházel zmíněné mezeře, a první element na dalším řádku je takový element, který následuje za zmíněnou mezerou po přeskočení všech bezprostředně následujících mezer a dalších tzv. odstranitelných elementů. Vidíme tedy, že při vlastním zlomu v mezeře tato mezera mizí a v jednotlivých řádcích se už nevyskytuje. Navíc zanikají i případné další mezery, které těsně následují. Uvažujme například:

```
1 Konec řádku.␣\hskip3em Další text
```

V případě, že se řádkový zlom provede v mezeře těsně za tečkou (viz ␣), mizí tato mezera a společně s ní mizí i mezera `\hskip3em`. Slovo „Další“ bude tedy bez mezery zahajovat nový řádek. Z následujícího výkladu algoritmu pro vyhledávání místa zlomu vyplyne, že TEX navíc nemůže v naší ukázce nikdy zlomit v mezeře `\hskip3em`, takže nikdy nezůstane žádná mezera na konci předchozího řádku. □

Jistý typ elementů označíme jako *odstranitelný* (angl. discardable). Jedná se o takové elementy, které mizí, pokud se v nich provede zlom. Mezi odstranitelné elementy patří:

- Výplněk typu ⟨*glue*⟩.
- Kern, tj. pevný výplněk.
- Penalta, tj. trest za zlom v daném místě.

Zde je výčet všech míst, ve kterých TEX může provést zlom:

- Ve výplňku typu ⟨*glue*⟩, pokud nepředchází odstranitelný element.
- V kernu, pokud za ním bezprostředně následuje výplněk typu ⟨*glue*⟩.
- V penaltě bez závislosti na tom, co předchází a co následuje.
- V místě dělení slov (`\discretionary`).

Odstranitelný element, ve kterém byl proveden zlom, mizí společně se všemi bezprostředně následujícími odstranitelnými elementy. Na dalším řádku (stránce) je tedy seznam zahájen prvním neodstranitelným elementem, který za řadou odstranitelných elementů následuje.

Zlom v prvním a druhém případě je zcela potlačen uvnitř matematického seznamu. Znamená to, že v matematickém seznamu je možné zlomit jen v penaltě, která je dána buď explicitně (pomocí `\penalty`), nebo implicitně (viz `\relpenalty` a `\binoppenalty`).

Při vyhledávání míst zlomu v horizontálním a vertikálním seznamu není v algoritmech pro sestavování odstavců nebo stránek žádný rozdíl. Výjimkou je poslední případ (`\discretionary`), který má smysl jen v horizontálním seznamu.

Ve všech ostatních místech v seznamu, která nesplňují výše uvedené podmínky, zlom není povolen. □

Pod pojmem *penalta* si můžeme představit bezrozměrnou značku v seznamu nesoucí číselnou hodnotu. Penalta se může vložit do seznamu explicitně pomocí primitivu `\penalty` následovaného číslem. Většinou se tak děje prostřednictvím nějakého makra — uživatele tímto pojmem nezatěžujeme. Například `\penalty50` vloží do seznamu penaltu o hodnotě 50. V případě, že by se provedl zlom v místě penalty, pak do algoritmů pro vyhodnocení „nejvhodnějšího zlomu“ (viz sekce 6.4 a 6.6) vstupuje toto číslo jako jistý trest za takto provedený zlom a může ovlivnit výpočet. Penalta je celé číslo, obvykle v intervalu $\langle -10\,000, 10\,000 \rangle$. `\penalty0` znamená „žádný trest“, tj. zlom je stejně vhodný, jako by byl proveden v mezeře. Rostoucí kladné hodnoty označují rostoucí „nevhodnost“ takového zlomu. `\penalty10000` a více znamená absolutně nevhodný zlom, který se neuskuteční v žádném případě. Naopak záporné hodnoty vyjadřují určité doporučení k provedení zlomu. Konečně výskyt `\penalty-10000` a méně si vynutí zlom za všech okolností.

Hodnoty penalt jsou kvantifikovány ve stejných jednotkách jako hodnota badness boxu. Zlom v penaltě o hodnotě $n > 0$ při nulové badness je tedy algoritmem zlomu interpretován podobně, jako zlom v mezeře nebo v nulové penaltě, při které badness boxu nabývá hodnoty n. Tato vlastnost vyplyne ze vzorců, které používají algoritmy pro výpočet demerits řádkového zlomu a ceny stránkového zlomu (strany 228, 240).

Hodnota `\penalty10000` (v plainu zkratka `\nobreak`) se může jevit na první pohled jako zbytečná, protože tam není vůbec povolen zlom. Proč ji tedy psát? Chceme-li například zabránit nebo omezit zlom v mezeře typu ⟨*glue*⟩, pak nám stačí uvést těsně před takovou mezeru penaltu a máme zaručeno, že v mezeře se neprovede zlom. Před mezerou totiž předchází odstranitelný element (sice penalta), a proto je v mezeře zlom zakázán. Všichni známe zkratku pro „nezlomitelnou“ mezeru v podobě vlnky, která je v plainu definována takto:

```
\def\break{\penalty-10000 } \def\nobreak{\penalty10000 }
\catcode`\~=\active \def~{\nobreak\ }
```

V mezeře `\nobreak\␣` se zlom nikdy neprovede, protože předchází odstranitelný element — penalta. Ta má hodnotu 10 000, takže v ní taky není dovolen zlom. □

Pokud se sejde více penalt těsně za sebou, algoritmus zlomu si najde tu s nejmenší hodnotou a ostatní jsou ignorovány. Například `\nobreak\break` je totéž jako `\break`. Můžeme tedy říci, že více penalt těsně za sebou splyne v jednu penaltu s hodnotou, která je rovna minimu hodnot uvedených penalt.

Penalta s hodnotou $-10\,000$ a méně je výjimečná v tom smyslu, že není v horizontálním seznamu považována za odstranitelný element. Ve vertikálním seznamu ovšem tato penalta odstranitelným elementem je. Takže například `\break\break` způsobí ve vertikálním seznamu jen jeden přechod na novou stránku, ale v horizontálním seznamu se provedou dva řádkové zlomy těsně za sebou. Obdržíme prázdný řádek společně s hlášením `Underfull`. Box s prázdným řádkem totiž obsahuje jen `\leftskip` a `\rightskip`, které většinou mají hodnoty roztažení nulové.

Plno LaTeXových uživatelů pracuje neopatrně s příkazem `\\`. Zjistili, že tento příkaz způsobí uvnitř odstavce přechod na další řádek a používají jej i na konci odstavce, aby dosáhli prázdného řádku mezi odstavci. Na terminálu pak mají mnoho hlášení `Underfull \hbox badness 10000` a je jim to úplně jedno. Jedno je jen kolečko u trakaře, a proto si podrobně vysvětlíme, co se stalo.

Pokud je sekvence „`\\`“ z LaTeXu použita v rámci zpracování běžného odstavce, je definována jako `\hfil\break`, tj. vlož pružnou mezeru na vyplnění řádku až do jeho konce a proveď řádkový zlom. To uvnitř odstavce funguje, ale na konci odstavce nastanou potíže. Algoritmus řádkového zlomu totiž vloží na konec odstavce posloupnost `\nobreak\hfil\break` pro vytvoření mezery východového řádku a pro závěrečný řádkový zlom v odstavci. Při zápisu sekvence „`\\`“ na konec odstavce

máme tedy za sebou nejprve materiál z této sekvence a potom materiál, který vloží algoritmus řádkového zlomu:

```
4 ... konec odstavce.\hfil\break\nobreak\hfil\break
```

Zlom se nejprve provede v prvním `\break` a zmizí všechny následné odstranitelné elementy, tj. `\nobreak\hfil`. Poslední `\break` není v horizontálním módu odstranitelný a provede poslední řádkový zlom. Efekt je tedy stejný, jako by se setkaly `\break\break` těsně za sebou. Na terminálu máme `Underfull`. Pokud chce LaTeXový uživatel vložit pod odstavcem mezeru velikosti prázdného řádku, naučte ho prosím používat `\bigskip`, nebo (jako v této knize) `\parskip=\baselineskip`. Pokuste se mu velmi nahlas zdůraznit, že přece nemůže ignorovat hlášení typu `Underfull`. Mně osobně to ale někdy připadá jako boj s větrnými mlýny. □

• **Příklady.** V závěru této sekce uvedeme dvě ukázky převzaté z TeXbooku. Ukázky se zaměřují na poněkud méně běžné vlastnosti, které jsou důsledkem popsaného algoritmu. Představme si, že potřebujeme vytvořit odstavec, který končí textem s podpisem, jako to je ukázáno v tomto odstavci. Přitom chceme, aby text s podpisem byl vložen do prostoru mezery východové řádky, pokud se tam vejde. Pokud se nevejde, bude text s podpisem na dalším řádku, rovněž vpravo. Tuto skutečnost by mělo makro ošetřit automaticky. *Ferdinand Mravenec*

Uživatel připojí ke konci odstavce sekvenci `\podpis`, použitou třeba takto:

```
5 ... makro ošetřit automaticky. \podpis Ferdinand Mravenec
```

Makro `\podpis` má poměrně jednoduchou definici:

```
6 \def\podpis #1\par{\unskip
7   \nobreak\hfill\penalty71\hskip2em\hbox{}\nobreak\hfill
8   \hbox{\it #1\/}\par}
```

Jestliže se text podpisu vejde do východové řádky, pak v řadě výplňků, penalt a dalšího materiálu z řádku 7 nedojde ke zlomu. Vidíme, že minimální mezera před textem podpisu je 2 em (`\hskip2em`) a z obou stran se vyplní pružné mezery `\hfill`. Tyto mezery potlačí mezeru z východového řádku (`\parfillskip=0pt plus1fil`), takže text podpisu je odsunut doprava. Penalta má hodnotu 71, protože musí být $\texttt{\linepenalty}^2 + \texttt{\penalty}^2 > \texttt{\finalhyphendemerits}$.

Pokud se text podpisu včetně mezery 2 em nevejde do prostoru východové řádky, zlom se provede v místě `\penalty71`. První výplněk `\hfill` se natáhne do konce řádku a další řádek je zahájen prvním neodstranitelným elementem, kterým je v tomto případě `\hbox{}`. Vpravo od tohoto boxu je další pružný výplněk `\hfill`, který odsune text podpisu doprava a potlačí mezeru z východového řádku.

Ferdinand Mravenec

V TEXbooku je ještě v tomto příkladě přidáno `\finalhyphendemerits=0`, aby algoritmus řádkového zlomu neměl problémy v souvislosti s rozděleným slovem v předposledním řádku odstavce. Při kladném `\finalhyphendemerits` by se algoritmus řádkového zlomu někdy snažil vytvořit na konci zbytečně o jeden řádek více, aby přesunul rozdělené slovo z předposledního řádku do řádku třetího od konce. O tomto parametru viz sekci 6.4. □

„Visící interpunkcí“ rozumíme způsob sazby, kdy tečky, čárky a uvozovky přečnívají přes zrcadlo sazby doprava nebo doleva. Levé uvozovky mohou přečnívat doleva, čárky a tečky mohou v případě, kdy jsou zcela na pravé straně bloku odstavce, přečnívat doprava. Ilustraci takové sazby vidíte v tomto odstavci. Takové řešení interpunkce není v elektronické typografii příliš obvyklé. Zřejmě tento požadavek plno elektronických sázítek prostě nezvládá. Určitě se takový typ sazby hodí pro „slavnostní účely“, nebo též v případě, kdy je v textu mnoho uvozovek a interpunkce, přičemž chceme zdůraznit blokové zarovnání odstavců. Zkuste porovnat pravou hranu tohoto odstavce třeba s následujícím odstavcem, který začíná slovy „Když je proveden zlom“. Subjektivně se jistě jeví tento odstavec zarovnanější než citovaný následující odstavec, kde je „visící interpunkce“ vypnuta.

Levé české uvozovky jsou v této ukázce realizovány znakem `\char254`, který má šířku `\wlqq`. Kolem tohoto znaku je vložena sekvence výplňků a dalšího materiálu takto:

```
9  \kern\wlqq \hbox{}\kern-\wlqq \char254{}
```

Když je proveden zlom v mezeře před levou uvozovkou, pak zmizí též `\kern\wlqq` a další řádek začíná prázdným boxem. Za tímto boxem je přítomen záporný kern, takže se nám znak s uvozovkami „vystrčí“ ze sazby. Pokud ale není v mezeře před uvozovkou zlom, pak se jednotlivé kerny před a za prázdným boxem kompenzují a žádná záporná mezera se nevyskytne.

Tečka, čárka a další interpunkce, která má být vystrčena vpravo, je vložena prostřednictvím sekvence:

```
10  .\kern-\wdot \kern\wdot{}
```

Pokud se provede zlom těsně za takovou tečkou, pak jedině ve druhém kernu (`\kern\wdot`), protože za tímto kernem následuje výplněk typu ⟨*glue*⟩. Pak ale zůstane v činnosti první `\kern-\wdot`, který vystrčí tečku doprava ze sazby. V prvním kernu ani v mezeře za těmito dvěma kerny nelze zlom provést. Pokud není za tečkou zlom řádku, pak se uvedené dva kerny kompenzují a žádná záporná mezera se neprojeví.

Uvedeme si nyní celé makro, kterým lze visící interpunkci zařídit. Uživatel píše pro uvozovky sekvenci `\uv{text}` a tečky a čárky zapisuje obvyklým způsobem.

```
\newdimen\wlqq   \setbox0=\hbox{\char254} \wlqq=\wd0
\newdimen\wrqq   \setbox0=\hbox{\char255} \wrqq=\wd0
\newdimen\wdot   \setbox0=\hbox{.}        \wdot=\wd0
\newdimen\wcomma \setbox0=\hbox{,}        \wcomma=\wd0
\def\comma{,\kern-\wcomma \kern\wcomma}
\def\period{.\kern-\wdot \kern\wdot}
\def\uv{\bgroup
   \ifvmode \leavevmode
   \else \kern\wlqq \hbox{}\fi
   \kern-\wlqq \char254
   \aftergroup \closequotes \let\next=}
\def\closequotes{\char255\kern-\wrqq\kern\wrqq}
\catcode`\.=13 \catcode`\,=13 \let.=\period \let,=\comma
```

V úvodní části měříme rozměry uvozovek a tečky a čárky. Pak definujeme sekvenci `\comma` a `\period` způsobem, o kterém jsme mluvili. Makro `\uv` rozlišuje mezi případem, kdy jsou uvozovky zcela na začátku odstavce (`\ifvmode`). V takovém případě se vloží před levé uvozovky pouze záporný kern. Jinak se před levé uvozovky vloží elementy, které jsme rozebírali výše. Zavírací uvozovky (`\closequotes`) mohou též vykukovat ze sazby, a proto je použit analogický postup jako při tečce a čárce. Nakonec uvedeme tečku a čárku do aktivního stavu.

S tímto typem interpunkce souvisí také jinak řešený spojovník pro dělení slov, který musí rovněž vyčnívat z bloku sazby. O tom, jak to zařídit, si povíme v následující sekci.

Povšimněme si, že kerning mezi posledním písmenem ve větě a tečkou nebo mezi levou uvozovkou a následujícím písmenem zůstává zachován. Ovšem musíme být opatrní už jen kvůli tomu, že máme aktivní tečku a čárku. Také přechody na jiné druhy písma (s jinými rozměry uvozovek a interpunkce) nebyly v makru řešeny. □

6.2. Zlom v místě `\discretionary`

Primitiv `\discretionary` vkládá do horizontálního nebo matematického seznamu značku, která způsobí odlišné chování textu podle toho, zda je nebo není v této značce proveden zlom. Primitiv má tři parametry:

```
\discretionary{⟨pre-break⟩}{⟨post-break⟩}{⟨no-break⟩}
```

Každý z parametrů může být prázdný nebo obsahuje horizontální materiál. Parametr ⟨*no-break*⟩ se použije, pokud se v místě značky neprovede zlom. Parametry ⟨*pre-break*⟩ a ⟨*post-break*⟩ se použijí, pokud se řádkový zlom provede. V takovém případě se ⟨*pre-break*⟩ připojí na konec řádku před zlom a ⟨*post-break*⟩ zahajuje další řádek. Například:

```
25 Zu\discretionary{k-}{k}{ck}er
```

způsobí, že zlovo Zucker se v případě dělení napíše jako Zuk-/ker. Stejný efekt můžeme zapsat též takto:

```
26 Zu\discretionary{k-}{}{c}ker
```

V němčině asi nebudeme taková speciální slova zapisovat přímo prostřednictvím primitivu \discretionary, ale použijeme nějakou zkratku pomocí makra. Často se též používá preprocesor, který vyhledá tato citlivá slova v textu a nahradí je příslušnými značkami podle makra.

Povel \discretionary pro dělení běžných slov vypadá takto:

```
27 \discretionary{-}{}{}
```

a má svou primitivní zkratku: „\-“. Explicitní místa pro dělení slov můžeme tedy zapsat takto: `ex\-pli\-cit\-ní mí\-sta pro dě\-lení slov`. Obvykle takovou věc nemusíme dělat, protože to za nás udělá algoritmus vyhledávání míst dělení slov podle vzorů (viz následující sekci).

Horizontální materiál v parametrech \discretionary nesmí obsahovat vnořené \discretionary, \penalty, výplňky typu ⟨*glue*⟩, ani přechod do matematického módu pomocí `$...$`. Dalším omezením povelu \discretionary je jeho použití v matematickém módu. I tam jsou parametry tohoto povelu interpretovány v horizontálním, a nikoli matematickém módu. Navíc tam musí být parametr ⟨*no-break*⟩ prázdný. Příklad použití tohoto povelu v matematickém módu je uveden v sekci 5.2, strana 160. □

Uvedeme si jednoduché aplikace primitivu \discretionary:

```
28 \def\tisíc{\discretionary{}{tisíc}{\kern.2em000}}
29 \def\až{\discretionary{\kern.3333em až}{}{--}}
```

Pokud píšeme třeba `125\tisíc`, pak se vytiskne 125 000 v případě, že není proveden zlom v mezeře za číslem 125. Pokud se za tímto číslem řádek zlomí, na dalším řádku máme slovy vypsáno „tisíc“. To odpovídá bývalé oborové normě pro základní pravidla sazby. Podobně, pokud napíšeme `110\až 135`, pak se vytiskne obvykle 110–135, ale pokud by se měl provést zlom v místě pomlčky, dostaneme na prvním řádku „110 až“ a na dalším řádku pokračuje číslovka „135“.

Řešení pro pomlčku mezi čísly prostřednictvím \discretionary má navíc jednu výhodu. Přímé použití `110--135` totiž za běžných okolností umožní řádkovému zlomu ukončit řádek těsně za pomlčkou a dostáváme patvar 110–/135. Použití `\až`

tento problém vylučuje. Problém dělícího místa za pomlčkami a spojovníky budeme diskutovat v této sekci ještě později. □

V předchozím textu jsme se dopustili jedné nepřesnosti. Primitiv „\-“ není přesným ekvivalentem k \discretionary{-}{}{}, ale ve skutečnosti se jedná o \discretionary{⟨*hyphenchar*⟩}{}{}. Znak ⟨*hyphenchar*⟩ je přitom definován v rezervovaném registru každého zavedeného fontu. Přístup do tohoto registru je umožněn prostřednictvím primitivu \hyphenchar. Například po:

```
30 \hyphenchar\font=‘\@
```

bude povel \- ekvivalentem k \discretionary{@}{}{}. Navíc algoritmus, který vyhledává místa dělení slov podle vzorů, vkládá do nalezených míst právě povel \-, takže od této chvíle budou všechna rozdělená slova končit nikoli spojovníkem, ale zavináčem. □

Nastaví-li se \hyphenchar fontu na hodnotu -1, v daném fontu není povoleno dělení slov. Například chceme zajistit, aby nebyla dělena slova v textech psaných písmem \tt. Pak stačí psát: \hyphenchar\tentt=-1. □

Registr \hyphenchar má ještě jednu odlišnou funkci. Vyskytne-li se ve vstupním textu znak ⟨*hyphenchar*⟩ nebo ligatura, která končí na ⟨*hyphenchar*⟩, pak za tímto elementem je automaticky připojeno \discretionary{}{}{}. Nepříjemný důsledek: napíšeme-li třeba slovo končící na „-li“, pak může být rozděleno tak, že na konci řádku máme spojovník a na dalším řádku jen „li“. To není dobré. Bohužel uvedené dvě odlišné funkce registru \hyphenchar se nedají od sebe oddělit, takže vznikají problémy. Uvedeme několik řešení těchto problémů.

Řešení 1. Nastavíme \exhyphenpenalty=10000. Za každý řádkový zlom typu \discretionary dostane algoritmus zlomu penaltu podle registru \hyphenpenalty při neprázdném ⟨*pre-break*⟩ a \exhyphenpenalty při prázdném ⟨*pre-break*⟩. Změna registru \exhyphenpenalty tedy neovlivní běžné dělení slov, protože v těchto případech máme v parametru ⟨*pre-break*⟩ spojovník.

Řešení 2. Nastavíme alternativní \hyphenchar, který ve fontu vypadá úplně stejně, ale není na ASCII pozici znaku „-“. V případě fontů kódovaných podle $\mathcal{CS}$-fontů stačí psát:

```
31 \hyphenchar\font=156
```

Je-li nyní na vstupu „je-li“, pak znak „-“ už není ⟨*hyphenchar*⟩, takže TEX za něj nepřipojuje \discretionary{}{}{}. Přitom v místech dělení slov se použije \discretionary{\char156}{}{}, což dopadne zcela stejně jako při standardním nastavení.

Obě uvedená řešení sice zabrání nesprávnému dělení za spojovníkem, ale nedovolí rozdělit slovo se spojovníkem vůbec. Podle pravidel české sazby je přitom možno v místě spojovníku dělit, ale je nutno opsat spojovník na začátek dalšího řádku, například propan-/-butan. Uvedeme tedy další řešení problému slov se spojovníkem:

Řešení 3. Místo `propan-butan` pišme třeba `propan\=butan`. Přitom máme definováno:

```
\def\={\discretionary{-}{-}{-}}
```

Uvedenou sekvenci používám nejraději, protože graficky označuje jakoby zdvojení běžně používaného primitivu `\-`. Skutečnost, že původně v plainu měla tato sekvence význam pro akcent tvaru n̄ mi soukromě nevadí, protože jsem tento akcent ještě nikdy nepoužil.

Řešení 4. Aby uživatel nemusel na spojovníkový problém myslet, dá se přidělit znaku „-“ aktivní kategorie a definovat jej analogicky, jako `\=` v předchozí ukázce. Abychom nepřišli o tvorbu pomlček z „--“ a „---“, je potřeba v makru udělat trochu práce. Často též potřebujeme napsat záporné číslo (třeba `\vskip-1pt`). Aby to fungovalo, omezíme aktivní činnost znaku „-“ jen na horizontální mód. Několik málo čísel, která píšeme v horizontálním módu, budeme muset při záporné hodnotě opsat sekvencí `\minus`. Autorem tohoto makra pro LaTeX je Jiří Zlatuška. Zde uvedeme jen zjednodušenou variantu jeho makra.

```
\def\splithyphen{\discretionary{-}{-}{-}}
\def\minus{-} % znak - bude aktivní, -1pt píšu jako \minus1pt
\def\ligtwohyphens{--}
\def\ligthreehyphens{---}

\catcode`-=13
\def -{\ifhmode \expandafter \onehyphen  \else \minus\fi}
\def\onehyphen{\futurelet\nextchar \doonehyphen}
\def\doonehyphen{%
   \ifx -\nextchar \expandafter \twohyphen
   \else \splithyphen \fi}
\def\twohyphen #1{\futurelet\nextchar \dotwohyphen}
\def\dotwohyphen{%
   \ifx -\nextchar \ligthreehyphens \expandafter\removetoken
   \else \ligtwohyphens \fi}
\def\removetoken #1{}
```

V první části jsou definována makra pro konstrukce, které po změně kategorie znaku minus nebudou dosažitelné. Pak přepínáme kategorii minus na aktivní a definujeme tento znak jako neaktivní minus mimo horizontální mód a jako `\onehyphen` v horizontálním módu. Použití `\expandafter\onehyphen\else` způsobí, že se nejprve

ukončí konstrukce `\if` a teprve pak vystartuje makro `\onehyphen`, které si vezme jako parametr nikoli token `\else`, ale token, který je za aktivním znakem minus skutečně napsaný.

Makro `\onehyphen` si pomocí `\futurelet` přečte následující token. Je-li shodný se znakem minus, provede se `\twohyphen`, jinak se vloží `\discretionary`. Makro `\twohyphen` smaže následující token, o němž už víme, že je znakem minus (viz parametr `#1`, který není nikdy použit). Potom načte prostřednictvím `\futurelet` další token a zkoumá (v rámci makra `\dotwohyphen`), zda tento token není znak minus. Pokud ano, vymaže ho (`\removetoken`) a vysází dlouhou pomlčku (`\ligthreehyphens`). Jinak znak ponechá a vysází kratší pomlčku (`\ligtwohyphens`).

Nahlédnete-li do skutečného Zlatuškova makra, bude se vám jevit podstatně složitější. V některých ohledech je tato složitost možná zbytečná. Například je tam ošetřen přechod z `\dotwohyphen` na něco jako `\threehyphen`. Nebo tam najdete sedm `\expandafter` v místech, kde větvení podmínek expanduje na neaktivní znak minus. Makro také ošetřuje vnitřní a odstavcový horizontální mód, což vede k další úrovni vnořených podmínek, ze kterých se pak vyskakuje nikoli jedním `\expandafter` (jako v našem případě), ale až sedmi `\expandafter`. Test na to, jak zrovna v LaTeXu pracuje sekvence `\protect`, je zase vesměs LaTeXovská záležitost. Na druhé straně makro zjišťuje podle `\righthyphenmin`, zda se povolí rozdělit „-li“ nebo ne. To jsme si pro jednoduchost odpustili. Také `\splithyphen` je v makru definováno sofistikovaněji v závislosti na hodnotě `\hyphenchar` takto:

```
49 \def\splithyphen{\ifnum\hyphenchar\font>-1
50    \discretionary{\char\hyphenchar\font}{-}{-}\else
51    \discretionary{-}{-}{-}\fi}
```

Toto řešení umožňuje používat alternativní `\hyphenchar`.

Řešení 5. Do kódu pro vygenerování formátu napíšeme před načtení `\patterns` řádek:

```
52 \lccode`-=`-
```

a do `\patterns` přidáme položku: „`7-8`“. Pak znovu vygenerujeme formát. Nakonec nastavíme stejně jako v řešení 2 alternativní ⟨*hyphenchar*⟩, aby v zápise `propan-butan` nebyl vložen za znakem „`-`“ `\discretionary{}{}{}`. Protože jsme na řádku 52 řekli, že znak minus bude ve smyslu následující sekce 6.3 písmenem a do vzorů dělení jsme přidali možnost dělit před tímto písmenem, bude skutečně naše slovo rozděleno jako propan-/-butan. Všimneme si, že není potřeba přidělit znaku minus kategorii 11, takže tento znak je možno použít i v číselných konstantách obvyklým způsobem. Nevýhoda tohoto řešení spočívá v tom, že je dovoleno dělení i v jiných místech slova se spojovníkem, tj. pro-/pan-butan nebo propan-bu-/tan.

Ve všech ostatních uvedených řešeních bylo v místě spojovníku použito explicitně `\discretionary`, což zcela potlačuje v takovém slově dělení na jiném místě. □

Nyní ukážeme, co je potřeba učinit, abychom při sazbě s visící interpunkcí měli ze sazby vystrčeno i rozdělovací znaménko. Použijeme k tomu fonty obsahující alternativní spojovník. Například při kódování podle $\mathcal{CS}$-fontů je alternativní spojovník na pozici 156.

Nejprve konvertujeme metriky použitých fontů do čitelné podoby pomocí `tftopl` a vyhledáme tam záznam o alternativním spojovníku. Ten je v $\mathcal{CS}$-fontech na pozici 156, tj. oktalově 234:

```
(CHARACTER O 234
   (CHARWD R 0.333334)
   (CHARHT R 0.430555)
   )
```

Zaměníme hodnotu u `CHARWD` za `0.0` a konvertujeme zpět do formátu `tfm` pomocí `pltotf`. Pak v TEXu stačí napsat:

```
53 \hyphenchar\font=156
```

nebo v csplainu použijeme makro `\extrahyphens`, které nastaví všem běžně používaným fontům alternativní ⟨*hyphenchar*⟩.

Samozřejmě nesmíme zapomenout na to, že změna v `tfm` se nepromítne do fontů, které jsou trvale zavedeny ve formátu. V takovém případě je potřeba přegenerovat formát. Nepomůže pouze následné zavedení pomocí `\font`, protože TEX se neobtěžuje načítat fonty, které už má jednou přečteny. □

6.3. Vyhledání míst pro dělení slov

Abychom přesně vymezili pojem *slovo* v TEXu a mohli mluvit o tom, ve kterých slovech se místa pro dělení hledají a jak, zavedeme pro tuto sekci pojem *písmeno* a pojem *další znak*. *Písmeno* je znak, který má nenulové `\lccode`. Je tedy možné provádět konverze z velkých písmen na malá, což je při hledání míst pro dělení slov nutné. *Další znak* je znak s nulovým `\lccode`. Například tečka nebo čárka je (obvykle) další znak.

Slovo je maximální souvislá skupina písmen. TEX hledá v horizontálním seznamu ve slově místa pro dělení, pokud jsou splněny tyto podmínky:

- Všechna písmena slova musí být sázena stejným fontem.
- `\hyphenchar` tohoto fontu spadá do rozsahu 0 až 255.

- Před slovem (po přeskočení dalších znaků) musí být mezera typu ⟨*glue*⟩.
- Za slovem (po přeskočení dalších znaků) nesmí být linka, box nebo písmeno.
- Obsahuje-li slovo explicitní `\discretionary`, dělí se jen v něm.

Při rozlišování slov pracuje TEX s již vytvořeným horizontálním seznamem, a nikoli s posloupností tokenů. Proto je jedno, zda písmeno bylo vloženo jako znak nebo pomocí povelu `\char` či sekvencí z `\chardef`.

V částech horizontálního seznamu, které vznikly konverzí z matematického seznamu, se místa pro dělení slov nehledají. Ze sekce 6.1 víme, že v těchto místech navíc není dovolen zlom v mezerách. Aby TEX odlišil úseky horizontálního seznamu, které vznikly z matematického seznamu, obklopuje tyto úseky značkami `\mathon` a `\mathoff`. Tyto značky vidíme například při použití primitivu `\showlists`.

Ligatury v horizontálním seznamu se při vyhledávání míst pro dělení slov zpětně převádějí na odpovídající skupiny písmen, ze kerých byly ligatury sestaveny. Slovo je nakonec sázeno s ligaturou, pokud není v místě ligatury rozděleno. Při rozdělení slova v místě ligatury je sazba nahrazena jednotlivými písmeny. Pokud je ale ligatura vložena explicitně pomocí `\char`, TEX neumí zpětně zjistit, z jakých písmen je složena, a převedení na jednotlivá písmena se neprovede.

Implicitní kerningové informace ve slovech, které vznikají na základě tabulky kerningových párů z použitého fontu, nehrají při rozlišování slov a míst dělení žádnou roli, tj. jako by tam nebyly. Je-li v místě takového kernu provedeno dělení slova, kern mizí a je případně nahrazen jiným implicitním kernem se znakem ⟨*hyphenchar*⟩. Na druhé straně explicitní kerny vložené do seznamu prostřednictvím povelů `\kern` nebo `\/` se uvnitř slov nesmí vyskytovat. Tyto kerny narušují podmínku, že slovo je souvislá skupina písmen.

Při vyhledávání míst dělení TEX zamění ASCII hodnoty písmen za jejich hodnoty z `\lccode`. Tím je zaručeno, že slova budou dělena stejně, i když jsou napsána jednou pomocí malých písmen a jednou pomocí velkých. Výjimku může způsobit jen nastavení registru `\uchyph` na nulu, což potlačuje dělení slov, která začínají velkým písmenem. □

Nyní si procvičíme pojem slova na příkladech. Předpokládejme, že znak „:“ má nulové `\lccode` (tj. další znak) a písmena `s`, `l`, `o`, `v` jsou skutečně malá písmena, tj. mají `\lccode` shodné s ASCII hodnotou znaku.

```
54 \language=5 \righthyphenmin=2 \hyphenchar\tentt=-1
55
56 slovo  % první slovo v odstavci se nedělí, nepředchází mezera
57 :::slovo::        % ano, toto slovo může být rozděleno
58 slovo:slovo       % dvě slova, ani v jednom se nedělí
59 $slovo$           % nedělí se, protože nebylo v horiz. seznamu
```

```
sl{ov}o           % dělí se, skupiny nehrají roli
{\bf s}lovo       % nedělí se, protože fonty jsou různé
{\bf slovo}       % dělí se
{\tt slovo}       % nedělí se, protože \hyphenchar=-1
\kern0pt::slovo   % před slovem není mezera, nedělí se
slovo::\kern0pt{} % dělí se
slovo::\hbox{}    % nedělí se
slovo::\kern0pt\hbox{} % dělí se
slov\-o           % dělí se jen explicitním způsobem
slov-o            % dělí se jen explicitním způsobem
s\char`\l ovo     % to je totéž, jako "slovo", dělí se
sl\/ovo           % dvě slova oddělená explicitním kernem
sl\penalty0ovo    % dvě slova oddělená penaltou
```

□

TEX projde před druhým průchodem algoritmu řádkového zlomu (viz následující sekci 6.4) celý horizontální seznam a přidá ekvivalenty k `\-` do všech míst, ve kterých je možno dělit slova. Nyní si vyložíme algoritmy, podle kterých tato místa najde.

Pravidlo 1. TEX hledá místa pro dělení jen ve slovech, ve kterých jsou splněny podmínky uvedené výše.

Pravidlo 2. Nikdy není automaticky doplněno místo pro dělení v levém a pravém okraji slova. Velikost těchto okrajů se vyjadřuje počtem písmen a je uložena v registrech `\lefthyphenmin` a `\righthyphenmin`. V češtině máme obvykle `\lefthyphenmin=2` a `\righthyphenmin=3`, což zabraňuje dělení typu o-/bluda nebo pomate-/ná.

Pravidlo 3. Je-li slovo zaneseno v tabulce `\hyphenation`, realizují se místa dělení odtud. Příklad takové tabulky:

```
\hyphenation{Post-Script Post-Scripto-vým
             Post-Scripto-vého nezlom}
```

Povel je aditivní, tj. další primitiv `\hyphenation` s dalším seznamem slov přidává seznam slov do již stávající tabulky. Přiřazení je navíc globální a je vztaženo k hodnotě registru `\language` (jak popíšeme později). Povel je možno použít nejen v iniTEXu, ale též v produkční verzi TEXu, tj. například v úvodní části dokumentu.

Slova se v `\hyphenation` oddělují mezerou $\boxed{\sqcup}_{10}$ a místa pro dělení slov se vyznačují tokenem $\boxed{-}_{12}$ bez závislosti na `\lccode` tohoto znaku a bez závislosti na hodnotě `\hyphenchar`. Písmena ve slovech mají kategorii 11 nebo 12 a musí mít nenulovou hodnotu `\lccode`. Písmena lze též psát prostřednictvím povelu `\char` nebo sekvencí z `\chardef`. To ovšem nebývá obvyklé.

Hned při načítání tabulky jsou všechna písmena konvertována prostřednictvím `\lccode` na malá písmena, tj. nebylo podstatné, zda jsme napsali `Post-Script` nebo `post-script`.

Je-li slovo v tabulce zaneseno vícekrát, a přitom pokaždé s jinými místy dělení, pak platí poslední výskyt.

Nevýhoda těchto tabulek pro český jazyk je zřejmá. Slovo se musí zcela shodovat s předlohou v tabulce, takže v tabulce musíme mít všechny tvary odpovídající všem příponám. Problém jsme naznačili v naší ukázce, kde jsme slovo PostScript zapsali do tabulky vícekrát.

Pravidlo 4. Místa pro dělení slova se naleznou prostřednictvím tabulky vzorů `\patterns`. Tento algoritmus popíšeme později. Zjednodušeně řečeno, stačí do tabulky `\patterns` přidat vzor „`.p6o6s6t5s6c6r6i6p6t`“ a máme slovo PostScript správně děleno včetně všech přípon, které k tomu slovu v češtině připojujeme. Tabulku vzorů je možné zavést do TEXu jen při generování formátu (v iniTEXu) a není aditivní.

Jednotlivá pravidla jsou uvedena se sestupnou prioritou. Znamená to, že při dělení slova podle pravidla 3 nebo 4, bude toto dělení vždy v souladu s pravidlem 2. Dále výskyt slova v tabulce `\hyphenation` způsobí, že pro toto slovo nebudou hledána místa dělení podle `\patterns`. Do `\hyphenation` uživatel zapisuje výjimky, tj. slova, která chce dělit jinak, než by je rozdělil algoritmus pracující se vzory v `\patterns`. Zatímco třeba slovo „nezlom“ by se dělilo podle `\patterns` jako „ne-/zlom“, nebude děleno vůbec, protože je zapsáno na řádku 74 bez místa dělení. □

• **Více jazyků současně.** TEX umožňuje pracovat až s 256 různými tabulkami `\patterns` a `\hyphenation`. Mezi jednotlivými tabulkami se přepíná pomocí registru `\language`. Jednotlivé tabulky většinou odpovídají různým jazykům a používají se ve vícejazyčných dokumentech. Jinou aplikací jsou různé tabulky pro jeden jazyk, které se liší podle zrovna použitého kódování fontu.

Číslo tabulky `\patterns` nebo `\hyphenation`, do které se přiřazují data prostřednictvím stejnojmenných primitivů, odpovídá momentální hodnotě registru `\language`. Je-li hodnota tohoto registru mimo interval $\langle 0, 255\rangle$, TEX se chová stejně, jako by tato hodnota byla rovna nule.

Například csplain má rezervováno pro české tabulky číslo 5, které je navíc uloženo do registru `\czech` (aby se to nemuselo pamatovat). Pro americkou angličtinu je vyhrazena tabulka s číslem 0. Při startu csplainu je nastaveno `\language=0`. Pokud tedy uživatel napíše na začátek svého dokumentu:

```
\hyphenation{ab-so-lut-ism}
\language=\czech
\hyphenation{ab-so-lu-tis-mus}
```

pak první slovo (`ab-so-lut-ism`) přidal do anglické tabulky, zatímco druhé slovo (`ab-so-lu-tis-mus`) přidal do tabulky české.

Která tabulka se při hledání míst dělení slov použije? To rovněž závisí na hodnotě registru `\language`. TEX umožňuje v jednom odstavci pracovat s větším množstvím tabulek, ale vyhledávání míst dělení slov se děje až v okamžiku činnosti algoritmu řádkovém zlomu (při povelu `\par`). Proto TEX už v průběhu vytváření horizontálního seznamu vkládá do tiskového materiálu značky typu `\setlanguage`, které nesou číslo použité tabulky. Tyto značky se do seznamu vloží při každé změně registru `\language`.

Uvedeme příklad. Předpokládejme, že registr `\language` je nastaven na hodnotu 5. Pokud napíšeme:

```
Absolutismus (anglicky {\language=0 absolutism}) označuje ...
```

pak na začátku odstavce je značka `\setlanguage5`, za slovem „anglicky“ je značka `\setlanguage0` a před slovem „označuje“ máme v horizontálním seznamu znova značku `\setlanguage5`. TEX pak při hledání míst dělení slov prochází horizontální seznam zleva doprava a přepíná použité tabulky podle hodnot značek `\setlanguage`.

Značky nesou kromě čísla tabulky ještě údaje o hodnotách registrů `\lefthyphenmin` a `\righthyphenmin` v okamžiku vložení značky. Proto lze přepínat i tuto vlastnost v závislosti na jazyku. Například:

```
\def\angl{\language=0 \righthyphenmin=3 }
\language=\czech \righthyphenmin=2
Absolutismus (anglicky {\angl absolutism}) označuje ...
```

způsobí, že se česká slova dělí s `\righthyphenmin=2`, zatímco v angličtině je nastavena hodnota 3.

Pozorný čtenář může namítnout, že máme v ukázce chybu: že je potřeba nejprve nastavit registr `\righthyphenmin` a teprve potom „pohnout“ s registrem `\language`. Má pravdu, i to bude fungovat, ale naše řešení bude fungovat rovněž. Značka `\setlanguage` se totiž do seznamu vkládá po změně `\language` s jistým zpožděním: až před sazbou prvního písmene nebo dalšího znaku, který v seznamu následuje. □

• **Vzory dělení `\patterns`.** Popíšeme nyní podrobně algoritmus na vyhledávání míst dělení pomocí `\patterns`. Data v tabulce `\patterns` obsahují (po případné

expanzi) tzv. *vzory*. Tyto vzory odpovídají úsekům slov, které se v daném jazyku vyskytují a jsou pro dělení slov důležité. Jednotlivé vzory jsou v datech odděleny mezerami $\boxed{\sqcup_{10}}$. Mezi písmeny a na začátku a na konci každého vzoru mohou být číslice 0 až 9 (kategorie 12). Budeme jim říkat *čísla dělení*. Pokud v některém místě číslo dělení chybí, uvažuje se implicitně nula. Písmena jsou tokeny kategorie 11 nebo 12, které mají nenulové `\lccode`. Povel `\char` pro písmeno není povolen. Na začátku nebo na konci vzoru se může objevit znak „.“, který označuje začátek nebo konec slova. Příklad:

```
\patterns{
.po1      % předpona po (po-vel)
.po2d1    % předpona pod (pod-strčit)
c6h       % dvojhláska ch kdekoli uvnitř slova
pá1       % slabika pá (vypá-rat, opá-sat)
pá2d1     % slovní základ pád (pád-ný)
is1mus.   % přípona is-mus (absolutis-mus)
}
```

Při hledání míst dělení TEX přepíše vyšetřované slovo do pracovní oblasti a přidá na začátek a konec slova speciální znaky, které označujeme tečkami. Mezi všechny znaky vloží v pracovní oblasti nulová čísla dělení. Například slovo „podpálit“ je v pracovní oblasti připraveno ve tvaru $\boxed{.{}_0\mathtt{p}_0\mathtt{o}_0\mathtt{d}_0\mathtt{p}_0\mathtt{á}_0\mathtt{l}_0\mathtt{i}_0\mathtt{t}_0.}$.

Nyní TEX projde všechny vzory v tabulce `\patterns`, které jsou shodné s nějakým úsekem vyšetřovaného slova. Pokud je ve vzoru na některém místě vyšší číslo dělení než na odpovídajícím místě v pracovní oblasti, přepíše se v pracovní oblasti číslo dělení podle vzoru. Nakonec tedy čísla dělení v pracovní oblasti odpovídají maximům čísel ze všech vyhovujících vzorů. Například první vzor $\boxed{.{}_0\mathtt{p}_0\mathtt{o}_1}$ našemu slovu vyhovuje. Proto se pracovní oblast změní na $\boxed{.{}_0\mathtt{p}_0\mathtt{o}_1\mathtt{d}_0\mathtt{p}_0\mathtt{á}_0\mathtt{l}_0\mathtt{i}_0\mathtt{t}_0.}$. Dále z naší ukázky vyhovují vzory $\boxed{.{}_0\mathtt{p}_0\mathtt{o}_2\mathtt{d}_1}$ a $\boxed{{}_0\mathtt{p}_0\mathtt{á}_1}$, takže se pracovní oblast nakonec změní na $\boxed{.{}_0\mathtt{p}_0\mathtt{o}_2\mathtt{d}_1\mathtt{p}_0\mathtt{á}_1\mathtt{l}_0\mathtt{i}_0\mathtt{t}_0.}$.

Všude tam, kde v pracovní oblasti zůstalo liché číslo dělení, je dělení povoleno a v sudých číslech je dělení zakázáno. V našem příkladě se tedy slovo „podpálit“ může rozdělit jako „pod-pá-lit“. Tím jsou místa dělení určena. Sudá čísla dělení ve vzorech tedy potlačují v daném místě zlom, zatímco lichá čísla dělení označují místa, kde je zlom povolen. Větší číslo dělení může zrušit platnost menších čísel z jiných vzorů. □

Ačkoli lze při troše píle sestavit třeba pro češtinu celkem rozumné vzory dělení heuristickým způsobem (postupně vyjmenujeme všechny předpony, přípony, slovní základy a slabiky), je účelnější použít program `patgen`. Tomuto programu se předloží rozsáhlé slovníky daného jazyka, v nichž jsou všechna místa dělení ve slovech vyznačena. Po několika průchodech nad těmito slovníky program vytvoří data

pro \patterns. Úspěšnost takových vzorů dělení pak závisí jen na kvalitě vstupních slovníků. Je překvapivé, že patgen je většinou schopen komprimovat obrovské množství vstupní informace ze slovníků do vzorů dělení \patterns velikosti pár desítek kilobytů. Navíc si většinou vystačí s čísly dělení v rozsahu 0 až 4 (viz například české vzory dělení). Právě v této komprimaci vstupní informace realizované programem patgen spočívá hlavní síla popsaného algoritmu.

Při činnosti programu patgen je potřeba specifikovat, pro jaké hodnoty registrů \lefthyphenmin a \righthyphenmin se mají vzory dělení vygenerovat. Vzory dělení, které vytvořil Pavel Ševeček a které jsou použity v csplainu, byly generovány pro \lefthyphenmin=\righthyphenmin=2. V csplainu je použita zkratka \chyph, která „zapíná češtinu“. Toto makro jednak nastaví \language na 5 (tj. přepne do českých vzorů dělení), a dále nastaví \lefthyphenmin=2 a \righthyphenmin=3, což lépe odpovídá pravidlům dobré sazby. Při sazbě do úzkých sloupců je možné nastavit registr pro pravý okraj na dvojku, protože vzory dělení pro takovou hodnotu připraveny jsou. □

Data pro \patterns se většinou zapisují v sedmibitovém tvaru prostřednictvím TEXovských sekvencí \v, \', \r apod. Sekvence \r je pro náš jazyk důležitá, protože označuje kroužek (ring) a další dvě čtenář určitě zná. Před načtením dat pro \patterns, která jsou tedy zapsána způsobem nezávislým na použitém kódování akcentovaných znaků ve fontech v TEXu, je potřeba vložit makra definující použité osmibitové kódování. Tato makra zajistí, aby se sekvence \v apod. s následujícím písmenem expandovaly do odpovídajícího osmibitového kódu. Například \v r se expanduje na $\boxed{\text{ř}}_{11}$, přičemž pod znakem „ř“ se skrývá nějaký konkrétní kód v použitém kódování. Dále samozřejmě makra nastavují \lccode‘ř=‘ř.

TEX může načíst stejné vzory dělení vícekrát za sebou pod různým \language a při různé definici maker deklarujících osmibitové kódování. Tím je možno načíst například české vzory dělení jednak v kódování $\mathcal{CS}$-fontů a jednak třeba v kódování T1. Tento trik je použit ve Zlatuškově počeštění LATEXu. V dokumentu je pak možno přepínat mezi fonty různě kódovanými změnou registru \language. Aby bylo kódování vstupního textu dokumentu jednotné, je potřeba navíc zvolit nějaké pevné vstupní kódování. Dále se všem znakům s kódem nad 128 přiřadí aktivní kategorie. Tyto znaky se expandují do kódu podle zrovna vybraného fontu. Tím je překódování mezi vstupem a výstupním fontem děláno na úrovni maker. To je sice v LATEXu obvyklé, ale vede to k některým problémům. Je lépe se takovému makro-tanci vyhnout a mít vstupní kódování shodné s kódováním fontů, případně překódovat na úrovni input procesoru nebo virtuálních fontů. □

Jak se po-dí-vat na všech-na mís-ta dě-le-ní, kte-rá TEX na-šel? Jed-nou mož-nos-tí je po-u-žít makro plainu \showhyphens. Ne-bo si udě-lá-me makro, kte-ré vy-sá-zí ce-lý od-sta-vec tak, ja-ko je ten-to. Na za-čá-tek od-stav-ce na-pí-še-me \showhyphenpar. Dá-le bez vy-ne-chá-ní řád-ku po-kra-ču-je text od-stav-ce na-psa-ný ob-vyk-lým způ-so-bem. Budeme-li psát \showhyphenpar před kaž-dý

od-sta-vec, ve-li-ce zá-hy zjis-tí-me, jak kva-lit-ní jsou po-u-ži-té vzo-ry dě-le-ní \patterns. Makro je de-fi-no-vá-no ná-sle-dov-ně:

```
\def\showhyphenpar{\bgroup
   \def\par{\setparams\endgraf\composelines}
   \setbox0=\vbox\bgroup \noindent\hskip0pt\relax}
\def\setparams{\rightskip=0pt plus1fil
   \linepenalty=1000 \pretolerance=-1 \hyphenpenalty=-10000}
\def\composelines{\setbox2=\hbox{}%
   \loop \setbox0=\lastbox \unskip \unpenalty
   \ifhbox0 % je už \vbox vyprázdněn?
     \global\setbox2=\hbox{\unhbox0\unskip\unhbox2}
   \repeat
   \egroup % vyprázdněný \vbox ukončím
   \exhyphenpenalty=10000 % \box2 vyšoupnu do vnějšího seznamu
   \emergencystretch=4em \leavevmode\unhbox2 \endgraf\egroup}
```

Makro je poněkud komplikované, protože TEX neumí nabídnout výsledek vyhledávání míst dělení do sazby ve formě viditelných značek. Nejprve je vytvořen odstavec v pracovním \vboxu se speciálními parametry. Tento odstavec je zvláštní tím, že má ukončen řádek v každém místě dělení slov a v žádné mezeře. Tím dosáhneme sazby znaku \hyphenchar. Řádky jsou ale poměrně krátké. Proto pomocí \composelines vytvořený vertikální seznam znovu rozebereme a sestavíme do jednoho dlouhého boxu (\box2), který znovu zpracujeme do odstavce. Tentokrát dělíme naopak pouze v mezerách. □

Česká pravidla sazby dovolují dělit slovo při jeho pravém okraji tak, že na následujícím řádku mohou zůstat jen dvě písmena v případě, kdy se k těmto písmenům připojuje interpunkce (tečka, čárka). Jinak na dalším řádku musí být aspoň tři písmena. Tuto skutečnost algoritmy TEXu neumějí jednoduše ošetřit, protože levý a pravý okraj je počítán jen v rámci slova a nejsou zahrnuty další znaky. Nicméně jistá možnost tady je: Volíme \righthyphenmin=3 a přidělíme tečce, čárce a další interpunkci význam písmene:

```
\lccode`;=`;  \lccode`:=`;
\lccode`,=`;  \lccode`.=`;  \lccode`!=`; \lccode`?=`;
```

Dále v \patterns zdvojíme všechny vzory, které odpovídají příponě (mají na konci tečku) a do alternativního vzoru vložíme před koncovou tečku středník. □

Na závěr si ukážeme jeden trik, jak zapsat do dat v \patterns znak ve významu písmene, který má ale v \patterns rezervovanou ASCII hodnotu. Jedná se o znak tečky a dále o číslice. Využije se toho, že při načítání \patterns se okamžitě provádějí konverze všech písmen na malá podle hodnoty \lccode. Těsně před načtením nastavíme třeba:

```
105 \lccode`@=`.
```

a v datech vyjádříme skutečnou tečku pomocí znaku „@“. Po načtení \patterns je možné vrátit \lccode znaku „@“ zpátky na nulu. □

6.4. Řádkový zlom

V této sekci se budeme zabývat jedním z nejzajímavějších algoritmů TEXu, který zcela určitě předběhl svou dobu. Při hledání míst pro řádkový zlom k vytvoření odstavce TEX totiž nepostupuje jednotlivě řádek po řádku, ale hledá globální řešení problému s minimálním součtem chyb (badness a dalších parametrů) na všech řádcích.

Po obdržení povelu \par v odstavcovém módu TEX zakončí nejprve horizontální seznam prostřednictvím \unskip (odebere případnou mezeru), dále přidá: \penalty10000\hskip\parfillskip (mezera pro východový řádek). Nakonec seznam uzavře vložením \penalty-10000, což si vynutí závěrečný řádkový zlom.

Nyní se spustí rutina, která v takto kompletovaném horizontálním seznamu najde nejlepší místa pro řádkový zlom. Tato rutina pracuje nejvýše ve třech průchodech, jak je popsáno níže. Úkolem rutiny je (v obecném případě) najít jednotlivé řádkové zlomy tak, aby se každý řádek dal vložit do horizontálního boxu o šířce \hsize. Výjimky z tohoto pravidla je možné stanovit prostřednictvím registrů \leftskip, \rightskip, \hangindent, \hangafter nebo prostřednictvím primitivu \parshape. O těchto výjimkách se zmíníme později. Abychom zahrnuli i možnost výjimek, budeme v dalším textu místo o \hsize mluvit obecně o *požadované šířce*.

1. průchod. Vyhodnotí se všechny možné řádkové zlomy, při nichž současně platí: (A) všechny řádky s požadovanou šířkou mají badness menší nebo rovnu \pretolerance a (B) nepoužije se dělení slov. Pokud řešení s těmito podmínkami neexistuje, rutina přejde do druhého průchodu. Pokud aspoň jedno řešení existuje, zvolí rutina řešení s nejmenším demerits a řádkový zlom je ukončen. O pojmu „demerits“ viz níže.

2. průchod. Vyhodnotí se všechny možné řádkové zlomy, při nichž platí, že všechny řádky s požadovanou šířkou mají badness menší nebo rovnu \tolerance. V tomto průchodu se berou v úvahu i místa pro dělení slov. Proto TEX před zahájením tohoto průchodu projde celý horizontální seznam a přidá do něj ekvivalenty k \- podle tabulek vzorů dělení slov (\patterns) a podle slovníku výjimek (\hyphenation). Pokud řešení s podmínkou na „toleranci“ neexistuje, rutina přejde do třetího průchodu. Jinak rutina zvolí řešení s nejmenším demerits a řádkový zlom je ukončen.

3. průchod. Tento průchod byl přidán do TEXu až od verze 3.0. Celková hodnota roztažení každého řádku se před výpočtem badness zvětší o `\emergencystretch` a dále se postupuje stejně jako v druhém průchodu. Mezery se chovají, jakoby si v každém řádku mezi sebe rovným dílem rozdělily dodatečnou hodnotu roztažení z `\emergencystretch`. Pokud řešení ani v tomto případě s podmínkou na danou „toleranci“ neexistuje, rutina ponechá nějaký řádek „`Overfull`“. Jaký je v takovém případě zvolen řádkový zlom, nebudeme dále rozebírat, protože nežádoucí efekt je potřeba stejně ručně odstranit. Pokud řešení existuje, rutina zvolí řešení s nejmenším demerits. Ačkoli při výpočtu badness jednotlivých řádků byla brána v úvahu přidaná hodnota `\emergencystretch` a takto vypočtené hodnoty badness byly použity i při celkovém výpočtu demerits, při závěrečné kompletaci řádků do vertikálního seznamu se znovu přehodnotí badness, jako by přidané hodnoty natažení neexistovaly. Tyto výsledky se použijí pro tisk případných varování typu „`Underfull`“ na terminálu a do souboru `log`.

Poznamenejme, že třetí průchod byl do TEXu přidán hlavně z toho důvodu, že funkce badness má v hodnotě 10 000 nespojitou derivaci a od tohoto bodu je konstantní. Proto nedokáže rozlišovat mezi „hodně“ natažených mezerami a „příšerně“ natažených mezerami. Takže volba `\tolerance=10000` sice vede (skoro vždy) k výsledkům už ve druhém průchodu, ale algoritmy pro výpočet demerits mají tendenci kumulovat hodnoty badness 10 000 do jediného řádku, místo aby je souvisle rozložily do více řádků. Tím dostáváme jeden řádek s „příšernými“ mezerami, místo více řádků s více méně souvisle ale „hodně“ natažených mezerami. □

• **Výpočet hodnoty demerits.** Čím vyšší demerits, tím větší je „celková nevhodnost“ konkrétního řešení řádkového zlomu v odstavci. Z předchozího víme, že rutina řádkového zlomu vybírá mezi možnými řešeními to, které má nejmenší demerits. Uvedeme nyní výpočet této hodnoty. Demerits celého odstavce je součet demerits ze všech řádků odstavce plus dodatečná demerits za výskyty určitých jevů z registrů `\adjdemerits`, `\doublehyphendemerits` a `\finalhyphendemerits`. Demerits jednotlivého řádku se počítá podle vzorce:

$$d = (l + b)^2 \pm p^2$$

kde l je konstantní hodnota z registru `\linepenalty`, b je badness boxu s řádkem o požadované šířce a p hodnota penalty v místě zlomu. Znaménko minus se ve vzorci použije při záporném p a znaménko + při nezáporném. Pro $p \leq -10\,000$ je výjimečně hodnota d rovna jen $(l + b)^2$. Tato nespojitost funkce demerits nám nevadí, protože v těchto místech je TEX nucen zlomit za jakýchkoli okolností.

Vidíme, že hodnoty demerits se počítají v kvadrátech penalt. Dále si uvědomíme, že tajuplná konstanta l může ovlivnit celkový počet řádků odstavce. Za každý řádek dostane řádkový zlom demerits aspoň l^2, což při velkém l vede k minimalizaci počtu řádků odstavce. O dalších možnostech, jak ovlivnit počet řádků odstavce, viz primitiv `\looseness` v části B.

Hodnota p je určena buď explicitně při zlomu v místě `\penalty`, nebo implicitně z registrů `\hyphenpenalty`, `\exhyphenpenalty` při rozdělení slova. Při zlomu v mezeře je $p = 0$.

Za každou dvojici „vizuálně nekompatibilních“ řádků v odstavci, které následují těsně za sebou, je přidáno dodatečné `\adjdemerits`. Vizuálně nekompatibilní dvojice řádků je případ, kdy v jednom řádku jsou mezery příliš staženy a v druhém příliš roztaženy. Přesněji, každý řádek dostane číslo: `3`, `2`, `1` nebo `0`. Toto číslo závisí na poměru stažení, resp. roztažení, mezer takto: jsou-li mezery staženy na víc než 50 %, tj. $b \geq 13$, má řádek číslo `3` (zúžený). Při $b < 13$ (je jedno, zda se jedná o stažení nebo roztažení) má řádek číslo `2` (vhodný). Při roztažení mezi 50 až 100 %, tj. $b \in \langle 13, 100)$, má řádek číslo `1` (roztažen) a pro $b \geq 100$ je řádek opatřen číslem `0` (velmi roztažen). Dvojice řádků nazýváme *vizuálně nekompatibilní*, pokud se hodnoty těchto čísel liší o více než 1.

Dodatečné `\doublehyphendemerits` je přidáno za každou dvojici po sobě následujících řádků, v nichž je užito dělení slov. Dodatečné `\finalhyphendemerits` je přidáno za rozdělení slova na konci předposledního řádku odstavce. □

• **Sestavení odstavce.** V závěrečné fázi algoritmus řádkového zlomu vloží jednotlivé řádky do vertikálního seznamu. Za každý řádek se případně vloží materiál z `\vadjust`, pokud v daném řádku existuje. Před každou meziřádkovou mezeru se vloží penalta o hodnotě `\interlinepenalty`, která je mezi prvním a druhým řádkem navíc zvětšená o `\clubpenalty` a mezi předposledním a posledním o `\widowpenalty`. Končí-li řádek rozděleným slovem, je dále přičtena `\brokenpenalty`. Je-li hodnota penalty nula, nevkládá se do vertikálního seznamu vůbec. Zmíněné penalty mohou ovlivnit pozdější stránkový zlom.

Úplně nakonec TEX pronuluje registry `\looseness` a `\hangindent`. Registr `\hangafter` nastaví na jedničku. Také zruší případné nastavení z `\parshape`. Tím jsou algoritmy, které se skrývají za povelem `\par`, ukončeny. □

Sepíšeme si nyní seznam všech registrů TEXu, které ovlivňují řádkový zlom. V závorce je uvedena hodnota, která je použita ve formátu plain.

registr	*poznámka*
`\hsize` (6.5 in)	požadovaná šířka řádku
`\leftskip` (0 pt)	mezera vložená dovnitř řádků zleva
`\rightskip` (0 pt)	mezera vložená dovnitř řádků zprava
`\hangindent` (—)	jinak definovaný tvar odstavce
`\hangafter` (—)	jinak definovaný tvar odstavce
`\parshape` (—)	jinak definovaný tvar odstavce
`\parindent` (20 pt)	odstavcová zarážka
`\parfillskip` (0pt plus 1fil)	mezera východového řádku

`\font` (`cmr10`)	mezislovní mezera podle použitého fontu nebo podle `\spaceskip` a `\xspaceskip`
`\pretolerance` (100)	maximální badness řádku 1. průchodu
`\tolerance` (200)	maximální badness řádku 2. a 3. průchodu
`\emergencystretch` (0pt)	dodatečné natažení v 3. průchodu
`\linepenalty` (10)	parametr l pro výpočet demerits řádku
`\patterns` (`hyphen.tex`)	tabulka dělení slov
`\hyphenation` (—)	tabulka výjimek dělení slov
`\language` (0)	volba jazyka (konkrétní tabulky dělení slov)
`\hyphenpenalty` (50)	penalta za běžné rozdělení slova
`\exhyphenpenalty` (50)	penalta za rozdělení s prázdným „pre-break“
`\hfuzz` (0.1pt)	dovolené „přečnívání“ boxu
`\hbadness` (1000)	mez, od které vidíme `Underfull \hbox`
`\adjdemerits` (10 000)	za vizuálně nekompatibilní řádky
`\doublehyphendemerits` (10 000)	za dva následující řádky s rozdělením slova
`\finalhyphendemerits` (5 000)	za rozdělení v předposledním řádku

Problematika, zda dělit či nedělit slova, jak a s jakou penalizací, byla vyložena v sekcích 6.1 a 6.3. Parametry, které určují tvar odstavce, budeme diskutovat v následující sekci 6.5. Nyní se zaměříme na jemnosti registrů pro řízení algoritmu řádkového zlomu. Předpokládejme volbu nějaké konkrétní šířky sloupce (tj. `\hsize`). Velmi dobré náměty k této problematice vyslovil Phil Taylor ve svém článku *Pragmatický přístup k odstavcům*, který byl přeložen v TEXbulletinu 3/94.

Při sazbě do užších sloupců je výchozí nastavení registrů z plainu nevyhovující. Narychlo lze tento problém řešit nastavením `\emergencystretch` na kladnou hodnotu, např. `1em`, a ignorovat hlášení `Underfull`. Phil Taylor ale navrhuje nejprve experimentovat s přidáním `\tolerance` a nepustit TEX do třetího průchodu, tj. ponechat `\emergencystretch=0pt`. Doporučený postup spočívá v nastavení `\tolerance` na 9 999, tj. není dovoleno pouze badness 10 000. S touto hodnotou provedeme první pokus zalomení dokumentu. Samozřejmě u této hodnoty nezůstaneme.

Na prvním místě je potřeba se zaměřit na výskyty `Overfull` a je nutné je řešit individuálně. Někdy stačí dodat nové místo pro rozdělení slova pomocí `\-`, jindy budeme zvažovat, proč musí být takto dlouhý matematický vzorec uveden v textu odstavce apod.

Dále si všimneme největší hodnoty badness v celém dokumentu a pokusíme se nastavit `\tolerance` o jedničku menší. Kupodivu i v tomto případě může algoritmus najít řešení (sice s větší demerits, ale s menším maximem badness z jednotlivých řádků). Tento experiment se dá opakovat tak dlouho, až najdeme nejmenší `\tolerance`, při níž se ještě neobjeví `Overfull`.

Některé odstavce je nutné řešit individuálně např. přidáním `\emergencystretch`. Nesmíme zapomenout lokální nastavení uzavřít do skupiny a skupinu ukončit až po `\par` (nebo prázdném řádku).

Doporučuje se při výše popsaném experimentování volit `\hbadness` tak, abychom na terminálu viděli jen nejkřiklavější případy, na které se zrovna musíme zaměřit. □

Dříve nebo později narazíme na případ, kdy TEX zalomí odstavec z lidského hlediska nepochopitelným způsobem. V takovém případě se hodí nastavit `\tracingparagraphs` na jedničku a podívat se do souboru `log`, co říká algoritmus řádkového zlomu. Jeho výpis může být pro začátečníky poměrně nepřehledný, proto jej zde rozebereme. Například při trasování řádkového zlomu tohoto odstavce dostáváme:

```
@firstpass
[][]\tenrm Dříve nebo později narazíme na případ, kdy T[]X
zalomí odstavec z lidského hlediska
@ via @@0 b=69 p=0 d=6241
@@1: line 1.3 t=6241 -> @@0
@secondpass
[][]\tenrm Dříve nebo poz-ději na-ra-zíme na pří-pad, kdy T[]X
za-lomí od-sta-vec z lid-ského hle-
@\discretionary via @@0 b=5 p=50 d=2725
@@1: line 1.2- t=2725 -> @@0
diska
@ via @@0 b=69 p=0 d=6241
@@2: line 1.3 t=6241 -> @@0
@emergencypass
[][]\tenrm Dříve nebo poz-ději na-ra-zíme na pří-pad, kdy T[]X
za-lomí od-sta-vec z lid-
@\discretionary via @@0 b=185 p=50 d=50525
@@1: line 1.0- t=50525 -> @@0
ského
@ via @@0 b=33 p=0 d=1849
@@2: line 1.1 t=1849 -> @@0
hle-
@\discretionary via @@0 b=1 p=50 d=2621
@@3: line 1.2- t=2621 -> @@0
diska
@ via @@0 b=69 p=0 d=6241
@@4: line 1.3 t=6241 -> @@0
ne-po-cho-pi-tel-ným způ-so-bem. V ta-ko-vém
pří-padě se hodí na-
@\discretionary via @@1 b=80 p=50 d=20600
@@5: line 2.1- t=71125 -> @@1
```

```
sta-
@\discretionary via @@1 b=18 p=50 d=13284
@@6: line 2.1- t=63809 -> @@1
vit
@ via @@1 b=3 p=0 d=10169
@ via @@2 b=155 p=0 d=27225
@@7: line 2.0 t=29074 -> @@2
.........................................atd.
stavce
@ via @@18 b=8 p=0 d=324
@ via @@19 b=8 p=0 d=324
@ via @@20 b=25 p=0 d=11225
@@30: line 5.2 t=40282 -> @@18
do-
@\discretionary via @@20 b=0 p=50 d=2600
@@31: line 5.2- t=42798 -> @@20
stá-
@\discretionary via @@20 b=38 p=50 d=4804
@@32: line 5.3- t=45002 -> @@20
váme:
@\par via @@21 b=0 p=-10000 d=5100
@\par via @@22 b=0 p=-10000 d=5100
.........................................atd.
@\par via @@30 b=0 p=-10000 d=100
@\par via @@31 b=0 p=-10000 d=5100
@\par via @@32 b=0 p=-10000 d=5100
@@33: line 6.2- t=40382 -> @@30
Underfull \hbox (badness 2426) in paragraph at lines 204--211
\tenrm hle-diska ne-po-cho-pi-tel-ným způ-so-bem. V ta-ko-vém
pří-padě se hodí na-sta-vit
```

Popíšeme si, co se stalo. Ukážeme si tím další tajemství algoritmu řádkového zlomu. Hledání minima hodnoty demerits je totiž prováděno překvapivě efektivní metodou. Poznamenejme, že v naší ukázce byly použity hodnoty `\pretolerance`=150, `\tolerance`=250 a `\emergencystretch`=20pt.

První i druhý průchod narazil při zpracování druhého řádku odstavce na neřešitelný problém v podobě nedělitelného boxu s textem „`\tracingparagraphs`“. Proto byly tyto průchody předčasně ukončeny (viz `@firstpass` a `@secondpass`). Každá značka s označením `@@`číslo je potenciální bod zlomu. Číslování probíhá průběžně od nuly, přitom `@@0` značí začátek odstavce. Před každým řádkem „`@@`“ je seznam jiných řádků s jediným znakem `@`, které vypisují, od kterého předešlého bodu zlomu (`via`) je možné ke stávajícímu bodu zlomu dospět, aniž by byla překročena (pre)`\tolerance`. Pro každý řádek `@` je spočítána badness (`b`), uvedena penalta (`p`) a demerits (`d`). Údaj `\discretionary` označuje, že se v tomto místě

rozděluje slovo. Protože každý bod zlomu `@@` nese s sebou celkovou sumu demerits počítáno až po tento bod (`t`), je možné se zpětně v jednotlivých řádcích typu „`@`“ vrátit k odpovídajícím předchozím bodům zlomu a na základě toho napočítat novou celkovou sumu demerits. V tomto okamžiku se vybere jen jediný řádek typu „`@`“, který vede k minimální celkové sumě demerits a do řádku „`@@`“ se zanese výsledný odkaz na ten „nejlepší“ předchozí bod zlomu za šipku a vypíše se nový součet demerits. Dále se vypíše pořadové číslo takto vzniklého řádku (`line`) a za tečkou je hodnota pro počítání vizuální nekompatibility.

Na konci odstavce (označeno pomocí `@\par`) algoritmus zjistil, že nejvhodnější předposlední bod zlomu má číslo 30. V tomto bodu zlomu si přečteme, že nejvhodnější předchozí bod zlomu má číslo 18 atd., až po bod `@@0`. Tím máme zrekonstruováno řešení s nejmenším celkovým součtem demerits. A je hotovo.

Všimneme si, že ve třetím průchodu TEX už pro první řádek našel daleko více možností pro body zlomu. Je to tím, že je do mezer přidána pružnost a hodnoty badness vycházejí podstatně menší, než v prvním a druhém průchodu. Také si všimneme, že teprve v druhém průchodu jsou označena všechna místa pro dělení slov.

Čtenáři určitě neušlo, že druhý řádek v naší ukázce vyšel kvůli přítomnosti boxu s textem „`\tracingparagraphs`“ s celkem velkými mezerami (badness 2 426). Nikde jinde v této knize takto velké mezery nejsou. Rozhodl jsem se totiž spojit výhody faktu, že jsem v případě tohoto dokumentu autorem a sazečem v jedné osobě. Když byla badness větší než 2 000, řešil jsem případ vždy individuálně. Pro výpis na terminál jsem měl tedy nastaveno `\hbadness=2000`. Protože jsem nepovolil dělit řídicí sekvence uváděné v textu odstavce, občas jsem na problém řádku s větší badness narazil. V takovém případě jsem mírně přeformuloval text. Tento přístup nelze samozřejmě použít, je-li sazečem někdo jiný než autor textu. Tehdy je potřeba dopředu zvážit, co lze a co ne. Pokud by například autor obdobného textu, jako je tento dokument, trval na tom, že mezery se nesmí natahovat víc než např. badness rovno xy a navíc se nesmí zlomit text v ukázkových řídicích sekvencích (nebo v matematických vzorcích v textu nebo v delším logu firmy, které se v textu často vyskytuje), pak musí sazeč autorovi slušně vysvětlit, že to bez přeformulování některých míst v textu nebude možné. □

6.5. Tvar odstavce

V této sekci se budeme věnovat parametrům, které určují tvar odstavce. V první řadě se jedná o registr `\parindent`, který obsahuje šířku prvního vloženého boxu pro odstavcovou zarážku. Dále je to registr `\parfillskip`, který označuje velikost mezery ve východové řádce. Právě touto mezerou se nyní budeme zabývat.

Například v knížce [7] jsem pro zadní stranu obálky nastavil `\parfillskip=0pt`. Tím jsem dosáhl ve všech odstavcích nulové mezery v posledním řádku. Každý odstavec má pak nutně obdélníkový tvar. To nebývá příliš obvyklé, protože takto nastavená mezera východového řádku bude velmi často dělat problémy při zalamování. Uvedeme proto poněkud praktičtější příklad. Nastavení:

```
106 \parfillskip=2em plus1fil
```

vede k tomu, že každý východový řádek bude obsahovat mezeru na konci o šířce aspoň 2 em. Takové nastavení je nezbytné, volíme-li `\parindent=0pt` a mezi odstavce neklademe žádnou vertikální mezeru. Čtenář totiž musí nějakým způsobem vědět, že zrovna končí odstavec.

Mezi kritéria dobré sazby patří požadavek na to, aby text ve východovém řádku odstavce nebyl kratší, než zarážka následujícího odstavce. Pokud bychom v mezeře východového řádku omezili hodnotu roztažení, nepomůžeme si, protože TEX dovolí roztahovat i nad uvedenou mez. Na druhé straně TEX nedovolí překročit stažení a toho využijeme v následujícím triku:

```
107 \parfillskip=\hsize
108 \advance\parfillskip by -1.5\parindent
109 \advance\parfillskip by 0pt minus \parfillskip
110 \advance\parfillskip by 0pt minus -1em
```

Po těchto operacích bude mít registr `\parfillskip` základní velikost rovnu šířce sazby minus 1,5 krát odstavcová zarážka. Hodnota natažení bude nulová a hodnota stažení bude stejná jako základní velikost, jen zmenšená o 1 em. V tomto případě TEX dovolí vytvořit mezeru ve východovém řádku o velikosti v rozsahu od 1 em do `\hsize` minus 1,5 krát odstavcová zarážka. □

Zalamovací algoritmus vkládá zleva, resp. zprava, dovnitř každého boxu s řádkem mezery podle registrů `\leftskip`, resp. `\rightskip`. Obvykle mají tyto registry nulové hodnoty, ale dají se s nimi dělat některá kouzla. Obsahují-li tyto registry hodnoty stažení nebo roztažení, samozřejmě se použijí už při vyhodnocování badness řádku v rámci výpočtu demerits. Sazbu na pravý praporek bychom tedy zařídili nastavením `\rightskip=0pt plus1fil` a analogicky pro levý praporek. Lepší nastavení, viz makro `\raggedright` v části B. □

V popiscích pod obrázky se nám někdy hodí sazba odstavce do bloku, ovšem poslední východový řádek je centrovaný (jako v tomto odstavci). Tento efekt nám zajistí následující nastavení registrů:

```
111   \leftskip=0pt plus1fil \rightskip=0pt plus-1fil
112   \parfillskip=0pt plus2fil \parindent=0pt
```
□

Oblomení pravoúhlých obrázků textem odstavce je možné zařídit prostřednictvím registrů \hangindent a \hangafter. První z nich je typu ⟨*dimen*⟩ a druhý typu ⟨*number*⟩. Při nulovém \hangindent se nic neděje. Jinak bude prvních \hangafter řádků ponecháno beze změny a ostatní budou odsazeny a zúženy o \hangindent. Výsledkem je tedy bílý obdélník, který vykrajuje text odstavce zleva dole, jako v tomto případě: \hangindent=50pt, \hangafter=4.

Bílý obdélník je možno prostřednictvím zmíněných registrů dostat i do ostatních tří „rohů“ odstavce. Má-li být vlevo, volíme \hangindent kladné, má-li být vpravo, volíme stejnou hodnotu \hangindent, ale se znaménkem minus. Podobně záporné \hangafter $= -n$ znamená, že se odsazení týká prvních n řádků v odstavci a ostatní řádky zůstanou nezměněny. V tomto odstavci jsme volili \hangindent=50pt a \hangafter=-2.

Registry \hangindent a \hangafter nám dobře poslouží v případě, že chceme do odstavce vsadit větší iniciálu. Tehdy dokonce uvítáme, že hodnota \hangindent je rutinou pro sestavení odstavce vždy vynulována, protože iniciálu obvykle nevsazujeme do každého odstavce. Vadí-li nám toto vynulování, místo \par pišme {\par}. Rutina pak pronuluje hodnotu lokálně a po ukončení skupiny se registr vrátí k původní hodnotě.

Odstavce, ve kterých máme jen první řádek poněkud vysunut, jako v tomto případě, je vhodné řešit nikoli prostřednictvím \hangindent a \hangafter, ale použít globální nastavení například pomocí:

```
113 \parindent=-2em \leftskip=2em
```

□

Pokud chceme oblomit textem větší obrázky, většinou narazíme na problémy s následným stránkovým zlomem. Obrázky lze totiž touto metodou ukotvit jen k textu odstavce, ale nikoli k podkladu strany. Proto je rozumné vkládat takové obrázky až v době, kdy je dokument připraven k definitivnímu zpracování a pracovat interaktivně: podle toho, kam vyšel text, tam ukotvit k textu obrázek. Při takovém řešení už není možné automaticky přeformátovat text do jiného rozměru tiskového zrcadla, ani dělat dodatečně změny v dokumentu. Jiný přístup, který vychází z předpokladu, že na každé straně je pevný počet řádků, uvádím jako ukázku na konci této sekce. □

Zcela obecný tvar odstavce je možné deklarovat primitivem \parshape. Za touto řídicí sekvencí musí následovat číslo ⟨*number*⟩, které udává, kolik řádků bude v odstavci zpracováno speciálním způsobem. Označme toto číslo n.

Za ním musí následovat n dvojic typu ⟨*dimen*⟩. První údaj z každé dvojice udává velikost odsazení a druhý šířku příslušného řádku. První dvojice patří prvnímu řádku, druhá druhému atd. Šířka musí být kladná, ale odsazení může být i záporné — pak řádek vyčnívá doleva přes levou hranici tiskového zrcadla. Je-li v odstavci

více než n řádků, jsou všechny další zpracovány stejným způsobem, jako n-tý. Je-li jich méně, přebytečné údaje z \parshape se nepoužijí. Tento odstavec byl zpracován pomocí:

```
\newdimen \s \s=.7\hsize
\parshape 5 .3\hsize\s .2\hsize\s .1\hsize\s 0pt\s 2cm10cm
```
□

Uvedeme si nyní příklad na praktické použití tohoto primitivu. Nechť máme dosti dlouhý odstavec a chceme jím oblomit pravoúhlý obrázek tak, aby několik prvních řádků bylo zpracováno beze změny, pak budou řádky odsunuté a zúžené a konečně na konci odstavce zase bude několik řádků usazeno obvyklým způsobem na plnou šířku. S tímto úkolem si registry \hangindent a \hangafter neporadí. Proto použijeme \parshape. Vytvoříme makro \oblom, které se volá se třemi parametry třeba takto:

```
\oblom 5cm od 3 odsadit 5
```

což znamená „oblom obrázek široký 5 cm počínaje třetím řádkem odstavce, přičemž celkem bude odsazeno 5 řádků“. Zde je slíbené makro:

```
\catcode`\@=11
\newcount\shapenum \newcount\tempnum
\newdimen\ii \newdimen\ww
\def\oblom #1 od #2 odsadit #3 {\par \ii=#1 \ww=\hsize
   \ifdim\ii>\z@ \advance\ww by-\ii
   \else \advance\ww by\ii \ii=\z@ \fi
   \shapenum=1 \tempnum=0 \def\shapelist{}
   \loop \ifnum\shapenum<#2 \edef\shapelist{\shapelist\z@\hsize}
      \advance\shapenum by1 \repeat
   \loop \edef\shapelist{\shapelist\ii\ww}
      \advance\tempnum by1 \ifnum\tempnum<#3 \repeat
   \advance\shapenum by#3 \edef\shapelist{\shapelist\z@\hsize}
   \doshape}
\catcode`\@=12
\def\doshape{\parshape \shapenum \shapelist}
```

Do registru \ii uložíme hodnotu odsazení a do registru \ww šířku odsazovaného řádku. Uživatel může napsat též \oblom -5cm, což bude znamenat odsazení vpravo. Proto pomocí \ifdim na řádku 121 ošetříme tyto alternativy. Pak postupně naplníme pracovní makro \shapelist jednotlivými dvojicemi typu ⟨*dimen*⟩. Nejprve se vloží #2 − 1 dvojic typu \z@\hsize (\z@ je zkratka v plainu pro 0pt). Dále vložíme #3 dvojic typu \ii\ww (tj. vlastní odsazení). Poslední dvojice \z@\hsize vrátí tvar odstavce do původního stavu. Registr \shapenum udává počet vložených dvojic. Proto nakonec stačí spustit \doshape jako \parshape \shapenum \shapelist. Makro \oblom napíše uživatel těsně před odstavec, kterého se týká. Už jsme si

uvedli, že ostatní odstavce nebudou změněny, protože algoritmus řádkového zlomu vždy zruší na konci své činnosti případné nastavení \parshape.

Toto makro sice funguje, ale brzy s ním přestaneme být spokojeni. Rádi bychom, aby pokyn \oblom měl širší platnost a mohl zasáhnout i do více odstavců pod sebou. Jakmile třeba napíšeme \oblom 5cm od 1 odsadit 30, chtěli bychom mít jistotu, že bude odsazeno 30 řádků bez ohledu na to, kolika odstavců se to bude týkat. Řešení vede přes použití registru \prevgraf, ve kterém máme po ukončení odstavce informaci o počtu řádků zrovna ukončeného odstavce. Tento registr je na začátku odstavce vždy nastaven na nulu a po ukončení odstavce se do něj přičítá počet nově vytvořených řádků. Pokud změníme v odstavcovém módu hodnotu tohoto registru třeba na dvojku, bude při formátování odstavce ignorován první a druhý údaj z \parshape a teprve třetí údaj ovlivní první řádek odstavce. Stačí tedy upravit naše makro \doshape následovně:

```
\newcount\globpar
\def\doshape{\globpar=0 \def\par{\ifhmode\shapepar\fi}}
\def\shapepar{\prevgraf=\globpar \parshape\shapenum\shapelist
   \endgraf \globpar=\prevgraf
   \ifnum \prevgraf>\shapenum \let\par=\endgraf \fi}
```

Na konci každého odstavce je pomocí makra \par (alias \shapepar) nastavena nejprve velikost \prevgraf na hodnotu součtu řádků z předchozích odstavců (\globpar) a dále je zopakováno \parshape. Odstavec se v místě \endgraf naformátuje podle údajů v \parshape počínaje údajem číslo \prevgraf + 1. Dále se přičte k \prevgraf počet nově vytvořených řádků. Abychom o tento důležitý údaj při zahájení nového odstavce nepřišli, ukládáme jej do \globpar. Pokud už je v sazbě více řádků než údajů v \prevgraf, vracíme sekvenci \par původní primitivní význam. To znamená, že další odstavce už nebudou makrem \oblom ovlivněny.

V makru jsme neřešili problém samotného vložení obrázku. Obrázek můžeme vložit vzhledem ke spodní hraně odstavce, ve kterém bylo naposledy použito \parshape. Všechny požadované rozměry jsme schopni v makru vypočítat. Chyba ale může nastat v případě, kdy nám tato spodní hrana posledního odstavce „uteče“ na další stránku. Důkladnější by bylo navrhnout makro, které umožní uživateli při sazbě pevného počtu řádků na každé stránce deklarovat prostřednictvím pojmů „číslo strany“, „počet řádků shora na straně“ všechna místa pro oblomení všech použitých obrázků. Makro by mohlo pracovat podobně jako makro z naší ukázky. Navíc by spolupracovala výstupní rutina, která by do prázdných míst vkládala obrázky.

V TEXovských archivech jsou k mání makra, která umožní uživateli zadat tvar odstavce příjemnějším způsobem než přímo prostřednictvím parametrů \parshape. Například vložením souřadnic několika bodů křivky. Takové makro už ukazovat nebudeme. Pokud má čtenář chuť a čas, určitě si takovou věc udělá jako cvičení. □

6.6. Stránkový zlom

Každý element, který je vložen do vertikálního seznamu v hlavním vertikálním módu, prochází dvěma paměťovými oblastmi. Po svém zařazení do seznamu se ocitá v *přípravné oblasti* (originální termín: recent contributions) a odtud jej TEX přemisťuje do *aktuální strany* (current page). K přemístění materiálu z přípravné oblasti do aktuální strany dochází po dávkách pouze při vyvolání tzv. algoritmu *plnění strany* (page builder).

Algoritmus plnění strany vykonává tyto úkoly:

- Převádí materiál z přípravné oblasti do aktuální strany.
- Počítá tzv. *cenu zlomu* (cost) ve všech možných místech zlomu strany.
- Hledá v aktuální straně nejlepší místo zlomu (podle ceny zlomu).
- Najde-li nejlepší místo zlomu, spustí algoritmus *uzavření strany*.
- Nenajde-li ani po vyčerpání přípravné oblasti nejlepší místo zlomu, předává zpět řízení hlavnímu procesoru.

Algoritmus uzavření strany dělá následující činnost:

- Materiál v aktuální straně za místem zlomu přenáší zpět do přípravné oblasti.
- Ostatní materiál v aktuální straně kompletuje jako `\box255`.
- Obsah aktuální strany vyprázdní.
- Vyvolá výstupní rutinu (`\output`).
- Po ukončení výstupní rutiny znovu startuje algoritmus plnění strany.

Algoritmus plnění strany je vyvolán při činnosti hlavního procesoru za těchto okolností:

- Po vložení `\parskip` při zahájení odstavce.
- Po ukončení odstavce a vložení jeho řádků do vertikálního seznamu.
- Po vložení předchozích řádků odstavce při zahájení display módu.
- Po ukončení display módu a vložení výsledné rovnice do vertikálního seznamu.
- Po vstupu boxu (`\hbox`, `\vbox`, `\copy`, `\box`) do vertikálního seznamu.
- Po kompletaci tabulky `\halign`.
- Po vložení `\penalty` do vertikálního seznamu.

Ve všech uvedených případech se jedná o vertikální seznam hlavního (nikoli vnitřního) vertikálního módu. To je právě ten seznam, který je rozdělen na přípravnou oblast a aktuální stranu a podléhá stránkovému zlomu. □

Pojmy, které jsme zde zavedli, si ilustrujeme na jednoduchém příkladě. Představme si, že máme dokument obsahující dva odstavce. První odstavec se vejde na první

stranu a z druhého odstavce se tam vejde jen prvních několik řádků. Zbytek druhého odstavce je na druhé straně. Nakonec je dokument ukončen povelem `\end`. Probereme, jak takový dokument zpracovávají uvedené algoritmy.

Po sestavení prvního odstavce povelem `\par` se výsledný vertikální seznam zařadí do přípravné oblasti. Pak TEX inicializuje algoritmus plnění strany. Tento algoritmus převede materiál do aktuální strany a přiřadí k možným místům zlomu (v místech mezířádkových mezer) cenu zlomu. O této funkci si povíme později. Zatím algoritmus nenašel nejvhodnější místo zlomu, tudíž jeho činnost končí. Hlavní procesor pokračuje ve čtení vstupní fronty.

Při zahájení druhého odstavce se do přípravné oblasti vloží `\parskip` a znovu se vyvolá algoritmus plnění strany. Ten zařadí `\parskip` do aktuální strany a ukončí činnost.

Po sestavení druhého odstavce algoritmus plnění strany převádí z přípravné oblasti do aktuální strany řádky druhého odstavce. Až se v aktuální straně objeví řádky, ze kterých je jasné, že zlom za nimi by způsobil přetečení strany, zlom strany je stanoven v místě nejnižší ceny zlomu. TEX nyní vyvolá algoritmus uzavření strany. Ten nejprve přemístí zpět do přípravné oblasti materiál, který v aktuální straně zůstává za místem zlomu. Pak kompletuje zbylý materiál v aktuální straně do boxu 255. Nyní se vyvolá výstupní rutina. Ta (obvykle) po připojení čísla strany provede primitivní povel `\shipout`, kterým se `\box255` (i s případnými dalšími náležitostmi) zapíše jako první strana do `dvi`.

Po činnosti výstupní rutiny TEX znovu vyvolá algoritmus plnění strany. V rámci něj se převede z přípravné oblasti do aktuální strany zbylý materiál. Zatím nebylo nalezeno nejvhodnější místo zlomu. Proto algoritmus plnění strany svou činnost končí a hlavní procesor čte poslední povel ze vstupní fronty.

Tímto povelem je `\end`, který vkládá do přípravné oblasti `\vfill \penalty`-2^{30} (přesněji viz část B). Tím se vyvolá algoritmus plnění strany. Jakmile se penalta přesunula do aktuální strany, TEX zjistí, že to je nejvhodnější místo pro provedení stránkového zlomu (díky té značně záporné hodnotě penalty). Proto stranu kompletuje jako `\box255` a vyvolá výstupní rutinu. Ta vytiskne do `dvi` druhou stranu. Pak se znovu vyvolá algoritmus plnění strany. Ten ale nemá co dělat, protože je přípravná oblast prázdná. TEX v této situaci ukončí svou činnost. □

Přípravná oblast vždy zachovává pořadí elementů tak, jak byly postupně vytvořeny v hlavním vertikálním módu. Proto jsou do nové aktuální strany nejprve zavedeny ty elementy, které byly odloženy do přípravné oblasti při uzavírání předchozí strany. Potom teprve vstupuje z přípravné oblasti do aktuální strany případný další materiál, který tam čekal už před činností algoritmu uzavření strany. □

• **Algoritmus plnění strany.** Při postupném plnění aktuální strany TEX pracuje s těmito primitivními registry:

- `\pagetotal` — přirozená výška materiálu v aktuální straně.
- `\pagegoal` — cílová výška strany (obvykle rovna `\vsize`).

TEX v této souvislosti používá ještě další primitivní registry. Vesměs všechny začínají slovem `\page...`, viz část B.

Nejprve je aktuální strana prázdná. V tomto stavu TEX při převádění elementů z přípravné oblasti do aktuální strany ignoruje všechny odstranitelné elementy (tj. ⟨*glue*⟩, kern nebo penalta). Tyto elementy tedy nenávratně mizí. Důsledek: na začátku strany není nikdy odstranitelný element (mezera, penalta) jako první.

Jakmile vstupuje do aktuální strany první box nebo linka, TEX nastaví registr `\pagegoal=\vsize` a `\pagetotal=0pt`. Před první box nebo linku je do aktuální strany vložena mezera počítaná podle `\topskip`. Od této chvíle už není při přesunu materiálu z přípravné oblasti do aktuální strany nic ignorováno.

Přidávaný materiál do aktuální strany mění hodnotu `\pagetotal`. Tento registr v každém okamžiku odpovídá přirozené výšce teoretického `\vbox`u, v němž je uložen veškerý materiál z aktuální strany. Měněny jsou i další registry s názvem `\page...` □

Nyní uvedeme výpočet *ceny zlomu*. Víme, že tato hodnota je přidělena všem možným místům zlomu v aktuální straně. Všechna možná místa zlomu ve vertikálním seznamu jsou uvedena v sekci 6.1. Stručně řečeno, jedná se o všechny penalty s hodnotou menší než 10 000, dále některá ⟨*glue*⟩ a výjimečně kerny.

TEX v každém možném místě zlomu strany počítá hodnotu badness boxu (označujeme ji písmenem b), stejně jako při:

```
137 \vbox to\pagegoal{⟨aktuální strana po počítané místo zlomu mimo něj⟩}
```

Při vyhodnocení místa zlomu v ⟨*glue*⟩ nebo kernu má *cena zlomu* (označujeme ji písmenem c) tuto hodnotu:

$$c = b.$$

Nezapomeňme, že funkce b má maximum 10 000 a dále je při větším podtečení boxu konstantní. Při přetečení boxu je $b = \infty$. I tyto hodnoty přebírá funkce c.

Je-li místo zlomu určeno nenulovou penaltou $p \in (-10\,000, 10\,000)$ a současně je $b < 10\,000$, počítá se cena zlomu jako součet:

$$c = b + p.$$

Dále je c zvětšeno o hodnotu `\insertpenalties` při `\insertpenalties` $< 10\,000$ a $b < 10\,000$. Toho si zatím nevšímejme a předpokládejme `\insertpenalties=0`, což je obvyklá hodnota. O tomto registru se zmíníme až v následující sekci.

Je-li hodnota penalty $p \geq 10\,000$, cena zlomu se vůbec nepočítá, protože se nejedná o možné místo stránkového zlomu. Je-li naopak $p \leq -10\,000$, provede se v tomto místě stránkový zlom vždy, když není $c = \infty$. □

Algoritmus plnění strany postupně přesunuje elementy do aktuální strany a přiřazuje k možným místům zlomu cenu zlomu. Zatím nejeví snahu hledat definitivní místo zlomu. Tuto snahu projeví až v okamžiku, kdy poprvé vyšla $c = \infty$, nebo při $p \leq -10\,000$.

Při $c = \infty$ TEX určí definitivní polohu stránkového zlomu v místě s nejmenší cenou zlomu na aktuální straně. Je-li takových míst více, TEX volí poslední z nich. Všimneme si, že pokud je $c = \infty$ hned v prvním možném místě zlomu na straně, provede se zlom zde. To je jediný případ, kdy výsledný box se stranou vychází jako přetečený. Daleko obvyklejší je podtečený box.

Při $p \leq -10\,000$ se definitivním místem zlomu stává tato penalta, pokud ale není v tomto místě $c = \infty$. □

Vyzkoušejte si zapnout `\tracingpages=1`. V souboru `log` najdete na řádcích začínajících procentem přesnou informaci o průběhu cenové funkce (`c`) v kontextu s badness (`b`), se zaplněním strany v počítaném místě `\pagetotal` (`t`) a s cílovou výškou strany `\pagegoal` (`g`). Je to přehledné a instruktivní. Hvězdička u symbolů `b` a `c` označuje hodnotu ∞.

Probereme si typický průběh cenové funkce. Skoro za všemi řádky je $c = 10\,000$, protože je taková i badness b (podtečený box). V takovém případě se nepřičítají hodnoty penalt. Cenová funkce je tedy konstantní. Dále za posledními zhruba čtyřmi řádky bude už $b < 10\,000$. Box bude podtečen už jen „málo“. Při nulových penaltách zde klesá cenová funkce od hodnoty 10 000 k nule rychlostí podobné funkci $-x^3$ (viz výpočet funkce badness). Pak začnou pracovat případné hodnoty stažení a funkce nám zase roste až k hodnotě $b = 100$. V úseku, kde je $b < 10\,000$, může být průběh cenové funkce narušen vloženými penaltami, které mohou pouze zde ovlivnit definitivní podobu stránkového zlomu. Pak přichází řádek, za nímž už je $b = c = \infty$. TEX tedy určí polohu zlomu v místě s nejmenší cenou zlomu.

Uvedený případ odpovídá nastavení pružných výplňků mezi odstavci `\parskip` z plainu. Většinou ale chceme mít pevnou řádkovou osnovu a pak pružnost v těchto výplňcích zrušíme. Potom je cenová funkce konstantní (má hodnotu 10 000) až do

doby, kdy přijde místo zlomu, ve kterém je už strana přeplněna a je $c = \infty$. Definitivním místem zlomu je tedy poslední místo zlomu s $c = 10\,000$. Vidíme, že v tomto případě penalty s hodnotou $p \in (-10\,000, 10\,000)$ nemají v algoritmu žádný vliv. Pokud bychom chtěli ovlivnit výpočet i prostřednictvím takových penalt, nastavme pružné `\topskip` tak, aby v určitém počtu posledních řádků na straně bylo už $b < 10\,000$ a tam mohly promluvit i tyto penalty. Je ovšem otázka, zda je použití takových penalt v případě požadavku na pevnou řádkovou osnovu vůbec smysluplné. □

Může se stát, že po vyvolání algoritmu plnění strany máme `\pagetotal` větší než `\pagegoal`, ale algoritmus uzavření strany ještě nebyl vyvolán. Uvažujme tento příklad:

```
\vbox to\vsize{} % zcela zaplníme první stranu
\hbox{a}  % zde je vyvoláno plnění strany; strana je přeplněna
\message{Total:\the\pagetotal, Goal:\the\pagegoal}% Total > Goal
\hbox{b}  % zde se znovu volá plnění strany, uzavření 1. strany
```

Za boxem „a“ je strana už přeplněna, ale výstupní rutina ještě není vyvolána. Hlášení `\message` nám potvrdí, že je `\pagetotal` větší než `\pagegoal`. Proč tomu tak je? Algoritmus plnění strany neobdržel z přípravné oblasti ještě žádný element, který by byl možným místem zlomu, v němž by bylo vyhodnoceno $c = \infty$. Algoritmus tedy ještě nemá důvod hledat nejlepší místo zlomu. Teprve v mezeře před boxem `\hbox{b}` je vyhodnoceno $c = \infty$ a stránkový zlom se najde v mezeře před boxem `\hbox{a}`. □

• **Příklady.** Uvedeme nyní příklad, v němž budeme při návrhu typografie knihy počítat parametry určující stránkový zlom. Budeme dodržovat tzv. *řádkový rejstřík*. To znamená, že všechny řádky mají na stránce pevné místo v řádkové osnově a všechny vertikální rozměry se počítají na řádky. Musíme proto nulovat hodnoty stažení a natažení všech pružných výplňků ve vertikálním seznamu. Proti nastavení z plainu je nejdůležitější změnit: `\parskip=0pt`.

Nechť má strana n řádků velikosti `\baselineskip`. Jak bude velké `\vsize`? Například pro $n = 40$ pišme:

```
\vsize=39\baselineskip \advance\vsize by\topskip
```

Velikost `\vsize` je tedy $(n-1)$`\baselineskip` + `\topskip`. Hloubka posledního řádku se totiž do `\vsize` nezapočítává a výška prvního řádku je rovna přirozené velikosti z `\topskip`.

Představme si, že chceme dodržet řádkový rejstřík a současně chceme potlačit výskyt všech *parchantů*. To znamená, že se stránkový zlom nesmí provést za prvním řádkem ani před posledním řádkem víceřádkového odstavce. Jak to zařídit,

abychom měli i při potlačení parchantů na každé straně zcela stejný počet řádků, ukážeme až na straně 258. Obecně to není jednoduché. Proto v tomto příkladě dovolíme, aby strany mohly výjimečně mít $n+1$ nebo $n-1$ řádků. Řekneme si to podrobněji: Pokud při n řádcích vychází na následující stranu nahoru poslední řádek odstavce (parchant), nechť má stávající strana $n+1$ řádků (tj. řádek je z následující strany přesunut dolů na stranu ke zbytku odstavce). Pokud při n řádcích končí strana prvním řádkem dalšího odstavce, nechť má taková strana $n-1$ řádků (tj. řádek je přesunut na další stranu ke zbytku odstavce).

Volme pružné `\topskip` tak, aby algoritmus stránkového zlomu měl kolem přesného počtu n řádků ještě vůli plus minus jeden řádek. Pišme:

```
143 \topskip=10pt plus 1.5\baselineskip minus 1.5\baselineskip
```

Nyní je za n-tým řádkem badness nulová a za řádkem s číslem $n\pm1$ je badness rovna 30. Zkuste si to spočítat podle vzorečku pro hodnotu badness ze strany 99. Dále stačí volit:

```
144 \widowpenalty=10000  % nezlom před posledním řádkem odstavce
145 \clubpenalty =10000  % nezlom za prvním řádkem odstavce
146 \interlinepenalty=10 % lépe mezi odstavci než uvnitř odstavce
```

Uvnitř odstavce bude za řádky $n-1$, n, $n+1$ a $n+2$ postupně tato cena zlomu: 40, 10, 40, ∞. K hodnotě badness se zde přičítá `\interlinepenalty`, která je rovna deseti. Zlom se provede v místě $c=10$ a strana bude mít n řádků. Je-li na straně řádek $n+1$ posledním řádkem odstavce, bude cena zlomu za řádky $n-1$, n, $n+1$ a $n+2$ rovna postupně: 40, $\times$, 30, ∞. Křížek označuje místo, kde se cena zlomu vůbec nevyhodnocuje, protože se při penaltě 10 000 nejedná o možné místo zlomu. Nejlepší cena zlomu je v tomto případě pro $n+1$ řádků na straně. Je-li konečně n-tý řádek prvním řádkem nového odstavce, dostáváme cenu zlomu postupně: 30, $\times$, 40, ∞. Zlom se tedy provede za řádkem $n-1$.

Nyní se ještě musíme postarat o to, aby se při konečné sazbě strany neprojevila pružnost z `\topskip` nahoře, ale aby se případně upravila mezera dole mezi posledním řádkem strany a patou strany. To zařídíme ve výstupní rutině a více se o tom dozvíme v sekci 6.8. Problém vyřešíme předefinováním sekvencí `\pagebody` a `\makefootline` z výstupní rutiny plainu:

```
147 \def\pagebody{\vbox to\vsize{\unvbox255\vss}}
148 \def\makefootline{\baselineskip=30pt\line{\the\footline}}
```

Mezera `\vss` za materiálem z boxu 255 potlačí pružnost z `\topskip`. Dále se musíme postarat, aby pata strany byla od n-tého řádku na straně vzdálena o více než jeden řádek. Tím dostaneme místo pro případný řádek $n+1$. Zatímco plain nastavuje

v \makefootline hodnotu \baselineskip na 24 pt, my ji zde nastavíme na 30 pt. Podrobněji o výstupní rutině plainu viz stranu 262. □

Nakonec si ukážeme makro, které přepíná mezi sazbou na plnou šířku stránky a sazbou do sloupců. Makro je v této knize použito pro tabulky v sekci 5.4 a dále pro sazbu rejstříku. Uvidíme, že za jistých omezujících předpokladů nebude potřeba zasahovat do výstupní rutiny. Uživatel napíše třeba:

```
Tady je sazba na plnou šířku strany...
\begmulti 3
Tady je sazba do tří sloupců...
\endmulti
Tady je znovu sazba na plnou šířku strany...
```

Celá sazba mezi \begmulti a \endmulti je uzavřena do skupiny. Šířka sloupce \hsize je lokálně zmenšena tak, aby se mezi sloupce vešla mezera velikosti \colsep.

```
\newdimen\colsep  \colsep=2em % horiz. mezera mezi sloupci
\newcount\tempnum             % pracovní proměnná
\splittopskip=\baselineskip
\def\roundtolines #1{%% zaokrouhlí na celé násobky vel. řádku
   \divide #1 by\baselineskip \multiply #1 by\baselineskip}
\def\corrsize #1{%% #1 := #1 + \splittopskip - \topskip
   \advance #1 by \splittopskip \advance #1 by-\topskip}

\def\begmulti #1 {\par\bigskip\penalty0 \def\Ncols{#1}
   \setbox0=\vbox\bgroup\penalty0
   %% \hsize := šířka sloupce = (\hsize+\colsep) / n - \colsep
   \advance\hsize by\colsep
   \divide\hsize by\Ncols  \advance\hsize by-\colsep}
\def\endmulti{\vfil\egroup \setbox1=\vsplit0 to0pt
   %% \dimen1 := velikost zbylého místa na stránce
   \ifdim\pagegoal=\maxdimen \dimen1=\vsize \corrsize{\dimen1}
   \else \dimen1=\pagegoal \advance\dimen1 by-\pagetotal \fi
   \ifdim \dimen1<2\baselineskip
     \vfil\break \dimen1=\vsize \corrsize{\dimen1} \fi
   %% \dimen0 := výška n sloupcové sazby po rozdělení do sloupců
   %% = (\ht0 + (n-1)\baselineskip) / n, zaokruhleno na řádky
   \dimen0=\Ncols\baselineskip \advance\dimen0 by-\baselineskip
   \advance\dimen0 by \ht0 \divide\dimen0 by\Ncols
   \roundtolines{\dimen0}
   %% Rozdělit sazbu n sloupců do stránek nebo nerozdělit ?
   \ifdim \dimen0>\dimen1 \splitpart
   \else \makecolumns{\dimen0} \fi
   \ifvoid0 \else \errmessage{ztracený text ve sloupcích?} \fi
```

```
182     \bigskip}
183  \def\makecolumns#1{\setbox1=\hbox{}\tempnum=0
184     \loop \ifnum\Ncols>\tempnum
185        \setbox1=\hbox{\unhbox1 \vsplit0 to#1 \hss}
186        \advance\tempnum by1
187     \repeat
188     \hbox{}\nobreak\vskip-\splittopskip \nointerlineskip
189     \line{\unhbox1\unskip}}
190  \def\splitpart{\roundtolines{\dimen1}
191     \makecolumns{\dimen1} \advance\dimen0 by-\dimen1
192     %% \dimen0 := výška _zbylé_ n sloupcové sazby
193     %% \dimen1 := prázdné místo na stránce = (cca) \vsize
194     \vfil\break
195     \dimen1=\vsize \corrsize{\dimen1}
196     %% Rozdělit zbylou sazbu n sloupců do více stránek ?
197     \ifvoid0 \else
198        \ifdim \dimen0>\dimen1 \splitpart
199        \else \makecolumns{\dimen0} \fi \fi}
```

Makro `\begmulti` nejprve vloží do hlavního vertikálního seznamu mezeru oddělující právě ukončenou sazbu od sazby do sloupců. Následující `\penalty0` je důležitá, protože vyvolá algoritmus plnění strany a my můžeme dále v makru číst registr `\pagetotal`. Dále se uloží požadovaný počet sloupců do `\Ncols` a do boxu 0 se uloží obsah sazby mezi `\begmulti` a `\endmulti`. V tomto boxu je zmenšeno `\hsize`.

Makro `\endmulti` připojí na konec boxu 0 `\vfil`. Pomocí `\vsplit0 to0pt` se materiál boxu 0 odlomí v počáteční `\penalty0`, což způsobí, že se před první řádek v boxu 0 vloží mezera podle `\splittopskip`. O to nám v této operaci šlo. Dále uložíme do `\dimen1` velikost zbylého místa na stránce a do `\dimen0` celkovou výšku sloupcové sazby po jejím rozdělení do n sloupců. Tento údaj počítáme zaokrouhlen na řádky. Předpokládáme, že celá sloupcová sazba přísně dodržuje řádkový rejstřík a dovolí provést stránkový zlom mezi každým řádkem. Proto bude možno rozlomit tuto sazbu přesně v místech, která odpovídají výšce sloupce `\dimen0`. To je též důvod, proč jsme volili `\splittopskip` = `\baselineskip` a provedli úpravu materiálu boxu 0 pomocí `\vsplit0 to0pt`.

Na řádku 179 se ptáme, zda rozdělíme sloupcovou sazbu do více stránek či nikoli. Pokud ji nerozdělíme, provede se jednoduše makro `\makecolumns`, které rozdělí box 0 do jednotlivých boxů o výšce `\dimen0` pomocí primitivu `\vsplit` a výsledné sloupečky klade postupně vedle sebe do boxu 1. Toto makro po nalomení všech sloupečků usadí výsledný box 1 do vnějšího vertikálního seznamu. Má trochu starosti usadit jej do seznamu správně. Proto nejprve vloží do seznamu prázdný box, který bude mít účaří na řádkové osnově. Pak posune sázecí bod o `\splittopskip` nahoru a potlačí meziřádkovou mezeru použitím makra `\nointerlineskip`. Nyní

vloží box 1. Tím je dosaženo, aby účaří prvních řádků jednotlivých sloupců byla ve správné výšce.

Pokud se má rozdělit sloupcová sazba do více stránek, pracuje makro `\splitpart`, které zaokrouhlí zbylé místo na stránce na celý násobek řádků a podle tohoto údaje nechá vytvořit sloupečky makrem `\makecolumns`. Dále zmenšíme `\dimen0` o výšku právě vytvořené sloupcové sazby. Pak ukončíme stránku pomocí `\vfil\break`. Na nové stránce máme zbylé místo rovné velikosti `\dimen1=\vsize`. Pro účely sloupcové sazby, která má nastaveno větší `\splittopskip` než `\topskip`, potřebujeme provést korekci hodnoty `\dimen1` pomocným makrem `\corrsize`. Nyní opakujeme otázku, zda se do zbylého místa na stránce vejde zbylá sloupcová sazba. Pokud ne, voláme rekurzivně makro `\splitpart`.

Rekurzivní volání makra `\splitpart` může při větším množství stránek s vícesloupcovou sazbou zahltit paměť TEXu (viz konstantu *stack_size* v sekci 7.2). To se dá odstranit pomocí obratu s `\next`, které má význam `\splitpart` nebo jiný a je uveden na konci těla definice makra `\splitpart`. Podstatnější ale je, že celá sloupcová sazba se musí uložit do boxu 0, což u rozsáhlého díla může působit velké nároky na hlavní paměť TEXu (viz *mem_max* v sekci 7.2). Pokud tedy přepínáme do sloupcové sazby na delší dobu než zhruba desítky stránek, je nutno přistoupit k úpravě výstupní rutiny (viz stranu 271 v sekci 6.9). □

6.7. Plovoucí objekty typu `\insert`

Kdyby nebylo poznámek pod čarou (`\footnote`), byl by výklad algoritmu o stránkovém zlomu z předchozí sekce definitivní. Ovšem není tomu tak. Nejprve načrtneme ideu, co by měl TEX s poznámkami pod čarou dělat na primitivní úrovni. Pak teprve ukážeme, jak jsou tyto věci skutečně implementovány.

Idea 1. TEX narazí na poznámku pod čarou většinou v odstavcovém módu. Měl by umět (například pomocí makra) vložit do horizontálního seznamu značku pro čtenáře — viditelný odkaz na poznámku z textu. Dále by měl sestavit vertikální seznam s obsahem poznámky. Tento vertikální seznam by měl podržet ve zvláštní paměti a do horizontálního seznamu by měl vložit jen bezrozměrný vnitřní odkaz na toto místo v paměti.

Idea 2. Jakmile TEX přemisťuje vnitřní odkaz na poznámku z přípravné oblasti do aktuální strany, měl by zmenšit odpovídajícím způsobem `\pagegoal`, aby po teoretickém připojení poznámky pod stranu měla strana celkový rozměr zachován.

Idea 3. Při přemisťování vnitřního odkazu poznámky do aktuální strany by měl TEX v separátní části paměti „přelévat“ i vlastní text poznámky. Další poznámky na stejné straně by měl připojovat k textu už stávajících poznámek.

Idea 4. Pokud dojde při přelévání víceřádkového textu poznámky k situaci, kdy se celá poznámka společně se stávajícím textem strany už do zrcadla strany nevejde, měl by TEX umět převést jen část poznámky a zbytek pozdržet v „čekacím stavu“ pro novou stranu.

Idea 5. Při kompletování \box255 pro výstupní rutinu by měl TEX kompletovat i separátní část paměti s textem poznámek do smluveného boxu. Jedná se samozřejmě jen o texty těch poznámek, které se skutečně na stranu vešly. Pak výstupní rutina (řízená makrojazykem TEXu) může obsahovat zhruba takový algoritmus: Je-li smluvený box prázdný, sází se strana bez poznámek pod čarou. Není-li prázdný, připojí se pod \box255 vhodná mezera, dále vhodná čára, podle které se poznámky jmenují „pod čarou“, a konečně obsah smluveného boxu s textem poznámek. □

Pro splnění vytčeného cíle byl do TEXu implementován datový typ *insert*. Toto slovo nebudeme překládat a dokonce si dovolíme k němu připojovat české koncovky podle vzoru hrad: insert bez insertu. Pod tímto pojmem budeme označovat bezrozměrný odkaz na vertikální seznam uložený v separátní paměti (viz ideu 1).

Vložení odkazu typu insert do vytvářeného seznamu je realizováno primitivem \insert takto:

200 \insert⟨*number*⟩{⟨*vertikální materiál*⟩}

kde ⟨*number*⟩ určuje tzv. *třídu insertu*, o které budeme hovořit za chvíli. Dále ⟨*vertikální materiál*⟩ obsahuje povely pro sestavení vertikálního seznamu, podobně jako například ve \vboxu. Je-li tam zahájen odstavcový mód, je po dosažení koncové závorky „}“ tento mód uzavřen a je sestaven odstavec. Výsledek sazby ⟨*vertikálního materiálu*⟩ TEX ukládá do separátní paměti ve formě vertikálního seznamu.

V okamžiku činnosti algoritmu plnění strany musí být insert uložen jako samostatný element v přípravné oblasti. Je-li „ukryt“ uvnitř boxu, TEX jej ignoruje. Do přípravné oblasti se insert může dostat třemi způsoby: (1) Přímo, po povelu \insert v hlavním vertikálním módu. (2) Z horizontálního seznamu po povelu \par, kdy insert přechází bezprostředně za řádek, ve kterém byl dosud. (3) Po provedení \unvbox nebo \unhbox, byl-li insert v použitém boxu.

Každý insert má svou *třídu*, což je celé číslo v intervalu $\langle 0, 255)$. Tato třída například určuje číslo boxu, ve kterém bude předán převedený vertikální seznam výstupní rutině při uzavírání strany. Kdybychom měli v dokumentu jen poznámky pod čarou, vystačili bychom si s jedinou třídou (volili bychom třeba číslo 254). Texty poznámek bychom pak „odchytávali“ ve výstupní rutině v boxu 254 (viz ideu 5). Jenomže neplavou po stránkách jen poznámky pod čarou, ale též obrázky, tabulky, apod. Uživatel používá například makro plainu \midinsert nebo LATEXovské prostředí `figure`. Pro tyto plovoucí objekty vyčlení programátor makra další třídy.

Třída insertu n se váže na tyto registry TEXu:

- `\box`n — tam najde výstupní rutina výsledek vložený do aktuální strany.
- `\dimen`n — maximální povolená velikost výsledného boxu.
- `\count`n — určuje násobící koeficient při zmenšování `\pagegoal`.
- `\skip`n — mezera při vložení prvního insertu na stranu.

Pokud tedy programátor maker rozhodne rezervovat pro poznámky pod čarou třídu 254, pak vloží do `\count254`, `\dimen254` a `\skip254` příslušné informace, které ovlivní výpočet stránkového zlomu při zařazování insertů této třídy do aktuální strany. K významům jednotlivých údajů se nyní vrátíme podrobněji.

Registr `\dimen`n určuje maximální výšku kompletovaného boxu n s textem jednotlivých insertů dané třídy. Například nechceme, aby byla stránka vyplněna z poloviny jen poznámkami pod čarou. Raději ji vyplníme jen ze čtvrtiny a zbylé poznámky „odložíme“ do další strany. V takovém případě nastavujeme `\dimen`n`=0.25\vsize`.

Registr `\count`n se obvykle nastavuje na 1000, což určuje koeficient $f = 1$. Koeficient $f =$ `\count`$n/1000$ určuje tuto vlastnost TEXu: Je-li převeden do aktuální strany insert, pak `\pagegoal` bude zmenšen o celkovou výšku jeho obsahu pronásobenou f. Srovnejte ideu 2.

Příklady: U poznámek pod čarou v jednosloupcové sazbě volíme $f = 1$, zatímco u dvousloupcové sazby, kdy výstupní rutina má za úkol rozlomit obdržený box 255 do dvou sloupců a pod druhý připojit poznámky pod čarou, volíme $f = 0{,}5$. Konečně při implementaci okrajových poznámek, které nijak neovlivní vzhled sazby základního textu na straně, volíme $f = 0$.

Registr `\skip`n určuje „velikost mezery před první poznámkou na straně“. Přesněji: pokud je do aktuální strany zařazen insert jako první insert třídy n, je `\pagegoal` zmenšena o přirozenou výšku `\skip`n a teprve potom je tento registr zmenšen podle koeficientu z `\count`n. Programátor maker tím naznačuje, že výstupní rutina připojí tuto mezeru k celkové výšce strany právě tehdy, když bude `\box`n neprázdný. Tím zůstane zachována celková výška strany. □

Nyní budeme sledovat, jakými cestami se ubírá insert třídy n a jeho obsah daný vertikálním materiálem, na který insert ukazuje. Cesta začíná provedením povelu `\insert` a končí ve výstupní rutině, která si vyzvedne tento vertikální materiál v boxu n. Celé tajemství se odehrává při převodu značky insert z přípravné oblasti do aktuální strany. Víme, že algoritmus pro kompletování strany může některý materiál (za místem zlomu) vracet zpět do přípravné oblasti, takže putování insertů může být dosti strastiplné (jako Odysseovy cesty) a může se zdát komplikované. Jednotlivé situace popíšeme relativně samostatně jako pravidlo 1 až pravidlo 9.

Označení. Víme, že TEX rezervuje pro obsah budoucí strany, a tedy budoucího boxu 255, speciální místo v paměti, které nazýváme aktuální stranou. Analogicky TEX rezervuje pro každou třídu insertů n místo v paměti, které se stane při uzavření strany boxem n. Označme toto místo znakem b_n. Dále označme znakem n_i místo v paměti pro vertikální seznam, který TEX rezervuje pro obsah jednoho konkrétního insertu třídy n po povelu \insert. Tam tedy obsah insertu svou cestu začíná.

Pravidlo 1. Převádí-li TEX z přípravné oblasti do aktuální strany insert třídy n s obsahem n_i, provede nejprve tento test: Je-li b_n prázdné, zmenší \pagegoal o přirozenou velikost \skip$n$ a přidá hodnoty stažení a roztažení do celkového součtu strany (registry \pagestretch, \pageshrink a případně \pagefil(ll)stretch). Dále se snaží převést n_i do b_n pokud možno celé. To se nemusí podařit, viz pravidlo 2.

Pravidlo 2. TEX při zařazování insertu do aktuální strany testuje, zda se jeho obsah n_i celý vejde do b_n. Existují dvě potenciální překážky: (1) Po přidání n_i do b_n je celková výška b_n větší než \dimen$n$. (2) Nechť h je teoretická výška n_i počítaná jako celková výška n_i pronásobená koeficientem f z \count$n$. Pokud se materiál s teoretickou výškou h nevejde do zbytku strany, přesněji pokud

$$h > \texttt{\pagegoal} - \texttt{\pagetotal} - \texttt{\pagedepth} + \texttt{\pageshrink}$$

pak nastává druhá překážka.

Pravidlo 3. Nenastala-li žádná z překážek podle pravidla 2, TEX zmenší registr \pagegoal o hodnotu h, přemístí materiál n_i do místa b_n a insert označí příznakem *hotovo*.

Pravidlo 4. Existuje-li překážka podle pravidla 2, TEX musí materiál n_i rozdělit do dvou částí. První část vloží do b_n a druhou ponechá v čekacím stavu v n_i. Rozdělení do částí provede pomocí operace

\vsplit ⟨*materiál* n_i⟩ to ⟨*dimen*⟩

kde ⟨*materiál* n_i⟩ je výchozí obsah paměti n_i. K němu je před operací připojena na konec materiálu \penalty-10000, aby existovalo aspoň jedno možné místo zlomu. Velikost ⟨*dimen*⟩ je maximální povolená celková výška, pro kterou nenastává podle pravidla 2 ani překážka (1), ani (2). U překážky (2) se přitom při výpočtu výšky h nebere v úvahu \pageshrink.

Výsledek operace \vsplit obsahuje „ulomený“ materiál z n_i (viz primitiv \vsplit v části B). Výchozí obsah n_i je zmenšen o tento „ulomený“ materiál a následující odstranitelné elementy. Nahoru je pak do n_i přidána před první box nebo linku mezera podle \splittopskip.

„Ulomený“ materiál je převeden do b_n a `\pagegoal` je zmenšena o jeho celkovou výšku násobenou f. K tomu dojde i v případě, že se výsledek zlomu do strany nevejde (přetečený box při `\vsplit`). Dále TEX přičte k `\insertpenalties` penaltu v místě zlomu původního seznamu n_i. Tato hodnota dále ovlivní výpočet ceny zlomu c.

Ukazatel na obsah insertu se zdvojí. Jeden ukazatel ukazuje na „ulomený“ materiál do b_n a dostane příznak *hotovo*, zatímco druhý ukazuje na „zbylý“ materiál do n_i a dostane příznak *čekej*. Pokud některý z těchto ukazatelů ukazuje na prázdný materiál, je takový ukazatel zrušen.

Pravidlo 5. Po vložení ulomené části insertu podle pravidla 4 se paměť b_n zablokuje a další inserty se do ní nepustí. Je-li tedy převáděn do aktuální strany nový insert stejné třídy n obsahu n_i, a přitom b_n je zablokováno, TEX pouze přičte k registru `\insertpenalties` hodnotu `\floatingpenalty`. Takový insert dostává příznak *čekej*.

Pravidlo 6. Registr `\insertpenalties` je při zahájení zpracování každé strany nulován. Je-li jeho hodnota (případně změněná v pravidlech 4 a 5) menší než 10 000, je cena zlomu c počítána jako

$$c = b + p + \texttt{\insertpenalties}, \quad \text{pro } b < 10\,000,$$

Je-li ale `\insertpenalties` $\geq 10\,000$, je rovnou nastaveno $c = \infty$.

Pravidlo 7. Při uzavření strany se TEX nejprve zaměří na inserty, které jsou v aktuální straně až za místem zlomu a musí se tedy společně s dalším materiálem za místem zlomu vrátit do přípravné oblasti. Pokud jsou mezi nimi inserty s příznakem *hotovo*, mají smůlu. Zdaleka nejsou hotovy. Jejich obsah je odpojen od b_n a vrací se do n_i. Pak TEX převede zpět veškerý materiál za stránkovým zlomem včetně případných insertů do přípravné oblasti.

Pravidlo 8. Dále se TEX při uzavření strany zaměří na inserty, které jsou zahrnuty do stávající strany (před místem zlomu). Pokud jsou mezi nimi některé s příznakem *čekej*, jsou rovněž převedeny do přípravné oblasti. Při otevření nové strany budou znovu vstupovat do (vyprázdněné) aktuální strany jako první ještě před materiálem, který byl do přípravné oblasti přesunut z aktuální strany podle pravidla 7.

Pravidlo 9. Po ukončení práce podle pravidel 7 a 8 TEX kompletuje box 255, do něhož vloží zbylý obsah aktuální strany. Dále do boxů n vloží obsahy b_n a výsledek předá výstupní rutině. Existuje výjimka z tohoto pravidla při kladném `\holdinginserts` (viz část B). □

• **Příklady.** Ilustrujme popsaný algoritmus na příkladech. Začneme třídou pro poznámky pod čarou v plainu. Makro `\footnote` je definováno takto:

```
\newcount\interfootnotelinepenalty \interfootnotelinepenalty=100
\newinsert\footins
\skip\footins=\bigskipamount % mezera mezi textem a poznámkami
\count\footins=1000 % koeficient pro výpočet výšky zůstává 1
\dimen\footins=8in  % max. množství poznámek na jedné stránce
\def\footnote#1{\let\@sf\empty
  \ifhmode\edef\@sf{\spacefactor\the\spacefactor}\/\fi
  #1\@sf\vfootnote{#1}}
\def\vfootnote#1{\insert\footins\bgroup
  \interlinepenalty=\interfootnotelinepenalty
                % penalta mezi řádky u víceřádkové poznámky
  \splittopskip=\ht\strutbox  % určuje mezeru nahoře při \vsplit
  \splitmaxdepth=\dp\strutbox % určuje max. hloubku pro \vsplit
  \floatingpenalty=20000 % dáno k \insertpenalties v pravidle 5
  \leftskip=0pt \rightskip=0pt   % toto nastavení může být
  \spaceskip=0pt \xspaceskip=0pt %    momentálně jiné
  \textindent{#1}\footstrut\futurelet\next\fo@t}
\def\fo@t{\ifcat\bgroup\noexpand\next \let\next\f@@t
  \else\let\next\f@t\fi \next}
\def\f@@t{\bgroup\aftergroup\@foot\let\next}
\def\f@t#1{#1\@foot}
\def\@foot{\strut\egroup}
\def\footstrut{\vbox to\splittopskip{}}
```

Pomocí deklaračního makra \newinsert byla poznámkám přidělena nějaká třída. Její číslo budeme dále označovat jako \footins. Prakticky je \footins=254 (viz makro \newinsert v části B).

Samotné makro \footnote řeší pouze sazbu odkazu v textu. Zapamatuje si hodnotu \spacefactor, potom vysází značku #1 a za ní nastaví původní hodnotu \spacefactor.

Makro \vfootnote už značku nesází, ale otevírá činnost primitivu \insert. V rámci ⟨*vertikálního materiálu*⟩ povelu \insert je nejprve nastaveno několik registrů. V této souvislosti je potřeba zdůraznit, že hodnoty \splittopskip, \splitmaxdepth a \floatingpenalty si TEX pamatuje z doby, kdy je proveden povel \insert. Teprve později (v pravidlech 4 a 5) tyto hodnoty použije. Tyto hodnoty jsou tedy vázány na konkrétní insert.

Pomocí \textindent je nakonec vysázena značka poznámky (do prostoru odstavcové zarážky). Dále se testuje, zda uživatel napsal při použití makra \footnote závorku „{“, která otevírá vlastní text poznámky. Na konec textu poznámky je pak připojen \strut, aby dvě poznámky nad sebou nebyly na sebe nalepené. TEX totiž při postupném plnění oblasti b_n nevkládá mezířádkové mezery mezi obsahy jednotlivých insertů.

Shrneme to. Použití makra \footnote{⟨*značka*⟩}{⟨*text poznámky*⟩} expanduje na

```
⟨sazba značky se zachováním \spacefactor⟩
\insert\footins{⟨nastavení registrů⟩
  ⟨značka v místě odstavcové zarážky⟩\strut⟨text poznámky⟩\strut}
```

Další věci týkající se sazby poznámky se odehrávají ve výstupní rutině. Ukážeme tu část výstupní rutiny, která se věnuje poznámkám.

```
\def\pagecontents{\ifvoid\topins\else\unvbox\topins\fi
  \dimen0=\dp255 \unvbox255 % Zde tiskneme vlastní text strany
  \ifvoid\footins\else      % jsou přítomny poznámky pod čarou?
    \vskip\skip\footins     % mezera mezi textem a poznámkami
    \footnoterule           % čára, podle které se to jmenuje
    \unvbox\footins \fi     % vložení obsahu poznámek
  \ifr@ggedbottom \kern-\dimen0 \vfil \fi}
\def\footnoterule{\kern-3pt
  \hrule width 2truein \kern 2.6pt } % \hrule je vysoká .4pt
```

Zde vidíme makro \pagecontents, které je ve výstupní rutině použito pro sazbu těla strany. Na řádku 227 je realizovaná sazba plovoucích obrázků a tabulek. To je další třída insertů v plainu (\topins), které se budeme věnovat za chvíli.

Po sazbě vlastního textu strany z boxu 255 se ptáme, zda máme připojit poznámky pod čarou (tj. zda box \footins je neprázdný). Pokud ano, vložíme nad poznámky mezeru, jak bylo domluveno v registru \skip\footins. Dále nakreslíme čáru \footnoterule a konečně vložíme box \footins s textem poznámek. □

Nyní zkusíme TEX potrápit na experimentální sazbě poznámek pod čarou v tomto příkladě:

```
\tracingpages=1 \tracingoutput=1
\vsize=34pt  % tři řádky: 10pt(\topskip) + 2*12pt(\baselineskip)
\def\\{\hfil\break}
První \\  Druhý \\
Třetí\footnote{$^1$}{poznámka} řádek\footnote{$^2$}{dva\\řádky}
\end
```

Je to velmi nepravděpodobný případ: strana obsahuje prostor jen pro tři řádky a ve třetím řádku jsou odkazy na dvě poznámky pod čarou. První poznámka je jednořádková a druhá dvouřádková. A to je celý dokument. Na tomto příkladě si podrobně ilustrujeme putování insertů podle uvedených pravidel 1 až 9.

Po vyzkoušení příkladu zjistíme, že první strana zůstala podtečená se dvěma řádky, druhá je přetečená. Obsahuje třetí řádek s odkazy na poznámky a pod ním obě dvě

poznámky. To jsou dohromady tři řádky plus mezera nad poznámkami, která nám způsobí přetečení. Druhá poznámka není celá, její druhý řádek pokračuje na (jinak prázdné) třetí straně.

Označíme insert 1 jako první poznámku a insert 2 druhou. Insert 1 má svůj obsah v n_1 a insert 2 v n_2. Po kompletování odstavce máme v přípravné oblasti tři řádky. Za třetím řádkem jsou bezprostředně připojeny insert 1 a insert 2. Tyto inserty „migrovaly“ po sestavení odstavce ze třetího řádku do vertikálního seznamu.

První strana. V mezeře za prvním řádkem je $c = 10\,000$ a za druhým taky. Nyní přichází do aktuální strany třetí řádek. Máme tedy \pagetotal=\pagegoal=34 pt. Ovšem bezprostředně za těmito řádky přicházejí inserty. Protože mezi třetím řádkem se značkami pro čtenáře a vlastními inserty není možné místo zlomu, nemůže poznámka skončit na jiné straně, než její odkaz.

První insert se podle pravidla 2 na stranu nevejde a je rozdělen podle pravidla 4. Nejprve je ale podle pravidla 1 zmenšen \pagegoal o 12 pt, což je hodnota \skip\footins. Dosadíme-li nyní do vzorečku podle pravidla 2, máme

$$h > 24\,pt - 34\,pt - 0\,pt = -12\,pt$$

Registr \pagedepth má momentálně hodnotu 0 pt, protože text: „Třetí[1] řádek[2]“ nemá žádnou hloubku. Protože je $f = 1$, provede TEX s insertem 1 podle pravidla 4 \vsplit⟨n_1⟩ to -12pt. Protože není nad textem poznámky žádný možný bod zlomu, bude výsledkem operace přetečený box výšky 12 pt (výška podpěry v poznámce). Zlom se provedl až v penaltě $-10\,000$ připojované na konec materiálu automaticky. Výsledný box má výšku 12 pt a o tuto výšku se zmenší \pagegoal. Nyní je tedy už \pagegoal = 10 pt. V souboru log o této události čteme:

```
% split254 to -12.0,12.0 p=-10000
```

První údaj za „to“ je požadovaná velikost, na kterou „byla snaha“ materiál zlomit a druhý údaj (12pt) je výsledná velikost, na kterou se zlom nakonec „podařil“.

Druhý insert už podle pravidla 5 zůstává v aktuální straně s příznakem „čekej“. Registr \insertpenalties se po prvním insertu podle pravidla 4 zvedl na hodnotu $-10\,000$, protože taková byla penalta při \vsplit. Nyní přičítáme hodnotu 20 000 podle pravidla 5, takže registr \insertpenalties má nakonec hodnotu 10 000.

Za třetím řádkem přichází materiál z povelu \end (viz část B, heslo \end). Nejprve prázdný box, dále \vfill. V této mezeře je cena zlomu $c = \infty$, protože je malá hodnota \pagegoal a došlo by k přeplnění boxu. Zlom je tedy realizován před třetím řádkem a třetí řádek i oba inserty putují zpět do přípravné oblasti. První z nich putuje podle pravidla 7, tj. ačkoli má příznak *hotovo*, je jeho obsah odpojen od b_n a vrácen do n_1.

Ve výstupní rutině dojde k podtečení boxu strany, protože strana má jen dva řádky, ale `\vsize` si žádá tři.

Druhá strana. Do aktuální strany vstoupí znovu třetí řádek a za ním insert 1. Ten zmenší `\pagegoal` o 12 pt podle pravidla 1 a celý se do strany vejde podle pravidla 3. Zbývající místo na straně má velikost 0 pt. Skutečně, v tento okamžik máme ve straně jeden řádek, jednu mezeru ve velikosti řádku a jednu poznámku obsazující poslední řádek.

Druhý insert se do strany nevejde a bude tedy rozlomen podle pravidla 4. V souboru `log` o tom čteme:

```
% split254 to 0.0,8.5 p=400
```

Tento insert má ve svém obsahu dva řádky a vidíme, že se „odlomil“ první řádek. Ukazatel dostane podle pravidla 4 příznak *hotovo* a vzniká další ukazatel s příznakem *čekej*, který ukazuje na zbytek odlomeného textu.

Dále přichází prázdný box a `\vfill` z povelu `\end`. V mezeře `\vfill` je sice cena zlomu $c = \infty$, ale přesto se zde zlom provede. Je to z toho důvodu, že je to první možné místo zlomu na celé straně. Výsledkem je strana, obsahující třetí řádek, první poznámku a první část druhé poznámky. Druhá část druhé poznámky má příznak *čekej*, a proto se vrací do přípravné oblasti.

Výstupní rutina vytvoří druhou stranu a na terminálu vidíme hlášení: `Overfull \vbox (4.5pt too high) has occurred while \output is active`.

Třetí strana. Do aktuální strany se znovu dostává insert s druhou částí poznámky. Nyní se tam vejde podle pravidla 3. Za ním TEX ignoruje zbylou `\penalty`-2^{30}. Je to totiž odstranitelný element, a přitom ještě v aktuální straně není žádný box nebo linka. Pak se TEX podívá znovu do čtecí fronty, kde se podruhé uplatní povel `\end`. Ten do přípravné oblasti vloží prázdný box následovaný `\vfill \penalty`-2^{30}. Zlom se provede v závěrečné penaltě. Tím vzniká poslední strana a všechny paměťové oblasti se vyprázdnily. TEX tedy při třetím načtení povelu `\end` skutečně končí svou činnost. □

Plain pracuje ještě s jednou třídou insertů: `\topins`. Tato třída je rezervována na větší obrázky nebo tabulky, které se nevejdou do textu a budou proto přesunuty na další stranu. Pro uživatele jsou připraveny tři řídicí sekvence `\topinsert`, `\pageinsert` a `\midinsert`. Používají se takto:

```
\topinsert ⟨vertikální materiál⟩ \endinsert
\pageinsert ⟨vertikální materiál⟩ \endinsert
\midinsert ⟨vertikální materiál⟩ \endinsert
```

Makro \topinsert přesune ⟨*vertikální materiál*⟩ do horní části stávající strany, pokud se tam vejde. Tj. pokud místo, kde byl \topinsert použit, neuteče na následující stranu. Jinak je ⟨*vertikální materiál*⟩ přesunut do horní části příští strany. \pageinsert se chová jako \topinsert, jenom s tím rozdílem, že příští strana je celá rezervována pro přesunutý materiál. Konečně makro \midinsert vloží ⟨*vertikální materiál*⟩ do místa, kde je makro \midinsert uvedeno, tj. nic nepřemisťuje. To ovšem platí pouze tehdy, když se tam tento materiál celý vejde (tj. po jeho vložení do strany se nepřekročí \pagegoal). Jinak se \midinsert chová jako \topinsert, takže materiál bude nahoře na příští straně.

Makra jsou implementována takto:

```
\newinsert\topins
\skip\topins=0pt   % do strany není přidávána žádná mezera
\count\topins=1000 % f = 1
\dimen\topins=\maxdimen % není omezeno množství insertů
\newif\ifp@ge \newif\if@mid
\def\topinsert{\@midfalse\p@gefalse\@ins}
\def\midinsert{\@midtrue\@ins}
\def\pageinsert{\@midfalse\p@getrue\@ins}
\def\@ins{\par\begingroup\setbox0=\vbox\bgroup} % start a \vbox
\def\endinsert{\egroup % finish the \vbox
  \if@mid \dimen0=\ht0
    \advance\dimen0 by\dp0 \advance\dimen0 by12pt
    \advance\dimen0 by\pagetotal \advance\dimen0 by-\pageshrink
    \ifdim\dimen0>\pagegoal\@midfalse\p@gefalse\fi\fi
  \if@mid \bigskip\box0\bigbreak
  \else\insert\topins{\penalty100 % floating insertion
    \splittopskip=0pt \splitmaxdepth=\maxdimen \floatingpenalty=0
    \ifp@ge \dimen0=\dp0
       \vbox to\vsize{\unvbox0\kern-\dimen0}% depth is zero
    \else \box0\nobreak\bigskip\fi}\fi\endgroup}
```

Přepínač \ifp@ge má hodnotu „true“, pokud má insert obsadit celou následující stranu. Dále \if@mid říká, že se v makru máme pokusit vložit ⟨*vertikální materiál*⟩ tam, kde je, a teprve pokud se to nezdaří, použijeme vlastnosti insertů.

Na řádcích 253 a 254 vidíme, že veškerý ⟨*vertikální materiál*⟩ je vložen do boxu 0 a teprve pak se rozhoduje, co s ním. Při \if@mid změříme box 0 a teoreticky jej přidáme do strany. Prakticky tedy dáme do \dimen0 celkovou výšku boxu, přidáme \pagetotal určující stávající zaplnění strany, ubereme \pageshrink obsahující stažitelnost stávající strany a ptáme se, zda výsledek je menší než \pagegoal. Pokud ano, pak na řádku 259 materiál do strany skutečně vložíme.

Ve všech ostatních případech použijeme primitiv `\insert`. Při `\ifp@ge` (rezervovat celou stranu) vkládáme do insertu box výšky `\vsize`, takže se na příští stranu určitě nic jiného nevejde. Jinak vkládáme `\box0`, který od dalšího textu oddělíme mezerou `\bigskip`.

Proč ⟨*vertikální materiál*⟩ z tohoto insertu může utéci na příští stranu? Pokud se nevejde na stávající stranu, pak je podle pravidla 4 proveden `\vsplit`. Ten ale odlomí materiál v místě `\penalty100` (viz řádek 260). Odlomená část neobsahuje nic a zbylá část dostává příznak *čekej*. Jak s těmito inserty třídy `\topins` naloží výstupní rutina, jsme už ukázali na straně 252, řádek 227. □

Další ukázku použití insertů — pro poznámky na okraji — odložíme do sekce 6.9, kde už budeme vědět více o výstupní rutině. □

6.8. Výstupní rutina

Výstupní rutina je posloupnost tokenů, které jsou uloženy v registru `\output`. V sekci 6.6 bylo řečeno, že výstupní rutinu vyvolává algoritmus pro uzavření strany. Toto „vyvolání“ fakticky znamená, že TEX otevře skupinu, v rámci níž budou přiřazení lokální. Dále expanduje obsah proměnné `\output` a výsledek zpracuje v hlavním procesoru obvyklým způsobem. Nakonec uzavře skupinu a ukončí tím činnost výstupní rutiny. Řízení je znovu předáno algoritmu plnění strany.

Zatímco do algoritmů plnění strany a uzavření strany nemůže programátor maker nikterak vstoupit, výstupní rutina mu to vynahradí. Zde programátor pomocí maker TEXu navrhuje konečný vzhled jednotlivých stran dokumentu.

Výstupní rutina získává od algoritmu pro uzavření strany tyto informace

- Obsah kompletované strany (`\box255`).
- Obsahy insertů vložených do této strany (`\box`n).
- Polohy značek `\mark` v sestavené straně (`\firstmark`, `\topmark`, `\botmark`).
- Hodnota penalty, ve které byl proveden stránkový zlom (`\outputpenalty`).

Výstupní rutina může používat následující výstupy

- Výstup strany do `dvi` pomocí `\shipout` (to bývá obvyklé).
- Výstup materiálu zpět do přípravné oblasti (to nebývá obvyklé).
- Ukládání informací do registrů, jak je obvyklé i při běžném zpracování. □

Velmi jednoduchá výstupní rutina by mohla vypadat takto:

```
\output={\shipout\vbox{%
    \line{\strut\rm Alfa-Omega\hfil\the\pageno}
    \hrule \bigskip \box255}
    \global\advance\pageno by1}
```

V této ukázce výstupní rutina řeší dva úkoly. (1) Do `dvi` ukládá stranu jako `\vbox` obsahující záhlaví s textem „Alfa-Omega“ (což by mohla být třeba značka nakladatele) a po pravé straně je číslo strany `\pageno`. Pod záhlavím je linka `\hrule`, která je k záhlaví připojena na hloubku podpěry `\strut`. Pak následuje vertikální mezera `\bigskip` a pod ní obsah strany `\box255`, jak ji připravil algoritmus uzavření strany. Všimneme si, že pro sazbu záhlaví je explicitně použit přepínač fontu `\rm`. Nejsme si totiž nikdy jisti, za jaké situace bude výstupní rutina vyvolána, tj. zda vždy bude aktuální font sazby nastaven na `\rm`.

(2) Výstupní rutina zvedne číslo strany `\pageno` o jedničku, aby i následující strana byla očíslována odpovídajícím číslem. Plain nastavuje registr `\pageno` alias `\count0` na jedničku, takže první strana bude mít číslo 1 a další strany budou mít takové číslo, jaké připraví výstupní rutina v předchozím běhu. Zvednutí čísla o jedničku provedeme globálně, protože se celá výstupní rutina odehrává uvnitř skupiny. □

V dalším textu se zaměříme podrobněji na rozbor jednotlivých vstupních a výstupních informací, se kterými výstupní rutina pracuje.

• **Stav boxu 255.** Výška boxu odpovídá hodnotě `\pagegoal` v místě zlomu. Pružné mezi řádkové výplňky jsou nataženy na tuto výšku, pokud to jde. Jinak je spodní hrana materiálu v boxu jinde než účaří samotného boxu (třeba při podtečení strany v případě, že neexistuje žádná pružná mezera v materiálu). Důležité je, že při přípravě takového boxu algoritmus sestavení strany nepodá žádnou zprávu o úspěšnosti či neúspěšnosti sestavení. Pokud se tedy chceme dozvědět, jak úspěšný byl stránkový zlom, musíme kompletování boxu 255 ve výstupní rutině zopakovat. Třeba takto:

```
\vbox to\ht255{\unvbox255}
```

Nyní se při hodnotě badness větší než `\vbadness` nebo při přeplnění objeví zpráva `Underfull/Overfull \vbox has occured while Output is active`.

Chceme-li změřit přirozenou výšku materiálu v boxu 255, pišme:

```
\setbox255=\vbox{\unvbox255} \dimen0=\ht255
```

Pomocí boxu 255 jsme tedy schopni zjistit `\pagegoal` i `\pagetotal` pro dané místo zlomu. Proč se na tyto registry nemůžeme ve výstupní rutině ptát přímo pomocí řídicích sekvencí `\pagegoal` a `\pagetotal`? Tyto registry totiž mají v době činnosti výstupní rutiny hodnoty, které dosáhly třeba po zahrnutí většího množství

materiálu do aktuální strany. Nezapomeňme, že materiál za místem zlomu je vrácen zpět do přípravné oblasti. Například pokud je v tomto nepoužitém materiálu insert, který zmenšuje `\pagegoal`, pak tento registr obsahuje hodnotu, která neodpovídá skutečně provedenému místu zlomu. □

Zpětné měření boxu ve výstupní rutině můžeme využít například při sázení knížky, na které nám velmi záleží. Tam většinou chceme jednak dodržet řádkový rejstřík a jednak zakážeme pomocí `\widowpenalty` a `\clubpenalty` parchanty. Dále bychom si přáli, aby na každé straně byl stejný počet řádků. Je zřejmé, že to je příliš mnoho požadavků najednou a pravděpodobně se napoprvé nepodaří všechny splnit. Volme velmi malou pružnost v `\topskip`, žádnou pružnost v dalších vertikálních mezerách a ve výstupní rutině pišme `\vbox to\ht255{\unvbox255}`. Pak se projeví všechny strany, kde chybí z důvodu potlačení parchantů poslední řádek, jako „Underfull“. Na terminálu okamžitě vidíme, o které strany se jedná. Dále můžeme pomocí `\looseness` experimentovat s jinými variantami sazby některých odstavců, abychom nežádoucí „Underfull“ potlačili. Pracujeme odpředu díla postupně dozadu. Je to pracné, musíme dílo opakovaně TEXovat a výsledek práce je použitelný pro konkrétní volbu zrcadla strany. Bohužel, jiné řešení při tak náročných požadavcích asi není v TEXu možné. □

• **Značky na stranách pro plovoucí záhlaví.** Výstupní rutina může pracovat s primitivními sekvencemi `\topmark`, `\firstmark` a `\botmark`, které se chovají jako makra bez parametrů expandující na určité posloupnosti tokenů. Makra pro sazbu kapitol, záhlaví hesel ve slovnících apod. mohou pracovat s primitivem `\mark`. Tento primitiv se používá takto:

```
271  \mark{⟨obsah značky⟩}
```

Primitiv vloží do sestavovaného seznamu tiskového materiálu bezrozměrnou značku s obsahem ⟨*obsah značky*⟩. Tento ⟨*obsah značky*⟩ je posloupnost tokenů a je v okamžiku činnosti primitivu `\mark` expandován, jako by se jednalo o `\edef`. Je-li `\mark` použit v odstavcovém módu, přesouvá se značka do vertikálního módu až při kompletaci odstavce, podobně jako to dělá insert a `\vadjust`.

Po kompletaci boxu 255 mohou sekvence `\topmark`, `\firstmark` a `\botmark` expandovat na ⟨*obsah značky*⟩ podle tohoto pravidla:

- `\topmark` odpovídá poslední značce z předchozího boxu 255.
- `\firstmark` se váže na první značku v sestaveném boxu 255.
- `\botmark` odkazuje na poslední značku v sestaveném boxu 255.
- Ostatní značky v boxu 255 mezi `\firstmark` a `\botmark` jsou nepřístupné.

Pokud není v boxu 255 použita žádná značka, zůstávají registry `\firstmark` a `\botmark` u svých původních hodnot. To není příliš přesně řečeno, takže ještě jednou a přesněji:

Při startu TEXu mají \topmark, \firstmark a \botmark prázdný obsah (expandují podobně jako makro \empty). V algoritmu uzavření strany se pak v rámci kompletování boxu 255 odehrávají tyto činnosti: (1) Obsah \botmark se přesune do \topmark. (2) Není-li v boxu 255 žádná značka, registry \firstmark a \botmark zůstávají u své původní hodnoty. Jinak \firstmark bude expandovat na ⟨*obsah značky*⟩, která se v boxu 255 vyskytuje jako první, a \botmark na ⟨*obsah značky*⟩, která se v boxu 255 vyskytuje jako poslední. Pokud je v boxu 255 jediná značka, bude \firstmark i \botmark expandovat na totéž. (3) Uvedená přiřazení provádí TEX globálně. □

Ukážeme si nyní, k čemu je to dobré. V této knížce je použito plovoucí záhlaví. Na každé liché straně je v záhlaví název aktuální sekce. Rozebereme si makra, která se o to starají. Nejprve uvedeme zjednodušenou variantu sazby názvu sekce pomocí makra \sub:

```
\def\sub #1 \par{\par\bigskip
   \global\advance\secnum by1
   \noindent{\secfonts \the\chapnum.\the\secnum. #1}\par\nobreak
   \mark{\the\chapnum.\the\secnum. #1}}
```

Makro zvětšuje číslo sekce o jedničku. Pak vysází název sekce včetně čísel ve zvětšeném fontu \secfonts a dále za \nobreak připojí \mark, takže ⟨*obsah značky*⟩ odpovídá názvu sekce včetně čísla. Zde využíváme toho, že se \the\secnum expanduje v okamžiku činnosti primitivu \mark.

Ve výstupní rutině budeme pro pravé záhlaví pracovat s makrem \pravezahlavi, které definujeme takto:

```
\def\headrule{\leaders\hrule height3pt depth-2.6pt\hfil}
\def\pravezahlavi{\headrule \kern3pt {\it \firstmark\/}}
```

Tato kniha využívá výstupní rutiny plainu, která do záhlaví expanduje obsah proměnné typu ⟨*tokens*⟩ s názvem \headline. Proto stačí psát:

```
\headline={\ifodd\pageno \pravezahlavi \else \levezahlavi \fi}
```

Zde se ptáme, zda je strana lichá. Pokud ano, použijeme \pravezahlavi, jinak sázíme \levezahlavi. Pravé záhlaví tiskne vedle pružné čáry text z \firstmark, takže tam budeme mít název právě probírané sekce. Levé záhlaví tiskne název kapitoly. Makro pro zahájení nové kapitoly (\chap) a makro \levezahlavi vypadá takto:

```
\def\chap #1 \par{\vfill\break
   \global\advance\chapnum by1 \global\secnum=0
   {\chapfonts \the\chapnum. #1}\par
```

```
282     \xdef\chapmark{Kapitola \the\chapum. #1}
283     } % konec definice \chap
284 \def\levezahlavi{{\it\chapmark\/}\kern3pt \headrule}
```

Vidíme, že zde si podáváme informaci o textu kapitoly v makru `\chapmark`, které nijak nesouvisí se značkami pro plovoucí záhlaví. Můžeme si to dovolit, protože kapitola vždy zahajuje novou stranu (viz `\vfill\break`). Nikdy tedy nedojde k nežádoucímu posunu mezi okamžikem, kdy je definován `\chapmark` pomocí `\xdef`, a časem, kdy výstupní rutina sází záhlaví. Naopak sekce nezačínají vždy na nové straně. Proto je v případě sekcí využití značek pro plovoucí záhlaví nutné.

Na stránce, kde je zahájena kapitola, potřebujeme tisk záhlaví potlačit. Proto přidáme do makra `\chap` vynulování registru `\headline`, ovšem s příznakem, aby na další stránce už zase vše pracovalo.

```
285 \def\chap #1 \par{⟨řádky 279 až 282 ponecháme stejné⟩
286     \headline={\hfil\starthead}
287     } % konec definice \chap
288 \def\starthead{\global\headline=
289   {\ifodd\pageno \pravezahlavi \else \levezahlavi \fi}}
```

Při první činnosti výstupní rutiny na začátku kapitoly se `\headline` tiskne jako `\hfil` (prázdné záhlaví) a navíc se expanduje `\starthead`. Toto makro přepíše hodnotu `\headline` na obsah, který už známe z řádku 278. Přepsání je globální, protože se odehrává uvnitř výstupní rutiny. Příští výstupní rutina už tedy bude pracovat se „standardním“ obsahem `\headline` a záhlaví se objeví. □

Všimneme si, že pokud nová sekce začíná uprostřed strany, záhlaví na této straně referuje na novou sekci a ne na dobíhající sekci, jejíž text se na straně také vyskytuje. V některých aplikacích se může tato vlastnost jevit jako nežádoucí. Například v části B je pro čtenáře určitě užitečnější, pokud záhlaví referuje na dobíhající text hesla než na první nové heslo na straně. Bez převracení stránek pak čtenář ví, k jakému heslu patří text ze začátku strany. Jak takovou věc zařídíme? V záhlaví použijeme místo `\firstmark` primitiv `\topmark` a makro pro zahájení hesla musí dovolit odlomit svou značku `\mark` na předchozí stranu takto:

```
290 \def\heslo #1 {\par\mark{#1}\bigskip\noindent{\bf #1} }
```

Pokud heslo začíná zcela nahoře na straně, je proveden zlom v místě `\bigskip` a značka hesla zůstává na předchozí straně. Proto `\topmark` skutečně referuje na toto heslo. Pokud heslo dobíhá na začátku strany, je jeho značka rovněž na předchozí straně a tudíž bude `\topmark` referovat na dobíhající heslo. Následující heslo se do záhlaví na této straně nepropracuje. □

• **Výstupy výstupní rutiny.** Nejčastěji používá výstupní rutina pro výstup primitiv `\shipout`, který zapisuje výsledek práce TEXu jako jednu stranu do `dvi`. Někdy ale využijeme též zadní vrátka výstupní rutiny a vracíme materiál do přípravné oblasti. Právě tuto možnost rozebereme důkladněji.

Výstupní rutina pracuje ve vertikálním módu. Pokud se v ní použijí povely, které vytvářejí elementy vertikálního seznamu, je výsledný vertikální seznam po ukončení činnosti výstupní rutiny zařazen do přípravné oblasti. Odtud vstupuje v rámci algoritmu plnění strany do aktuální strany. Zajímá nás nyní pořadí, v jakém bude materiál z přípravné oblasti převáděn zpět do aktuální strany. Zde je odpověď:

- Inserty, vrácené do přípravné oblasti, protože měly status *čekej*.
- Vertikální materiál vyprodukovaný výstupní rutinou.
- Element zlomu strany.
- Materiál, který byl do přípravné oblasti vrácen za místem zlomu.
- Materiál, který v přípravné oblasti čeká a ještě nevstoupil do aktuální strany.

Toto pořadí jednotlivých částí přípravné oblasti je závazné a každá uvedená část může být za jistých okolností prázdná. Výjimku tvoří element zlomu strany, který je přítomen vždy. O něm si nyní povíme podrobněji.

Pokud je zlom strany proveden mimo penaltu, je element zlomu strany roven elementu, ve kterém byl proveden stránkový zlom. V takovém případě má registr `\outputpenalty` hodnotu 10 000. Pokud byl zlom proveden v penaltě, je její hodnota před vyvoláním výstupní rutiny uložena do `\outputpenalty` a element zlomu je roven `\penalty10000`.

Element zlomu strany je vždy odstranitelným elementem. Pokud tedy výstupní rutina nevytvoří žádný box nebo linku, bude poté v algoritmu plnění strany ignorován, protože nepředchází box nebo linka. To je nejběžnější situace. Proto jsme zatím ve výkladu algoritmů stránkového zlomu jeho výskyt v přípravné oblasti nezmiňovali. Pokud ale výstupní rutina vyprodukuje do vertikálního seznamu box nebo linku, není element zlomu strany ignorován a objevuje se znovu v aktuální straně. □

Napíšeme-li například:

```
291 \output={\unvbox255}
```

pak se po ukončení výstupní rutiny objeví veškerý materiál znovu v přípravné oblasti a znovu vstupuje do aktuální strany. Protože nebyly změněny žádné parametry sazby (například změna `\vsize`, dodatečná změna registru `\output`), dopadne stránkový zlom stejně jako prve a výstupní rutina znovu vrátí obsah strany do přípravné oblasti. Tušíme tedy, že jsme vytvořili nekonečnou smyčku a nepředvedli příliš praktický příklad. Z experimentování máme ale jeden pozitivní výsledek —

seznámíme se s registry `\deadcycles` a `\maxdeadcycles`. TEX nám totiž vynadá tímto způsobem:

```
! Output loop---25 consecutive dead cycles.
```

Jestliže při činnosti výstupní rutiny nebyl použit primitiv `\shipout`, TEX zpozorní, protože taková výstupní rutina není v pravém slova smyslu výstupní. Prakticky TEX v takovém případě přičte do registru `\deadcycles` jedničku. Původně má tento registr hodnotu nula a po každé výstupní rutině, která použila `\shipout`, se `\deadcycles` znovu nuluje. Jakmile registr `\deadcycles` dosáhne hodnoty `\maxdeadcycles`, TEX ohlásí uvedenou chybu. Plain nastavuje `\maxdeadcycles` na hodnotu 25. □

• **Rozbor výstupní rutiny plainu.** Tato rutina je definována následovně:

```
\newtoks\headline \headline={\hfil} % headline is normally blank
\newtoks\footline \footline={\hss\tenrm\folio\hss}
\def\advancepageno{\ifnum\pageno<0 \global\advance\pageno by-1
  \else\global\advance\pageno by1 \fi} % increase |pageno|
\def\folio{\ifnum\pageno<0 \romannumeral-\pageno
           \else\number\pageno \fi}

\output={\plainoutput}
\def\plainoutput{\shipout\vbox{\makeheadline
  \pagebody\makefootline}\advancepageno
  \ifnum\outputpenalty>-20000 \else\dosupereject\fi}
\def\pagebody{\vbox to\vsize{\boxmaxdepth=\maxdepth
  \pagecontents}}
\def\makeheadline{\vbox to0pt{\vskip-22.5pt
  \line{\vbox to8.5pt{}\the\headline}\vss}\nointerlineskip}
\def\makefootline{\baselineskip=24pt\line{\the\footline}}
\def\dosupereject{\ifnum\insertpenalties>0
  \line{}\kern-\topskip\nobreak\vfill\supereject\fi}
```

Proměnná typu ⟨*tokens*⟩ s názvem `\headline` je použita pro sazbu záhlaví (viz řádek 306) a proměnná `\footline` obsahuje sazbu paty strany. Hned u deklarace jsou proměnné naplněny výchozími hodnotami, tj. záhlaví strany je prázdné a v patě strany je stránková číslice `\folio` uprostřed. Například makro `\nopagenumbers` nastavuje obsah `\footline` na `\hfil`, takže po jeho použití bude prázdná i pata strany. Jiné vhodné nastavení paty strany může vypadat třeba takto:

```
\footline={\ifodd\pageno \hfill\fi \bf\folio \hfil}
```

Nyní bude stránková číslice sázena polotučným řezem (\bf) a na pravých stránkách bude vpravo (pracuje \hfill), zatímco na levých stránkách bude vlevo (pracuje jen \hfil).

Česky psané knihy mají prvních několik stránek nečíslovaných a pak čísla stránek pokračují, jako by první strany byly číslovány. Nejprve tedy použijeme \nopagenumbers a dále v místě, kde začíná číslování stránek, napíšeme buď přímo řádek 310, nebo použijeme makro \pagenumbers, které si definujeme například takto:

```
311 \def\pagenumbers{%
312     \footline={\ifodd\pageno \hfill\fi \bf\folio \hfil}}
```

V knihách vydávaných v Americe je zvykem číslovat i strany v úvodu knihy s obsahem, předmluvou, povídáním o tom, jak číst tuto knihu, apod. Ovšem tyto strany se číslují římskými číslicemi, zatímco vlastní text knihy se čísluje znova od jedničky arabskými číslicemi. Proto Knuth místo jednoduchého \the\pageno použil makro \folio. Toto makro (viz řádek 296) tiskne \pageno arabsky, je-li kladné. Římsky je sázeno $-$\pageno, je-li \pageno záporné. Na začátku knihy tedy stačí nastavit \pageno na číslo -1 a budeme mít číslování římské. Skutečně, makro \advancepageno, které je definováno na řádku 294 a použito na řádku 301, v případě záporného \pageno tento registr o jedničku snižuje.

Vlastní výstupní rutina plainu, neboli makro \plainoutput, řeší tři úkoly. Za prvé vysází obsah strany, za druhé pohne se stránkovou číslicí pomocí \advancepageno a za třetí při \outputpenalty $\leq -20\,000$ provede \dosupereject. O druhé činnosti jsme už mluvili, zaměříme se nyní na první a třetí.

Obsah strany je sázen pomocí primitivu \shipout a skládá se ze tří částí. První část (záhlaví) vyrobí \makeheadline, druhou část (vlastní text strany) vytvoří makro \pagebody a konečně poslední část (patu strany) připojí makro \makefootline. Při studiu \makeheadline zjistíme, že záhlaví je posunuto nahoru od referenčního bodu o 22,5 pt přičemž je podepřeno podpěrou vysokou 8,5 pt. Proto je účaří záhlaví nad referenčním bodem vzdáleno o 14 pt. Vzdálenost tohoto účaří od účaří prvního řádku textu strany vychází z důvodu \topskip $= 10$ pt na 24 pt, což tedy znamená, že je mezi záhlavím a prvním řádkem vynechán právě jeden dvanáctibodový řádek. Samozřejmě jen tehdy, je-li výška prvního řádku textu strany menší než 10 pt a výška záhlaví menší než 8,5 pt.

Při průzkumu makra \makefootline zjistíme, že mezi účařím boxu s obsahem strany (tj. účařím posledního řádku textu strany) a účařím paty strany je rovněž vzdálenost 24 pt, tedy je vynechán jeden řádek. Stane se tak díky lokálnímu nastavení \baselineskip na hodnotu 24 pt. Ovšem pozor! V tomto makru plainu je

jedna chybička, která se projeví při zapnutí \offinterlineskip. Pokud je výstupní rutina vyvolána v době, kdy je zapnuto \offinterlineskip, je nastaveno \lineskiplimit na \maxdimen a z toho důvodu nebude nikdy pracovat \baselineskip, ale vždy zabere \lineskip, které je nastaveno na nulu. Pak bude pata strany přilepena přímo k dolnímu okraji strany. Tuto chybu plainu zjistíme, pokud chceme sázet třeba tabulky s nastaveným \offinterlineskip přes více stran.

Vlastní text strany je zpracován v makru \pagecontents. Toto makro jsme ukázali v předchozí sekci na straně 252 na řádku 227. Zde je jednak použit box 255 a dále je nahoru připojen obsah insertu \topins, pokud je neprázdný. Dole je sestavena poznámka pod čarou, pokud je box \footins neprázdný.

Nakonec si povíme, co dělá ve výstupní rutině plainu makro \dosupereject. Toto makro spolupracuje s makrem \supereject, které většinou používáme v kontextu \vfill\supereject na ukončení kapitoly. Tento zápis expanduje na \vfill\penalty-20000. V uvedené penaltě se určitě provede stránkový zlom a výstupní rutina pak má v \outputpenalty číslo −20 000. Proto se ve výstupní rutině aktivuje makro \dosupereject, které vrací do přípravné oblasti za prázdným boxem znovu \supereject, tj. znovu penaltu −20 000. Tuto činnost provede ale jen tehdy, když zůstávají neuplatněné inserty, což pozná dotazem na \insertpenalties. Tím vzniká cyklus, ve kterém se postupně uplatní všechny dosud nevytištěné inserty. Pak teprve cyklus končí. K čemu to potřebujeme? Na konci kapitoly většinou chceme vysázet všechny obrázky a tabulky (insert třídy \topins), které se dosud nevešly do sazby a byly stále přesouvány na další strany. Nová kapitola je pak zahájena s čistým štítem.

Pokud na konci dokumentu nepoužijeme \supereject, ale pouze \end, nemusíme se bát, že zůstanou nějaké inserty nevysázeny. Povel \end je totiž vybaven analogickou schopností, jakou mezi kapitolami řeší makro \dosupereject. Z toho důvodu se přiznám, že moc dobře nechápu, proč je makro \bye definováno jako \par\vfill\supereject\end, když vlastně totéž dokáže samotný primitiv \end. □

6.9. Ukázky různých výstupních rutin

O výstupních rutinách, jak píše Knuth v TEXbooku, by se dala napsat celá kniha. Tuto vizi se podařilo naplnit panu Bechtolsheimovi [1], který celý čtvrtý díl své monografie věnoval výstupním rutinám. Přiznám se, že jsem toto dílo neviděl, a proto všechny zde uvedené ukázky jsou z mé vlastní dílny a nejsou (možná bohužel) nikým inspirovány.

Našim cílem nebude napsat o výstupních rutinách knihu, ale pouze v jedné sekci soustředit příklady z rozličných oblastí. Příklady nekladou nárok na univerzální použití (například mohou přestat fungovat poznámky pod čarou). Záměrem totiž

nebylo vytvářet univerzální obludy. Především jsem chtěl, aby kód jednotlivých příkladů byl dostatečně přehledný. Čtenář se tak dozví, jak věci fungují, a může si makra z ukázek dále upravit podle svých konkrétních požadavků. Myslím si, že to je cennější deviza, než nabídnout uživateli jednu univerzální výstupní rutinu, do které není pořádně vidět, a říci: „Vážený uživateli, nesnaž se to pochopit, pouze to používej“. Přesně tomuto (LaTeXovému) přístupu jsem se chtěl vyhnout vlastně v celé své knize. □

• **Sazba na praporek vpravo a vlevo.** V tomto odstavci vidíte sazbu na pravý praporek. Znamená to, že řádky jsou vlevo zarovnány pod sebe a vpravo nikoli. Takovou věc zařídíme makrem plainu `\raggedright`, které vkládá do `\rightskip` pružnou mezeru `\hfil`. Při sazbě knihy na praporek nám typografové doporučují, aby na pravých stránkách v otevřené knize byla sazba na pravý praporek a na levých stránkách na praporek levý.

Uvedený požadavek vlastně znamená, že se mezi pravým a levým praporkem přepíná i uprostřed odstavce — totiž v místě, kde je proveden stránkový zlom. K tomu nám už makro `\raggedright` nebude stačit. Teprve výstupní rutina ví naprosto přesně, zda se materiál soustředěný do boxu 255 objeví na liché nebo sudé stránce. Tato rutina bude mít tedy nový úkol. Pokud je sázena lichá (neboli pravá) strana, ponechá box 255 beze změny. Pokud je ale sázena sudá (neboli levá) strana, rozebere výstupní rutina box 255 na jednotlivé řádky, do každého z nich přidá zleva `\hfill` a zase box sestaví dohromady. Pak jej teprve vytiskne do `dvi`. Na levé stránce budeme mít skutečně levý praporek. Pokud navíc nastavíme v dokumentu `\raggedright`, budeme mít pravé stránky na pravý praporek. Levé strany zůstávají na levý praporek, protože dodatečně vložené `\hfill` je víc, než `\hfil` z makra `\raggedright`.

Nejprve uvedeme makro, které dokáže rozebrat box na řádky a do nich vložit zleva `\hfill`.

```
313 \def\toright#1{\setbox1=\vbox{\unvbox#1
314   \skip0=\lastskip \unskip \global\setbox#1=\vbox{\vskip\skip0}
315   \loop \setbox0=\lastbox \skip0=\lastskip \unskip
316      \advance\skip0 by\lastskip \unskip
317      \advance\skip0 by\lastskip \unskip \unpenalty
318      \ifhbox0  \setbox0=\hbox to\hsize{\hfill\unhbox0}
319         \global\setbox#1=\vbox{\vskip\skip0 \box0 \unvbox#1}
320      \repeat}%
321      \ifdim\ht1>0pt
322          \showboxbreadth=100 \showboxdepth=1 \showbox1 \fi}
```

V obvyklé sazbě vertikálního materiálu se za sebou postupně nalézá box, penalta, ⟨*glue*⟩, ⟨*glue*⟩ a znova box, penalta, ⟨*glue*⟩, ⟨*glue*⟩. Za boxem může být totiž penalta z `\widowpenalty` nebo z analogického registru. Dále může následovat ⟨*glue*⟩

z \parskip. Pak určitě následuje ⟨glue⟩ z \baselineskip nebo \lineskip. Potom teprve přichází další box. Někdy se dokonce sejdou za sebou tři výplňky typu ⟨glue⟩. První je vložena nějakým makrem (například za nadpisem), druhá přichází z \parskip a poslední z \baselineskip.

Při rozebírání boxu na řádky používáme \lastbox pro odebrání boxu, \lastskip pro zjištění hodnoty poslední ⟨glue⟩, \unskip pro odebrání poslední ⟨glue⟩ a konečně \unpenalty pro odebrání poslední penalty. Hodnoty ⟨glue⟩ musíme registrovat a při zpětném sestavení je mezi řádky znovu vracet. Hodnoty penalt můžeme s klidem zapomenout, protože penalty už svoji roli sehrály při hledání nejlepšího stránkového zlomu. Musíme ale tyto penalty odebírat (\unpenalty), abychom se propracovali k dalšímu boxu, který leží před takovou penaltou.

Z uvedeného plyne, že zezadu budeme postupně odebírat v pořadí \lastbox, \unskip, \unskip, \unskip, \unpenalty a tak pořád dokola až do vyprázdnění rozebíraného boxu. Pokud je v seznamu méně ⟨glue⟩ nebo chybí penalta, použitý primitiv se nezlobí. Pouze neudělá nic. Před provedením \unskip vždy přečteme hodnotu ⟨glue⟩ pomocí \lastskip a sčítáme ji v registru \skip0. Pak ji zpětně vložíme do výsledného boxu #1 jako \vskip\skip0.

Cyklus ukončíme testem na \ifhbox0. To je pravda právě tehdy, když se podařilo odebrat v těle cyklu řádek pomocí \lastbox. Pokud se to povedlo, pak vkládáme do řádku doleva \hfill a takto upravený řádek vracíme do boxu #1.

Nastavíme ještě \raggedright a poměrně jednoduše předefinujeme výstupní rutinu:

```
323 \output={\ifodd\pageno \else \toright{255} \fi \plainoutput}
324 \raggedright
```

V makru \toright je na řádku 321 použit jednoduchý test, zda se box 1 podařilo celý rozebrat (ptáme se na výšku boxu). Pokud se to nepodařilo, objeví se o tom zpráva na terminálu (díky \showbox). Navíc v \log u okamžitě vidíme, proč se box nepodařilo rozebrat. Většinou vadí značka z \write nebo \mark. První typ značky můžeme schovat do boxu a odebrat ji jako box, ovšem značku typu \mark nelze asi v této aplikaci použít.

Makra z tohoto příkladu se velmi hodí při sazbě poznámek na okraj. Poznámky na okraji vpravo by totiž měly být na pravý praporek a poznámky vlevo na levý. V takovém případě budeme ve výstupní rutině na levých stránkách transformovat box s okrajovými poznámkami. Můžeme k tomu použít makro \toright. K poznámkám na okraji uvedeme příklad v této sekci později. □

• **Vyrovnávání pravé strany podle zaplnění levé.** Na straně 242 jsme uvedli, jak potlačit výskyt parchantů. Pokud nepoužijeme řešení ze strany 258, ale zůstaneme u řešení ze strany 242, máme strany zaplněny různým počtem řádků. Většina stran má n řádků, ale některé mohou mít $n+1$ řádků a jiné zase $n-1$ řádků. Je nehezké, když na dvou protilehlých stranách v knize je různý počet řádků. Chtěli bychom, aby se počet řádků na protilehlých stranách lišil v rámci možností co nejméně. Znamená to, že pokud levá strana má $n+1$ řádků, pak pravá by měla mít implicitně také $n+1$ řádků a pouze při problémech s parchantem bude mít jen n řádků. V tomto případě vůbec nedovolíme, aby měla pravá strana jen $n-1$ řádků.

Právě popsaný požadavek může řešit výstupní rutina, pokud ji vybavíme jistou inteligencí.

```
\topskip=10pt plus 1.5\baselineskip minus 1.5\baselineskip
\vsize=39\baselineskip \advance\vsize by\topskip
\widowpenalty=10000  % nezlom před posledním řádkem odstavce
\clubpenalty =10000  % nezlom za prvním řádkem odstavce
\interlinepenalty=10 % lépe mezi odstavci než uvnitř odstavce
\newskip\oritopskip  \oritopskip=\topskip
\newskip\orivsize    \orivsize  =\vsize

\def\pagebody{\vbox to\orivsize{\unvbox255\vss}}
\def\makefootline{\baselineskip=30pt\line{\the\footline}}

\output={\ifodd\pageno \pravastrana \else \levastrana \fi
   \plainoutput}
\def\levastrana{\setbox255=\vbox{\unvbox255} % měříme výšku
   \dimen0=\ht255
   \advance \dimen0 by 0.5\baselineskip
   \ifdim\dimen0<\orivsize                          %% n-1 řádků
      \global\advance\vsize by-\baselineskip
      \global\topskip=10pt minus 1.5\baselineskip
   \else \advance\dimen0 by-\baselineskip
         \ifdim\dimen0>\orivsize                    %% n+1 řádků
            \global\advance\vsize by\baselineskip
            \global\topskip=10pt plus 1.5\baselineskip
         \fi        %% pro n řádků ponechám původní nastavení
   \fi}
\def\pravastrana{\global\vsize=\orivsize
   \global\topskip=\oritopskip}
```

V makru `\levastrana` zjišťujeme, na kolik řádků tato strana vyšla. Pokud vyšla na $n+1$ řádků, nastavíme pro sestavení následující strany `\vsize` o jeden řádek větší a navíc upravíme `\topskip`, aby nebylo možno takovou stranu přeplnit (rušíme část

„minus“). Pokud vyšla levá strana na $n-1$ řádků, pak zmenšujeme pro následující stranu \vsize o jedničku a z \topskip rušíme část „plus“.

Makro \pravastrana pouze vrací pro \vsize a \topskip původní nastavení. Proto bude následující levá strana sázena zase v rozmezí $n \pm 1$ řádků. Naše makro ještě trochu upravíme:

```
\def\pravastrana{\setbox255=\vbox{\unvbox255}
   \dimen0=\ht255
   \advance\dimen0 by 0.5\baselineskip
   \ifdim\dimen0<\vsize \hlaseni \fi
   \advance\dimen0 by -\baselineskip
   \ifdim\dimen0>\vsize \hlaseni \fi
   \global\vsize=\orivsize \global\topskip=\oritopskip}
\def\hlaseni{\message{strany mají různý počet řádků}}
```

Podle tohoto hlášení poznáme, že dvě protilehlé strany nemají stejný počet řádků. Můžeme tedy dále upravovat sazbu knihy tak, že přidáváme k některým odstavcům \looseness. Přitom opakujeme TEXování tak dlouho, až nebude v knize existovat jediná nežádoucí dvojice protilehlých stránek s různým počtem řádků. □

• **Tabulky přes více stránek.** Představme si, že pracujeme s velkými tabulkami, které se nám nevejdou na jedinou stranu. Chtěli bychom, aby se takové tabulky dokázaly „rozpadnout“ do více stran, a přitom na konci každé strany byla tabulka uzavřena vodorovnou linkou a na začátku každé nové strany byla znovu nahoře připojena hlavička tabulky. Použití našeho makra bude třeba takové:

```
\beglongtable
\halign{\strut\vrule\quad\hfil #\hfil\quad\vrule
      &&\quad\hfil #\hfil\quad\vrule\cr
 \bf Hlava 1 & \bf Hlava 2 & \bf Hlava 22 \cr
 \noalign{\penalty-10001}
⟨data⟩ & ⟨data⟩ & ⟨data⟩ \cr
⟨atd., obludné množství dat⟩ \cr}
\endlongtable
```

Celá tabulka je tedy vložena mezi sekvence \beglongtable a \endlongtable. Staráme se o případné svislé linky v tabulce (viz deklaraci řádku v \halign), ale nestaráme se o vodorovné linky. Místo toho za prvním boxem tabulky klademe smluvenou \penalty-10001, čímž dáváme najevo, že jsme v tabulce vytvořili box se záhlavím. Chceme, aby se záhlaví na začátku každé strany opakovalo. Je principiálně jedno, zda jsme tabulku vytvořili pomocí \halign nebo jakkoli jinak. Podstatné je jenom to, aby za prvním boxem v materiálu mezi \beglongtable a \endlongtable byla vložena smluvená penalta hodnoty −10 001.

Makro si nejprve vezme první box před smluvenou penaltou. Nakreslí nad něj a pod něj vodorovnou linku (vzniká záhlaví tabulky) a začne klást do strany další řádky tabulky. Pokud sazba dosáhne konce strany, makro nakreslí vodorovnou linku, kterou tabulku uzavře. Na následující straně zopakuje záhlaví tabulky: nakreslí tedy vodorovnou linku, zkopíruje box se záhlavím a pod něj vloží znovu vodorovnou linku. Pak pokračují další řádky tabulky. Pokud se znovu dosáhne konce strany, činnost se opakuje. Na úplném konci tabulky makro uzavře sazbu vodorovnou linkou a vloží `\bigskip`. Za `\endlongtable` pokračuje běžný text. Makro navíc celou tabulku na všech stranách centruje na vertikální osu podle `\hsize`.

Tím jsme si řekli, co by makro mělo umět. Nyní to skutečně uděláme. Samotné `\beglongtable` zahájí novou skupinu a v rámci ní předefinuje výstupní rutinu. Tato rutina po `\penalty-10001` odpojí z boxu 255 poslední řádek se záhlavím a uloží si ho do vyhrazeného boxu. Po dosažení konce strany bude tato rutina kreslit požadované linky a výsledné části tabulky (odpovídající postupně jednotlivým stranám) bude ukládat do rezervovaného boxu pod sebe. Mezi tyto části tabulky rutina vloží `\penalty0`. V místě `\endlongtable` ukončíme skupinu a vracíme se tedy k původní výstupní rutině, která se stará o záhlaví, patu strany, stránkové číslice a podobné legrácky. V tuto chvíli „vypustíme“ do hlavního vertikálního módu box s nastřádanými částmi tabulky. Zlom strany se najde v místech `\penalty0`.

Poté, co jsme načrtli ideu, ukážeme řešení a rozebereme některé podrobnosti:

```
\newbox\tabbox       % zde ukládáme hotové části tabulky
\newbox\headb        % zde je uloženo záhlaví tabulky
\newbox\pagebox      % pro zapamatování textu před tabulkou
\newdimen\orivsize   % pro uložení originální hodnoty \vsize
\newdimen\tabsize    % hodnota \vsize v režimu tvorby tabulky
\newdimen\dist       % posun doprava kvůli centrování tabulky

\def\beglongtable{\bigskip
  \ifinner \errmessage{Velká tab. musí být v hlavním módu}\fi
  \begingroup  \offinterlineskip  \orivsize=\vsize
  \topskip=0pt  \holdinginserts=1 \maxdeadcycles=1000
  \output={\globaldefs=1 \ifnum\outputpenalty=-10001 \savehead
                         \else \savetable \fi}}
\def\endlongtable{\penalty-10000 % naposledy speciální \output
  \global\vsize=\orivsize        % návrat k původní \vsize
  \endgroup  \unvbox\pagebox \unvbox\tabbox
  \bigskip}
\def\savehead{%
   \setbox\pagebox=\vbox{\unvbox255 \setbox\headb=\lastbox}
   \tabsize=\vsize  %% \tabsize := \vsize-(\ht+\dp)\headb-4.7pt
   \advance\tabsize by-\ht\headb \advance\tabsize by-\dp\headb
   \advance\tabsize by-4.7pt
```

```
390     \vsize=\pagegoal %% \vsize := \pagegoal - \pagetotal - 4.7pt
391     \advance\vsize by-\pagetotal \advance\vsize by-4.7pt
392     \ifdim\vsize<\bigskipamount \vsize=\tabsize \fi
393     \dist=\hsize     %% \dist := (\hsize - \wd\headb) / 2
394     \advance\dist by-\wd\headb \divide\dist by2 }
395   \def\savetable{\setbox\tabbox=\vbox{\unvbox\tabbox
396           \moveright\dist\vbox{%
397              \hrule\copy\headb\hrule\unvbox255\hrule}
398           \vfil\penalty0}
399     \vsize=\tabsize}
```

Nová výstupní rutina bude veškerá svá přiřazení později potřebovat znovu, proto tato přiřazení musí být globální. Abychom nemuseli pořád psát `\global`, nastavíme jednou pro vždy registr `\globaldefs` na pozitivní hodnotu (řádek 379). Toto nastavení samotné je přitom lokální, takže po ukončení skupiny výstupní rutiny budou znovu všechna nová přiřazení lokální.

Smluvená penalta $-10\,001$ si určitě vynutí stránkový zlom. V té chvíli bude výstupní rutina pracovat jako `\savehead`. Rozebereme si nyní činnost tohoto pomocného makra. Na řádku 386 ukládáme stávající stav strany (boxu 255) do rezervovaného místa `\pagebox`. Použijeme jej až po sestavení celé tabulky v hlavním vertikálním módu znovu. Pomocí `\lastbox` odebereme z této strany box, který tvoří hlavičku tabulky (`\headb`).

Na řádku 390 změříme zbytek místa na straně a podle tohoto zbytku nastavíme nové `\vsize`. Do této výšky budeme „nabírat“ jednotlivé řádky tabulky. Až bude výška naplněna, dostane se výstupní rutina ke slovu znovu. K Pišvejcově konstantě 4,7 pt jsme dospěli takto: $3 \times 0{,}4 + 3{,}5 = 4{,}7$, kde 0,4 pt je tloušťka horizontální linky (kterou makro doplní na každé straně na třech místech) a 3,5 pt je hloubka podpěry posledního řádku tabulky na straně. Pod něj bude připojena závěrečná linka tabulky. Chceme, aby tato linka byla maximálně ve výšce účaří posledního řádku, pokud srovnáváme stránky s běžným textem.

Takto vypočítané `\vsize` je pro „jedno použití“ jen pro dojezd první strany s tabulkou. Na dalších stranách už nebude nahoře předchozí text z `\pagebox` a bude tedy `\vsize` obecně větší. Napočítáme si je dopředu do `\tabsize`. Od originálního `\vsize` musíme odečíst kromě Pišvejcova čísla 4,7 pt ještě výšku plus hloubku boxu s hlavičkou. Pokud na řádku 392 zjistíme, že se do stávající strany nevejde ani první řádek tabulky, zahájíme celou tabulku až na další straně a nastavujeme tedy `\vsize` rovnou na hodnotu `\tabsize`.

V závěru makra `\savehead` počítáme horizontální posun `\dist`, o který posuneme všechny díly tabulky doprava, aby tabulka byla centrována podle `\hsize`. Šířku tabulky zjistíme podle šířky boxu se záhlavím.

Naše přechodná výstupní rutina je připravena provést ukončení strany v makru `\savetable`. Tam postupně přidáváme do `\tabbox` celý díl tabulky jako `\vbox` posunutý o `\dist`. V tomto boxu je linka, kopie boxu se záhlavím, linka, obsah strany a závěrečná linka (viz řádek 397). Pod box vkládáme `\vfil\penalty0`, což je budoucí místo stránkového zlomu.

Makro `\endlongtable` vyvolá naposledy přechodnou výstupní rutinu (pomocí vložení penalty $-10\,000$). Máme tedy jistotu, že tato rutina sestaví aspoň jeden díl tabulky. Po ukončení skupiny makro vloží zpětně do vertikálního seznamu `\pagebox` následovaný `\tabbox`. Druhý box se přitom rozpadne v algoritmu stránkového zlomu na jednotlivé díly tabulky.

V tomto příkladě si všimneme ještě jedné novinky. V rámci skupiny pro sazbu tabulky nastavujeme na řádku 378 `\holdinginserts` na jedničku. Při tomto nastavení TEX nekompletuje inserty do boxů, ale nechává jejich odkazy uvnitř sestaveného boxu 255. Skutečně, tyto inserty budeme ještě jednou potřebovat: v okamžiku, kdy `\pagebox` znovu vracíme do vertikálního seznamu. Proto budou fungovat i poznámky pod čarou, jejichž odkazy jsou v `\pagebox`. Dále nastavujeme `\maxdeadcycles` na hodnotu 1000, čímž dáváme najevo, že může být výstupní rutina tisíckrát vyvolána bez použití `\shipout`. Můžeme tedy dělat tabulky, které obsadí až tisíc stránek. Pokud samozřejmě dřív nezahltíme paměť TEXu. □

• **Vícesloupcová sazba.** Jeden přístup k vícesloupcové sazbě byl ukázán v příkladě na straně 244. Tam nebylo potřeba zasahovat do výstupní rutiny. Makro se hodí pro sazbu krátkých rejstříků (řádově desítky stránek) a pro tabulky. Nevýhodou makra je skutečnost, že celou sazbu musíme nejprve podržet v TEXovské paměti a teprve potom se upravuje a tiskne. To narazí při větším množství stránek na kapacitní možnosti TEXu. Výhodou makra zase byla skutečnost, že dovoluje uživateli velmi pružně přepínat mezi sazbou na plnou šířku stránek a sazbou do sloupců. Ve většině případů mi toto makro po mírné modifikaci postačilo. Chci-li ale navrhnout typografii vícesloupcové encyklopedie nebo sázet telefonní seznam Telecomu, bude potřeba navrhnout výstupní rutinu s ohledem na požadavek vícesloupcové sazby.

Nejjednodušší řešení vícesloupcové sazby by mohlo vypadat například takto:

```
400 \newcount\Ncols  \Ncols=3    % Počet sloupců sazby
401 \newcount\ncols  \ncols=1    % Počítá právě sestavovaný sloupec
402 \newdimen\colsep \colsep=2em % Mezery mezi sloupci
403 \newbox\cbox                 % Tam ukládáme jednotlivé sloupečky
404 \def\multioutput{%
405   \ifnum\ncols<\Ncols \savebox \else \printout \fi}
406 \def\savebox{\global\setbox\cbox=\columns
407   \global\advance\ncols by1 }
408 \def\printout{\setbox255=\vbox{\columns}
409   \hsize=⟨celková šířka n sloupcové sazby⟩
```

```
410   \plainoutput \global\ncols=1 }
411 \def\columns{% připojí další sloupeček z \box255:
412   \hbox{\ifvoid\cbox\else \unhbox\cbox\kern\colsep \fi \box255}}
```

Po zmenšení \hsize a nastavení \output={\multioutput} začne TEX sázet do sloupců. Zvětšení \hsize je na řádku 409 provedeno lokálně, takže má vliv pro sestavení záhlaví a paty strany v \plainoutput, ale po ukončení výstupní rutiny se vrátí \hsize zpětně ke globálnímu rozměru, který odpovídá šířce sloupce.

Pozastavme se u řádku 412. Tam je připojen sloupeček s výškou \pagegoal, která je konstantně rovna \vsize. Případ poznámek pod čarou ve vícesloupcové sazbě nebudeme pro jednoduchost uvažovat. Pokud bychom chtěli vidět, jak úspěšný byl sloupcový zlom, místo \box255 pišme \vbox to\ht255{\unvbox255}. Jestliže požadujeme řádkový rejstřík a zakazujeme výskyt vdov a sirotků, pak se po tomto obratu dozvíme, které sloupečky nedopadly zrovna nejlépe. Dále můžeme použít myšlenky z příkladu o vyrovnání pravé a levé strany (viz stranu 267).

Při povelu \end TEX opakuje volání výstupní rutiny tak dlouho, až je splněno \deadcycles=0. Nehrozí tedy předčasné ukončení TEXu se ztrátou informace uložené v \cbox. Na poslední stránce se proto mohou objevit nějaké sloupečky (vpravo) prázdné. Na druhé straně \vfil\eject ukončí jenom jeden sloupec a ne celou stranu. Vytvoříme si makro \ejectpage, které například na konci kapitoly ukončí sazbu celé stránky.

```
413 \mathchardef\pagepenalty=10201  % smluvená hodnota
414 \def\ejectpage{\par \penalty-\pagepenalty}
415 \def\multioutput{\ifnum\outputpenalty=-\pagepenalty \printout
416   \else \ifnum\ncols<\Ncols \savebox \else \printout \fi \fi}
```

Ještě bychom rádi, aby byl začátek kapitoly nejprve zahájen na plnou šířku sazby a teprve po použití makra \startcols se rozjela vícesloupcová sazba až do konce kapitoly, kde se použije \endcols. V takovém případě musíme postupovat podobně, jako u předchozího příkladu s tabulkou. Nejprve si zapamatovat do rezervovaného \pagebox původní obsah strany, dále změřit zbytek místa na straně a potom teprve sázet do sloupců. Schematicky by se problém dal řešit takto:

```
417 \newbox\pagebox  \newdimen\Vsize
418 \def\startcols{\bigskip\penalty0
419   {\output={\global\setbox\pagebox=\vbox{\unvbox255}}\vfil\break}
420   \begingroup
421     \Vsize=\vsize  % zapamatuji si původní výšku strany
422     \vsize=⟨zmenšená velikost o výšku boxu⟩ \pagebox⟩
423     \hsize=⟨zmenšená na šířku sloupce⟩
424     \output={\multioutput}}
425 \def\endcols{\vfil \penalty-\pagepenalty \endgroup}
```

```
% musíme upravit definici z řádku 408:
\def\printout{\setbox255=\vbox{\columns}
  \ifvoid\pagebox \else
     \setbox255=\vbox{\unvbox\pagebox \unvbox255}
     \global\vsize=\Vsize \fi
  \hsize=⟨celková šířka n sloupcové sazby⟩
  \plainoutput \global\ncols=1 }
```

Na počátku je aktuální nastavení `\output={\plainoutput}`. Trikem z řádku 419 uložíme obsah aktuální strany do `\pagebox`. Pak otevřeme skupinu a v ní změníme `\vsize`, `\hsize` a `\output`. Základní myšlenka výstupní rutiny `\multioutput` byla předvedena před chvílí. Toto makro musíme v případě `\printout` vybavit dodatečnou inteligencí, aby poznalo, zda je voláno poprvé s kratším `\vsize`. V takovém případě doplní stranu nahoře o obsah z `\pagebox`. □

Všimli jsme si, že makro `\begmulti` ze strany 244 zarovná po ukončení vícesloupcové sazby jednotlivé sloupečky do pokud možno stejné výšky. Tomuto fenoménu budeme říkat *vyrovnání sloupců*. Zrovna předvedené makro to neumí. Toto makro vyplní na konci kapitoly nejprve celý první sloupec. Pokud zbyl ještě nějaký materiál, začne s druhým sloupcem, pak s třetím atd. Zda to chceme nebo ne, záleží samozřejmě na návrhu typografie dokumentu.

Pokud chceme, aby naše makro vyrovnalo sloupce v místě použití značky `\endcols`, musíme na problém jít zcela jiným způsobem. Při výskytu sekvence `\endcols` se totiž už zalomilo prvních k sloupců (pro $k \leq n$) na zrovna zpracovávané straně. My bychom ale chtěli veškerý tento materiál spojit zpětně do jednoho `\vbox`u a rozlomit jej na n stejně vysokých sloupců. To dost dobře nejde, protože velikosti meziřádkových mezer v místě zlomu už jsou ztraceny a nejde tedy zrekonstruovat původní sazbu v jednom dlouhém sloupci. Zkusíme tedy toto řešení:

```
\newdimen\orivsize
\orivsize=\vsize     % původní hodnota \vsize
\vsize=\Ncols\vsize  % n násobek původní \vsize
\hsize=⟨zmenšená na šířku sloupce⟩
\output={\Multioutput}}
```

Nová výstupní rutina `\Multioutput` přijme materiál v boxu 255 až po vytvoření sloupečku s n násobnou výškou, než je požadovaná. Výstupní rutina sama bude tento sloupeček rozdělovat do n sloupců pomocí `\vsplit` podobným způsobem, jako v makru `\begmulti` na straně 244.

Problém vyrovnání sloupců automaticky ale není zcela jednoduše řešitelný. Představme si, že při dvousloupcové sazbě máme rozdělit sloupeček přesně napůl, ale uprostřed je větší nezlomitelný element (rovnice, nadpisy, zákaz parchanta apod.). Pokud je zlomeno pod tímto elementem, je levý sloupec přetečený. Pokud je zlomeno

před ním, je levý sloupec nezaplněný a z pravého přetéká sazba na další stranu. Vyrovnat se s problémem skutečně poctivě by vyžadovalo dosti složitá makra, která by opakovaně zkoušela \vsplit s různými možnostmi nastavení výšky sloupce a konvergovala by k jistému optimu. Už jen formálně definovat toto optimum za všech okolností je komplikované. V následující ukázce uvedeme jen drobný střípek, který může vést k řešení tohoto problému.

Podíváme-li se do Zlatých stránek, zjistíme, že sloupce jsou na konci vyrovnané, ovšem za cenu toho, že mezi každým řádkem je pružná mezera (\baselineskip). Takže o řádkovém rejstříku vůbec není řeč. V naší ukázce vyjdeme právě z předpokladu, že řádky mají mezi sebou dostatečnou pružnost a sloupce se přizpůsobí v rámci možností požadované výšce. Nechť $\vsize = n \times$ \colsize, kde \colsize je skutečná výška jednoho sloupce na straně. Box 255 přichází tedy do výstupní rutiny v n násobné výšce \colsize. Pokud se jedná o poslední stranu, dostaneme o tom zprávu ve formě \outputpenalty=-\balancepenalty. V takovém případě budeme vyrovnávat sazbu do sloupců stejné velikosti.

```
\output={\ifnum\outputpenalty=-\balancepenalty \balancuj
         \else \vypln \fi}
\def\rozdelsloupce{% rozdělí box 255 do n sloupců výšky \colsize
   \setbox1=\hbox{} \ncols=0
   \loop \ifnum\Ncols>\ncols
     \global\setbox1=\hbox{%
         \unhbox1 \vsplit255 to\colsize \hskip\colsep}
     \advance\ncols by1
   \repeat}
\def\vypln{\rozdelsloupce \Tisk
   \unvbox255} % pokud něco zbylo, vracím zpátky !!
\def\balancuj{\setbox0=\vbox{\unvcopy255} % záložní kopie
   \colsize=\ht0 \divide\colsize by\Ncols % požadovaná výška
   \loop {\rozdelsloupce}                 % zkus rozdělit
     \ifvbox255              % zůstal nevytištěný zbytek?
       \advance\colsize by.2\baselineskip % zvětším cílovou výšku
       \global\setbox255=\copy0           % vrátím zpět
   \repeat                                % a zkusím znovu
   \Tisk}
\def\Tisk{\shipout\vbox to⟨xy⟩{⟨záhlaví⟩\box1 \vfil ⟨pata strany⟩}
                \advancepageno}
```

Zajímavé místo v tomto makru je na řádku 448, kde vracíme nespotřebovaný zbytek do vertikálního seznamu. Nikde totiž není řečeno, že když box výšky \vsize rozdělíme třeba na tři kousky přesně třetinové výšky (pomocí \vsplit), tak vyčerpáme veškerou původní sazbu z boxu. S podobným problémem se potýkáme při závěrečném vyrovnávání sloupců v makru \balancuj, kde nejprve zkusíme rozdělit

box přesně na n stejných částí. Pokud něco zbylo, zvětšíme drobet cílovou výšku sloupců a provádíme činnost znovu.

Při vícesloupcové sazbě můžeme narazit na další problémy. Třeba v předchozí ukázce nám s přebytečným materiálem, který se vrací zpět do vertikálního seznamu, utíká i `\botmark`. Místo něj můžeme v záhlaví použít `\splitbotmark`, který obsahuje skutečně poslední značku `\mark` v posledním sloupci, protože tento sloupec vznikl poslední operací `\vsplit`. □

• **Kombinace sazby do sloupců s obrázky.** V předchozí ukázce jsme naznačili, že existují dva přístupy k programování vícesloupcové sazby. Nyní v tomto příkladě ukážeme, že to není tak docela pravda, že existuje ještě aspoň jeden další přístup k tomuto problému.

Při sazbě novin nebo časopisů máme na vstupu texty jednotlivých článků. Dále obrázky a reklamy, které lze podle potřeby zvětšit či zmenšit. Výstupem činnosti sazeče by měly být jednotlivé strany novin nebo časopisu, které mají pevný rozměr. Všechny strany by měly být zcela zaplněny určitým systémem kladení sloupečků sazby do strany a kombinováním takové sazby s obrázky. Podle toho, jak vychází text jednotlivých článků, sazeč zvětší či zmenší obrázek, aby strana byla vhodně zaplněna. Přitom obrázky se neváží ke konkrétnímu odstavci v textu, ale spíše dokreslují celkový vzhled sazby. V některých periodikách dokonce bývají obrázky opakovaně na stejném místě, což určuje charakter periodika.

Na co nejefektivnější řešení takového úkolu jsou zaměřeny především interaktivní programy pro přípravu elektronické sazby, které bývají postaveny na principu WYSIWYG. Sazeč myškou nastrká na stranu, kterou vidí na obrazovce, jednotlivé komponenty sazby. Tento přístup je sice TEXu cizí, ovšem ukážeme, že i v TEXu lze poměrně efektivně stanovený úkol řešit.

Následující příklad nedotáhneme do takové obecnosti, aby se pomocí předvedených maker dala rovnou sázet Mladá fronta Dnes. Pokusíme se ale, aby případné zobecnění bylo již celkem názorné. Jako příklad jsem volil sazbu svého článku „*Kam se poděla dobrá typografie?*“, který vyšel v časopise Mensa 5/96. Časopis pracuje s dvousloupcovou sazbou, kterou kombinuje s obrázky. Příklad dále redukujeme na ukázku jediného článku v tomto časopise. Článek nejprve v TEXu celý načteme do samostatného `\vbox`u a potom z něj budeme postupně při plnění jednotlivých stran „ukrajovat“ požadovaný počet řádků. Použijeme k tomu primitiv `\vsplit`. Kdybychom pracovali s více články současně, načetli bychom je do více samostatných `\vbox`ů. Z nich bychom „ukrajovali“ sloupečky pro jednotlivé strany.

Algoritmus stránkového zlomu je tedy zcela pod kontrolou uživatele. Až je strana zaplněna jednotlivými sloupečky a obrázky, vystoupí do výstupní rutiny po explicitním použití `\eject`. Výstupní rutina pak pouze připojí záhlaví a patu strany podle

návrhu typografie zpracovávaného periodika. V našem případě se jedná o časopis Mensa. Zde je výstupní rutina:

```
\def\datum{5/1996}
\def\mensaoutput{\shipout\vbox{%
   \hrule height2pt \kern2pt\hrule %% V záhlaví jsou dvě linky
   \vbox to188mm{\kern4pt\unvbox255\vfil}    %% Obsah strany
   \hrule height1.5pt \kern5pt
   \hbox to\hsize{\patafont                  %% Patička
      \ifodd\pageno MENSA \datum\hfil\the\pageno
      \else \the\pageno\hfil MENSA \datum\fi}}
   \global\advance\pageno by 1\relax}        %% Paginace
\hsize=124mm  %% Šířka strany. Výšku \vsize nepotřebujeme
\output={\mensaoutput}
```

Makro `\datum` je potřeba měnit pro každé číslo časopisu. Jeho obsah se totiž, jak vidíme v makru `\mensaoutput`, sází do paty strany. Na lichou stranu doleva a na sudou doprava. Nyní ukážeme další část makra, kde definujeme použité fonty a parametry sazby odstavce:

```
\font\rm=cptmr at9pt \font\bf=cptmb at9pt
\font\it=cptmri at9pt  \rm                % Times-Roman-*
\font\patafont=cptmr at11pt
\font\titlfont=cphvb at11pt               % Helvetica-Bold
\font\autfont=cptmrc at9pt                % Small-Caps
\font\tensy=cmsy10 at9pt \textfont2=\tensy % Matematika v CM

\newbox\celybox              % \vbox s textem článku
\newdimen\colsize            % šířka sloupce
\newdimen\colsep             % mezera mezi sloupci
\newdimen\picsize            % pomocné, pro velikost obrázků
\newif\ifframe \newif\ifpic  % režim tisku obrázků

\chyph \righthyphenmin=2     % české dělení slov
\colsize=6cm  \colsep=4mm  \parindent=1em
\emergencystretch=2em \hbadness=2000
\pretolerance=300 \tolerance=1000 \exhyphenpenalty=10000
\doublehyphendemerits=100000 \finalhyphendemerits=10000000
\parskip=0pt  \baselineskip=11pt
\medskipamount=.5\baselineskip \bigskipamount=\baselineskip
```

Text každého článku budeme mít ve zvláštním souboru. V našem případě budeme mít článek o typografii v souboru `typo.tex`. Naším úkolem je připravit makra, která tento článek načtou ze souboru do `\vbox`u (například použitím `\natahni[typo]`) a potom z něj budou „ukrajovat“ části podle stanoveného počtu řádků. Například

`\ulom[28]` „odkrojí“ z vrcholu `\vbox`u s článkem 28 řádků, které v místě použití makra `\ulom` vystoupí ve formě `\vbox` nebo `\vtop` (podle potřeby).

```
490 \splittopskip=8pt plus 8pt  \topskip=0pt
491 \def\natahni[#1]{\testvoidbox \setbox\celybox=\vbox{%
492       \let\end=\endinput \hsize=\colsize \penalty0 \input #1
493       \par\hfill\autfont\Autor\unskip \vfil} %% konec \vbox
494    \setbox0=\vsplit\celybox to0pt}
495 \def\ulom[#1]{\setbox0=\vsplit\celybox to #1\baselineskip
496    \pouzdro{\unvbox0}}
497 \let\pouzdro=\vtop %% implicitně vystupuje \ulom jako \vtop
498 \def\korektura{\output={\plainoutput} \unvbox\celybox}
499 \def\testvoidbox{\ifvoid\celybox\else
500    \message{Pozor: část textu ztracena!}\fi}
```

Makro `\testvoidbox` kontroluje, zda `\natahni` nepřepisuje v boxu `\celybox` zbytek nepoužitého předchozího článku. Pokud ano, vypíše se varování na terminál. Pak je do boxu `\celybox` načten celý soubor `#1`. Nakonec je připojeno jméno autora článku, jak bývá v periodiku zvykem.

Na řádku 494 neodlomíme z boxu s článkem nic. Děláme to jenom proto, aby „zbytek“ v boxu `\celybox` byl opatřen nahoře mezerou podle `\splittopskip`. To budeme při dalším „ukrajování“ článku potřebovat.

Makro `\ulom` „ukrojí“ z boxu s článkem právě `#1` řádků. Při hledání zlomu bude pracovat pružnost ze `\splittopskip`, která se nastaví na hodnotu 3 pt, což je přesně rozdíl mezi `\baselineskip` a přirozenou velikostí z `\topskip`. Výsledek se „zapouzdří“ do `\vtop` nebo `\vbox` podle aktuálního významu sekvence `\pouzdro`.

Pokud nebudeme sázet celý časopis, ale budeme chtít pouze vytisknout článek pro sloupcovou korekturu, použijeme makro `\korektura` (viz řádek 498).

Uvnitř článku (tj. v souboru `typo.tex`) používá autor smluvené TEXovské sekvence:

```
501 \def\titul #1 \par{\gdef\Titul{#1}}
502 \def\autor #1 \par{\gdef\Autor{#1}}
503 \def\sub #1 \par{\par\hbox{}\nobreak\vskip-\medskipamount
504    {\bf#1}\par\nobreak\medskip}
```

Makro `\titul` uloží do separátního makra `\Titul` název článku. Obsah tohoto makra použijeme až při návrhu vzhledu strany a připojíme jej třeba nad všechny sloupce současně. To známe z novinových „palcových titulků“.

Podobně makro `\autor` uloží do vyhrazeného makra `\Autor` ctěné jméno autora. Toto makro je použito v `\natahni` na konci článku. Konečně makro `\sub` bude

sázet nadpis dílčího úseku článku. Všimneme si, že tento nadpis nenaruší řádkový rejstřík.

V další části našeho příkladu se budeme zabývat problematikou obrázků. Veškeré obrázky budeme zpracovávat ve formátu EPSF (Encapsulated PostScript). Toto je nejrozšířenější formát pro profesionální zařazování obrázků do sazby. Máme-li obrázky připraveny v jiném formátu, použijeme konverzní program a samozřejmě korekční program, pomocí něhož odstraníme z obrázku kazy, nastavíme barevnou saturaci, kontrast a další věci. V tuto chvíli už předpokládáme, že každý obrázek má tyto věci úspěšně za sebou.

K obrázku připojíme popisky. Vytvoříme makro, pomocí něhož budeme pohodlně definovat vztah mezi názvem obrázku (názvem souboru s obrázkem) a jeho popiskem. Makro se jmenuje `\popis` a ukázku jeho použití vidíme později na řádcích 521 až 527. Dále definujeme makro `\obr`, které v místě použití sází obrázek `#2` na stanovenou šířku podle parametru `#1`.

```
505 \input epsf  %% Standardní makro pro práci s EPSF obrázky
506 \def\popis[#1]#2{\expandafter\def\csname obr:#1\endcsname{#2}}
507 \def\obr#1[#2]{\vbox{\hsize=#1 \epsfxsize=\hsize
508    \ifframe\else \let\ramekl=\relax \let\ramekr=\relax \fi
509    \ifpic\else
510       \def\special##1{\raise5.5pt\hbox{\ \rm#2.eps}}\fi
511    \ramekl\epsfbox{#2.eps}\ramekr
512    \nadpopisem \leftskip=0pt plus1fil \rightskip=0pt plus-1fil
513    \parfillskip=0pt plus2fil \it\noindent
514    \csname obr:#2\endcsname}}
515 \def\ramekl{\vbox\bgroup\hrule\kern-.4pt
516    \hbox\bgroup\vrule\kern-.4pt}
517 \def\ramekr{\vrule\kern-.4pt\egroup\hrule\kern-.4pt\egroup}
518 \def\nadpopisem{\kern1pt}
```

Makro `\obr` nastavuje na řádku 507 požadovanou šířku obrázku. Z registru `\epsfxsize` bude makro `\epsfbox` číst požadovaný horizontální rozměr obrázku. Vertikální rozměr se dopočítá tak, aby nedošlo k deformaci obrázku. Na řádku 511 tiskneme vlastní obrázek použitím makra `\epsfbox`, které je definováno v balíku `epsf.tex`. Toto makro se podívá do souboru s obrázkem na údaj `BoundingBox`, kde se dozví rozměry obrázku. Dále expanduje na určitý `\special`, který ve výstupním `dvi` dává ovladači `dvips` pokyn, co má udělat. Ovladač podle pokynu nastaví nový grafický stav PostScriptu s transformací souřadnic tak, aby měl obrázek požadovanou velikost. Dále ovladač zařadí v místě `\special` do výstupního PostScriptu vlastní data obrázku.

Slušné prohlížeče `dvi` (například v UNIXu) umějí v místě takového `\special` s obrázkem vyvolat Ghostscript a nechat si od něj obrázek vyrastrovat. Výsledek pak

vloží do náhledu na obrazovce. Bohužel, Ghostscript pracuje se skutečnými daty obrázku a nikoli jen s rychlým náhledem. To nás může u velkých bitmapových a barevných obrázků zdržovat při práci. Z toho důvodu se naše makro `\obr` stará i o alternativní sazbu obrázku. Primitivní povel `\special` je na řádku 510 předefinován tak, že místo obrázku vidíme jen text s názvem souboru. Navíc při volbě rámečků kolem obrázků (`\ifframe`) můžeme vidět hranici obrázků.

```
519 \picfalse \frametrue     % Obrázky přeskoč, kresli jen rámečky
520 %\pictrue  \framefalse   % Obrázky vykresli bez rámečků
```

Z řádku 520 zrušíme vpředu komentářový znak až při definitivním tisku, nebo v době, kdy máme čas si počkat, až se všechny obrázky v UNIXovém prohlížeči vykreslí.

Při návrhu našeho makra jsme už udělali dost velký kus práce. Proto nyní budeme sklízet plody této práce. Článek včetně obrázků a popisků k obrázkům zavedeme do TEXu přehledně takto:

```
521 \natahni [typo]    %%%% Kam se poděla dobrá typografie?
522 \popis [gutenber] {Johannes Gensfleisch Gutenberg (1395--1468)}
523 \popis [linotype] {Schéma řádkového odlévacího stroje Linotype}
524 \popis [madona]   {Bruselská madona, tisk o jednom listu (1418)}
525 \popis [setkani]  {Setkání různých technologií}
526 \popis [bible]    {42 řádková Gutenbergova bible s ručně
527                     domalovanými iniciálami (1452--1455)}
```

Konečně navrhneme vzhled jednotlivých stran. Postupně „ulamujeme“ části natažené článku a metodou `\hbox{\vbox{..}\kern..\vbox{..}}` vytváříme konečný vzhled strany.

```
528 %% Zkratky:
529 \def\cl{\hskip\colsep} \def\eject{\vfil\break}
530 \def\bye{\testvoidbox\end} % kontrola, zda je článek celý
531 \def\datum{5/1996} %% Číslo periodika
532 \pageno=37         %% Strana. Zahajuje tuto část časopisu
533                                                  %% Strana 37
534 \vglue5pt
535 \centerline{\titlfont \Titul}\vskip-2pt  % sazba titulku
536 \hbox{\ulom[46]\cl\vtop{\ulom[19]        % dva sloupce
537    \obr\colsize[gutenber]}}              % ve druhém obrázek
538 \eject
539                                                  %% Strana 38
540 \hbox{\let\pouzdro=\vbox   % dva sloupce, v prvním obrázek
541    \vbox{\obr\colsize[linotype]\bigskip\ulom[27]}\cl\ulom[48]}
542 \eject
```

```
                                                    %% Strana 39
\hbox{\ulom[30]\cl\ulom[30]}  % dva sloupce, pod nimi obrázek
\picsize=\hsize \advance\picsize by-3.1cm
\centerline{\obr\picsize[setkani]}
\eject
                                                    %% Strana 40
\hbox{\vtop{\ulom[24]\kern9pt          % dva sloupce
   \obr\colsize[madona]}\cl\ulom[48]}  % v prvním obrázek
\eject
                                                    %% Strana 41
\hbox{\ulom[48]\cl % dva sloupce, v druhém obrázek
   \vtop{\ulom[23]\kern7pt\obr\colsize[bible]}}
\eject
                                                    %% Strana 42
\hbox{\ulom[48]\cl\ulom[48]} % standardní dva sloupce.
\eject  \bye
```

K tomuto výsledku je třeba dospět interaktivně. V jednom okénku operačního systému pracujeme s primitivy \hbox, \vbox a s makry \ulom, \obr a ve vedlejším okénku po TEXování vidíme v prohlížeči výsledek. Podle toho, jak to vychází, přidáváme nebo ubíráme počty řádků v \ulom a zvětšujeme nebo zmenšujeme \obr. Pokud nám na časopisu hodně záleží, doplňujeme podle výsledku do vybraných odstavců \looseness, abychom se zbavili parchantů. Pokud chceme dělat efekty s obtékáním obrázků (tj. přechodně je sloupec s textem vedle obrázku zúžen), můžeme použít makro \oblom, které jsme předvedli v sekci 6.5 na straně 236. □

• **Poznámky na okraji.** V některých publikacích se dávají krátké texty na okraj stránky vedle hlavního textu. Například v současných počítačových publikacích je tento grafický prvek velmi častý. Tyto publikace jsou totiž často inspirovány americkými návrhy typografie, kde se používají v knize na naše poměry až příliš velké okraje.

Poznámky na okraji lze řešit přímo pomocí \vadjust (viz heslo \vadjust v části B). Nemusíme používat inserty. Potřebujeme pouze zjistit, zda jsme na pravé straně (pak bude poznámka na pravém okraji), nebo na levé straně (pak bude poznámka na levém okraji). K tomu stačí použít vlastnosti primitivu \write a zpracovávat dokument dvěma průchody. Takové řešení jsem podrobně popsal v [7] a nebudu je zde znovu opakovat.

V tomto posledním příkladě si ukážeme výstupní rutinu, která využívá inserty. Jednu takovou už známe. V sekci 6.8 je rozebrána výstupní rutina plainu, která pracuje se dvěma třídami insertů. Jednu používá na poznámky pod čarou a druhou na plovoucí objekty (tabulky a obrázky). Zde zkusíme pomocí insertů implementovat poznámky na okraji strany.

Naše řešení bude mít několik omezení. Mezi nejdůležitější omezení patří skutečnost, že nedovolíme do strany vkládat pružné mezery. Budeme totiž vedle boxu s textem strany vytvářet box s poznámkami na okraji. Vertikální mezery mezi poznámkami budou v tomto boxu pro jednoduchost pevné. Výstupní rutina pak už jen snadno připojí takový box s poznámkami vpravo nebo vlevo do boxu 255.

Použití našeho makra bude jednoduché. Kdekoli uvnitř odstavce napíšeme sekvenci `\marginalie{`⟨*text*⟩`}`. Při sazbě strany se dostane ⟨*text*⟩ na okraj tak, že první řádek ⟨*textu*⟩ bude ve stejné výšce, jako je řádek, ve kterém je `\marginalie` použita. Samotný ⟨*text*⟩ je sázen jako odstavec s menším `\hsize` a na praporek. Pokud je ⟨*text*⟩ tak dlouhý, že přesáhne konec strany, bude pokračovat na okraji příští strany. Pokud se budou dva ⟨*texty*⟩ z důvodu „hustého“ použití `\marginalie` na okraji překrývat, dostaneme o tom od TEXu varování.

Definujeme třídu insertu `\margins`. Po sestavení strany budou v boxu `\margins` všechny okrajové poznámky včetně správných vertikálních mezer mezi nimi.

```
\newinsert\margins
\dimen\margins=\vsize  % po zaplnění b_n se poznámka zlomí
\count\margins=0       % vložení poznámky nezmenšuje \pagegoal
\skip\margins=0pt      % ani první poznámka nezmenší \pagegoal
\newdimen\marsize \marsize=50pt % šířka sazby pro poznámky
```

Výstupní rutina bude po uzavření strany pouze klást vedle boxu 255 box `\margins`. Na pravé straně vpravo a na levé vlevo:

```
\def\pageoutput{\shipout\vbox{\vbox to\vsize{\hbox{%
         \ifodd\pageno \pageright \else \pageleft \fi}\vss}
      \bigskip \line{\hfil\the\pageno\hfil}} % pata strany
   \advancepageno}
\def\pageright{\vtop{\zrule \unvbox255}\kern10pt
               \vtop{\zrule \unvbox\margins}}
\def\pageleft{%
   \ifvoid\margins
   \else {\hsize=\marsize \toright\margins}% levý praporek
         \kern-\marsize \kern-10pt
         \vtop{\zrule \unvbox\margins}\kern10pt
   \fi   \vtop{\zrule \unvbox255}}
\def\zrule{\hrule height0pt \relax}
\hoffset=20pt % zvětšíme levý okraj, aby se tam poznámky vešly
```

Pomocí neviditelné linky `\zrule` a primitivu `\vtop` usadíme oba boxy vedle sebe tak, aby horní okraje boxů byly ve stejné výšce. Při sazbě levé strany (viz `\pageleft`) navíc převedeme zarovnání boxu s poznámkami na levý praporek. K tomu jsme využili makro `\toright` ze strany 265.

Nejvíce práce nám dá přidání správných mezer mezi poznámky do boxu \margins. K tomu účelu vyvoláme explicitně výstupní rutinu ve speciálním režimu. V tomto režimu se nebude tisknout obsah strany, ale změříme současné zaplnění strany. Výstupní rutina se tedy bude větvit:

```
578 \output={\ifnum\outputpenalty=-10013 \addspacetomargins
579           \else                         \pageoutput \fi}
```

Výklad činnosti rutiny při \addspacetomargins zatím odložíme a uvedeme definici uživatelského makra \marginalie:

```
580 \def\marginalie #1{\ifvmode \indent \fi
581    \vadjust{\penalty-10013}%
582    \insert\margins{\raggedright \hsize=\marsize
583       \noindent\vrule height\topskip width0pt #1}}
```

Vidíme, že \marginalie nejprve požádá výstupní rutinu, aby změřila stav zaplnění strany. Po explicitním \penalty-10013 se totiž aktivuje \addspacetomargins. Teprve pak se vloží do vertikálního materiálu insert s vlastním textem poznámky.

Nyní už zbývá jen ukázat makro \addspacetomargins a rozebrat jeho činnost.

```
584 \def\addspacetomargins{%
585   \ifvoid\margins \dimen0=0pt % \dimen0 bude vložená mezera
586   \else \dimen0=-\ht\margins \advance\dimen0 by-\dp\margins \fi
587   \advance\dimen0 by\pagetotal \advance\dimen0 by-\topskip
588   \ifdim\dimen0<0pt
589      \message{Pozor: marginálie se překrývají!}\fi
590   \insert\margins{\vskip\dimen0}
591   \unvbox255 }
```

V \dimen0 vypočítáme velikost mezery, kterou je potřeba do boxu \margins vložit. Při zatím prázdném \margins bude tato mezera rovna \pagetotal. Tento registr obsahuje *nezkresleně* současné zaplnění strany, protože výstupní rutina byla vyvolána explicitně penaltou. Mezeru dále zmenšíme o \topskip, protože to je výška podpěry, která je obsažena na začátku každého textu poznámky (viz makro \marginalie, řádek 583). Při neprázdném boxu \margins zmenšíme ještě počítanou mezeru o součet výšky a hloubky tohoto boxu.

Na řádku 588 testujeme, zda nevyšla požadovaná mezera záporně. Pokud ano, je potřeba upozornit na to, že se budou poznámky překrývat.

Nejzajímavější jsou poslední dva řádky našeho makra. Zde vracíme do přípravné oblasti nejprve insert s napočítanou mezerou a potom celý stávající obsah strany. Za tímto materiálem přichází (jak víme ze sekce 6.8) element zlomu, který je v tuto

chvíli roven penaltě 10 000. Proto se zlom strany nedovolí těsně za řádkem, který referuje na poznámku na okraji.

Box `\margins` nebyl v této fázi výstupní rutiny použit, proto zůstává obsah odpovídající paměti b_n zachován. Odkazy na inserty už v boxu 255 neexistují. To nám pro tuto třídu insertů nevadí. Algoritmus plnění strany převede po ukončení `\addspacetomargins` do aktuální strany nově připravený insert, který doplní do b_n napočítanou mezeru. Pak teprve přichází insert, který byl sestaven v makru `\marginalie`. Obsah tohoto insertu se připojí do b_n pod mezeru. Pokud se přeplní b_n tak, že je jeho výška větší než `\dimen\margins=\vsize`, pak se insert rozlomí a pokračuje na další straně.

V této ukázce jsme použili několik nečistých obratů. Obsah boxu 255 jsme vraceli z výstupní rutiny zpět do přípravné oblasti, přitom jsme ale nenastavili `\holdinginserts` na kladnou hodnotu. Z toho důvodu jsme přišli o všechny odkazy na inserty. S naším řešením tedy nebude korektně fungovat třída insertů pro poznámky pod čarou a pro plovoucí objekty. V TEXbooku totiž není dokumentovaná tato vlastnost paměti b_n: pokud výstupní rutina nepoužije odpovídající box, zůstává b_n nezměněna. Některé algoritmy tedy nemají v této situaci definované chování, zvláště pokud inserty mění `\pagegoal`. To ale nebyl v případě poznámek na okraji náš případ. Kdybychom nastavili `\holdinginserts` na jedničku, pak nám TEX sice ponechá v boxu 255 odkazy všech insertů, ale vůbec nám neumožní ve výstupní rutině nahlédnout do místa b_n. Tudíž nemůžeme měřit při výpočtu mezery mezi poznámkami výšku předchozího materiálu z b_n.

Další problém: Může se stát, že se nám insert s textem poznámky vrátí celý zpět do přípravné oblasti, protože nejlepší místo zlomu zůstalo před řádkem, ve kterém byl insert použit. Pak ale byl výpočet mezery v makru `\addspacetomargins` zbytečný. Naše makro to přitom nepozná. Insert se pak objeví na následující straně hned nahoře. Řádek, ze kterého jsme na insert odkazovali, může ale být výjimečně druhým řádkem na straně. To je jediná situace, kdy poloha poznámky na okraji neodpovídá přesně místu, kde byla poznámka vytvořena. Je to velmi výjimečná situace a chyba makra v tomto případě není příliš podstatná.

Může se stát, že chceme vedle hlavního textu sázet na okraji strany (třeba drobnějším písmem) ještě jeden relativně nezávislý text, který se samostatně zalamuje do stránek. To je přesně to, co dokáže předvedený typ insertu. Pokud oba sloupce (hlavní i komentářový) začínají po zahájení kapitoly ve stejné výšce, není potřeba dále výšku textů v insertu měřit. Stačí nastavit `\dimen\margins=\vsize` a máme vystaráno. Makro je tedy podstatně jednodušší. Příklad takové koncepce sazby můžeme najít třeba v knize *Toulky českou minulostí* pana Petra Hořejše. □

7. Různé

7.1. Jak TEX pracuje se soubory

TEX v operačním systému pracuje s těmito vstupními a výstupními zdroji (v závorce je uveden odpovídající primitiv):

- Textový vstupní proud (`\input`, `\endinput`).
- Příkazový řádek systému.
- Vstup předzpracovaného binárního formátu `fmt`.
- Vstup binárních souborů `tfm` (`\font`).
- Textový výstup protokolu zpracování do souboru `log` a na terminál.
- Binární výstup do souboru `dvi` (`\shipout`).
- Binární výstup do souboru `fmt` (`\dump` při iniTEXu).
- Textové vstupy z pracovních souborů (`\read`).
- Textové výstupy do pracovních souborů (`\write`).

• **Textový vstupní proud.** Informace z tohoto zdroje jsou postupně zpracovány input procesorem a dalšími procesory TEXu tak, jak jsme popsali v první a druhé kapitole. Jeden textový soubor je považován za hlavní (`\jobname`) a ostatní jsou z něj (obvykle) průběžně načítány (`\input`, `\endinput`). V těchto „vnořených“ souborech může být znovu použito `\input`. Po `\endinput` z hlavního souboru, případně po jeho celém přečtení a nedosažení povelu `\end`, TEX přechází ke čtení dalších řádků vstupního proudu z terminálu v interaktivním režimu. To dá najevo promptem ve tvaru hvězdičky. □

• **Příkazový řádek systému.** Zde se rozhoduje, jak se bude jmenovat hlavní soubor vstupního proudu. Také se zde může specifikovat jméno binárního formátu, který má být před zahájením zpracování vstupního proudu načten. Způsob práce s příkazovým řádkem je závislý na systému, takže je potřeba nastudovat dokumentaci ke konkrétní instalaci TEXu. Například stránky `man` v UNIXu nebo `tex.doc` v emTEXu pro DOS a OS/2.

Obvykle platí tato pravidla: (1) Pokud je na příkazovém řádku za zápisem pro spuštění programu znak `&`, pak je následující text (až po mezeru nebo konec řádku) interpretován jako název binárního souboru `fmt`, který je před startem zpracování vstupního proudu načten. Přípona `fmt` je nepovinná. (2) Dále na příkazovém řádku následuje ⟨*zbytek příkazového řádku*⟩. Pokud je toto místo prázdné, TEX přejde do interaktivního režimu a první řádek vstupního proudu (a případně i další) bude nutno napsat z terminálu. Začíná-li ⟨*zbytek příkazového řádku*⟩ znakem s kategorií 0, je celý interpretován jako první řádek vstupního proudu. Nezačíná-li

⟨*zbytek příkazového řádku*⟩ znakem s kategorií 0, je interpretován jako název hlavního souboru vstupního proudu. Přesněji, před ⟨*zbytek příkazového řádku*⟩ je vložen povel `\input` a tento celek je interpretován jako první řádek vstupního proudu.

Například v rámci balíku emTEX v DOSu voláme program `tex386.exe`. Příkazový řádek, který spouští TEX s formátem `csplain.fmt` a zahajuje čtení hlavního souboru `dokument.tex`, může vypadat:

```
> tex386 &csplain dokument
```

Pokud chceme vyzkoušet druhou možnost interpretace vstupního řádku, pišme:

```
> tex386 &csplain \hsize=10cm \input dokument
```

V UNIXu voláme program `virtex` a můžeme třeba psát:

```
$ virtex '&csplain \hsize=10cm \input dokument'
```

což udělá totéž jako v předchozí ukázce. Apostrofy `'...'` jsou pouze záležitostí UNIXového shellu. V příkazovém řádku z pohledu TEXu se apostrofy po zpracování shellem už nevyskytují. Nebudeme zde rozebírat podrobně pozadí jednotlivých systémů, takže uvedeme jen poslední ilustraci pro případ UNIXu, kdy je přítomný link `csplain -> virtex`. Pak vyvolání TEXu s formátem `csplain.fmt` s hlavním vstupním souborem `dokument.tex` lze psát stručně:

```
$ csplain dokument
```

□

• **Jméno úlohy (`\jobname`)** je totožné se jménem hlavního souboru, po odmyšlení případné přípony. Toto jméno je v TEXu použito pro sestavení názvu výstupního souboru a souboru pro protokol zpracování (`\jobname.dvi`, `\jobname.log`). Při `\dump` v iniTEXu vzniká soubor `\jobname.fmt`. Jméno úlohy je jednoznačně určeno, pokud příkazový řádek obsahuje jen jméno hlavního souboru. V ostatních případech je věc trochu sporná, a proto zde uvedeme přesná pravidla.

Jméno úlohy je určeno názvem prvního textového souboru, který se na základě pokynů z příkazového řádku podařilo otevřít ke čtení. Pokud takový soubor neexistuje (například nebylo zvoleno správné jméno souboru a činnost byla předčasně ukončena), zůstává implicitní název jména úlohy „`texput`“.

Pokud je potřeba použít `\jobname` dřív, než dojde k úspěšnému otevření prvního textového souboru, zůstává jméno úlohy ve tvaru „`texput`“. Vyzkoušejte si třeba na příkazovém řádku UNIXu v běžném shellu napsat:

```
$ csplain '\shipout\hbox{abc}    \input dokument'
$ csplain '\edef\jmeno{\jobname} \input dokument'
```

V obou případech zůstává i po úspěšném otevření souboru `dokument.tex` jméno úlohy „`texput`“. Na druhé straně třeba při:

```
$ csplain '\input prvni \input dalsi \input posledni \end'
```

se po úspěšném otevření souboru `prvni.tex` stává jménem úlohy „`prvni`“. Stejný efekt má zápis:

```
$ csplain prvni '\input dalsi \input posledni \end'
```

Soubor `log` je otevírán až v okamžiku, kdy je definitivně známo jméno úlohy. Proto například po:

```
$ csplain '\message{ahoj} \input dokument'
```

je jméno úlohy `dokument`. V souboru `dokument.log` máme text příkazového řádku, ale chybí samotné slůvko „`ahoj`“, které má být výstupem činnosti povelu `\message`. Toto slůvko najdeme jen na terminálu. Ze stejných důvodů nehledejme v `dokument.log` informaci o přetečeném boxu:

```
$ csplain '\hbox to0pt{a} \input dokument'
```

Pokud je ⟨*zbytek příkazového řádku*⟩ prázdný nebo není použit povel `\input`, TEX přechází hned zpočátku do interaktivního režimu. V tomto případě zůstává jméno úlohy „`texput`“. □

• **Hledání vstupních souborů v systému.** I tato záležitost je závislá na konkrétní implementaci TEXu v operačním systému. Obvykle ale platí toto pravidlo: vstupní textový soubor čtený povely `\input` a `\read` je nejprve hledán v „pracovním místě systému“, například v aktuálním adresáři. Není-li tam nalezen, pak je hledán v místě systému podle „systémové proměnné“ `texinput` (například v adresáři, kde jsou instalovány běžné vstupy pro TEX). Pro binární soubory `fmt`, respektive `tfm`, platí podobné pravidlo, tj. nejprve jsou hledány v pracovním místě systému a potom podle systémové proměnné `texfmt`, respektive `textfm`. Výjimky jsou možné a jsou velmi časté, proto je nutné prostudovat dokumentaci k používané implementaci TEXu. □

• **Příklady.** Uvedeme si nyní příklad na práci s primitivem `\input`. Sestavíme makro, které bude uchovávat jméno zrovna čteného souboru ze vstupního proudu. Podobně jako primitiv `\jobname` vrací název úlohy, vytvoříme makro `\currfile`, které bude vracet název zrovna čteného souboru. Kupodivu tato věc není do TEXu implementovaná jako primitiv.

Problém budeme řešit předefinováním primitivu `\input` na makro, které si uchová název svého argumentu. Protože musíme zajistit i návrat k původnímu názvu, sotva

se ukončí činnost jednoho povelu \input, budeme implementovat jednoduchou zásobníkovou strukturu názvů souborů. Při zahájení \input vložíme název stávajícího souboru do zásobníku a při ukončení \input se vrátíme k původnímu názvu ze zásobníku.

```
\def\stack{} % na začátku je zásobník prázdný
\def\push #1{\xdef\stack{#1, \stack}}      % vlož do zásobníku
\def\pop {\expandafter \separe \stack\end} % vylov ze zásobníku
\def\separe #1, #2\end #3{\xdef #3{#1}\xdef\stack{#2}}
\xdef\currfile{\jobname}
\let\oriinput=\input
\def\input #1 {\push\currfile
   \xdef\currfile{#1}\oriinput #1 \pop\currfile}
```

Původní primitiv \input jsme schovali do \oriinput a makro \input jsme definovali s jedním parametrem separovaným mezerou. Aby mohlo být makro funkční, musí být přítomné v hlavním souboru. Makro bohužel nerozlišuje zápis názvu souboru bez přípony (kdy se doplní přípona .tex) a s příponou. To není těžké doplnit, viz stranu 37, makro \setfilename. Je ovšem třeba si uvědomit, že makro s parametrem separovaným mezerou se v mnohém liší od primitivu s parametrem, jež je před vyhodnocením nejprve expandován (viz syntaktické pravidlo ⟨*file name*⟩ v části B). □

V LaTeXu (2_ε) je rovněž předefinován primitiv \input. Původní význam je uložen do \@@input a nový význam je definován takto:

```
\def\input{\@ifnextchar\bgroup\@iinput\@@input}
```

Význam tohoto makra můžeme číst takto: „pokud následuje otevírací závorka skupiny, pracuj jako LaTeXovské \@iinput, jinak pracuj jako primitivní \input“. Znamená to, že pokud je uživatel zvyklý z LaTeXu všude psát kučeravé závorky a napíše \input{⟨*název souboru*⟩}, pak se LaTeX postará o mnoho věcí. Má připraveno vlastní chybové hlášení při nenalezení souboru, název souboru zpracuje podle zapsané či nezapsané přípony, dále zapíše název souboru do pracovního makra \@listfiles a pak konečně pomocí primitivního \input načte soubor. Diametrálně odlišná situace nastane, pokud uživatel napíše jednoduše \input ⟨*název souboru*⟩. Pak se prostě provede primitivní \input a LaTeX nevyvíjí žádné doplňující aktivity. □

Primitiv \input má jednu nepříjemnou vlastnost. Pokud se nepodařilo otevřít soubor s příslušným jménem (například soubor nebyl v systému nalezen), TEX začne komunikovat s uživatelem a nedá pokoj, dokud uživatel nenapíše jméno platného souboru. To nám mnohdy nevyhovuje. Proto vytvoříme makro \softinput, které se chová jako \input, ale pokud soubor neexistuje, vypíše pouze varovné hlášení a jde dál. Využijeme přitom vlastnost primitivu \openin, který nevyvolá chybu ani

při nenalezeném souboru. Programátor maker pak může pomocí \ifeof zjistit, zda byl soubor nalezen či nikoli.

```
\newread\testin
\def\softinput #1 {\let\next=\relax \openin\testin=#1
  \ifeof\testin \message{Warning: the file #1 does not exist}%
  \else \closein\testin \def\next{\input #1 }\fi
  \next}
```

□

• **Textové vstupy pomocí \read.** Tento přístup k souborům se v TEXovských úlohách používá velmi zřídka. Pokud je totiž připraven pracovní soubor pomocí \write, je jeho struktura už koncipována tak, aby soubor mohl být následně přečten v hlavním vstupním proudu pomocí \input. Možnosti \read využijeme například tehdy, když potřebujeme vytvořit makro, které interaktivně komunikuje s uživatelem. Tam se ale zase nejedná o vstupní soubory. Ukázku takového makra najdeme v souboru `testfont.tex`. V příkazovém řádku UNIXu pišme:

```
$ tex testfont
```

TEX se nejprve zeptá uživatele na název testovaného fontu a pak mu nabídne, co s fontem lze dále provádět. Zda se má tisknout pokusný text, vytvořit tabulka fontu, načíst nový font nebo ukončit činnost. Právě pro tuto interakci s uživatelem je použit primitiv \read. □

• **Práce s primitivem \write.** Naproti \read má primitiv \write zcela nezastupitelnou úlohu při řešení křížových odkazů všeho druhu. Jednoduchou ukázku použití tohoto primitivu jsme už uvedli na stránce 57. Tam jsme ovšem narazili na problém, že se nám při přípravě podkladu pro sazbu obsahu expandoval název kapitoly na jednotlivé primitivy. Právě nyní je vhodná příležitost se tomuto problému podrobněji věnovat.

V LATEXu jsou všechna makra, která se mohou vyskytovat v nadpisech a mohou tedy být předmětem expanze v argumentu \write, opatřena zabezpečením proti nežádoucí expanzi. Dělá se to pomocí sekvence \protect. Taková makra jsou v LATEXu označována jako „bytelná“ (robust). Sekvence \protect pak v průběhu zpracování střídá dva významy. Buď má význam \relax (při sazbě běžného textu), nebo má význam \noexpand (v output rutině). Věc je poněkud složitější, protože význam sekvence \protect je v LATEXu měněn na mnoha dalších místech podle situace a podle potřeby. To už ale nebudeme do podrobnosti rozebírat a odkážeme na studium skutečného kódu LATEXu. Uvedeme tedy jen zjednodušeně, jak sekvence \protect pracuje.

Například sekvence \LaTeX je v LATEXu 2.09 definována takto:

```
\def\LaTeX{\protect\p@LaTeX}
\def\p@LaTeX{{\reset@font\rm L\kern-.36em%
    \raise.3ex\hbox{\sc a}\kern-.15em%
    T\kern-.1667em\lower.7ex\hbox{E}\kern-.125emX}}
```

To znamená, že se sekvence `\LaTeX` při běžném použití, kdy `\protect` funguje jako `\relax`, expanduje na `\p@LaTeX` a logo se vysází. Na druhé straně v době expanze argumentu `\write` pracuje `\protect` jako `\noexpand` a ve výstupním souboru najdeme sekvenci `\p@LaTeX` a nikoli obsah těla definice `\p@LaTeX`. To je přesně to, co jsme potřebovali. Samozřejmě nesmíme zapomenout před opětovným načtením souboru nastavit kategorii znaku „`@`“ na 11 (písmeno).

V novém LaTeXu už není nutno měnit kategorii tohoto znaku, protože „bytelné“ příkazy jsou definovány ještě zajímavějším způsobem. Zaměřme se na vlnku v novém LaTeXu. Ta je definována jako `\nobreakspace{}`. Přitom `\nobreakspace` je definována pomocí LaTeXovského `\DeclareRobustCommand` takto:

```
\DeclareRobustCommand{\nobreakspace}{\leavevmode\nobreak\ }
```

Makro `\DeclareRobustCommand` je poměrně složité (znovu odkazuji na vlastní kód LaTeXu), nicméně v závěru jeho činnosti se v našem příkladě provede toto:

```
\def\nobreakspace{\protect nobreakspace␣ }
\def nobreakspace␣ {\leavevmode\nobreak\ }
```

Až na ten tanec s mezerou v identifikátoru se vlastně nestalo nic nového. Makro použilo sekvenci `\protect`, takže výsledné makro `\nobreakspace` se chová jako „bytelné“ (robust) a je možno je použít v nadpisech, které se expandují v argumentech `\write`. Uživatel prakticky v nadpise použije vlnku a do pracovního souboru se po `\write` expanduje sekvence `\nobreakspace␣`. Ta mezera za identifikátorem přitom není při opětovném načítání pracovního souboru podstatná a není ani nutné nastavovat kategorii znaku „`@`“, jak tomu bylo u starého LaTeXu. □

Poučíme se z uvedených triků se sekvencí `\protect` a zkusíme si něco podobného udělat v plainu. Například budeme chtít, aby se nám vlnka v pomocném souboru pro obsah expandovala na `\vlnka`. Vyjdeme z kódu ze strany 57. Potřebný kód nyní znovu přepíšeme a v některých místech jej upravíme.

```
\newtoks\pagetoks \pagetoks={\the\pageno}
\newcount\chapnum \newcount\secnum
\newwrite\toc
\immediate\openout\toc=\jobname.toc

\def\vlnka{\leavevmode\nobreak\ }
\def~{\protect\vlnka {}} \let\protect=\relax
```

```
29
30 \def\sec #1 \par{\advance\secnum by1
31    \bigskip\noindent{\bf \the\chapnum.\the\secnum. #1}\par
32    \nobreak\medskip
33    {\def\protect{\noexpand\noexpand\noexpand}%
34     \edef\act{\write\toc{\noexpand\string\noexpand\tocline
35        {\the\chapnum.\the\secnum}{#1}{\the\pagetoks}}}\act}}
```

Všimneme si, že jsme zde v době činnosti `\edef\act` definovali sekvenci `\protect` jako `\noexpand\noexpand\noexpand`. Proto se například při nadpise „`0~problematice chroustů`“ makro `\act` definuje jako:

```
36 \write\toc{\string\tocline
37   {2.13}{0\noexpand\vlnka {}problematice chroustů}{\the\pageno}}
```

a do souboru se tedy uloží:

```
38 \tocline {2.13}{0\vlnka {}problematice chroustů}{73}
```

□

Povídání o `\protect` už bylo dost. Nyní uvedeme trošku jiné řešení uvedeného problému, které se mi na úrovni plainu velmi osvědčilo. Na `\protect` mohu s klidem zapomenout. Po prvním vytvoření podkladů pro obsah (například v souboru `\jobname.toc`) nahlédnu do tohoto souboru a okamžitě vidím, které TEXovské příkazy autor díla použil v nadpisech. Zkontroluji, zda žádný z těchto příkazů není použit ve výstupní rutině a nastavím jim ve výstupní rutině význam `\relax`. Protože expanze argumentu `\write` probíhá v okamžiku `\shipout`, tj. v okamžiku činnosti výstupní rutiny, zůstanou pak všechny sekvence ve významu `\relax` neexpandovány.

Například autor díla použil v nadpisech sekvence `\uv` pro uvozovky, `\TeX` pro logo a vlnku pro mezeru se zakázaným zlomem. Pak stačí napsat:

```
39 \output={\let\uv=\relax \let\TeX=\relax \let~=\relax
40   \plainoutput}
```

a máme po starosti. Nastavení nového významu je lokální jen ve výstupní rutině. Ještě se musíme postarat o to, aby se nám makra neexpandovala už v době činnosti makra `\sec` při `\edef\act`. Proto naše makro ještě mírně upravíme:

```
41 \newtoks\nadpis
42 \def\sec #1 \par{\advance\secnum by1
43    \bigskip\noindent{\bf \the\chapnum.\the\secnum. #1}\par
44    \nobreak\medskip
45    \nadpis={#1}%
46    \edef\act{\write\toc{\noexpand\string\noexpand\tocline
```

```
47      {\the\chapnum.\the\secnum}{\the\nadpis}{\the\pagetoks}}}
48    \act}
```

Zde jsme využili toho, že `\the\nadpis` se v `\edef` expanduje jen na obsah proměnné ⟨*tokens*⟩ a tyto tokeny se v okamžiku `\edef` dále neexpandují. □

V následujícím příkladě ukážeme makro, které kontroluje číslování poznámek pod čarou. Dříve bylo zvykem číslovat poznámky pod čarou na každé stránce znova od jedničky. Zařídit tuto vlastnost v TEXu není tak jednoduché, protože v době, kdy je třeba do textu odstavce vložit číslo poznámky jako odkaz, ještě nevíme, jak dopadne rozdělení odstavce na stránky. Proto budeme při prvním průchodu zpracování TEXem vkládat poznámky číslované průběžně v celém díle, upozorníme na to uživatele, zaznamenáme do pracovního souboru prostřednictvím `\write` informace o polohách poznámek na jednotlivých stránkách a teprve v druhém průchodu po načtení pracovního souboru budeme číslovat poznámky definitivně.

Návrh makra začneme od popisu struktury pracovního souboru. Označíme jej například jménem `\jobname.foo`. Každá poznámka do něj vloží sekvenci `\onefootnote` a na konci každé strany tam budeme vkládat sekvenci `\newpage`. Pro dvoustránkový text se čtyřmi poznámkami by mohl soubor vypadat takto:

```
49 \onefootnote
50 \newpage
51 \onefootnote
52 \onefootnote
53 \onefootnote
54 \newpage
```

což bude znamenat, že na první straně je jedna poznámka a na druhé tři. Pro načtení takového souboru budeme pracovat se dvěma registry typu ⟨*number*⟩.

```
55 \newcount\fnum  % globální číslo poznámky v celém textu
56 \newcount\pnum  % číslo poznámky (na každé straně od jedné)
57 \def\onefootnote{\advance\fnum by1 \advance\pnum by1
58   \expandafter\edef\csname f:\the\fnum\endcsname{\the\pnum}}
59 \def\newpage{\pnum=0 }
```

Skutečně, po načtení pracovního souboru budeme mít definovány všechny sekvence tvaru `f:`⟨*globální číslo poznámky*⟩. Obsahem jejich definic je definitivní číslo poznámky v rámci strany.

Nyní se pustíme do definice vlastního makra `\footnote`, které je použito v textu v místě poznámky. Makro se použije bez značky pro odkaz, pouze s textem poznámky.`\footnote{Například takto.}` Pro jednoduchost využijeme původní

makro plainu, ačkoli je skoro nevyhovující (nezmenšuje totiž řez písma). To ovšem není problém, který by nás momentálně zaměstnával.

```
60 \let\orifootnote=\footnote % původní makro plainu pro poznámku
61 \def\footnote{\global\advance\fnum by1
62   \expandafter \ifx \csname f:\the\fnum\endcsname \relax
63         \message{Warning: Footnote \the\fnum\space isn't OK}%
64         \edef\footsig{\the\fnum}%
65   \else \edef\footsig{\csname f:\the\fnum\endcsname}%
66   \fi
67   \write\fout{\string\onefootnote}%
68   \orifootnote{$^{\footsig}$}}
```

Makro pomocí `\ifx` zjistí, zda je sekvence `f:`⟨*globální číslo poznámky*⟩ definovaná. Pokud ano, použije se číslo poznámky odtud. Pokud ne, makro vypíše varování a použije globální číslo poznámky. Pomocí `\write` je do pracovního souboru `\fout` zapsána sekvence `\onefootnote`. V další části makra obsloužíme pracovní soubor:

```
69 \newwrite\fout
70 \fnum=0  \softinput \jobname.foo
71 \fnum=0  \immediate\openout\fout=\jobname.foo
```

Nesmíme zapomenout otevřít soubor `\fout` k zápisu až poté, co byl načten a tím byly využity informace z předchozího průchodu. Zde jsme použili makro `\softinput`, které najdeme na straně 288.

Konečně musíme upravit výstupní rutinu tak, aby po `\shipout` (kdy proběhne příslušný počet zápisů sekvencí `\onefootnote`) byla do pracovního souboru vložena oddělovací sekvence `\newpage`.

```
72 \output={\plainoutput
73   \immediate\write\fout{\string\newpage}}
```

Pokud chceme značit poznámky tak, že první poznámka na straně bude mít jednu hvězdičku, druhá dvě atd., pak stačí předefinovat makro `\onefootnote` z řádku 57:

```
74 \def\onefootnote{\advance\fnum by1 \advance\pnum by1
75   \expandafter\edef\csname f:\the\fnum\endcsname{\ifcase\pnum
76     \or *\or **\or ***\or \dag\or \ddag\else \S\fi}}
```

Je třeba pozměnit ještě řádek 68, protože v tomto případě není potřeba použít matematický exponent. Hvězdičky jsou totiž v textovém fontu mírně posazeny nahoru a jsou už připraveny k přímému použití ve významu značek pro poznámky pod čarou. Na použití křížků (`\dag`) se názory různí. Šikovný programátor maker si už s konkrétními požadavky typografa poradí. □

V závěru této sekce se zaměříme na problematiku kontroly konzistence pracovních souborů. Může se totiž stát, že se ani po několikerém průchodu TEXem obsah pracovního souboru neustálí na konečné podobě. Uvažujme předchozí příklad a představme si, že na jedné stránce jsou tři poznámky pod čarou a odkaz na poslední poznámku je až na posledním řádku stránky, ovšem zatím má tvar jediné hvězdičky. V dalším průchodu bude mít tento odkaz tvar tří hvězdiček, ale to se už sazba nevejde do posledního řádku a odkaz se objeví jako první poznámka na další straně. Pak by ale odkaz měl mít jen jednu hvězdičku. Pokud se tak stane, poznámka se nám v dalším průchodu vrátí zpět jako třetí poznámka na původní stranu, ale s jednou hvězdičkou. A tak pořád dokola.

Je to velmi výjimečný fenomén, ale byla by ostuda, kdybychom jej nedokázali uhlídat. Proto náš předchozí příklad obohatíme o práci s jistým „kontrolním součtem“ obsahu pracovního souboru. Nejprve upravíme makro `\newpage` z řádku 59:

```
77 \newcount\chnum \newcount\checksum
78 \def\newpage{\multiply\pnum by\pnum \multiply\pnum by\fnum
79   \advance\chnum by\pnum \pnum=0 }
```

Při čtení souboru `foo` tedy počítáme kontrolní součet `\chnum`. Nyní stačí zapsat tento kontrolní součet do příští verze pracovního souboru a příště jej znovu počítat a navíc kontrolovat se zapsaným údajem. Upravíme tedy řádky 69–71 takto:

```
80 \newwrite\fout
81 \fnum=0  \softinput \jobname.foo
82 \ifnum\checksum=\chnum \else
83    \message{Warning: \jobname.foo may be inconsistent!!}\fi
84 \fnum=0  \immediate\openout\fout=\jobname.foo
85 \immediate\write\fout{\string\checksum=\the\chnum}
```

Když nám i po třetím průchodu bude TEX hlásit, že soubor `foo` není konzistentní, nastal asi problém podobného typu, jako jsme naznačili před chvílí. Pak je potřeba si uchovat dvě po sobě jdoucí verze pracovního souboru a porovnat je například UNIXovým `diff`. Tím máme podchyceno místo, ve kterém se problém odehrává. Dále musíme řešit problém manuálně, například lokální změnou některých parametrů sazby. □

Všimneme si, že i LATEX nás někdy upozorní, že křížové reference mohly být pozměněny, ať tedy pustíme sazbu ještě jednou. Zde je to uděláno poněkud jiným způsobem. V místě `\begin{document}` je čten původní `aux`, pak je vytvářen nový a při dosažení `\end{document}` se nový `aux` zavře (`\immediate\closeout`) a LATEX jej znovu načte pomocí `\input`. Přitom překontroluje, zda je poslední odkaz shodný s posledním odkazem ze starého souboru `aux`. □

7.2. Struktura paměti TEXu

Už jste se setkali s hláškou „`TeX capacity exceeded, sorry`“ a s dodatkem „`If you really absolutely need more capacity, you can ask a wizard to enlarge me`“ ? Po přečtení této sekce se stáváte oním „wizardem“, který ví, kam sáhnout, jaký kus TEXu zvětšit nebo zmenšit a proč. □

TEX pracuje s několika paměťovými poli, jejichž velikost je pevně nastavena. Tento program totiž vznikl v době, kdy nebyly příliš rozšířeny operační systémy, které dokáží přidělit aplikaci potřebné množství paměti dynamicky podle aktuální potřeby (například prostřednictvím systémových volání v UNIXu). TEX tedy při startu alokuje fixní množství paměti a s tou pracuje po celou dobu činnosti. Velmi často tedy není paměť využita zcela. Přitom operační systém nic netuší a nemůže paměť přidělit zrovna potřebnější aplikaci. To bychom mohli TEXu vytknout.

Na druhé straně má toto řešení některé výhody. Především je TEX portovatelný i do operačních systémů, které neumějí dynamicky přidělovat paměť jednotlivým aplikacím. U systémů, které nepracují s procesy paralelně, to ani nemá žádný smysl. Další výhodou může být skutečnost, že TEX si od systému vyžádá paměť jednou pro vždy a neobtěžuje pak systém při své činnosti dalšími paměťovými nároky.

Nevýhoda fixního přidělení paměti je jasná. Musíme program startovat tak, aby mu všechna paměťová pole pro bezchybnou činnost stačila, ovšem také tak, aby nebyly překročeny systémové a hardwarové možnosti výpočetního prostředku, na němž TEX spouštíme. U operačních systémů, které zpracovávají více úloh najednou, nesmíme dovolit aplikaci s TEXem obsadit veškerou dostupnou paměť, protože by si v tu chvíli žádná jiná aplikace ani neškrtla. □

Ve většině implementací se nastavení limitů paměťových polí TEXu provádí staticky před kompilací TEXu ve zdrojovém souboru `tex.web`, respektive přesněji ve změnovém souboru `tex.ch`. Všechny limity TEXu jsou soustředěny do sekcí 11 a 12 WEBovského zdroje. Pokud chceme upravit nějakou velikost paměťového pole TEXu, provedeme tedy změnu v `tex.ch` a kompilujeme TEX znovu do spustitelné podoby. Jak se to dělá, záleží na konkrétní platformě. Obecně se musí programem `tangle` a dalšími prostředky konvertovat WEBovský zdrojový text do zdrojového textu programovacího jazyka, jehož kompilátor je v dané platformě implementován. U `web2c` instalace (pro UNIX) stačí napsat `make TeX`.

V některých implementacích TEXu je možné nastavit požadované rozměry některých paměťových polí při každém startu TEXu prostřednictvím parametrů z příkazového řádku. Příkladem může být balík emTEX, který obsahuje implementaci TEXu a METAFONTu pro DOS, OS/2 a MS Windows. Přesto je v balíku dodáváno několik variant spustitelného programu TEX (`tex.exe`, `tex386.exe`, `htex386.exe`), které se od sebe liší volbou tzv. „paměťového plánu“, o kterém si povíme později.

Jednotlivé varianty TEXu v tomto balíku mohou nastavovat paměťové požadavky jen podle svých možností. Třeba s variantou `tex.exe` příliš díru do světa neuděláme, zatímco s variantou `htex386.exe` můžeme jít až na samotné hranice možností současných procesorů řady INTEL. Pro ilustraci, hlavní paměťové pole TEXu může alokovat v `htex386.exe` velikost až 32 MB paměti, zatímco u `tex.exe` nejvýše 256 kB. Zato nám `tex.exe` poběží bez odkládání na disk i na přístrojích typu XT s procesorem 8086 a se zkostnatělým segment/offset adresováním. □

Pokud generujeme iniTEXem formát, nesmíme zapomenout, že soubor `fmt` je binárním obrazem paměťových polí TEXu v okamžiku povelu `\dump`. Tento formát je možné použít jen takovým TEXem, který má všechna paměťová pole nastavena na stejné velikosti, jaké byly použity v době iniTEXu. Pokud nedodržíme toto pravidlo, dočkáme se zprávy `Fatal format file error, I'm stymied`. TEX o sobě prohlašuje, že je zaslepen a že se v předloženém formátovém souboru nevyzná.

Na druhé straně jsou formátové soubory `fmt` mnohdy přenositelné i mezi různými platformami, na kterých běží TEX. Stačí splnit jediný požadavek: jednotlivá paměťová pole TEXu nastavit v obou platformách na stejnou velikost. □

Pokud volíme `\tracingstats=1`, pak se v `log`u dozvíme nastavení většiny limitů v konkrétní instalaci TEXu. Zároveň vidíme, jak jsme se k těmto limitům přiblížili při zpracování našeho dokumentu. Následující tabulka shrnuje všechny limity TEXu. V levém sloupci tabulky jsou názvy limitů podle WEBovského zdrojového textu TEXu. V prostředním sloupci je označení příslušného limitu podle výpisu v `log`u po kladném `\tracingstats` a v pravém sloupci je stručné vysvětlení. Podrobnější vysvětlení významu jednotlivých názvů uvedeme později.

mem_max	`words of memory of *`	hlavní paměť TEXu
stack_size	`stack positions out of *i`	počet vstupních proudů
nest_size	`stack positions out of *n`	počet sémantických úrovní
param_size	`stack positions out of *p`	současně čtené parametry
buf_size	`stack positions out of *b`	buffer pro vstupní řádky
save_size	`stack positions out of *s`	použije při ukončení skupiny
font_max	`for fonts, out of - for *`	počet fontů
font_mem_size	`font info out of *`	společné místo pro data fontů
pool_size	`string characters out of *`	společné místo pro řetězce
max_strings	`string out of *`	počet řetězců
hash_size	`control sequences out of *`	počet řídicích sekvencí
trie_size	`iniTeX`	společné místo pro `\patterns`
trie_op_size	`iniTeX: ops out of *`	počet čísel vzorů v `\patterns`
max_trie_op	`iniTeX`	čísla vzorů pro jeden jazyk
hyph_size	`hyph. exceptions out of *`	počet výjimek v `\hyphenation`
file_name_size		délka názvu souboru
max_in_open		počet vstupních souborů
dvi_buf_size		výstupní buffer

Pokud je v prostředním sloupci slovo `iniTeX`, dozvíme se odpovídající údaj ze souboru `log` pouze při iniTEXu. Pokud je prostřední sloupec prázdný, dozvíme se příslušný údaj pouze pohledem do zdrojového kódu TEXu. □

Zkuste se podívat do výpisu v `log`u při kladném `\tracingstats`. Dozvíte se, jak jsou nastaveny jednotlivé paměťové oblasti v té verzi TEXu, kterou používáte. Ovšem ta čísla nám, jako programátorům, moc neřeknou. Není tam uvedeno, *v jakých jednotkách* jsou velikosti uvedeny. Nás by zajímalo, kolik ta legrace zabírá bytů, neboli bajtů. Této jednotce rozumíme a víme, kolik nás to stojí, když běžíme kupovat nový paměťový čip. V obchodě se vyjadřujeme pomocí jednotek MB, ovšem při rozboru jednotlivých paměťových polí TEXu si (naštěstí) vystačíme s jednotkou B (byte). K pochopení problematiky převodu mezi jednotkami použitými v TEXu a jednotkou B je potřeba zmínit paměťové typy TEXu *memory_word*, *halfword* a *quarterword*.

Typ *memory_word* (budeme značit písmenem m) používá TEX k alokaci jednoho paměťového místa ve své hlavní paměti. Požaduje toto minimum: (1) Musí se tam vejít čtyřbytový typ integer nebo (2) dva údaje typu *halfword* nebo (3) čtyři údaje typu *quarterword*. Typ *halfword* (označíme h) je použit jako ukazatel do hlavní paměti nebo třeba pro datový typ token. Rezervujeme-li tedy pro něj 2 B, bude hlavní paměť omezena velikostí $65\,536\,m$ (tj. $2^{16}\,m$). Konečně typ *quarterword* (označíme q) používá TEX většinou pro uchování jednoho znaku a vystačíme si s velikostí 1 B. Údaje o požadavcích na velikosti h a q jsou deklarovány ve WEBovské sekci 110 pomocí proměnných *max_halfword* a *max_quarterword*.

Pokud se smíříme s velikostí hlavní paměti jen $65\,536\,m$, pak vystačíme s $m = 4$ B, $h = 2$ B a $q = 1$ B. Toto je nejmenší možný paměťový plán TEXu a je použit například v `tex.exe` pro procesory 8086. S omezeními z toho plynoucími (zvláště na velikost hlavní paměti) se dá vystačit při běžné práci s plainem nebo se starým LATEXem. Nový LATEX se už do tak malé hlavní paměti nevejde. Knuth navrhuje i jiný paměťový plán: $m = 5$ B, $h = 2{,}5$ B a $q = 1$ B. Zde při použití čtyř údajů typu q v jednom m budeme mít nevyužité místo. To nám nevadí. Maximální použitelnost hlavní paměti se nám ale zvětšila na čtyřnásobek. Horší je, že většina překladačů nedovede pracovat s datovým typem velikosti 2,5 B. Například ve `web2c` instalaci čteme v jednom z klíčových hlavičkových souborů `texd.h`:

```
86 typedef integer halfword ;
87 typedef unsigned char quarterword ;
```

Protože deklarátor `integer` alokuje 4 B a do $1\,m$ se mají vejít dva údaje typu h (`halfword`), je ve `web2c` instalaci použit paměťový plán: $m = 8$ B, $h = 4$ B a $q = 1$ B. □

V závěrečné části rozebereme jednotlivé názvy limitů z naší tabulky podrobněji. Vpravo od názvu je pro ilustraci uvedena hodnota z instalace `web2c` a je tam též

uvedena jednotka B, m, h, q nebo i. Poslední z nich označuje velikost paměti pro datový typ `integer`, což bývá obvykle rovno 4 B. S ostatními jednotkami jsme se už seznámili. Pokud víme, jak vypadá paměťový plán našeho TEXu, dokážeme si z těchto jednotek spočítat, kolik nás to „stojí“ v bytech.

• *mem_max* 262 140 m
TEX používá pole, které nazýváme hlavní pamětí TEXu. Zde si TEX hospodaří svými vlastními prostředky. Dokáže si pro data sám alokovat místo v tomto poli a pokud už data nejsou potřeba, sám místo v paměti uvolní. Je to takové dynamické přidělování paměti sám sobě.

Konstanta *mem_max* nemusí označovat absolutní velikost hlavní paměti. Ve zdrojovém kódu se pracuje ještě s konstantou *mem_min*, která určuje dolní hranici pole. Ta je obvykle rovna nule. Počet alokovatelných míst velikosti m v hlavní paměti pak je přesně roven $mem\_max - mem\_min + 1$. V případě standardní instalace `web2c` se jedná o hodnotu 262 141 m.

TEX ukládá do hlavní paměti posloupnosti tokenů (tj. například těla definic maker). Dále tam ukládá tiskový materiál, který je kompletován do boxů a který je z paměti uvolněn až při tisku strany pomocí `\shipout`. Proto jsme schopni překročit limit hlavní paměti buď použitím obrovského množství rozsáhlých definic, nebo podržením rozsáhlého množství tiskového materiálu v paměti bez provedení `\shipout`. □

• *stack_size* 300 (2 q + 4 h)
Zásobník s názvem *input_stack* o velikosti *stack_size* obsahuje informace o otevřených *vstupních proudech*. Základní vstupní proud přichází z řádku vstupního souboru. Pokud se tam objeví makro, které se má expandovat, je založen nový vstupní proud, který je ztotožněn s tělem definice makra. Pokud v tomto těle je makro, které se má expandovat, je založen další vstupní proud.

Jestliže došlo k nějaké chybě při zpracování, TEX vykreslí na obrazovku do řádků stav jednotlivých vstupních proudů. V místě chyby jsou řádky roztrženy na dva, aby bylo vidět, kde se zpracování přerušilo. Řadový uživatel většinou rozumí jen tomu poslednímu řádku, který je shodný s právě čteným řádkem. Další vstupní proudy, které jsou nad tímto řádkem, ukazují stav expanze vnitřních maker. Takový výpis nejlépe ilustruje, co to je vstupní proud.

Počet vypsaných vstupních proudů při chybovém hlášení můžeme regulovat hodnotou registru `\errorcontextlines`.

Pokud se má provést expanze makra, které je napsáno v definici jako poslední token těla definice, je nejprve uzavřen vstupní proud, který odpovídal stávajícímu tělu definice, a pak je teprve zahájen nový vstupní proud, který čte z těla definice nového makra. TEX tedy v takovém případě šetří zásobník vstupních proudů.

Překročení kapacity *stack_size* dosáhneme snadno rekurzivním voláním makra. Například: `\def\a{\a b} \a`. Pokud bychom v tomto příkladě do těla definice `\a` nenapsali `b`, bylo by makro `\a` na konci těla definice a nový vstupní proud by se otevíral na stejné úrovni jako původní starý. Například při `\def\a{\a} \a` máme věčný cyklus, který nevytváří žádné nové paměťové požadavky. Při `\def\a{b\a} \a` se nepřekročí *stack_size*, ale *mem_max*, protože se zaplní horizontální seznam znaky bbbb... □

• *nest_size* $40\,(4\,i + 2\,h)$

Hlavní procesor TEXu, jak víme ze sekce 3.4, pracuje v jednom ze šesti módů (vertikální, horizontální, matematický, vše ve dvou variantách). Po ukončení módu M se TEX musí umět vrátit do módu, ve kterém byl před vstupem do módu M. K tomu potřebuje zásobník, jehož velikost je nastavena na *nest_size*. Uvedenou hodnotu 40 přesáhneme například tehdy, pokud napíšeme dvacet do sebe vnořených konstrukcí typu `\hbox{\vbox{\hbox{\vbox{...}}}}`. □

• *param_size* $60\,h$

Když TEX načítá parametr, ukládá čtenou posloupnost tokenů do hlavní paměti. Pak si ale musí poznačit, kde tato posloupnost v hlavní paměti začíná. Právě k tomu slouží pole o velikosti *param_size*.

Víme, že jedno makro může pracovat maximálně s devíti parametry, takže na *param_size* nejsou obvykle kladeny velké nároky. Teoreticky se ale může stát, že makro s devíti parametry volá ve svém těle další makro s devíti parametry. V tu chvíli už potřebujeme, aby *param_size* bylo rovno aspoň číslu 18. □

• *buf_size* 3 000 B

TEX pracuje s tzv. bufferem, který obsahuje jednotlivé vstupní znaky. Při načtení dalšího řádku ze souboru uloží celý tento řádek do bufferu. Ovšem nemusí jej klást od začátku bufferu. Pokud je v předchozím souboru například řádek s povelem `\input soubor.tex` dlouhý n znaků, je buffer obsazen po celou dobu čtení řádků ze `soubor.tex` těmito n znaky a za nimi jsou připojovány a odpojovány znaky z jednotlivých řádků souboru.

Na začátku tohoto bufferu je vždy obsah příkazového řádku. Pokud má tento řádek třeba 16 znaků a je čten jediný soubor s maximální délkou řádku 50 znaků, bude maximální využití `buf_size` 66 B.

Prostor v bufferu je TEXem rovněž využíván při sestavování názvu řídicí sekvence při činnosti `\csname...\endcsname`. Pokud máme dlouhé řídicí sekvence, nebo máme dlouhé řádky obsahující `\input` a několikrát do sebe vnořené, můžeme překročit kapacitu parametru *buf_size*. □

• *save_size* $4\,000\,m$

Jakmile TEX vstoupí do skupiny `{...}` a v ní provede nějaké lokální přiřazení do

registru, který má nějakou jinou globální hodnotu, musí si TEX zapamatovat tuto globální hodnotu, aby se po opuštění skupiny mohl ke globální hodnotě bezproblémově vrátit. K tomu využívá zásobník o velikosti *save_size*.

Popsanou činnost TEXu lze sledovat při nastavení registru `\tracingrestores` na kladnou hodnotu. Pak v souboru `log` vidíme veškeré aktivity TEXu spojené s restaurováním původních hodnot lokálně přepsaných registrů.

Každý údaj, který je potřeba si zapamatovat pro pozdější restaurování původní hodnoty, obsadí dvě pozice velikosti m v zásobníku. V první pozici jsou údaje o typu přiřazení a odkaz na přiřazovaný registr. V druhé pozici je odkaz do hlavní paměti TEXu, kde je uschována vlastní hodnota registru, kterou bude potřeba po uzavření skupiny obnovit. Znamená to, že při velikosti zásobníku 4 000 jej celý zaplníme, pokud lokálně přiřadíme 2 000 maker a registrů všech možných typů, které by bylo potřeba po ukončení skupiny restaurovat. □

• *font_max* 255 (4 *q* + 4 *h* + 12 *i* + 3 B)
Maximální množství fontů použitelných v jednom dokumentu. Uvedenou hodnotu 255 nelze zcela jednoduše zvýšit. TEX je totiž vybaven pouze algoritmy, které do `dvi` zapisují jen jednobytové číslo fontu. Kdybychom chtěli využít vlastnosti formátu `dvi` a používat třeba čísla fontu ve dvou bytech, musíme v TEXu některé algoritmy upravit. To je uděláno například v `htex386.exe` z balíku emTEX, který dovolí použít 759 fontů současně.

Množství paměti rezervované pro údaj o jednom fontu (v závorce) je pouze odhad. Přesněji viz WEBovskou sekci 549 zdrojového kódu TEXu. Hlavní data fontu jsou ukládána jinam — do společného pole velikosti *font_mem_size*. □

• *font_mem_size* 100 000 *m*
Pole této velikosti obsahuje data všech načtených fontů. Jakmile TEX zavádí nový soubor `tfm` do paměti, data ukládá na ještě neobsazené místo do tohoto pole. Kapacitu pole překročíme tehdy, pokud budeme zavádět velké množství fontů s rozsáhlými metrikami.

Na tomto místě je dobré upozornit na to, že originální plain byl v souvislosti se zaváděním fontů zaměřen na optimalizaci rychlosti a ne paměti. Proto je už v čase iniTEXu zavedeno množství fontů, které navíc vůbec nemají identifikátor (viz `\preloaded` v části B). Pokud toto zavedení z formátu plainu zrušíme, ušetříme značné množství paměti pro své fonty. Můžete si všimnout, že fonty `\preloaded` už v csplainu nejsou. □

• *pool_size* 124 000 B
Veškeré textové řetězce, se kterými TEX pracuje, jsou uloženy v poli označovaném jako `pool`. Po startu iniTEX naplní toto pole výchozími hodnotami ze souboru `tex.pool` a teprve potom je schopen s uživatelem komunikovat. V tomto souboru

má veškerý svůj vyjadřovací slovník všech chybových hlášení a nápověd. Pokud program potřebuje vytisknout na terminál nějaký text, není v místě kódu programu tento text uložen fyzicky, ale je tam jen odkaz na místo v `poolu`. Výjimku tvoří jen několik základních hlášení typu „nenašel jsem soubor `tex.pool`“.

V `poolu` jsou kromě (někdy i veselých) hlášení určených uživateli i všechny identifikátory primitivních řídicích sekvencí a všechna klíčová slova, která má TEX rozpoznávat na vstupu. V průběhu činnosti iniTEXu se `pool` plní novými identifikátory maker. Při `\dump` se `pool` (společně s dalšími paměťovými poli) ukládá do formátu `fmt`. Proto TEX načítající formát nemusí hledat soubor `tex.pool`. Veškerý vyjadřovací slovníček má totiž uložen v `poolu` ve formátu.

Pokud pracujeme s TEXem bez načtení formátu, pak hlášení o „`string characters`“ v `logu` (při kladném `\tracingstats`) ukazuje nikoli plnou velikost `poolu`, ale pouze použitelnou velikost, která je zmenšena o základní TEXovský slovník, který byl načten ze souboru `tex.pool`. Při práci s TEXem po načtení formátu je znovu toto hlášení změněno a ukazuje použitelnou velikost „zbytku“ `poolu` po uložení základního slovníku a slovníku použitého formátu. □

• *max_strings* 15 000 *i*

TEX klade do `poolu` jednotlivé řetězce těsně za sebou bez oddělovací značky. Aby se v tom vyznal, pracuje ještě s jedním polem, kde má sekvenčně za sebou ukazatele na začátky řetězců v `poolu`. Vlastní řetězec je pak v kódu programu reprezentován odkazem do tohoto pole ukazatelů. Tam TEX zjistí jednak, kde v `poolu` řetězec začíná a jednak délku řetězce. Tuto druhou kvantitu zjistí TEX snadno podle toho, kam ukazuje následující ukazatel v poli ukazatelů.

Konstanta *max_strings* tedy udává velikost pole ukazatelů do `poolu` a ve svém důsledku maximální použitelný počet řetězců v programu. Nás toto číslo zajímá v souvislosti s množstvím různých identifikátorů, které můžeme při definování maker použít. Shodný řetězec (třeba použitý i v jiném významu) není do `poolu` nikdy znovu zanesen. Takže se šetří jednak `pool`em samotným, jednak polem ukazatelů do `poolu`.

Podobně jako v případě `poolu`, není v souboru `log` (při kladném `\tracingstats`) celková kapacita počtu řetězců, ale už jen údaj zmenšený o slovník z `tex.pool` (při iniTEXu), nebo o slovník z formátu. Například má-li TEX místo pro 15 000 různých řetězců, pak po načtení `tex.pool` (iniTEXem) zbývá ještě prostor pro 13 705 různých řetězců a po načtení formátu `csplain` se prostor zmenší na 12 983 možných dalších řetězců. □

• *hash_size* 9 500 (2*m*)

Toto číslo určuje maximální počet víceznakových řídicích sekvencí, které lze v TEXu použít. TEX samozřejmě uloží text identifikátoru do `poolu` a zanese o tom údaj do

pole ukazatelů (viz *max_strings*). Ovšem v případě identifikátoru řídicí sekvence to nestačí. TEX si o takovém identifikátoru potřebuje zaznamenat další informace.

Je potřeba si poznamenat především význam řídicí sekvence (zda se jedná o makro, font či třeba sekvence z `\chardef`). Tato informace ve formě vhodného odkazu se ukládá do takzvané „tabulky ekvivalencí“ a jedno její místo rezervované pro jednu řídicí sekvenci má velikost $1\,m$.

Další informace, které si TEX o nové řídicí sekvenci ukládá, souvisí s možností pozdějšího rychlého vyhledání této řídicí sekvence podle čerstvě načteného identifikátoru. K tomuto účelu pracuje s tzv. „hledací tabulkou“. Z kontrolního součtu hledaného identifikátoru pak velmi rychle v této tabulce identifikátor najde. Algoritmus hledání se jmenuje *hash table algorithm* a podle toho má tato proměnná jméno. Velikost tabulky odpovídá maximálnímu množství různých řídicích sekvencí, které TEX umí vyhledat, a je tedy rovna *hash_size*. Pro jednu řídicí sekvenci je zde rovněž rezervováno místo $1\,m$. □

• *trie_size* $30\,000\,(2\,h+q)$
Vzory dělení slov jsou při načítání iniTEXem baleny do zajímavé provázané struktury dat, které se říká `trie`. Každá pozice této struktury obsahuje ukazatel na další pozici, dále má odkaz na tzv. číslo vzoru (viz *trie_op_size*) a konečně nese jeden znak vzoru. Není ale účelem této knihy napsat „TEX: the program naruby“ (srovnej [5]), takže to nebudeme dále rozebírat.

Kolik místa zaberou vzory dělení jednoho konkrétního jazyka zjistíme z výpisu souboru `log` po načtení odpovídajících `\patterns` iniTEXem. Z toho si můžeme udělat představu, jak velké jsou naše paměťové nároky například při požadavku zavedení `\patterns` více jazyků. Pro ilustraci, české vzory dělení (podle pana Ševečka) jsou uloženy v textovém souboru velikosti 28,9 kB a po načtení iniTEXem zabírají v poli `trie` 4 579 pozic. Vidíme, že při nastavení *trie_size* na hodnotu 30 000 se nám do TEXu vejde zhruba 6 vzorů dělení srovnatelné složitosti, jako pro náš jazyk. Vzory dělení pro americkou angličtinu (které jsou součástí plainu) zaberou nepatrně více: 6 075 pozic v paměti. □

• *trie_op_size* $751\,h$
Každý jednotlivý vzor má v `\patterns` údaj o hodnotách zlomu (čísla dělení 0 až 9 mezi každým znakem vzoru a vpředu a vzadu, viz sekce 6.3). Pro n znakový vzor je tímto údajem $n+1$ ciferné číslo. Toto číslo nazýváme *číslem vzoru*. Několik různých vzorů může mít stejné číslo vzoru. Počet *různých* čísel vzoru je v souboru `log` po iniTEXu označen jako počet „`ops`“. Čísla vzoru jsou chytře komprimována a dvě stejná čísla vzoru nejsou nikdy kladena do paměti vedle sebe. Proto stačí pro jedno číslo vzoru velikost paměti $1\,h$ (nebo dokonce $1\,q$, viz *max_trie_op*). Pole, kam TEX ukládá komprimované informace o číslech vzorů, má maximální rozměr *trie_op_size*.

Pro český jazyk je ve vzorech `\patterns` použito pouze 63 čísel vzorů a třeba pro anglické vzory dělení (z plainu) je iniTEXem alokováno 181 čísel vzorů. □

• *max_trie_op* ovlivní paměťový plán pole `trie`
Tato konstanta označuje maximální počet čísel vzorů pro jeden jazyk. Pokud ji nastavíme na hodnotu 255 (maximální hodnota typu q) a žádný jazyk tuto hodnotu nepřesáhne, TEX použije úspornější paměťový plán pro pole `trie` (viz *trie_size*). Jednotka paměti tohoto pole pak je pouze 1 h+2 q. Toto byl původní Knuthův záměr a pro anglický, český i slovenský jazyk je vyhovující. Ukázalo se ale, že omezení 255 různých čísel vzorů na jeden jazyk je nevyhovující pro němčinu. Proto dnes vesměs všechny implementace TEXu používají poněkud méně úsporný paměťový plán pro pole `trie`, ovšem dovolí více variant čísel vzorů pro jeden jazyk. □

• *hyph_size* 607 ($i + h$)
Tato proměnná označuje maximální počet slov, který je možno použít ve slovníku výjimek. Jednotlivá slova si TEX ukládá do `poolu`, a informace o místech dělení do hlavní paměti. Dále TEX potřebuje tabulku, která obsahuje jednak odkazy do pole ukazatelů do `poolu` a jednak odkazy do hlavní paměti na informace o místech dělení. Tato tabulka má velikost *hyph_size*.

• *file_name_size* 255 B
Maximální délka názvu souborů, se kterými TEX pracuje, je věc závislá na konkrétním operačním systému. Do této délky se započítávají i názvy případných adresářů. □

• *max_in_open* 15
Maximální počet současně otevřených vstupních souborů. Například po `\input a1` z příkazové řádky je otevřen jeden soubor, po `\input a2` ze souboru `a1.tex` jsou už otevřeny dva soubory a po `\read` ze souboru `a2.tex` se otevírá třetí soubor. □

• *dvi_buf_size* 16 384 B
Zápis do `dvi` se neprovádí po jednotlivých bytech, ale tento výstup je ukládán do bufferu velikosti *dvi_buf_size*. Buffer je rozdělen na dvě stejně velké poloviny. Jakmile je některá polovina zaplněna, je celá naráz zapsána do výstupního `dvi`.

Velikost tohoto bufferu neovlivňuje nikterak kapacitní možnosti TEXu. Větší buffer může pouze urychlit zápis do souboru, a tím i činnost TEXu. Zde se tedy TEX sám stará o jednoduchou „cache“ nad výstupním souborem. □

7.3. Formát metriky fontu tfm

Poslední dvě sekce této knížky — formát vstupní metriky tfm a výstupního souboru dvi — obsahují už poměrně dost technické záležitosti. Může se ovšem stát, že se to bude někomu hodit. Rozhodně znalost těchto formátů umožní čtenáři lepší pochopení vlastností TEXu samotného.

Binární formát tfm lze dešifrovat do čitelné textové podoby pomocí programu tftopl. V této podobě můžeme dělat úpravy a výsledek zpět převést do binárního formátu pomocí programu pltotf (viz například [7]).

Informace jsou v tfm členěny do paměťových slov délky 4 B. Každý tfm má tedy délku souboru v bytech dělitelnou čtyřmi. V jednom paměťovém slově může být uložena informace o rozměru ve formě *racionální číslo*×*jednotka*. Zápis racionálního čísla je interpretován v pevné řádové čárce, kterou je možno si představit uprostřed druhého bytu, tj. za dvanáctým bitem. Označme hodnoty jednotlivých bytů (zleva doprava) písmeny a, b, c, d. Pak odpovídající kladné racionální číslo má hodnotu $2^4\,a + 2^{-4}\,b + 2^{-12}\,c + 2^{-20}\,d$. Záporné číslo se ukládá jako dvojkový doplněk.

Toto racionální číslo je dále násobené jednotkou. Jednotka je implicitně rovna parametru *design_size*, což označuje velikost, ve které byl font navržen, a tudíž v této velikosti font nejlépe vypadá. Pokud uživatel žádá font v jiné velikosti (klíčovým slovem at nebo scaled), použije TEX jednotku odpovídajícím způsobem upravenou. Veškeré rozměrové údaje TEX při čtení metriky přepočítává podle této jednotky a dále s ohledem na to, že sám pracuje s racionálními čísly s pevnou řádovou čárkou umístěnou přesně mezi druhým a třetím bytem ve čtyřbytovém paměťovém slově.

Existují dvě výjimky, kdy TEX nenásobí zaváděné racionální číslo jednotkou, ale pouze upravuje řádovou čárku: (1) Parametr *design_size* samotný. Ten je vyjádřen v jednotce pt. (2) Parametr \fontdimen1, určující sklon písma, je bezrozměrný údaj interpretovaný takto: Pokud posuneme bod sazby nad účaří o jednu nějakou jednotku, pak jej musíme posunout ještě doprava o \fontdimen1 stejných jednotek, abychom respektovali sklon písma.

Ostatní informace v tfm, které nevyjadřují rozměr, jsou většinou baleny do jednotlivých čtyřbytových slov po skupinách, jak uvidíme za chvíli. Paměťová slova (tj. 4 B) budeme v dalším textu označovat písmenem w. □

Soubor tfm je rozdělen na úseky, které na sebe bezprostředně navazují.

- První úsek délky 6 w obsahuje informace o velikosti ostatních úseků.
- Hlavička fontu, délka lh.
- Úsek znaků ve fontu, délka $ec - bc + 1$.
- Data o šířkách znaků, délka nw.

- Data o výškách znaků, délka *nh*.
- Data o hloubkách znaků, délka *nd*.
- Data o italických korekcích znaků, délka *ni*.
- Ligační a kerningový program, délka *nl*.
- Data kerningových párů, délka *nk*.
- Informace o větvení typu `extensible`, délka *ne*.
- Parametry fontů `\fontdimen`, délka *np*.

Každý úsek může být prázdný. Výjimkou je samozřejmě první úsek a rovněž hlavička fontu musí mít aspoň 2 w. Jinak není metrika TEXem akceptovaná. □

• **První úsek** obsahuje délky dalších úseků, které jsou zapsány jako dvoubytový integer bez znaménka. Údaje jsou baleny po dvou do jednoho w:

- 1. w: délka celé metriky a *lh*
- 2. w: *bc* (nejnižší kód znaku ve fontu) a *ec* (nejvyšší kód znaku).
- 3. w: *nw* a *nh*.
- 4. w: *nd* a *ni*.
- 5. w: *nl* a *nk*.
- 6. w: *ne* a *np*.

Údaje *nw*, *nh* atd. vyjadřují délku příslušných úseků v jednotce w. Je-li tedy například $nw = 6$, úsek s daty o šířkách znaků je velký 6 w, tj. 24 B. □

• **Hlavička fontu** musí obsahovat v prvním údaji délky w kontrolní součet fontu. Tento údaj TEX zapisuje do výstupního souboru `dvi` a je zanesen též ve fontech formátu `pk`. Tiskové ovladače mohou podle tohoto údaje kontrolovat, zda načtený font formátu `pk` skutečně odpovídá metrice fontu, kterou TEX použil při sazbě.

Druhé slovo hlavičky musí obsahovat údaj *design_size*, o němž už zde byla řeč. Další slova hlavičky jsou nepovinná a TEX je při zavádění fontu ignoruje. Bývají tam obvykle napsány nezávazné informace o fontu v textové podobě. □

• **Úsek znaků ve fontu.** Každý údaj v tomto úseku délky w obsahuje informace o jednom znaku počínaje znakem s kódem *ec*. Pokud jsou ve fontu nevyužité kódy mezi kódem *ec* a *bc*, je odpovídající údaj nulový.

V jednom údaji délky w jsou baleny informace o znaku formou odkazů podle následující tabulky. (Jednotlivé byty číslujeme zleva doprava.)

- 1. B: Odkaz na šířku znaku do čtvrtého úseku (8 bitů).
- 2. B: Odkaz na výšku znaku (4 bity) a hloubku znaku (4 bity).
- 3. B: Odkaz na italickou korekci znaku (6 bitů) a tzv. *tag* (2 bity)
- 4. B: Tzv. *remainder* (8 bitů), jehož význam závisí na *tag*.

Je-li *tag* = 0, *remainder* není využit. Hodnota *tag* = 1 říká, že znak může zahajovat ligaturu nebo že za ním může být kern. To samozřejmě závisí na následujícím znaku sazby. Údaj *remainder* odkazuje do ligačního a kerningového programu. Je-li *tag* = 2, pak znak má následníka ve fontu ve smyslu sekce 5.3, strana 179. Údaj *remainder* obsahuje pozici následníka ve fontu. Je-li konečně *tag* = 3, pak znak má odkaz na čtyři další znaky (viz rovněž stranu 179) a *remainder* ukazuje do úseku větvení typu `extensible`. □

• **Úseky pro šířky, výšky a hloubky znaků a pro italické korekce** začínají vždy nulovým údajem. Proto nulový odkaz v údaji znaku rovnou znamená, že příslušný rozměr je nulový. Dále v těchto úsecích pokračují za sebou všechny použité rozměry znaků, přičemž v rámci jednoho úseku se dva stejné rozměry v metrice nikdy neopakují. Tím je dosaženo poměrně velké komprese dat i možnosti přímého použití takto strukturovaných informací sázecím programem. □

• **Ligační a kerningový program** je posloupnost povelů s parametry, přičemž jeden povel i s parametry obsadí velikost 1 w. Jakmile se vysází znak (označme jej symbolem α), který má odkaz do ligačního a kerningového programu, TEX se nejprve podívá, jaký znak v sazbě následuje (označme jej symbolem β). Jsou-li α i β ze stejného fontu, TEX vstoupí podle odkazu znaku α do ligačního a kerningového programu a začne vykonávat za sebou jdoucí povely. V těchto povelech je test na znak β nebo tam je značka konce programu `(STOP)`. Pokud TEX narazí na tuto značku, přičemž nebyl úspěšný žádný z testů na znak β, nebude se v sazbě dvojice znaků α, β nijak upravovat. Vyhovuje-li ale znak β testu v některém povelu, TEX upraví sazbu dvojice znaků α, β podle tohoto povelu. V takovém případě je vykonávání dalších povelů v ligačním a kerningovém programu pro dvojici α, β ukončeno.

Každý povel v ligačním a kerningovém programu obsahuje data balená do čtyř bytů:

- 1. B: Údaj *stop*. Určuje, zda se jedná o povel `(STOP)` a další věci.
- 2. B: *next_char* je kód testovaného znaku. Je-li β = *next_char*, uprav dvojici.
- 3. B: Údaj *op* má dvě části: 1 bit: zda kern nebo ligartura, 7 bitů: další věci.
- 4. B: Údaj *remainder* je dolní část odkazu nebo pozice ligatury.

Je-li *stop* = 0, povel nemá značku `(STOP)`. Je-li *stop* = 128 (`"80`), povel obsahuje značku `(STOP)`. To znamená, že se jedná o poslední povel v řadě povelů. Tento povel se ještě provede, ovšem pak TEX vykonávání povelů v ligačním a kerningovém programu pro dvojici α, β ukončí. Samozřejmě, za tímto povelem mohou následovat další povely ligačního a kerningového programu, do kterých TEX vstupuje podle odkazů jiných výchozích znaků.

Je-li *stop* = 255 (`"FF`), pak *next_char* určuje tzv. *hraniční znak* fontu. Tato věc nebývá ve fontech obvyklá, takže pokud se tím nechcete dál zabývat, přeskočte následující odstavec. Tento text se totiž může na první čtení jevit dost nečitelně.

Hraniční znaky, které způsobují automatickou změnu některých viditelných znaků na krajích slova, se budou asi hodit při sazbě v arabštině nebo v jiném exotickém jazyce. V běžných fontech není tato vlastnost využita.

TEX může pracovat se dvěma hraničními znaky: pravý a levý. Pravý klade (jen teoreticky) na konec každého slova (souvislé posloupnosti znaků ze stejného fontu bez mezer). Levý hraniční znak se klade analogicky na začátek každého slova. Tyto znaky fakticky nejsou sázeny, ale mohou ovlivnit sazbu při vyhodnocování ligatur nebo implicitních kernů s okrajovým skutečně viditelným znakem ve slově. Viditelný znak α se může na začátku slova proměnit v ligaturu nebo před ním může být implicitní kern, pokud levý hraniční znak má ve svém odkazu do ligačního a kerningového programu test na znak α. Rovněž se znak α na konci slova může proměnit v ligaturu nebo za ním může být implicitní kern, pokud tento znak má odkaz do ligačního a kerningového programu, kde je test na pravý hraniční znak. Pravý hraniční znak musí být v úseku ligačního a kerningového programu definován (pomocí $stop = 255$) zcela na začátku programu. Levý hraniční znak pak zcela na jeho konci. Levý hraniční znak navíc má odkaz do ligačního a kerningového programu, aby bylo známo, na jaké znaky se váže ligaturou nebo implicitním kernem. Tento odkaz je ve tvaru $256 \times op + remainder$.

Je-li $stop = 254$ (`"FE`), pak je *op* a *remainder* interpretován jako odkaz do jiné části ligačního a kerningového programu. Odkaz je ve formátu $256 \times op + remainder$. TEX skočí na toto místo ligačního a kerningového programu a tam pokračuje ve čtení povelů. Tato vlastnost umožňuje implementovat ligační a kerningové programy velikosti až 65 tisíc povelů. Přitom z jednobytového odkazu v údaji konkrétního znaku se dá odkázat jen na prvních 256 míst v tomto programu. Tam při velkém množství ligatur a kerningových párů může být nejprve odkaz na vzdálenější místo v programu a teprve tam TEX vykonává povely.

Je-li $op \geq 128$ (při $stop \leq 128$), pak se jedná o povel pro přidání kernu. Je-li v tomto případě $next\_char = \beta$, TEX přidá implicitní kern mezi α a β. Hodnotu kernu získá z úseku s daty kerningových párů z pozice $256 \times (op - 128) + remainder$.

Je-li $op < 128$ (při $stop \leq 128$), pak se jedná o povel k vytvoření ligatury. Je-li v tomto případě $next\_char = \beta$, TEX vloží znak (ligaturu) z pozice *remainder* mezi znaky α, β. Dále ověří, zda původní znaky α, β má ze sazby zlikvidovat podle tohoto algoritmu: Při sudém *op* odebere znak β. Při sudé hodnotě (*op* div 2) odebere znak α. Konečně z údaje (*op* div 4) zjistí, kolik následujících znaků sazby nemá podléhat tvorbě ligatur. Obvykle je v povelu pro ligaturu $op = 0$, takže ze sazby mizí α i β a zůstává tam jen ligatura, která se sama může podle následujícího znaku proměnit v jinou ligaturu. Jiný, komplikovanější příklad: Nechť je $op = 6$. Pak v sazbě zůstává α, za ním je ligatura s kódem *remainder*, dále chybí β a ligatura nevytvoří s následujícím znakem novou ligaturu. □

• **Úsek informací o větvení typu `extensible`.** Pokud má znak $tag = 3$, odkazuje jeho *remainder* do úseku informací o větvení typu `extensible`. Zde najde znak své čtyři následníky pro vytvoření závorky libovolné výšky. Každý byte ve čtyřbytovém slově odkazuje na pozici jednoho následníka ve fontu. První následník má význam horního okraje závorky, druhý tvoří střed, třetí spodní okraj a čtvrtým je vyplněn zbytek. Každý z těchto odkazů (s výjimkou posledního) může být nulový. To potom neznamená odkaz na následníka s pozicí nula, ale vyjadřuje to skutečnost, že příslušná část závorky není použita. □

Poslední úsek v metrice fontu obsahuje postupně jednotlivá `\fontdimen` počínaje `\fontdimen1`. Pokud má font méně než 7 těchto údajů, TEX zbytek doplní nulovými hodnotami. Víme totiž, že 7 údajů `\fontdimen` tvoří minimum potřebné pro sazbu v textovém režimu (horizontálním módu). □

• **Omezení, vyplývající z formátu `tfm`.** Maximální počet znaků v jednom fontu je 256. Maximální délka souboru `tfm` je 256 kB, tj. $(2^{16}-1)$ w. Maximálně 255 znaků může mít svou vlastní jedinečnou šířku znaku. Máme totiž možnost použít 255 nenulových odkazů na šířku (8 bitový odkaz). Nulový odkaz znamená, že znak ve fontu chybí. Znaky mohou mít dohromady maximálně 63 různých nenulových italických korekcí (6 bitový odkaz). Dále mohou mít maximálně 15 různých nenulových výšek a 15 různých hloubek (4 bitový odkaz). To je asi nejvíc svazující omezení. Víme ale, že výška kresby znaku nemusí mít s výškou v metrice nic společného. Výška znaku z metriky navíc velmi často nemá na celkový výsledek sazby žádný vliv. Pouze maximální výška a hloubka v jednom řádku může ovlivnit řádkování, ale velmi často ani tyto údaje řádkování neovlivní. Problémy ale nastanou, když chceme nad každý znak individuálně posazovat akcenty primitivem `\accent`. Pak potřebujeme, aby výšky znaků odpovídaly skutečně výškám kresby.

Maximální počet údajů typu kern/ligatura je omezen pouze maximální délkou metriky. Pokud jsou ostatní úseky malé, pak je možno uložit zhruba 60 tisíc těchto údajů. Počet *různých* hodnot implicitních kernů je omezen na polovinu délky metriky (cca 30 tisíc), protože minimálně stejně dlouhý musí být ligační a kerningový program.

Omezení 256 znaků v jednom fontu může být podstatné při sazbě v některých exotických jazycích (čínština). V poslední době se proto úspěšně rozvíjí projekt Ω, který navazuje na TEX, ale dovolí na vstupu i ve fontech používat 16 bitové znaky (tj. 65 536 různých znaků na vstupu a v každém fontu). □

7.4. Formát výstupního souboru `dvi`

Výstupní soubor `dvi` čtou ovladače jednotlivých výstupních zařízení: ovladač obrazovky = prohlížeč `dvi` a ovladač tiskárny = tiskový program `dvi`. Tyto programy tedy „zviditelní“ práci TEXu v podobě hotové sazby.

Existuje ještě program `dvitype`, který převede všechny informace z binárního souboru `dvi` do čitelné podoby. Pojmem „čitelná podoba“ v tomto případě nemyslíme výsledný vzhled sazby, ale textový výstup, který obsahuje všechny údaje o pohybu bodu sazby a o kódech jednotlivých znaků v relativně srozumitelné podobě. WEBovský zdrojový kód programu `dvitype` se též často používá jako výchozí bod při programování konkrétního ovladače. Zvláště skenovací algoritmy binárního `dvi` se z tohoto programu přebírají beze změny.

Binární formát `dvi` byl navržen tak, aby byly splněny tyto požadavky:

- Délka souboru by měla být v rámci možností co nejmenší.
- Soubor by měly být schopny snadno a rychle číst ovladače výstupních zařízení.
- TEX vytváří `dvi` sekvenčně, tj. nevrací se k jednou zapsané informaci.
- Ovladače mohou číst `dvi` sekvenčně, nebo vyhledávat požadované strany.

Jednotlivé informace jsou v `dvi` uloženy ve formě *povel*, *parametry*, *povel*, *parametry*, atd. *Povel* má vždy délku 1 B. Máme tedy možnost definovat 256 různých povelů. Později uvidíme, že je definováno jen 250 povelů. Každý povel je tedy určen číslem 0 až 249, ovšem my budeme povely označovat slovními přepisy, abychom se v tom vyznali. Tabulku mezi čísly a slovními přepisy uvedeme později.

Každý povel má jednoznačně určeno, kolik parametrů za ním následuje. Některé povely nemusí mít žádný parametr. Uvedeme několik příkladů povelů bez parametru. Povel *set_char_49* znamená, že se má vysázet znak s kódem 49 ze zrovna nastaveného fontu. To číslo 49 není parametr, ale je součástí povelu. Ve formátu `dvi` je skutečně definováno 128 povelů typu *set_char_i*. Jiný příklad: povel *fnt_num_3* znamená, že se má sazba nastavit na font číslo 3. Nebo povel *push* uloží stav bodu sazby do zásobníku a povel *pop* jej vyzvedne.

Uvedeme nyní příklady typických povelů s parametry. Povel *down_1* $a[1]$ posune bod sazby dolů o hodnotu, uvedenou v jednobytovém parametru a. Nebo povel *down_2* $a[2]$ posune bod sazby dolů o hodnotu, uvedenou ve dvoubytovém parametru a. Vidíme, že se zde šetří prostorem. Mohl by být totiž definován jen povel *down_4* $a[4]$, který má čtyřbytový parametr, do něhož se vejde už pořádná velikost posunu bodu sazby dolů. Pokud se ale posunujeme dolů jen „o málo“, TEX šetří s místem a *down_4* nahradí takovým povelem *down*, aby byla výsledná délka `dvi` co nejmenší.

Jiným typickým povelem s parametry je povel pro deklaraci čísla fontu. Abychom mohli použít třeba povel *fnt_num_3*, musíme nejprve deklarovat číslo fontu 3 povelem

$$\mathit{fnt\_def\_1}\ \ k[1]\ c[4]\ s[4]\ d[4]\ a[1]\ l[1]\ n[a+l]$$

kde v závorkách je uvedena délka každého parametru v bytech. Parametr k je číslo fontu (které zatím nebylo deklarováno), c je kontrolní součet z metriky fontu a s udává jednotku pro rozměrové údaje ve fontu. Při nezvětšeném fontu je $s = d$, kde d je *design_size* fontu. Dále a je délka prefixu názvu fontu (tj. údaje o adresářích) a l je délka samotného názvu fontu (například název `csr10` má délku 5). Konečně n obsahuje název fontu včetně případných adresářů. Poznamenejme, že bývá obvykle $a = 0$ a název se uvádí bez adresářů. Pak musí `dvi` ovladač vyhledat font v systému podle smluvených pravidel. Všimneme si, že v případě povelu *fnt_def_1*, má poslední parametr proměnlivou délku závislou na a a l. □

• **Záhlaví souboru `dvi` a použitá jednotka pro rozměrové údaje.** Záhlaví souboru `dvi` se nazývá *preamble* a je uvozeno povelem *pre*.

$$\mathit{pre}\ \ i[1]\ \mathit{numerator}[4]\ \mathit{denominator}[4]\ \mathit{magnification}[4]\ k[1]\ x[k]$$

Údaj i označuje číslo verze `dvi` a už od verze TEX82 má konstantní hodnotu $i = 2$. Dva následující parametry typu `integer` udávají velikost použité jednotky pro všechny rozměrové údaje v `dvi` podle vzorce:

$$\mathit{použitá\ jednotka} = \frac{\mathit{numerator}}{\mathit{denominator}} \times 10^{-7}\,\mathrm{m}$$

Je tedy pamatováno i na jiné sázecí programy, které mohou použít výstup do `dvi`. Programy nastaví *numerator* a *denominator* tak, aby mohly všechny údaje ve výstupním souboru `dvi` zapisovat v takové jednotce, která je jim nejbližší. TEX pracuje s jednotkou `sp` $= 2^{-16}$ `pt`. Proto volí pro zlomek *numerator*/*denominator* vždy stejnou hodnotu 25 400 000/473 628 672.

Všechny ostatní rozměrové údaje mají v souboru `dvi` formát *celé číslo* × *použitá jednotka*. Pokud je *celé číslo* záporné, je zapsáno běžným způsobem jako dvojkový doplněk. Množství bytů rezervovaných pro rozměrový údaj může být v `dvi` různé. To už jsme poznali u povelů *down_i*. Jednotlivé byty se čtou zleva doprava od vyšších řádů k nižším. Tomu se mezi programátory říká formát BigEndian. □

Koeficient dodatečného zvětšení nebo zmenšení rozměrů v `dvi` je dán hodnotou $f = \mathit{magnification}/1000$. TEX implicitně nastavuje $\mathit{magnification} = 1000$, tj. poměr jedna ku jedné. Tento údaj může být v TEXu změněn zápisem do registru `\mag` před výstupem první strany do `dvi`. Dále jej může ovladač ignorovat a považovat za důležitější hodnotu, kterou uživatel specifikuje v příkazovém řádku.

Poslední část záhlaví délky k bytů obsahuje textové informace o okolnostech vzniku tohoto souboru `dvi`. Vlastní sazbu tento údaj nikterak neovlivní. □

• **Členění souboru `dvi` na stránky.** Za záhlavím (preamble) začíná bezprostředně první strana. Každá strana je uvozena povelem:

$$\mathit{bop}\ c_0[4]\ c_1[4]\ c_2[4]\ c_3[4]\ c_4[4]\ c_5[4]\ c_6[4]\ c_7[4]\ c_8[4]\ c_9[4]\ p[4]$$

Toto je jediný případ, kdy se v `dvi` poněkud plýtvá místem. Pro číslo strany je rezervováno 10 čísel bez znaménka, každé má velikost 4 B. TEX do nich zapíše stav registrů `\count0` až `\count9`. V obvyklých dokumentech se používá pro číslo strany jen `\count0` alias `\pageno`. Ostatní čísla jsou rovna nule. Čísla strany zapsaná do těchto parametrů nemusí mít nic společného s číslem strany, které plyne z pořadí jednotlivých stran v souboru `dvi`.

Za povšimnutí stojí poslední parametr p, který ukazuje na pozici v `dvi`, kde lze najít povel *bop* předchozí strany. Jedná-li se o zcela první stranu v `dvi`, je $p = -1$. Například začíná-li první *bop* na padesátém bajtu (byte) v `dvi`, má $p = -1$ a následující *bop* uvozující druhou stranu má $p = 50$. □

Za povelem *bop* následují povely pro vytvoření sazby příslušné strany. Jsou tam deklarace fontů, které ještě nebyly deklarovány, dále povely, které pohybují s bodem sazby, a konečně povely, které přepínají fonty a kladou do sazby znaky podle kódu ve fontu.

Každá strana je ukončena povelem *eop*, který nemá žádné parametry. Za ním bezprostředně následuje *bop* nové strany. Za povelem *eop* poslední strany přichází závěr `dvi`, tzv. *postamble*. Závěr je uvozen povelem *post*. Dále následuje shrnutí deklarací všech použitých fontů a povel *post_post* uzavírá soubor `dvi`:

$\mathit{post}\ p[4]\ \mathit{numerator}[4]\ \mathit{denominator}[4]\ \mathit{magnification}[4]\ l[4]\ u[4]\ s[2]\ t[2]$
$\langle \mathit{deklarace\ fontů} \rangle$
$\mathit{post\_post}\ q[4]\ i[1]\ \langle \mathit{konec} \rangle$

Parametr p ukazuje na *bop* poslední strany a dále jsou zopakovány parametry ze záhlaví (preamble). Figurují tu ale nové parametry: l je výška nejvyšší strany v `dvi` a u je šířka nejširší strany. Tyto údaje většinou ovladače nepotřebují, ale co kdyby. Parametr s udává nároky na použití zásobníku pro uchování a vyzvednutí stavu bodu sazby. O zásobníku se zmíníme později. Konečně t je celkový počet stran v `dvi`.

V místě $\langle \mathit{deklarace\ fontů} \rangle$ jsou zopakovány všechny povely *fnt_def*, které definují čísla použitých fontů. Každé číslo fontu je tedy v `dvi` definováno právě dvakrát. Jednou uvnitř nějaké strany, kde je font poprvé použit, a podruhé v závěru `dvi`.

Povel *post_post* má v q odkaz na polohu povelu *post*. Dále je zopakována verze $i = 2$ ze záhlaví a pak přichází úplný ⟨*konec*⟩. Ten je realizován čtyřmi až sedmi byty s hodnotou 223 ("DF"). Počet těchto bytů je volen tak, aby délka celého souboru dvi byla dělitelná čtyřmi.

Na systémech s přímým přístupen na určité místo ve čteném souboru provede většina dvi ovladačů nejprve skok na konec souboru podle jeho délky. Tam ovladače vyhledají povel *post_post*. Z jeho parametru q zjistí, kde je povel *post*. Za tímto povelem načtou potřebné údaje a dále deklarace použitých fontů. V tuto chvíli ovladače většinou zkontrolují přítomnost těchto fontů v systému a případně i jejich kontrolní součty. Pak začnou pomocí parametrů p odskakovat zpětně dozadu po jednotlivých povelech *bop* až k první straně. Údaje o polohách začátků stran v dvi si poznamenají do paměti. Teprve potom začnou zpracovávat ty strany dvi, které si uživatel přeje. □

• **Pohyb bodu sazby na straně.** Poloha bodu sazby je určena dvěma souřadnicemi (h, v) v kartézském systému souřadnic. Zvětšení h znamená posunout bod sazby doprava a zvětšení v znamená posunout jej dolů. Například povel *down_1* 20 provede vlastně přiřazení $v := v + 20$ a poloha bodu sazby se posune o dvacet jednotek dolů. Nebo *down_1* −20 posouvá bod sazby o dvacet jednotek nahoru.

Na začátku každé strany je nastaveno $(h, v) = (0, 0)$. Předpokládá se, že ovladač umístí tento počátek souřadnic do levého rohu strany 1 in od levého i horního okraje papíru. Srovnejte heslo \shipout v části B.

Pokud se provádí sazba znaku například povelem *set_char_i*, znak se umístí tak, aby se jeho referenční bod kryl s bodem sazby a po vykreslení znaku se bod sazby posune o šířku znaku doprava. K tomu ovladač potřebuje znát šířky všech znaků, které jsou uloženy v tfm. Ovladači může postačit formát pk, protože tam jsou údaje o šířkách znaků (nikoli však o výškách a hloubkách) duplikovány se stejnou přesností jako v tfm.

Ovladač před sazbou každého znaku nebo linky provede zaokrouhlení polohy bodu sazby do jednotky „pixel“, což je jednotka akceptovatelná výstupním zařízením a odpovídá rozlišovací schopnosti výstupního zařízení. Pak je možné vykreslit rastrový obraz znaku do rastru výstupního zařízení jednoznačně. Při sazbě linky povelem *set_rule* nebo *put_rule* se zaokrouhluje šířka a výška linky na jednotky „pixelů“ vždy nahoru, aby byla linka ve výstupu viditelná i tehdy, pokud je podstatně „tenčí“, než šířka jednoho pixelu.

Existují povely, které umožní při používání stejných mezer (například mezislovních) zmenšit délku souboru dvi. Běžně bychom mohli například pro mezery velikosti 218 453 sp (mezislovní mezera fontu cmr10 bez stažení nebo roztažení) psát opakovaně povel *right_3* 218 453. To ale spotřebuje 4 B v souboru dvi na každou mezeru. TEX postupuje chytřeji. Při první mezeře, která se bude v řádku opakovat,

provede povel *w_3* 218 453. Tento povel nejen posune bod sazby o požadovanou velikost doprava, ale navíc uloží do pracovní proměnné w hodnotu parametru. Při dalších výskytech stejné mezery v řádku TEX použije už jen povel *w0*, který nemá parametr. Povel posune bod sazby doprava o hodnotu pracovní proměnné w. To zabere v `dvi` pouze 1 B. Pro oba směry (směr vertikální i horizontální) jsou pro tyto účely vyhrazeny dvě pracovní proměnné. Můžeme tak v jednom řádku nebo sloupci kombinovat dvě velikosti mezer pouze pomocí povelů bez parametru. Pro horizontální směr se jedná o povely *w0* a *x0* (proměnné w a x) a pro vertikální směr se jedná o povely *y0* a *z0* (proměnné y a z). Teprve pro ostatní (často nahodilé) mezery se použije *right_i* nebo *down_i*.

Další úspora místa vzniká tím, že TEX neklade nikdy dvě mezery vedle sebe. Například mezi jednotlivými řádky se bod sazby většinou posune o velikost odpovídající hloubce vysázeného řádku, pak o meziřádkovou mezeru a nakonec o velikost odpovídající výšce následujícího řádku. Podtrženo a sečteno, bod sazby se (většinou) posune o velikost `\baselineskip`. Tento součet TEX uloží do `dvi`. Navíc první takový posun na stránce deklaruje pomocí *y_3* a další posuny o řádek jsou už realizované jednobytovým povelem *y0*.

Stavem bodu sazby rozumíme jeho polohu a jeho možný pohyb při použití povelů *w0*, *x0*, *y0* a *z0*. Stav bodu sazby je tedy určen šesti proměnnými h, v, w, x, y, z. Na začátku každé strany musí ovladač přidělit těmto šesti proměnným nulové hodnoty.

Stav bodu sazby se může kdykoli schovat do zásobníku (pomocí povelu *push*) a později ze zásobníku vyjmout (povelem *pop*). TEX například před sazbou každého řádku provede povel *push*. Pak se bod sazby v řádku obvykle posune někam doprava. Před zahájením nového řádku provede TEX nejprve *pop*. Bod sazby se vrátil na začátek řádku podle stavu uloženého v zásobníku. Dále TEX provede posun bodu sazby dolů na účaří následujícího řádku. Potom povelem *push* uloží stav bodu sazby do zásobníku a začne tvořit další řádek.

Úroveň zaplnění zásobníku musí před koncem strany (povel *eop*) klesnout na nulu. To může ovladač `dvi` kontrolovat. Pokud se tak nestalo, ohlásí ovladač chybu konzistence souboru `dvi`. □

• **Tabulka všech povelů formátu `dvi`.** Uvedeme tabulku povelů včetně jejich parametrů. Nejprve je uvedeno číslo v desítkové a hexadecimální soustavě. Za ním je název povelu a případné parametry. Za pomlčkou je stručné vysvětlení povelu. Pokud jsme už povel zmiňovali dříve, je vysvětlení velmi stručné.

0–127 (`"0-7F`) *set_char_i* — Sazba znaku s kódem i z aktuálního fontu.
128 (`"80`) *set_1* $c[1]$ — Sazba znaku s kódem c (užitečné pro $c > 127$).
129 (`"81`) *set_2* $c[2]$ — Sazba znaku s kódem c (pro $c > 255$, v TEXu nepoužito).
130 (`"82`) *set_3* $c[3]$ — Sazba znaku s kódem c ($c > 65\,535$, v TEXu nepoužito).

131 ("83") *set_4* $c[4]$ — Sazba znaku s kódem c (pro 32 bitové fonty, nepoužito).
132 ("84") *set_rule* $a[4]$ $b[4]$ — Černý obdélník výšky a a šířky b. Dále $h := h + b$.
133–136 ("85-88") *put_i* $c[i]$ $(i = 1, \ldots, 4)$ — Jako *set_i*, ale bez změny h.
137 ("89") *put_rule* $a[4]$ $b[4]$ — Jako *set_rule*, ale bez změny h.
138 ("8A") *nop* — Nic nedělej. Analogie `\relax` v TEXu. Nepoužito.
139 ("8B") *bop* $c_0[4] \ldots c_9[4]\, p[4]$ — Začátek strany.
140 ("8C") *eop* — Konec strany.
141 ("8D") *push* — Uložení stavu bodu sazby do zásobníku.
142 ("8E") *pop* — Vyzvednutí stavu bodu sazby ze zásobníku.
143–146 ("8F-92") *right_i* $c[i]$ $(i = 1, \ldots, 4)$ — Posun bodu sazby doprava. Přesněji $h := h + c$. Povely se pro různá i liší jen velikostí argumentu.
147 ("93") *w0* — Posun bodu sazby doprava o hodnotu w, tj. $h := h + w$.
148–151 ("94-97") *w_i* $b[i]$ $(i = 1, \ldots, 4)$ — Posun bodu sazby doprava o hodnotu b, tj. $h := h + b$. Dále ulož $w := b$ pro použití v povelu *w0*.
152 ("98") *x0* — Posun bodu sazby doprava o hodnotu x, tj. $h := h + x$.
153–156 ("99-9C") *x_i* $b[i]$ $(i = 1, \ldots, 4)$ — Posun bodu sazby doprava o hodnotu b, tj. $h := h + b$. Dále ulož $x := b$ pro použití v povelu *x0*.
157–160 ("9D-A0") *down_i* $a[i]$ $(i = 1, \ldots, 4)$ — Posun bodu sazby dolů. Přesněji $v := v + a$. Povely se pro různá i liší jen velikostí argumentu.
161 ("A1") *y0* — Posun bodu sazby dolů o hodnotu y, tj. $v := v + y$.
162–165 ("A2-A5") *y_i* $a[i]$ $(i = 1, \ldots, 4)$ — Posun bodu sazby dolů o hodnotu a, tj. $v := v + a$. Dále ulož $y := a$ pro použití v povelu *y0*.
166 ("A6") *z0* — Posun bodu sazby dolů o hodnotu z, tj. $v := v + z$.
167–170 ("A7-AA") *z_i* $a[i]$ $(i = 1, \ldots, 4)$ — Posun bodu sazby dolů o hodnotu a, tj. $v := v + a$. Dále ulož $z := a$ pro použití v povelu *z0*.
171–234 ("AB-EA") *fnt_num_i* $(i = 0, \ldots, 63)$ — Následující sazba bude ve fontu č. i.
235 ("EB") *fnt1* $k[1]$ — Následující sazba bude ve fontu č. k (užitečné pro $k \geq 64$).
236 ("EC") *fnt2* $k[2]$ — Následující sazba bude ve fontu č. k (užitečné pro $k \geq 256$).
237 ("ED") *fnt3* $k[3]$ — Následující sazba bude ve fontu č. k (užitečné pro $k \geq 2^{16}$).
238 ("EE") *fnt4* $k[4]$ — Následující sazba bude ve fontu č. k (užitečné pro $k \geq 2^{24}$).
239–242 ("EF-F2") *xxx_i* $k[i]$ $x[k]$ $(i = 1, \ldots, 4)$ — V parametru x délky k je zapsán vzkaz v `dvi` pro konkrétní ovladač. V TEXu použito při implementaci primitivu `\special`. Například *xxx_1* se použije v případě délky vzkazu menší než 255 B. Nebo povel *xxx_4* je rezervován pro velmi dlouhé vzkazy.
243–246 ("F3-F6") *fnt_def_i* $k[i]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a + l]$ — Deklarace čísla fontu k. Povely se pro různá i liší velikostí čísla k. Standardní TEX používá jen *fnt_def_1*, takže v jednom dokumentu může být jen 256 různých fontů.
247 ("F7") *pre* $i[1]$ *numerator*$[4]$ *denominator*$[4]$ *magnification*$[4]$ $k[1]$ $x[k]$ — Záhlaví souboru `dvi` (preamble).
248 ("F8") *post* $p[4]$ *numerator*$[4]$ *denominator*$[4]$ *magnification*$[4]$ $l[4]$ $u[4]$ $s[2]$ $t[2]$ — Závěr souboru `dvi` (postamble).
249 ("F9") *post_post* $q[4]$ $i[1]$ ⟨*konec*⟩ — Úplný závěr souboru `dvi`.

Formát `dvi` byl navržen tak, aby z něj bylo možno provést sazbu i na jednoduchých rastrových výstupních zařízeních bez vlastní inteligence. Proto ve formátu

nenajdeme práci s barvou, nepravoúhlé souřadnice a další věci. Pokud je výstupní zařízení schopno akceptovat i tyto „speciality“, použijeme povel *xxx_i* (`\special`), který nese vzkaz pro ovladač výstupního zařízení. V argumentu tohoto povelu může být kód jazyka výstupního zařízení, například PostScriptu. Tím získáme přístup ke všem možnostem výstupního zařízení, které používáme.

Povel *xxx_i* se též používá například pro zařazování barevných PostScriptových obrázků do sazby. V parametru povelu nejsou obvykle data celého obrázku (to by nám `dvi` obrovsky zmohutnělo), ale je tam jen odkaz na název souboru s daty obrázku. Ovladač PostScriptového zařízení najde podle tohoto odkazu soubor s obrázkem a do výstupního kódu jej zařadí. □

Část B

Reference

1. Slovník syntaktických pravidel

V sekci 3.1 bylo vysvětleno, že hlavní procesor vyžaduje, aby vstupní informace měly přesně definovaný formát ve tvaru *povel*, *parametry*, *povel*, *parametry*. Přitom *parametry* mohou chybět nebo musí souhlasit s tzv. *syntaktickými pravidly*.

Uvedeme jen ta syntaktická pravidla, která jsou použita v této knize. Všechna syntaktická pravidla (syntactic quantities) jsou uvedena v TEXbooku v Backus Naurově formě v kapitolách 24 až 26. Na rozdíl od TEXbooku se vyhneme zápisu tabulek v Backus Naurově formě. Místo toho popíšeme každé syntaktické pravidlo slovy a výklad ilustrujeme na příkladech.

V části A jsme občas psali v ⟨*těchto závorkách*⟩ názvy syntaktických pravidel česky. Například ⟨*maska parametrů*⟩ nebo ⟨*deklarace řádku*⟩. Tato pravidla byla vysvětlena přímo v textu a nebudeme je znovu uvádět. Čtenář může vyhledat vysvětlující text k pravidlům s českými názvy podle závěrečného rejstříku. I v této příloze budeme u výkladu některých syntaktických pravidel pracovat s českými názvy pomocných pravidel (například ⟨*desetinné číslo*⟩) a vysvětlíme jejich význam přímo v textu.

• **Značení.** Pomocí znaků `{`, `}`, resp. `$`, budeme označovat tokeny kategorie 1, 2, resp. 3, přičemž na ASCII hodnotách tokenů nezáleží (*explicitní zápis*). Tyto znaky též mohou znamenat zástupné řídicí sekvence deklarované pomocí `\let` (*implicitní zápis*). Pokud je druhá možnost implicitního zápisu vyloučena, vždy na to upozorníme. Například znak „{“ označuje token $\boxed{\texttt{\{}}_1$ nebo $\boxed{\texttt{X}}_1$ nebo `\bgroup`. Na druhé straně tímto znakem neoznačujeme token $\boxed{\texttt{\{}}_{12}$

Pod pojmem *mezera* (v TEXbooku označováno jako ⟨*space token*⟩) rozumíme buď explicitní zápis tokenu kategorie 10, nebo implicitní zápis řídicí sekvence, která má význam tokenu kategorie 10. V příkladech budeme mezeru vyznačovat vaničkou (␣). Budeme-li v ukázkách uvažovat dvě nebo více mezer vedle sebe, nesmíme zapomenout, že v tuto chvíli se na věc díváme očima hlavního procesoru. Nelze tedy do textu vstupního souboru prostě napsat více mezer, protože jsou obvykle zredukovány token procesorem na mezeru jedinou. Například o pár řádků níže uvádíme „1␣␣␣Pt“, což můžeme realizovat ve vstupním souboru třeba takto:

```
\edef\act{\let\noexpand\spacetoken= \space} \act
% nyní je \spacetoken zástupná řídicí sekvence za mezeru
1 \space\spacetoken Pt
```

Jestliže v syntaktickém pravidle nebo ve slovníku primitivů zmiňujeme token, který musí mít jednoznačně danou kategorii i ASCII kód současně, zapisujeme jej například takto: `=`$_{12}$. Tyto tokeny nelze zastoupit jinou řídicí sekvencí, která má stejný význam pomocí `\let`.

V syntaktickém pravidle se může vyskytnout text tzv. *klíčového slova*. Následuje tabulka všech možných klíčových slov v TEXu. V prostředním sloupci jsou uvedena syntaktická pravidla, ve kterých příslušné klíčové slovo figuruje.

Klíčová slova	*syntaktické pravidlo*	*poznámka, primitiv*
`at, scaled`	⟨*at clause*⟩	`\font`.
`bp, cc, cm, dd, in,`		
`mm, pc, pt, sp`	⟨*dimen*⟩	rozměrové jednotky
`em, ex`	⟨*dimen*⟩	jednotky závislé na fontu
`by`	⟨*optional by*⟩	`\advance`, `\divide`, `\multiply`
`depth, height, width`	⟨*rule specification*⟩	`\hrule`, `\vrule`
`fil, fill, filll`	⟨*glue*⟩	stupeň pružnosti výplňku
`minus, plus`	⟨*glue*⟩	parametry pružnosti výplňku
`spread, to`	⟨*box specification*⟩	`\hbox`, `\vbox`, `\halign`, `\valign`
`to`		`\read`, `\vsplit`
`true`	⟨*dimen*⟩	jiná interpretace rozměrů

Pro klíčová slova platí následující pravidla:

- Před klíčovými slovy může vždy předcházet libovolné množství mezer.
- Klíčová slova nemají mezery mezi písmeny (výjimkou je `fill` a `filll`).
- Jednotlivá písmena mohou být malá i velká bez závislosti na `\lccode`.
- Jednotlivá písmena nelze nahradit implicitními řídicími sekvencemi.
- Jednotlivá písmena mají libovolnou kategorii.

Například „`1␣␣␣Pt`“ je sémanticky totéž co „`1pt`“. U posledního pravidla (o libovolné kategorii) připomínáme, že na vstup hlavního procesoru nepřicházejí všechny kategorie podle tabulky ze strany 20, ale jen ty, které nejsou zpracovány předchozími procesory. Musíme proto vyloučit kategorie s čísly 0, 5, 9, 13, 14 a 15.

Pokud uvádíme primitivní řídicí sekvence, předpokládáme, že mají původní (primitivní) význam. Identifikátory těchto sekvencí uvozujeme obvyklým „\“. Tyto primitivy lze nahradit jinými řídicími sekvencemi se stejným významem pomocí `\let`.

⟨*at clause*⟩ má tři formy zápisu:

- nula nebo více mezer
- `at`⟨*dimen*⟩
- `scaled`⟨*number*⟩

V případě, že je ⟨*at clause*⟩ prázdné (nula nebo více mezer), bude font zaveden ve velikosti podle *design_size* (základní velikost fontu). V případě klíčového slova `at` bude velikost fontu rovna ⟨*dimen*⟩ a konečně při použití klíčového slova `scaled` bude font zaveden ve velikosti *design_size* × ⟨*number*⟩/1000. Například font `csr10` má *design_size* = 10 pt. Proto mají následující dva zápisy stejný význam:

```
\font\twrm=csr10 at 12pt
\font\twrm=csr10 scaled 1200
```

⟨*balanced text*⟩ je libovolná posloupnost tokenů, ve které všechny explicitní „`{`“ párují s explicitními `}`. Přítomnost implicitních závorek tohoto typu (například `\bgroup` a `\egroup`) nemá vliv. Příklady:

```
abc123\bgroup              % ano, to je ⟨balanced text⟩
{{xy{z\egroup}\bla}\bla}   % to je také ⟨balanced text⟩
\bgroup bcd}               % toto není ⟨balanced text⟩
{ abc\egroup               % toto také není ⟨balanced text⟩
```

Pravidlo ⟨*balanced text*⟩ se často vyskytuje v kontextu, který je v TEXbooku označen jako ⟨*general text*⟩. V této knize tento pojem nezavádíme, nicméně ⟨*general text*⟩ je zkratka za:

```
⟨filler⟩{⟨balanced text⟩}
```

Zde platí důležitá výjimka. Pravá závorka v ⟨*general text*⟩ *musí* být jedině explicitní, zatímco levá může být implicitní. S tímto syntaktickým pravidlem se setkáváme u následujících primitivů: `\message`, `\errmessage`, `\hyphenation`, `\patterns`, `\lowercase`, `\uppercase`, `\mark`, `\write`, `\special`. Uvozující závorka pro ⟨*balanced text*⟩ je všemi primitivy načtena po případné expanzi. Proto je dovolena třeba taková zvrácenost:

```
\def\beg{\bgroup}
\message \beg Ahoj, \beg, \egroup \egroup }
% Vytvoří zprávu: "Ahoj, \bgroup, \egroup \egroup "
```

Tři primitivy (`\write`, `\lowercase`, `\uppercase`) nejprve čtou ⟨*balanced text*⟩ bez expanze a teprve potom jej (případně po další úpravě) expandují. V takovém případě musí závorky v ⟨*balanced text*⟩ párovat *před* případnou expanzí. V ostatních případech stačí, když závorky v ⟨*balanced text*⟩ párují po expanzi. Například:

```
\def\bexplicit{{\iffalse}\fi} % expanduje na explicitní "{"
\message {Toto \bexplicit je} balancovaný text}
% Vytvoří zprávu: "Toto {je} balancovaný text"
```

Stejný text by byl v případě primitivů `\write`, `\lowercase` a `\uppercase` zpracován jinak a skončil by chybou.

⟨*box*⟩ je výsledkem konstrukce těchto povelů (včetně parametrů): `\box`, `\copy`, `\vsplit`, `\hbox`, `\vbox`, `\vtop` a `\lastbox`. Příklady (odděleny středníky):

```
\box2;  \copy\count255;  \vsplit254 to 0.5\vsize;
\hbox{abc};  \vbox{\unvbox255};  \vtop{\kern3pt\hrule};
```

⟨*box or rule*⟩ je pravidlo, zahrnující pro `\leaders`, `\xleaders` a `\cleaders` tyto možnosti:

- ⟨*box*⟩
- `\hrule` ⟨*rule specification*⟩
- `\vrule` ⟨*rule specification*⟩

⟨*box specification*⟩ má tři formy zápisu:

- `to`⟨*dimen*⟩⟨*filler*⟩
- `spread`⟨*dimen*⟩⟨*filler*⟩
- ⟨*filler*⟩ neboli prázdná specifikace.

Významy jednotlivých forem zápisu ⟨*box specification*⟩, viz sekci 3.5. Příklady:

```
to12pt\relax    % zde je použito \relax z ⟨filler⟩
spread12pt␣␣    % první mezera z ⟨dimen⟩ a druhá z ⟨filler⟩
␣␣to12pt        % před klíčovým slovem mohou být mezery
```

⟨*control sequence*⟩ je libovolný token kategorie 13 nebo token typu řídicí sekvence. Například [~]$_{13}$ nebo „`\bla`“. Těmto tokenům v celé knize říkáme prostě *řídicí sekvence*.

⟨*delimiter*⟩ je zápis pro sazbu matematického znaku, který může být uveden jako argument primitivů `\left` a `\right`. Jedná se o tyto možnosti:

- ASCII znak s nezáporným `\delcode`
- `\delimiter`⟨*27-bit number*⟩

Pro stejné syntaktické pravidlo používáme též označení ⟨*delim1*⟩ a ⟨*delim2*⟩, abychom odlišili znak pro pravou a pro levou závorku.

⟨*dimen*⟩ se zapisuje ve tvaru

⟨*znaménko*⟩⟨*desetinné číslo*⟩⟨*jednotka*⟩

Pravidla pro ⟨*znaménko*⟩, viz heslo ⟨*number*⟩.

⟨*desetinné číslo*⟩ má několik forem zápisu. Jednak je to posloupnost desítkových cifer, jako u ⟨*number*⟩ a dále zápisy uvozené znaky [‘]$_{12}$, ["]$_{12}$ nebo [’]$_{12}$, viz ⟨*number*⟩. Také je možno použít konstantu z `\chardef` nebo `\mathchardef`. Tyto formy zápisu bychom mohli nazvat „celé číslo bez řádové čárky“. Kromě toho existuje ještě další možnost. Jedná se o posloupnost desítkových cifer kategorie 12 následovaná znakem [.]$_{12}$ nebo [,]$_{12}$ (znak značí desetinnou čárku)

a dále následuje další posloupnost desítkových cifer. Posloupnost cifer před desetinnou čárkou i za ní může být prázdná, ale nikdy ne obě současně. Uvedeme příklady pro ⟨*desetinné číslo*⟩. Příklady (odděleny středníky):

```
22 232;  "E8;  3.14;  3,14;  0.3333;  .3333;  2.
```

⟨*jednotka*⟩ je buď *implementovaná jednotka* (viz následující tabulku) nebo registr typu ⟨*number*⟩, ⟨*dimen*⟩ nebo ⟨*glue*⟩. Implementovaná jednotka (například `pt`) může být následovaná jednou nepovinnou mezerou. Uvedeme nyní tabulku všech implementovaných jednotek.

`pt`	monotypový bod	$1\,\mathrm{pt} = 1/72{,}27\,\mathrm{in} \doteq 0{,}35146\,\mathrm{mm} \doteq 0{,}93457\,\mathrm{dd}$
`pc`	pica	$1\,\mathrm{pc} = 12\,\mathrm{pt} \doteq 4{,}21752\,\mathrm{mm}$
`bp`	počítačový bod	$1\,\mathrm{bp} = 1/72\,\mathrm{in} \doteq 0{,}35278\,\mathrm{mm}$
`dd`	Didotův bod	$1\,\mathrm{dd} = 1238/1157\,\mathrm{pt} \doteq 1{,}07\,\mathrm{pt} \doteq 0{,}376\,\mathrm{mm}$
`cc`	cicero	$1\,\mathrm{cc} = 12\,\mathrm{dd} \doteq 4{,}512\,\mathrm{mm}$
`in`	palec (inch, coul)	$1\,\mathrm{in} = 72{,}27\,\mathrm{pt} = 25{,}4\,\mathrm{mm}$
`cm`	centimetr	$1\,\mathrm{cm} = 10\,\mathrm{mm} \doteq 28{,}4528\,\mathrm{pt}$
`mm`	milimetr	$1\,\mathrm{mm} = 1/25{,}4\,\mathrm{in} \doteq 2{,}84528\,\mathrm{pt} \doteq 2{,}6591\,\mathrm{dd}$
`sp`	přesnost TEXu	$1\,\mathrm{sp} = 1/65536\,\mathrm{pt} = 2^{-16}\,\mathrm{pt} \doteq 5{,}36 \cdot 10^{-9}\,\mathrm{m}$
`em`	velikost písma	je rovna hodnotě `\fontdimen6\font`
`ex`	výška malého x	je rovna hodnotě `\fontdimen5\font`

Následující příklady ukazují stejnou velikost ⟨*dimen*⟩. Jednotlivé ukázky jsou odděleny středníkem:

```
23 12␣pt␣;  1pc;  786432.sp;  .166044in;  4,21752␣mm
```

Před implementovanou jednotkou může předcházet klíčové slovo `true`, které považujeme za součást syntaktického pravidla ⟨*jednotka*⟩. Mezi `true` a vlastní jednotkou mohou být mezery. Slovo `true` nelze použít před jednotkami `em` a `ex`. Je-li před jednotkou použito `true`, je jednotka vynásobena převrácenou hodnotou koeficientu celkového zvětšení (viz heslo `\mag`).

⟨*jednotka*⟩ může být též registr typu ⟨*number*⟩, ⟨*dimen*⟩, ⟨*glue*⟩ nebo konstanta z `\chardef` a `\mathchardef`. Před takovouto veličinou mohou být mezery. Je-li veličina jiného typu, než ⟨*dimen*⟩, TEX provede konverzi následujícím způsobem: K typu ⟨*number*⟩ připojí jednotku `sp` a u typu ⟨*glue*⟩ ignoruje hodnotu stažení a roztažení. Možnost zápisu ⟨*jednotky*⟩ ve tvaru registru implementuje do makrojazyka TEXu jednoduché násobení racionálních čísel (viz stranu 80). Příklady:

```
24 0.5␣\hsize     % pronásobí \hsize koeficientem 0,5
25 1\parfilskip   % vezme jen základní velikost z \parfillskip
26 2\count255     % velikost rovna 2*\count255 sp
27 \%\%           % 37*37sp = 0.02089pt, protože \chardef\%=37
```

Syntaktické pravidlo ⟨*dimen*⟩ jako celek má ještě jednu formu zápisu: registr typu ⟨*dimen*⟩ nebo ⟨*glue*⟩, před kterým může stát ⟨*znaménko*⟩. V případě registru typu ⟨*glue*⟩ TEX provede konverzi na typ ⟨*dimen*⟩ tím způsobem, že ignoruje hodnoty stažení a roztažení.

⟨*equals*⟩ zahrnuje libovolné množství mezer ukončené nepovinným znakem =12. Za tímto znakem už případná mezera spadá do následujícího syntaktického pravidla. Většinou následuje číselná konstanta. Pak mezery za rovnítkem spadají do pravidla ⟨*znaménko*⟩ (viz ⟨*number*⟩ a ⟨*dimen*⟩).

⟨*file name*⟩ je pravidlo závislé na implementaci TEXu v konkrétním operačním systému. Při načítání ⟨*file name*⟩ probíhá vždy expanze. Nejprve jsou ignorovány případné mezery a pak je čten vlastní název souboru. Narazí-li se na neexpandovatelnou řídicí sekvenci (například `\relax`, `\pageno`), je načítání názvu souboru ukončeno. Obvyklejší je ukončovat název souboru jednou mezerou nebo více mezerami, které pracují jako separátor a v následujícím vstupu jsou ignorovány. Není ovšem vyloučeno, že v operačních systémech, kde jsou mezery v názvech souborů na denním pořádku, bude implementace ⟨*file name*⟩ vypadat jinak.

Například v instalaci `web2c` (v UNIXu) není vůbec možné načíst soubor obsahující v názvu mezeru. Ve WEBovské sekci 516 zdrojového textu TEXu je totiž pro ukončení čtení názvu přímý test na ASCII 32, takže nepomůže ani změna kategorií. V případě mezery v názvu je tedy nutné nejprve změnit název souboru prostředky operačního systému.

Při čtení názvu ⟨*file name*⟩ neprobíhá interpretace kategorií 1–4, 6–8, 11 a 12. Proto je možné otevírat soubory s názvy „`pok_1.tex`“, „`pok#$^&.tex`“ nebo „`{pok.tex`“ prostým zápisem názvu souboru. Třeba `\input {pok.tex`. Tento příklad nebude fungovat v LATEXu, protože tam je primitiv `\input` předefinován.

Po načtení názvu souboru provede TEX případnou zpětnou konverzi názvu podle kódování systému (podobně jako při zápisu do `log`, viz stranu 18). Toto je definitivní název souboru, který pak TEX vyhledává v systému.

Podmínky automatického připojování koncovky `.tex` (nebo `.tfm`) k názvu souboru jsou rovněž závislé na implementaci. Například emTEX připojí při povelu `\input` koncovku `.tex` právě tehdy, když název souboru neobsahuje tečku. Na druhé straně instalace z `web2c` (pro UNIX) provede následující tři průchody: (1) Pokud název souboru obsahuje tečku, zkusí soubor otevřít. Podařilo-li se, přeskočí následující průchody. (2) Pokud název nemá příponou `.tex`, připojí tuto příponu a zkusí soubor otevřít. Podařilo-li se soubor otevřít, přeskočí poslední průchod. Jinak název souboru vrátí do původního stavu, tj. odpojí z názvu přidanou příponu `.tex`. (3) Není-li v názvu tečka, zkusí soubor otevřít. Nepodařilo-li se dosud soubor otevřít, ohlásí chybu. Při otevírání

souboru `web2c` instalace vyhledává v systému poměrně komplikovaným způsobem (viz `site.h` a dokumentaci ke `kpathsea`). Obecně platí, že nejprve se program podívá do aktuálního adresáře a pak „někam do instalace“.

Uvedeme příklad, ve kterém je chování emTEXu a `web2c` naprosto odlišné. Nechť máme v systému soubor s názvem `xyz` (bez přípony). Pak v emTEXu povel `\input xyz` hledá soubor `xyz.tex` a náš soubor nebude nalezen, zatímco v UNIXu soubor nalezen bude. Protože název souboru neobsahuje tečku, přejde instalace `web2c` přímo k druhému průchodu, ve kterém se pokusí najít soubor `xyz.tex`. Pokud takový soubor neexistuje, najde soubor `xyz` v průchodu 3. Naopak povel `\input xyz.` (tečka na konci) v emTEXu úspěšně vyhledá soubor `xyz`, zatímco v UNIXu se soubor nalézt nepodaří, protože samozřejmě žádný soubor `xyz.` (podle průchodu 1) ani `xyz..tex` (podle průchodu 2) v systému neexistuje.

Další odlišností může být významnost velkých a malých písmen v názvu souboru. Pokud tuto kvalitu systém rozlišuje, TEX ji rozlišuje také. DOS ji ovšem nerozlišuje. Používáte-li například emTEX, prostudujte si pravidla zapisování názvů souborů v dokumentaci `tex.doc`. Zjistíte třeba, že můžete pro názvy souborů včetně adresářů používat obyčejná lomítka místo zpětných. Rovněž se emTEX postará o redukci dlouhých názvů na DOSové omezení $8+3$, takže v zájmu přenositelnosti je vhodné zapisovat názvy souborů vždy plným (třeba dlouhým) jménem.

⟨*filler*⟩ je libovolně dlouhá posloupnost mezer kombinovaná s povely `\relax`. Toto syntaktické pravidlo zahrnuje též možnost prázdného zápisu (což je asi nejčastější případ). Pravidlo umožňuje makrům expandovat na `\relax` většinou před zahájením skupiny nebo před ⟨*balanced text*⟩. Například:

```
28 \hbox {abc}  % Takhle by to napsal každý normální jedinec.
29 \hbox \space\relax\relax\space {abc} % Totéž ne zcela běžně.
```

⟨*font*⟩ je řídicí sekvence, která má význam přepínače fontu. Takový význam má při iniTEXu pouze primitiv `\nullfont`. Později tento význam dostanou též sekvence deklarované primitivem `\font` nebo ztotožněné se sekvencemi ve významu přepínače fontu pomocí `\let` a `\futurelet`.

Do pravidla ⟨*font*⟩ spadá kromě zmíněných řídicích sekvencí též samotný primitiv `\font`, který má v takovém kontextu význam právě aktuálního fontu.

Do tohoto pravidla dále spadají konstrukce `\textfont`⟨*4-bit number*⟩ až `\scriptscriptfont`⟨*4-bit number*⟩. Tyto konstrukce ovšem nelze použít jako přímý přepínač fontu. K tomu musíme využít vlastnosti primitivu `\the`. Například `\the\font` je přepínač aktuálního fontu a `\the\textfont2` přepíná do základní velikosti fontu rodiny 2.

⟨*general text*⟩ viz heslo ⟨*balanced text*⟩.

⟨*glue*⟩ má základní formu zápisu: ⟨*dimen*⟩⟨*hodnota roztažení*⟩⟨*hodnota stažení*⟩. První údaj ⟨*dimen*⟩ je povinný a označuje *základní velikost* mezery. Následující údaje mají samovysvětlující název a deklarují *hodnotu roztažení* a *hodnotu stažení* pro mezeru typu ⟨*glue*⟩. ⟨*hodnota roztažení*⟩ nebo ⟨*hodnota stažení*⟩ nebo obě mohou chybět. Místo nich může být libovolný počet mezer. Při chybějícím údaji je příslušná hodnota interpretována jako nulová. Pořadí údajů nelze prohodit.

Kromě toho do syntaktického pravidla ⟨*glue*⟩ spadá zápis registru typu ⟨*glue*⟩, před kterým může stát ⟨*znaménko*⟩ (viz heslo ⟨*number*⟩).

⟨*hodnota roztažení*⟩ je uvozena klíčovým slovem `plus`, které je následováno zápisem ⟨*zobecněné dimen*⟩. ⟨*hodnota stažení*⟩ je uvozena klíčovým slovem `minus`, za kterým rovněž následuje ⟨*zobecněné dimen*⟩.

⟨*zobecněné dimen*⟩ má všechny možnosti zápisu, jako ⟨*dimen*⟩, ale navíc rozšiřuje možnosti pravidla ⟨*jednotka*⟩. Pravidlo ⟨*jednotka*⟩ může kromě variant z ⟨*dimen*⟩ zahrnovat také zápis jednoho z klíčových slov `fil`, `fill` nebo `filll`. Za těmito klíčovými slovy může být libovolné množství mezer a mezi písmeny „`l`“ výjimečně také mohou být mezery. Je-li pro jednotku použito některé z uvedených klíčových slov, pak se do hodnoty roztažení, resp. stažení, ukládá bezrozměrné číslo tvaru ⟨*znaménko*⟩⟨*desetinné číslo*⟩ (viz základní formu zápisu ⟨*dimen*⟩). Toto číslo se označí jako hodnota roztažení, resp. stažení, *řádu* r, kde r je počet písmen „`l`“ v klíčovém slově pro jednotku. r může nabývat hodnot 1, 2, 3. Pokud je použita běžná jednotka z ⟨*dimen*⟩, klade se $r = 0$.

Příklady zápisů podle syntaktického pravidla ⟨*glue*⟩:

```
1pt  % hodnoty roz/s/tažení chybějí; aby nehrozilo
     % uživatelovo "plus", viz stranu 70
1pt␣␣␣  % místo hodnot roz/s-tažení jsou mezery
1pt plus0pt minus0pt  % totéž jako v předchozích příkladech
1ptplus4ptminus2pt  % před klíč. slovy jsou mezery nepovinné
1 pt plus 4 pt minus 2 pt % mezery povolené různými pravidly
1pt minus2pt        % chybí hodnota roztažení
1pt minus2pt plus4pt  % chybí roztažení, vytiskne "plus4pt"
0pt plus 1fill  % ukázka použití pravidla ⟨zobecněné dimen⟩
+ - + 2 pt plus - 3.14 fil minus + 2 fil  % to je taky možné
2pt plus fil  % chyba! před fil musí být ⟨desetinné číslo⟩
\baselineskip % pravidlo ⟨glue⟩ jako registr typu ⟨glue⟩
\hsize        % registr ⟨dimen⟩, hodnoty roz/s-tažení chybějí
\hsize minus .5\hsize % lze připsat hodnoty roz/s-tažení
\pageno  % chyba! typ ⟨number⟩ se nekonvertuje na ⟨dimen⟩
```

Uvědomíme si rozdílnost v těchto případech:

```
\baselineskip =  \baselineskip plus1pt % vytiskne "plus1pt"
\baselineskip = 1\baselineskip plus1pt % přiřadí roztažení
```

Zápis `\baselineskip` samotný spadá do syntaktického pravidla ⟨*glue*⟩, takže další text se tiskne. Na druhé straně zápis `1\baselineskip` je ve tvaru pravidla ⟨*dimen*⟩ a můžeme tedy přidávat hodnoty roztažení a stažení.

⟨*glue specification*⟩ je pravidlo zahrnující pro `\leaders`, `\xleaders` a `\cleaders` tyto možnosti (odděleny středníky):

- `\vskip`⟨*glue*⟩; `\vfil`; `\vfill`; `\vss`; `\vfilneg`;
- `\hskip`⟨*glue*⟩; `\hfil`; `\hfill`; `\hss`; `\hfilneg`;
- `\mskip`⟨*muglue*⟩.

Alternativy z prvního řádku je možno použít ve vertikálním módu, z druhého řádku v horizontálním a z druhého a třetího řádku v matematickém módu.

⟨*horizontal material*⟩ je libovolná posloupnost povelů hlavního procesoru (včetně jejich parametrů), které postupně vytvářejí horizontální seznam. Je možné použít všechny povely označené ve slovníku primitivů typem [h], [a] a [pre]. Povely typu [a] označují přiřazení hlavního procesoru, které přímo nevytvářejí materiál, ale jsou povoleny. Kromě toho je možno konstruovat přiřazení ve tvaru ⟨*register*⟩⟨*equals*⟩⟨*value*⟩, jak bylo vyloženo v sekci 3.3.

Toto syntaktické pravidlo se vesměs vyskytuje obklopeno závorkami `{...}`, které mohou být i implicitní. Pak musí koncová závorka v okamžiku zpracování uzavírat stejnou úroveň skupiny, jaká byla otevřena uvozující závorkou, a uvnitř ⟨*horizontal material*⟩ nesmí být tato skupina uzavřena předčasně.

⟨*math field*⟩ je základním syntaktickým pravidlem pro vytváření matematického seznamu. Toto pravidlo má dvě formy zápisu:

- ⟨*samostatný matematický znak*⟩
- ⟨*filler*⟩`{`⟨*math mode material*⟩`}`

⟨*samostatný matematický znak*⟩ zdaleka nemusí být jen jediný token. Toto syntaktické pravidlo zahrnuje všechny formy zápisu, které vedou na sazbu jediného znaku v matematickém seznamu. Přesněji, jedná se o následující možnosti: ⟨*znak*⟩ nebo `\char`⟨*8-bit number*⟩ nebo `\mathchar`⟨*15-bit number*⟩ nebo `\delimiter`⟨*27-bit number*⟩. Také jsou možné zástupné řídicí sekvence z `\chardef` a `\mathchardef`. ⟨*znak*⟩ je token kategorie 11 nebo 12, který způsobí sazbu matematického znaku podle `\mathcode`. Před zápisem podle pravidla ⟨*samostatný matematický znak*⟩ může být libovolné množství mezer, protože mezery se v matematickém módu ignorují.

⟨*math mode material*⟩ je vyložen níže jako další heslo.

Příklad. Navrhujeme makro `\cosi`, které vytváří sazbu v matematickém seznamu jako box. Chceme, aby uživatel mohl psát třeba `$x^\cosi$`, takže makro musí expandovat na ⟨*math field*⟩.

```
\def\cosi{\hbox{...}}    % zápis $x^\cosi$ havaruje
\def\cosi{{\hbox{...}}}  % nyní $x^\cosi$ bude fungovat
```

⟨*math mode material*⟩ je libovolná posloupnost povelů hlavního procesoru (včetně jejich parametrů), které postupně vytvářejí matematický seznam. Je možné použít všechny povely označené ve slovníku primitivů typem [m], [a] a [pre]. Povely typu [a] označují přiřazení hlavního procesoru, které přímo nevytvářejí materiál, ale jsou povoleny. Kromě toho je možno konstruovat přiřazení ve tvaru ⟨*register*⟩⟨*equals*⟩⟨*value*⟩, jak bylo vyloženo v sekci 3.3.

Toto syntaktické pravidlo se vesměs vyskytuje obklopeno závorkami `$...$` nebo `{...}`, které mohou být i implicitní. Pak musí koncová závorka v okamžiku zpracování uzavírat stejnou úroveň skupiny, jaká byla otevřena uvozující závorkou, a uvnitř ⟨*math mode material*⟩ nesmí být tato skupina uzavřena předčasně.

⟨*mudimen*⟩ má stejná pravidla jako ⟨*dimen*⟩. Ovšem ⟨*jednotka*⟩ musí být pouze klíčové slovo `mu`. Jiné jednotky nejsou povoleny. Jednotka `mu` má rozměr 1/18 ⟨*quad*⟩, což odpovídá tzv. *setové jednotce* dřívějších typografických strojů. ⟨*quad*⟩ označuje velikost písma, tj. 1 em matematického fontu rodiny 2. Viz stranu 157.

V případě registrů spadá do tohoto pravidla jen registr typu ⟨*muglue*⟩. TEX v takovém případě provede konverzi, tj. ignoruje hodnoty roztažení nebo stažení. Tento registr lze použít i v kontextu ⟨*jednotka*⟩.

⟨*muglue*⟩ má stejná pravidla jako ⟨*glue*⟩. Ovšem místo ⟨*dimen*⟩ je nutno všude použít jen ⟨*mudimen*⟩, tj. je povolena jen jednotka `mu`. V hodnotách roztažení a stažení jsou ještě povoleny jednotky `fil`, `fill`, `filll`. Není dovolena konverze z registrů jiných typů do ⟨*muglue*⟩. Příklady:

```
1mu plus 1fil minus 0mu % toto je v pořádku
1mu plus 1fil minus 0pt % chyba! není povolena jednotka pt
2.5\muskip0 minus1mu    % ⟨jednotka⟩ je registr typu ⟨muglue⟩
\thickmuskip            % ⟨muglue⟩ jako registr typu ⟨muglue⟩
```

⟨*number*⟩ má šest základních forem zápisu:

- ⟨*znaménko*⟩⟨*číslo*⟩⟨*jedna nepovinná mezera*⟩
- ⟨*znaménko*⟩`'`$_{12}$⟨*oktalové číslo*⟩⟨*jedna nepovinná mezera*⟩
- ⟨*znaménko*⟩`"`$_{12}$⟨*hexadecimální číslo*⟩⟨*jedna nepovinná mezera*⟩
- ⟨*znaménko*⟩`` ` ``$_{12}$⟨*token znak*⟩⟨*jedna nepovinná mezera*⟩
- ⟨*znaménko*⟩⟨*numerický registr*⟩
- ⟨*znaménko*⟩⟨*konstanta*⟩

⟨*jedna nepovinná mezera*⟩ označuje jednu nebo žádnou mezeru. Toto je velmi podstatným gramatickým jevem, protože mezera umožňuje oddělit zápis čísla od případně dalších povelů hlavního procesoru.

⟨*číslo*⟩ je souvislá posloupnost číslic, které musí mít kategorii 12. Pravidlo pro ⟨*oktalové číslo*⟩ je shodné (ovšem nejsou povoleny číslice `8` a `9`). Tímto způsobem zapisujeme číslo v oktalové notaci. Konečně ⟨*hexadecimální číslo*⟩

navíc může obsahovat „číslice“ `A` až `F`. Tyto „číslice“ mohou mít kategorii 11 nebo 12, ale nesmí se zapisovat jako malá písmena.

⟨*token znak*⟩ je buď řídicí sekvence, která musí mít jednoznakový identifikátor, nebo se jedná o token typu dvojice (ASCII, kategorie). Na kategorii nezáleží. Ačkoli tento token může být řídicí sekvencí nebo má kategorii 13, nebude TEXem expandován. Zápis `‘`$_{12}$ ⟨*token znak*⟩ znamená číslo ASCII kódu ⟨*token znak*⟩.

⟨*znaménko*⟩ je libovolně dlouhá posloupnost znaků z množiny {␣, `+`$_{12}$, `-`$_{12}$}. Pokud je v tomto zápise lichý počet znaků `-`$_{12}$, je číslo interpretováno jako záporné. Jinak je nezáporné.

Uvedeme nyní různé příklady zápisu čísla 65. Zápisy od sebe oddělujeme středníkem, který nijak nesouvisí se syntaktickým pravidlem ⟨*number*⟩.

```
65; 65␣; ’101; "41; ‘A; ‘A␣; ‘\A; +‘\A; ␣␣␣‘\A; ␣-++␣-␣‘\A
```

Nakonec popíšeme formu zápisu ⟨*number*⟩ jako ⟨*numerický registr*⟩ nebo ⟨*konstantu*⟩. ⟨*numerický registr*⟩ je libovolný registr typu ⟨*number*⟩, ⟨*dimen*⟩ nebo ⟨*glue*⟩, který ve slovníku primitivů označujeme jako [integer], [dimen] nebo [glue]. Rovněž za registr uvedeného typu považujeme řídicí sekvence deklarované primitivy `\countdef`, `\dimendef` nebo `\skipdef`. Konečně ⟨*konstanta*⟩ je libovolný token deklarovaný pomocí `\chardef` nebo `\mathchardef`. Před ⟨*numerickým registrem*⟩ nebo ⟨*konstantou*⟩ může stát ⟨*znaménko*⟩ podle syntaktických pravidel popsaných výše. Je-li ve znaménku lichý počet znaků `-`$_{12}$, bude výsledná hodnota vynásobena číslem -1. Není-li registr typu ⟨*number*⟩, provede se při interpretaci čísla typu ⟨*number*⟩ konverze. Viz sekci 3.3, strana 78. Příklady:

```
\pageno              % dává v tuto chvíli: 326
\%                   % jedná se o token z \chardef, dává: 37
\sum                 % token z \mathchardef: 4944
\baselineskip        % po konverzi je 12pt rovno: 786432
-\fontdimen2\tenrm   % dává po konverzi: -218453
```

Pravidla ⟨*4-bit nummber*⟩, ⟨*8-bit number*⟩, ⟨*15-bit number*⟩, ⟨*24-bit number*⟩, ⟨*27-bit number*⟩ jsou zcela shodná s pravidlem ⟨*number*⟩, ovšem navíc číslo musí být nezáporné a pro ⟨*n-bit number*⟩ musí být číslo menší nebo rovno $2^n - 1$. Jedná se tedy o přirozené číslo, které se dá implementovat do n bitového registru. Zopakujeme si mocniny dvou: ⟨*4-bit number*⟩ ≤ 15, ⟨*8-bit number*⟩ ≤ 255, ⟨*15-bit number*⟩ $\leq 32\,767$, ⟨*24-bit number*⟩ $\leq 16\,777\,215$ a konečně ⟨*27-bit number*⟩ $\leq 134\,217\,727$.

⟨*numeric variable*⟩ je registr typu ⟨*number*⟩, ⟨*dimen*⟩, ⟨*glue*⟩ nebo ⟨*muglue*⟩. Jedná se o primitivy, které mají u svého hesla typ [integer], [dimen], [glue], [muglue], ale nikoli s prefixem *restricted* nebo *read-only*.

Registr ⟨*numeric variable*⟩ vystupuje v parametrech primitivů `\advance`, `\multiply` a `\divide`. Tyto primitivy mění hodnotu registru. Proto je zákaz použití registrů určených pouze ke čtení pochopitelný. Méně pochopitelný je zákaz použití registrů označených jako *restricted*, do nichž je možno zapisovat. Považujme to jako kuriozitu TEXu a chceme-li třeba zvětšit hodnotu `\spacefactor` o 10, pišme:

```
% nelze použít \advance\spacefactor by 10
\count255=\spacefactor \advance\count255 by 10
\spacefactor=\count255
```

⟨*optional by*⟩ je nula nebo více mezer nebo klíčové slovo `by`. Jako příklad uvedeme různé zápisy pro ⟨*optional by*⟩ v kontextu `\advance\hsize`⟨*optional by*⟩`5cm`.

```
\advance\hsize 5cm     % v místě ⟨optional by⟩ nemusí být nic
\advance\hsize by5cm   % použití klíčového slova
\advance\hsize ␣␣by5cm % před klíč. slovem mohou být mezery
\advance\hsize ␣␣5cm   % ⟨optional by⟩ jsou mezery
\advance\hsize by␣5cm  % tato mezera je ⟨znaménko⟩ čísla 5
```

⟨*register*⟩ je libovolný registr, do kterého lze zapisovat. Ve slovníku primitivů jsou tyto registry označeny typem [integer], [dimen], [glue], [muglue] [tokens] a [font]. Do syntaktického pravidla ⟨*register*⟩ rovněž spadají řídicí sekvence ve významu registru deklarované pomocí `\countdef`, `\dimendef`, `\skipdef`, `\muskipdef`, `\toksdef`, `\let` a `\futurelet`.

⟨*rule specification*⟩ může být prázdné nebo je vyplněno mezerami nebo obsahuje údaje ⟨*výška*⟩, ⟨*hloubka*⟩ a ⟨*šířka*⟩. Tyto údaje se nemusí vyskytovat vůbec, mohou se vyskytovat v libovolném pořadí a mohou se vyskytovat i opakovaně. Objeví-li se nějaký údaj opakovaně, platí poslední výskyt.

Údaj ⟨*výška*⟩ zapisujeme jako `height`⟨*dimen*⟩, údaj ⟨*hloubka*⟩ píšeme ve tvaru `depth`⟨*dimen*⟩ a konečně ⟨*šířka*⟩ má formát `width`⟨*dimen*⟩.

Pravidlo ⟨*rule specification*⟩ se používá výhradně ve spojení s primitivy `\vrule` a `\hrule` pro vytvoření *linky*. Linka se chová jako černý obdélník o rozměrech ⟨*výška*⟩, ⟨*hloubka*⟩ a ⟨*šířka*⟩. Při umisťování do sazby se linka chová stejně jako box s odpovídajícímu rozměry. Je-li součet ⟨*výška*⟩ + ⟨*hloubka*⟩ nekladný nebo ⟨*šířka*⟩ je nekladná, není linka vůbec vidět. Nicméně pohyb aktuálního bodu sazby a rozměry tiskového materiálu může ovlivnit i neviditelná linka.

Není-li některý rozměr přímo specifikován pomocí odpovídajícího klíčového slova, pak má příslušný rozměr implicitní velikost podle této tabulky:

	výška	hloubka	šířka
`\hrule`	0,4 pt	0 pt	*
`\vrule`	*	*	0,4 pt

Hvězdičky v tabulce znamenají, že rozměr je dopočítán podle rozměru vnějšího boxu, ve kterém je linka použita, viz sekci 3.5. V takovém případě mluvíme o *neurčitém rozměru* linky. Například, vyzkoušíme-li:

```
\hbox{ab
 \vrule height4pt depth-2pt width5pt %všechny rozměry určeny
  cd
 \vrule %žádný rozměr není určen, výpočet až při kompletaci
  ef
 \vrule height10pt width0pt %je width0pt, nebude nic vidět
  gh}
```

pak na výstupu dostaneme:

ab cd ef gh

V tomto příkladě je výška boxu počítána podle nejvyššího elementu, kterým je neviditelná linka výšky 10 pt. Hloubka boxu bude odpovídat hloubce písmene „g“. První linka má explicitně zadané rozměry, zatímco druhá linka bude mít výšku a hloubku rovnu výšce a hloubce boxu. Šířka této linky bude 0,4 pt. Poslední linka díky své šířce 0 pt nebude vidět a bude mít hloubku rovnu hloubce kompletovaného boxu. Protože je tato linka „nejvyšším elementem“, určuje celkovou výšku boxu.

Nevyhovuje-li nám implicitní tloušťka linek 0,4 pt, můžeme využít skutečnosti, že poslední zápis údajů ⟨*výška*⟩ apod. má přednost.

```
\let\orihrule=\hrule  \let\orivrule=\vrule
\newdimen\defaultrulethickness \defaultrulethickness=0.5dd
\def\hrule{\orihrule height\defaultrulethickness}
\def\vrule{\orivrule width\defaultrulethickness}
```

V dalších makrech můžeme po této deklaraci používat řídicí sekvence `\hrule` a `\vrule` v naprosto stejném smyslu, jako jsme dosud pracovali se stejnojmennými primitivy. Pouze implicitní tloušťku čáry máme pod vlastní kontrolou. Napíšeme-li například `\vrule width0pt`, pak uvedený parametr ⟨*šířka*⟩ způsobí ignorování „implicitního“ `\defaultrulethickness`.

⟨*token*⟩ je libovolný token, jak bylo popsáno v sekci 1.3. Pro odlišení někdy též zapisujeme ⟨*token1*⟩ a ⟨*token2*⟩.

⟨*tokens*⟩ je libovolná (třeba prázdná) posloupnost tokenů. Pro odlišení někdy též zapisujeme ⟨*tokens1*⟩ a ⟨*tokens2*⟩.

⟨*vertical material*⟩ je libovolná posloupnost povelů hlavního procesoru (včetně jejich parametrů), které postupně vytvářejí vertikální seznam. Je možné použít všechny povely označené ve slovníku primitivů typem [v], [a] a [pre]. Povely typu [a] označují přiřazení hlavního procesoru, které přímo nevytvářejí materiál, ale jsou povoleny. Kromě toho je možno konstruovat přiřazení ve tvaru ⟨*register*⟩⟨*equals*⟩⟨*value*⟩, jak bylo vyloženo v sekci 3.3.

Toto syntaktické pravidlo se vesměs vyskytuje obklopeno závorkami `{...}`, které mohou být i implicitní. Pak musí koncová závorka v okamžiku zpracování uzavírat stejnou úroveň skupiny, jaká byla otevřena uvozující závorkou, a uvnitř ⟨*vertical material*⟩ nesmí být tato skupina uzavřena předčasně.

Při zpracování ⟨*vertical material*⟩ je povolen přechod do odstavcového módu (viz sekci 3.4) a zpět. V odstavcovém módu je možné používat povely typu [h] anebo přejít do matematického módu a používat povely typu [m]. Vyskytne-li se koncová závorka uzavírající ⟨*vertical material*⟩ v odstavcovém módu, je provedeno vynucené ukončení odstavce (strana 91).

Z pohledu hlavního procesoru můžeme zapsat syntaktické pravidlo pro vytvoření celého dokumentu takto: ⟨*vertical material*⟩`\end`.

2. Zkratky plainu

Plain používá z důvodu šetřením místa v hlavní paměti TEXu některé zkratky (viz stranu 75). Tyto zkratky většinou zhoršují čitelnost kódu makra. Proto jsem veškeré kódy maker publikované v této knížce pozměnil tak, aby zkratky neobsahovaly. Například místo „`\@ne`“ píši „`1␣`“.

Zkratky v plainu jsou dvojího druhu. (1) Jsou používány řídicí sekvence zastupující čísla. (2) Je vynechávána znak „=“ v případě syntaktického pravidla ⟨*equals*⟩ a slovo `by` u syntaktického pravidla ⟨*optional by*⟩. V těchto dvou případech jsem upravil publikovaný kód makra plainu takto: (1) Místo zástupných řídicích sekvencí jsou použita čísla. (2) Jsou použita rovnítka a slůvka `by` pokud možno všude, kde to syntakticky lze.

```
\chardef\@ne=1
\chardef\tw@=2
\chardef\thr@@=3
\chardef\sixt@@n=16
\chardef\@cclv=255
\mathchardef\@cclvi=256
\mathchardef\@m=1000
\mathchardef\@M=10000
\mathchardef\@MM=20000
\countdef\m@ne=22 \m@ne=-1

\newdimen\p@ \p@=1pt
\newdimen\z@ \z@=0pt
\newskip\z@skip \z@skip=0pt plus0pt minus0pt
\newbox\voidb@x

\def\lq{‘} \def\rq{’}                    % pro polámané nebo
\def\lbrack{[} \def\rbrack{]}            % nemožné klávesnice
\let\sp=^ \let\sb=_

\countdef\count@=255
\dimendef\dimen@=0
\dimendef\dimen@i=1
\dimendef\dimen@ii=2
\skipdef\skip@=0
\toksdef\toks@=0

\def\o@lign{\lineskiplimit\z@ \oalign}  % oprava maker \d a \b

```

```
107 \def\m@th{\mathsurround\z@}
```

Za pozornost stojí pomocné makro `\m@th`. V následujícím slovníku primitivů a maker jsou všechna pomocná makra (tj. obsahující znak „`@`“) uvedena u kódu toho makra, kterému slouží. Výjimkou je právě makro `\m@th`, které bychom museli obkreslovat na 26 místech, protože slouží mnoha různým makrům.

Uvedené zkratky a časté používání znaku `@` v pomocných makrech chápu jako nutnost, protože formát je určen i pro mnoho nepoučených uživatelů, před kterými je potřeba například zakrýt pracovní řídicí sekvence. Také paměťový prostor TEXu v době, kdy plain vznikal, byl dosti omezen. Proto se hodilo šetřit každým rovnítkem.

Mám ale pocit, že tyto důvody plno lidí ne zcela dobře pochopilo a nečitelností kódu plainu se inspirovalo i při publikování svých vlastních maker. Jinak si nelze vysvětlit, proč tito lidé mají tak rádi znak „`@`“ a nenávidějí znak „`=`“. Snadno se dá vytvořit zcela nečitelný kód makra. Viz například [9], ale tam to byl úsměvný záměr. Jestliže chceme, aby naše makra někdo četl, pokusme se používat rovnítka a vyvarovat znaku „`@`“, pokud to jen trochu bude možné.

3. Slovník primitivů a maker plainu

Za každým heslem je popis parametrů hesla podle syntaktických pravidel. Vpravo od hesla v hranaté závorce je typ hesla. Ve slovníku se vyskytují následující typy:

[h] — horizontal command: povel horizontálního módu.
[v] — vertical command: povel vertikálního módu.
[m] — math. command: povel matematického módu.
[exp] — expandable command: primitiv expanduje v expand procesoru.
[a] — assignment: přiřazení provedené hlavním procesorem ve všech módech.
[pre] — prefix: (`\global`, `\outer` atd.).
[spec] — speciální primitiv v `\halign` a `\valign`.
[integer] — integer parameter: primitivní registr typu ⟨*number*⟩.
[dimen] — dimen parameter: primitivní registr typu ⟨*dimen*⟩.
Podobně [glue], [muglue], [tokens].
Prefixy: *global* — přiřazení je vždy jen globální,
read-only — parametr je pouze ke čtení,
restricted — nelze použít jako ⟨*numeric variable*⟩.
[font] — font: přepínač fontu, viz syntaktické pravidlo ⟨*font*⟩.
[plain] — makro formátu plain.
[csplain] — makro formátu csplain, které není v plainu.

Například u hesla `\wd` je uveden parametr ⟨*number*⟩ a typ hesla *restricted* [dimen]. Tím je řečeno, že za primitivem `\wd` musí bezpodmínečně následovat (po expanzi) parametr podle syntaktického pravidla ⟨*number*⟩ a celek (tj. heslo včetně parametru) se chová jako registr typu ⟨*dimen*⟩. Třeba `\wd2` je registr typu ⟨*dimen*⟩. Dále například heslo `\kern` má parametr podle pravidla ⟨*dimen*⟩ a výsledkem je typ [h, v, m], tj. povel vkládající nějaký materiál do horizontálního, vertikálního nebo matematického seznamu. Protože je `\wd2` v souladu s pravidlem ⟨*dimen*⟩, je dovolena například konstrukce `\kern\wd2`, která jako celek vkládá cosi do tiskového materiálu.

Jedná-li se o primitivní registr, je vedle něj uvedena v kulaté závorce výchozí hodnota, jak ji nastavuje plain nebo iniTEX.

Ke každému heslu je připojen výklad včetně příkladů a ukázek. Výklad je stručný, pokud už byl související algoritmus vyložen v části A. V takovém případě může být v seznamu odkazů za znakem **Ol** na prvním místě číslo strany sázené kurzívou. Na této straně je možno najít další podrobnější informace o heslu.

Pokud heslo není primitivem, ale je makrem (typ hesla [plain] nebo [csplain]), jsou jeho parametry rovněž uvedeny v ⟨*této*⟩ ⟨*konvenci*⟩, ovšem nejedná se o názvy syntaktických pravidel. Význam parametrů je přímo vysvětlen u hesla. Dále je uveden

kód ze souboru `plain.tex`, kterým je makro definováno. Pomocné řídicí sekvence obsahující v identifikátoru znak „`@`“ nemají ve slovníku samostatnou pozici, ale jsou uvedeny u toho makra, ve kterém je příslušná pomocná řídicí sekvence použita. Viz též předchozí „zkratky plainu“.

Rovněž ve slovníku chybí řídicí sekvence, které jsou deklarovány pro matematickou sazbu pomocí `\mathchardef`, `\mathaccent`, `\delimiter` nebo jako makro pro sazbu složeného matematického symbolu. Tyto řídicí sekvence jsou shrnuty v sekci 5.4.

Za označením **Kn:** jsou uvedeny stránky z TEXbooku, odkazující na jmenované heslo. Jedná se o 19. vydání, October 1990. Stránky se shodují i pro jiná vydání popisující TEX 3.x. Tyto stránky jsou seřazeny podle důležitosti. Za označením **Ol:** jsou uvedeny stránky, na kterých je zmínka o hesle v této knížce. Čísla stránek mohou mít indexy, které odkazují na čísla řádků v ukázkách. Pokud indexy nejsou přítomny, je to často z toho důvodu, že je heslo používáno v knize velmi často a uvedení plné reference by způsobilo zaplnění textu horou čísel nevalné užitečnosti.

`~`13 [plain]

Vlnka. Definuje mezeru jako `\␣`, ve které není povolen řádkový zlom.

```
\def~{\penalty10000\ } % tie
```

Kn: 38, 51, 343, 353, **Ol:** $24_{32,\,37}$, 25_{46}, $27_{70–71}$, 31_{1}, $33_{22–23}$, $56_{254–255}$, 91_{152}, 211_{3}, 289_{28}, 290_{39}, 333_{108}, 335_{126}.

`'`12 [plain]

Protože je nastaveno `\mathcode‘\’="8000`, chová se znak (’) jako aktivní, ovšem pouze v matematickém módu. Expanduje na `^\bgroup\prime\egroup`, takže `$f'$` vede na f', neboli „derivace f“. Při druhé derivaci (dva znaky za sebou) se celá záležitost expanduje na `^\bgroup\prime\prime\egroup` a při vícenásobné derivaci analogicky. Navíc zápis `$f'^3$` expanduje na `$f^\bgroup\prime 3\egroup$`, takže je trojka i znak derivace ve společném exponentu. Výsledek pak dopadá podle uživatelova očekávání: f'^3, neboli „první derivace f umocněna na třetí“. Znak `\prime`, viz stranu 187.

```
{\catcode‘\’=\active \gdef’{^\bgroup\prim@s}}
\def\prim@s{\prime\futurelet\next\pr@m@s}
\def\pr@m@s{\ifx’\next\let\nxt\pr@@@s
            \else\ifx^\next\let\nxt\pr@@@t
             \else\let\nxt\egroup\fi\fi \nxt}
\def\pr@@@s#1{\prim@s} \def\pr@@@t#1#2{#2\egroup}
```

Mimo matematický mód má znak (’) speciální význam v syntaktickém pravidle ⟨*number*⟩ a jinak se chová obyčejně. Sází anglickou jednoduchou pravou uvozovku. Dva znaky za sebou (’’) vedou v CM fontech (a v $\mathcal{CS}$-fontech) na ligaturu (”), tj. pravá dvojitá anglická uvozovka.

Kn: 130, 305, 3–5, 51, 155, 201, 324, 357, 394–395, **Ol:** není nikdy použito.

\␣ [h]

Explicitní mezera. TEX vloží stejnou mezeru ⟨*glue*⟩, jako při povelu ␣10 pro \spacefactor=1000.

Kn: 285, 290, 10, 19, 86–87, 154, 283, 323, 351, 8, 73, 74, 163, 167, 381, **Ol:** *106*, 14, 22, 29, 89, 103, 157–158, 172, 211, 333, 355.

\+ [plain]

Řádek řízený tabulátory. Kód makra viz stranu 126.

Kn: 231–234, 249, 339, 354, **Ol:** 124, 125114–117, 119, 121, 126124, 129–130, 127134–135, 16065, 68, 16169.

\- [h]

Ekvivalent k \discretionary{*t*}{}{}. Místo pro rozdělení slova. Přitom znak *t* je definován jako \char\hyphenchar\font pro \hyphenchar\font v intervalu ⟨0, 255⟩ a *t*=„nic“ pro jiné hodnoty. Obvykle *t* reprezentuje rozdělovník (divis). Viz \hyphenchar, \defaulthyphenchar.

Kn: 455, 95, 283, 287, 292, **Ol:** 62337, 89, 14816, 216–217, 22168, 227, 230, 445785.

\/ [h]

Italická korekce. Každý znak má ve fontu údaj o mezeře, kterou je vhodné za tento znak vložit, následuje-li něco kolmého. Primitiv \/ přečte z fontu velikost této mezery a vloží ji za znak jako explicitní \kern.

Kn: 287, 292, 382, 455, 14, 64, 306, **Ol:** *305*, 1415–17, 311, 102–103, 194407, 195408–410, 2128, 220, 22171, 251207, 259277, 260284.

\%, \&, \#, \$, _ [plain]

Tyto řídicí sekvence umožní sazbu znaků se speciálními kategoriemi.

```
\chardef\%=‘\%
\chardef\&=‘\&
\chardef\#=‘\#
\chardef\$=‘\$
\def\_{\leavevmode \kern.06em \vbox{\hrule width.3em}}
{\catcode‘\_=\active \global\let_=\_}
```

Kn: 38, 43–44, 51, 53, 165, 202, 309, 356, **Ol:** 2130, 24, 2546, 32027, 32655, 355275.

\‘⟨*znak*⟩, \’⟨*znak*⟩, \=⟨*znak*⟩, \^⟨*znak*⟩, \.⟨*znak*⟩, \~⟨*znak*⟩, \"⟨*znak*⟩ [plain]

Akcenty definované v plainu primitivem \accent.

```
\def\‘#1{{\accent18 #1}}      à
\def\’#1{{\accent19 #1}}      á
\def\=#1{{\accent22 #1}}      ā
```

```
\def\^#1{{\accent94 #1}}      â
\def\.#1{{\accent95 #1}}      ȧ
\def\~#1{{\accent"7E #1}}     ã
\def\"#1{{\accent"7F #1}}     ä
```

Kn: 7–9, 24–25, 52–53, 55, 305, 335, 356, 387, 420, **Ol:** 348$_{226,\,233}$, 349.

`\,` , `\>`, `\;`, `\!` [plain]

Mezery v matematickém módu podle primitivních registrů.

```
\def\,{\mskip\thinmuskip}     % malá mezera
\def\>{\mskip\medmuskip}      % střední mezera
\def\;{\mskip\thickmuskip}    % větší mezera
\def\!{\mskip-\thinmuskip}    % záporná malá mezera
```

Kn: 167–173, 5, 171, 305, 357, **Ol:** 61$_{321,\,329}$, 62$_{334}$, 87$_{133–137}$, 133$_{179}$, 144$_{2,\,4}$, 147$_{12–13}$, 176$_{156,\,159,\,161}$, 190$_{356–357,\,367}$, 191$_{384–387,\,391,\,393}$, 192$_{394}$, 214$_{23}$, 342$_{200}$, 346$_{216}$, 355$_{278}$, 359$_{293,\,295}$, 367$_{344}$, 394$_{475,\,478}$, 406$_{574}$.

`\*` [plain]

Symbol × se objeví při rozdělení na konci řádku. Není-li rozdělen řádek, neobjeví se v sazbě nic (tak většinou sázíme násobení).

```
\def\*{\discretionary{\thinspace\the\textfont2\char2}{}{}}
```

Kn: 173, 357, **Ol:** 13$_{3}$, 14$_{18,\,21}$, 21$_{30}$, 25$_{48}$, 26$_{55,\,65–66}$, 46$_{133,\,136}$, 178$_{188}$.

`\aa`, `\AA` [plain]

Složené znaky å, Å.

```
\def\aa{\accent23a}
\def\AA{\leavevmode\setbox0=\hbox{h}\dimen0=\ht0
  \advance\dimen0 by-1ex
  \rlap{\raise.67\dimen0\hbox{\char'27}}A}
```

Kn: 356, **Ol:** není nikdy použito.

`\above` ⟨*dimen*⟩ [m]

Jako `\over`, navíc s možností definice vlastní tloušťky zlomkové čáry.

Kn: 292, 152, 444–445, 143, **Ol:** 152$_{40,\,43}$, 166, 407, 412.

`\abovedisplayshortskip` (plain: 0 pt plus 3 pt) [glue]

Vertikální mezera mezi textem a rovnicí (nad rovnicí), pokud poslední řádek textu je krátký, tj. nezasahuje do rovnice. Jinak je použito `\abovedisplayskip`.

Kn: 189, 274, 348, 415, **Ol:** *sekce 5.6*, 201, 202$_{437}$.

`\abovedisplayskip` (plain: 12 pt plus 3 pt minus 9 pt) [glue]

Vertikální mezera mezi textem a rovnicí (nad rovnicí), pokud není použita mezera z `\abovedisplayshortskip`.

Kn: 189, 190, 194, 274, 291, 348, 415, **Ol:** *sekce 5.6*, 198, 201, 202$_{436}$, 203$_{455}$, 204–205.

\abovewithdelims ⟨*delim1*⟩⟨*delim2*⟩⟨*dimen*⟩ [m]
Jako \above společně se závorkami ⟨*delim1*⟩ a ⟨*delim2*⟩ po stranách.

Kn: 292, 152, 444–445, **Ol:** 152$_{43}$.

\accent ⟨*8-bit number*⟩⟨*optional assignments*⟩⟨*character*⟩ [h]
Znak s kódem ⟨*8-bit number*⟩ (tzv. akcent) umístí nad ⟨*character*⟩. Parametr ⟨*character*⟩ je povel k vysázení znaku, tj. písmeno, \char⟨*number*⟩ nebo sekvence deklarovaná z \chardef. Pokud takový povel za \accent nenásleduje, není znám znak, nad kterým má být akcent usazen. V tomto případě se vysází akcent samostatně jako \char⟨*8-bit number*⟩ a \spacefactor se nastaví na 1000.

Ve fontu jsou akcenty vykresleny ve výšce, která je vhodná pro znaky vysoké 1 ex. Primitiv \accent změří výšku ⟨*character*⟩ a pokud je rovna 1 ex, položí pouze ⟨*character*⟩ a akcent přes sebe na společnou vertikální osu. Jinak navíc sníží nebo zvýší akcent o rozdíl mezi výškou ⟨*character*⟩ a hodnotou 1 ex. Tento pohyb provede podél (případně skloněné) osy znaku, jejíž sklon je definován ve \fontdimen1. Výsledný kompozitní znak má šířku rovnu šířce ⟨*character*⟩, i kdyby byla šířka akcentu větší.

⟨*optional assignments*⟩ umožňuje před ⟨*character*⟩ napsat libovolné množství povelů přiřazení, například \count255=13. Tato přiřazení se provedou před sestavením znaku s akcentem. Nebývá obvyklé tuto vlastnost využívat. Toto pravidlo má smysl jen v případě nastavení nového fontu, kterým bude sázeno písmeno. Nastavení fontu je totiž také přiřazení. Například:

```
137 \bf \accent20 \rm e
```

vysází písmeno „e“ s háčkem, přičemž písmeno je ve fontu \rm a akcent ve fontu \bf. Všimneme si, že \rm expanduje na dva povely přiřazení. Nejprve \fam=0, což je pro daný úkol nezajímavé, a potom povel \tenrm, který skutečně provede nastavení fontu. Viz též příklad u hesla \t.

Přítomnost primitivu \accent ve slově znemožňuje dělení slov podle vzorů v \patterns a \hyphenation. Pokud chceme například v češtině používat vzory dělení slov, nelze používat pro akcentované znaky \accent, ale je nutné použít osmibitový font. Primitiv \accent použijeme jen ve výjimečných slovech (například když citujeme cizí názvy s exotickými akcenty).

V plainu se makra \v a \' expandují na \accent. V csplainu je po použití makra \csaccents chování těchto maker předefinováno tak, že se expandují na příslušný osmibitový kód podle $\mathcal{CS}$-fontů. Pak dělení slov funguje i při „sedmibitovém“ zápise vstupního textu pomocí \v a \'.

Kn: 286, 9, 54, 86, 283, **Ol:** 89, 307, 334$_{121–123}$, 335$_{124–127, 133}$, 345$_{209}$, 348$_{232–237}$, 349, 369$_{377}$, 409$_{596}$, 423$_{668}$, 433–434, 439$_{756}$, 447$_{789}$, 451$_{802}$.

`\active` [plain]

Zkratka za číslo 13 (používá se v kontextu kategorie 13, tj. aktivní).

```
138 \chardef\active=13
```

Kn: 241, 343, 395, 421, **Ol:** 211_{3}, 333_{109}, 334_{120}, $408_{587–588}$, 409_{591}.

`\adjdemerits` (plain: 10 000) [integer]

„Demerits“ za vizuálně nekompatibilní řádky bezprostředně pod sebou.

Kn: 98, 273, 314, 348, **Ol:** 228–230.

`\advance` ⟨*numeric variable*⟩⟨*optional by*⟩⟨*value*⟩ [a]

Součet a přiřazení. Do registru ⟨*numeric variable*⟩ se uloží jeho hodnota zvětšená o specifikovanou ⟨*value*⟩. Typ registru ⟨*numeric variable*⟩ může být ⟨*number*⟩, ⟨*dimen*⟩, ⟨*glue*⟩ a ⟨*muglue*⟩. Údaj ⟨*value*⟩ zastupuje syntaktické pravidlo stejného názvu.

Kn: 276, 21, 118–119, 218, 256, 355, **Ol:** *79*, *sekce 3.3*, 29, 37–38, 41, 52–53, 57, 59, 62–63, 81, 84, 103, 106–107, 115, 121–123, 127, 180, 203, 207–208, 234, 236, 242, 244–245, 255, 257, 259, 262, 265, 267–271, 274, 276, 280, 282, 290–293, 317, 327, 335, 337, 375, 398, 400–401, 410–411, 427.

`\advancepageno` [plain]

Zvětší globálně stránkovou číslici o jedničku. Je-li záporná, zmenší o jedničku. Tisk záporných hodnot je realizován jako římské číslo hodnoty $-$`\pageno` (viz `\folio`).

```
139 \def\advancepageno{\ifnum\pageno<0
140         \global\advance\pageno by-1
141    \else\global\advance\pageno by1 \fi}
```

Kn: 257, 256, 362, 416, **Ol:** $262_{294,\,301}$, 263, 274_{458}, 281_{567}.

`\ae`, `\AE` [plain]

Sazba symbolů æ, Æ.

```
142 \chardef\ae="1A
143 \chardef\AE="1D
```

Kn: 52–53, vii, 17, 45–46, 239, 356, **Ol:** není nikdy použito.

`\afterassignment` ⟨*token*⟩ [h, v, m]

TEX si uloží ⟨*token*⟩ do přechodné paměti a vrátí jej do vstupní fronty až po provedení prvního přiřazení. Například:

```
144 \def\testtoken{\afterassignment\testznak \let\znak= }
145 \def\testznak{\ifcat\znak ... }
```

Tímto obratem jsme prostřednictvím `\let` přiřadili sekvenci `\znak` význam následujícího tokenu a hned po tomto přiřazení se do čtecí fronty zařadilo makro `\testznak`, které se při expanzi větví podle kategorie právě přečteného znaku.

Na rozdíl od \futurelet toto řešení nenechává přečtený token ve vstupní frontě.

Povel \afterassignment nemusí bezprostředně předcházet před přiřazovacím povelem. Více \afterassignment za sebou ale asi nemá smysl použít, protože platný je jen ten poslední.

Přiřazení prostřednictvím \setbox má v souvislosti s \afterassignment výjimku. „Pozdržený“ token se totiž nevkládá až za konec vkládaného boxu, jak by se dalo očekávat, ale hned za otevřením skupiny, která ohraničuje vkládaný box ještě před \everyhbox nebo \everyvbox. Token je tedy vložen v místě zahájení sestavování materiálu vkládaného boxu a v tu chvíli není přiřazení boxu ještě provedeno. Proto je obvykle potřeba další činnost dále pozdržet pomocí \aftergroup. Viz například sekci 3.7 v ukázce na řádku 238.

Kn: 279, 215, 352, 364, 376, 401, **Ol:** 72, 107_{192}, 113_{226}, 115_{237}, 126_{131}, 367, 371_{380}, $386_{436,\,438,\,443}$, 390_{454}, 410_{604}, 454_{806}.

\aftergroup ⟨*token*⟩ [h, v, m]

TEX si vloží ⟨*token*⟩ do přechodné paměti a vrátí jej do čtecí fronty až po ukončení skupiny, v rámci které byl \aftergroup použit.

Například české uvozovky píše uživatel jako \uv{slovo}. Kdyby bylo makro \uv definováno s parametrem, nedala by se uvnitř uvozeného textu realizovat změna kategorií například pro verbatim prostředí. Proto je makro \uv definováno použitím \aftergroup takto:

```
146 \def\uv{\bgroup\aftergroup\closequotes
147         \leavevmode\clqq\let\next=}
148 \def\closequotes{\unskip\crqq\relax}
```

Makro otevře skupinu, pak pomocí \aftergroup řekne TEXu, že při ukončení skupiny má vložit zavírací uvozovky, pak vloží otevírací uvozovky a nakonec pomocí \let\next= sejme nyní už nepotřebný token {$_1$. \clqq, resp. \crqq jsou makra na vysázení levých, resp. pravých českých uvozovek.

Pokud je v jedné skupině použito více než jedno \aftergroup, všechny tokeny z jednotlivých \aftergroup se vrátí po zavření skupiny zpět do čtecí fronty a sice ve stejném pořadí, v jakém byly za jednotlivými \aftergroup uvedeny.

Kn: 279, 215, 379, 363, 374, 377, **Ol:** 72, 92, 115_{238}, 214_{21}, 251_{220}, 338.

\allowbreak [plain]

Makro povolí v místě použití zlom.

```
149 \def\allowbreak{\penalty0 }
```

Kn: 174, 353, 396, **Ol:** není nikdy použito.

\atop [m]

Jako \over ovšem bez zlomkové čáry. Tj. umístí dva objekty nad sebe.

Kn: 292, 444, 152, 143, 145, 178, **Ol:** 152$_{39,\,42}$, 166, 407, 412.

`\atopwithdelims` ⟨*delim1*⟩⟨*delim2*⟩ [m]
Jako `\atop`, navíc kolem umístí závorky pružné velikosti podle ⟨*delim1*⟩ a ⟨*delim2*⟩ (použití např. pro kombinační čísla).

Kn: 292, 444, 152, 324, 360, **Ol:** 152$_{42}$, 344$_{202–203}$, 347$_{220}$.

`\b` ⟨*znak*⟩ [plain]
Udělá čárku pod písmenem. Například `\b a` vede na „a̲“.

```
\def\sh@ft#1{\dimen0=.00#1ex
  \multiply\dimen0 by \fontdimen1\font
  \kern-.0156\dimen0 } % compensate for slant
\def\b#1{{\lineskiplimit=0pt
    \oalign{\relax#1\crcr\hidewidth\sh@ft{29}%
    \vbox to.2ex{\hbox{\char22}\vss}\hidewidth}}}
```

Kn: 52, 356, **Ol:** 330$_{105}$, 363$_{303–304}$, 366$_{332–335}$, 378$_{400,\,405}$, 411$_{610}$.

`\badness` *read-only* [integer]
Hodnota badness pro naposledy sestavovaný box. Vzorec pro výpočet badness, viz stranu 99. Nekonečno je reprezentováno číslem 100 000.

Kn: 214, 271, 229, **Ol:** *98*, *sekce 3.5*, 101.

`\baselineskip` (plain: 12 pt) [glue]
Vertikální mezera mezi účařími dvou následujících řádků. Tato hodnota může být ignorována — viz `\lineskiplimit` a `\prevdepth`.

Kn: 80, 104, 194, 253, 274, 281, 342, 349, 78–79, 256, 351–352, 409, 414–415, **Ol:** *sekce 3.7*, 65, 72–73, 80, 108–110, 112–115, 138, 143, 173, 190, 192, 203, 212, 242–245, 252, 262–264, 266–268, 274, 276–277, 312, 323–324, 326, 342–343, 354, 387, 394, 407–411, 421.

`\batchmode` [a]
Nastaví se způsob zpracování, při němž TEX přeskakuje zcela všechny chyby jako při `\nonstopmode` a navíc nezobrazuje vůbec nic na terminálu. Viz též `\errorstopmode`.

Kn: 32, 277, 299, 336, **Ol:** 360–361, 406.

`\begingroup` [h, v, m]
Otevřít *skupinu* lze v TEXu dvěma způsoby. (1) Pomocí závorky `{`, což značí token kategorie 1 nebo zástupnou řídicí sekvenci `\bgroup`. (2) Použitím primitivu `\begingroup`. Vtip je v tom, že tyto dva způsoby se nesmí míchat. Tj. skupinu zahájenou způsobem (1) musím uzavřít závorkou `}` a skupinu zahájenou způsobem (2) je nutno uzavřít jedině pomocí primitivu `\endgroup` (a nikoli `\egroup`). Vhodným kombinováním obou metod lze lépe diagnostikovat případné omyly typu „zapomenutá závorka“.

Kn: 279, 21, 249, 262, 380, 407, 419, **Ol:** 25$_{48}$, 26$_{55,66}$, 40, 61$_{320}$, 84$_{115}$, 87$_{133}$, 127$_{141}$, 255$_{253}$, 269$_{377}$, 272$_{420}$, 339–340, 342$_{188}$, 356, 358.

\beginsection ⟨*text nadpisu*⟩\par [plain]

Jednoduché makro na zahájení sekce.

```
\outer\def\beginsection#1\par{%
  \vskip0pt plus.3\vsize\penalty-250
  \vskip0pt plus-.3\vsize \bigskip \vskip\parskip
  \message{#1}\leftline{\bf#1}\nobreak\smallskip\noindent}
```

Kn: 340–341, 355, **Ol:** není nikdy použito.

\belowdisplayshortskip (plain: 7 pt plus 3 pt minus 4 pt) [glue]

Vertikální mezera mezi rovnicí a textem (pod rovnicí), je-li poslední řádek před rovnicí krátký, tj. nezasahuje do rovnice. Jinak viz \belowdisplayskip.

Kn: 189, 274, 348, 415, **Ol:** *sekce 5.6*, 201, 202$_{439}$.

\belowdisplayskip (plain: 12 pt plus 3 pt minus 9 pt) [glue]

Vertikální mezera mezi rovnicí a textem (pod rovnicí), pokud není použito \belowdisplayshortskip

Kn: 189, 190, 194, 274, 291, 348, 415, **Ol:** *sekce 5.6*, 198, 201, 202$_{438}$, 203$_{456}$, 205, 342.

\bf, \bffam [plain]

Přepínač fontu do **polotučného řezu**. V matematickém módu se uplatní nastavení registru \fam na hodnotu \bffam a mimo matematický mód se uplatní přepínač \tenbf deklarovaný přímo primitivem \font.

```
\newfam\bffam \def\bf{\fam\bffam\tenbf} % \bf is family 6
\textfont\bffam=\tenbf \scriptfont\bffam=\sevenbf
                       \scriptscriptfont\bffam=\fivebf
```

Kn: 13–14, 164–165, 328, 351, 409, 414–415, **Ol:** 32–33, 35, 57, 64, 114, 119, 123, 129–130, 139, 141–143, 172–175, 180, 194, 221, 260, 262–263, 268, 276–277, 290, 336, 340, 365, 422, 427.

\bgroup [plain]

Alternativa k $\boxed{\{}_1$. Na rozdíl od přímého použití $\boxed{\{}_1$ nemá vliv na párování v ⟨*balanced text*⟩. Viz též \egroup, \begingroup, \endgroup.

```
\let\bgroup={
```

Protože zástupná řídicí sekvence neovlivní ⟨*balanced text*⟩ například pro těla definic, ale lze ji použít pro otevření boxu, můžeme psát třeba:

```
\def\otevribox{\hbox\bgroup abc }
\def\zavribox{\vrule\egroup}
\otevribox ... \zavribox % je shodné s \hbox{abc ... \vrule}
```

Kn: 363, 407, 421, 269, 351, 382, 385, **Ol:** 27–29, 40–42, 62, 84, 107, 128, 139, 141, 149, 196, 214, 226, 244, 251, 255, 278, 287, 316, 318, 333, 338–339, 412.

\big⟨*delimiter*⟩, \Big⟨*delimiter*⟩, \bigg⟨*delimiter*⟩, \Bigg⟨*delimiter*⟩ [plain]
Závorka proměnlivé velikosti, která se při použití těchto maker „natáhne“ do takové výšky, aby pokryla neviditelnou podpěru výšky (postupně 8,5 pt, 11,5 pt, 14,5 pt a 17,5 pt. V matematickém seznamu vystupuje samostatně jako atom typu Ord. Například použití dvojice `\bigg(...\bigg)` vytvoří závorky zvolené velikosti bez závislosti na velikosti matematické sazby mezi nimi. Navíc se sazba mezi těmito závorkami může rozdělit do více řádků. To při použití `\left(...\right)` není možné. Upozornění: příklad `\bigg(...\bigg)` není zcela v pořádku, protože závorky vystupují jako atomy typu Ord, zatímco je vhodnější (z důvodu automatického mezerování), aby závorky vystupovaly jako atomy typu Open a Close. K tomu je nutno použít `\biggl` a `\biggr` (viz následující heslo).

```
\def\big#1{{\hbox{$\left#1\vbox to8.5pt{}\right.\n@space$}}}
\def\Big#1{{\hbox{$\left#1\vbox to11.5pt{}\right.\n@space$}}}
\def\bigg#1{%
            {\hbox{$\left#1\vbox to14.5pt{}\right.\n@space$}}}
\def\Bigg#1{%
            {\hbox{$\left#1\vbox to17.5pt{}\right.\n@space$}}}
\def\n@space{\nulldelimiterspace=0pt \m@th}
```

Kn: 147, 171, 175, 196, 327, 359, 360, 414–415, **Ol:** 191–192, 341$_{174–175}$.

\bigl, \bigm, \bigr, \Bigl, \Bigm, \Bigr, \biggl, \biggm, \biggr, \Biggl, \Biggm, \Biggr [plain]
Makra se chovají analogicky jako `\big` až `\Bigg`, ovšem navíc v matematické sazbě vystupují jako atom typu Open (v názvu je „`l`“ jako „left“), Close (v názvu je „`r`“ jako „right“) a Rel (v názvu je „`m`“ jako „middle“). Správné použití závorek řízené velikosti tedy je: `\biggl(...\biggr)`.

```
\def\bigl{\mathopen\big} \def\bigr{\mathclose\big}
\def\bigm{\mathrel\big}          % první větší velikost
\def\Bigl{\mathopen\Big} \def\Bigr{\mathclose\Big}
\def\Bigm{\mathrel\Big}          % druhá větší velikost
\def\biggl{\mathopen\bigg} \def\biggr{\mathclose\bigg}
\def\biggm{\mathrel\bigg}        % třetí větší velikost
\def\Biggl{\mathopen\Bigg} \def\Biggr{\mathclose\Bigg}
\def\Biggm{\mathrel\Bigg}        % čtvrtá největší velikost
```

Kn: 146–147, 149–150, 155, 171, 175, 359, 437, **Ol:** 191, 192$_{395–396}$.

\bigbreak [plain]
Makro přidá do vertikálního seznamu mezeru velikosti `\bigskipamount` s hodnotou penalty −200 a odstraní případnou předcházející mezeru menší velikosti. Makro tuto činnost udělá jen tehdy, když nepředchází mezera větší nebo rovna

\bigskipamount. Vhodné například pro mezeru nad nadpisem sekce. Pokud totiž předchází před nadpisem menší mezera (třeba z \belowdisplayskip), je vhodné ji před sazbou nadpisu zrušit a pak teprve klást mezeru \bigskip a pod ní text nadpisu.

```
\def\bigbreak{\par\ifdim\lastskip<\bigskipamount
  \removelastskip\penalty-200\bigskip\fi}
```

Kn: 111, 116, 353, 363, **Ol:** 255_{259}, 396, 435.

\bigskip, \bigskipamount [plain]

Makro \bigskip vloží mezeru podle registru \bigskipamount. Tato velikost obvykle odpovídá výšce jednoho řádku. Proto můžeme říci, že \bigskip „vynechá jeden řádek“. Viz též \medskip a \smallskip.

```
\newskip\bigskipamount
\bigskipamount=12pt plus 4pt minus 4pt
\def\bigskip{\vskip\bigskipamount}
```

Kn: 70, 109, 111, 115–116, 123, 349, 352, 355, 407, 410–412, **Ol:** 33, 35, 57, 114, 119, 175–176, 212, 244–245, 255–257, 259–260, 269, 272, 279, 281, 290, 340, 342, 396, 435.

\binoppenalty (plain: 700) [integer]

Penalta se vloží za každý atom typu Bin. TEX tedy může rozdělit formuli za binárním operátorem. Výjimky, kdy se penalta do seznamu neklade, viz \relpenalty.

Kn: 446, 101, 174, 272, 322, 348, **Ol:** 160_{63}, 161_{71}, 210, 425.

\bordermatrix [plain]

Makro vytvoří matici včetně sloupečku vlevo od matice a řádku nad maticí. O použití makra — viz stranu 194.

```
\setbox0=\hbox{\tenex B} \p@renwd=\wd0 % šířka velké "("
\def\bordermatrix#1{\begingroup \m@th
  \setbox0=\vbox{\def\cr{\crcr\noalign{\kern2pt
      \global\let\cr=\endline}}%
    \ialign{$##$\hfil\kern2pt\kern\p@renwd
      &\thinspace\hfil$##$\hfil &&\quad\hfil$##$\hfil\crcr
      \omit\strut\hfil\crcr\noalign{\kern-\baselineskip}%
      #1\crcr\omit\strut\cr}}%
  \setbox2=\vbox{\unvcopy0 \global\setbox1=\lastbox}%
  \setbox2=\hbox{\unhbox1 \unskip \global\setbox1=\lastbox}%
  \setbox2=\hbox{$\kern\wd1 \kern-\p@renwd \left( \kern-\wd1
    \global\setbox1=\vbox{\box1\kern2pt}%
    \vcenter{\kern-\ht1 \unvbox0
      \kern-\baselineskip}\,\right)$}%
  \null\;\vbox{\kern\ht1 \box2}\endgroup}
```

Makro nejprve sestaví matici bez závorek a včetně řádku a sloupce kolem pomocí \halign (viz řádek 191). Přitom za první sloupec vloží mezeru \p@renwd (šířka největší alternativy pružné závorky) a před prvním řádkem bude prázdný řádek obsahující \strut následovaná zápornou mezerou -\baselineskip. Tím máme zaručeno, že makro bude přesně fungovat i s prvním řádkem nižším než \strut. Bohužel makro nebude dobře pracovat v případě prvního řádku vyššího než \strut.

Poslední řádek v \halign obsahuje pomocí \omit jediný box, uvnitř kterého je další \strut. Tento box má šířku prvního sloupce, což je pro další činnost důležité. Proto je tento box ze sestaveného materiálu (v boxu 0) postupně odebrán a uložen do box 1 (viz řádky 195 a 196). Nakonec je v boxu 2 vytvořena výsledná sazba. Nejprve je vložena mezera šířky prvního sloupce, pak se vrátíme o šířku závorky zpět a vložíme \left(. Dále se vrátíme na začátek boxu posunem doleva o šířku prvního sloupce. Dále přepočítáme box 1, aby jeho výška byla rovna výšce podpěry \strut plus 2 pt. Do sazby vložíme výsledek \halign posunutý nahoru o právě vypočítanou výšku boxu 1. Spodní část sazby musíme rovněž korigovat, neboť zde máme prázdný řádek. Sazbu pak uzavřeme pomocí \right). Celou záležitost umístíme do matematického seznamu jako \vbox. Přitom nahoře přidáváme do boxu mezeru, protože vnitřní sazba má první řádek vystrčen a je bezrozměrný. Nyní tomuto řádku dáváme rozměr.

Uvedu ještě jiné řešení tohoto problému, které jsem kdysi použil. Na vstupu ale musí být první (vystrčený) řádek napsán jako poslední. Pak tento řádek odebereme pomocí \lastbox a zbytek vložíme do \vcenter, kolem kterého jsou závorky. Levou závorku můžeme pomocí záporného kernu přistrčit analogicky, jako to bylo uděláno zde. Výslednou sazbu označíme jako \mathop a přidáme na konec zápis: \limits^⟨*odebraný řádek*⟩.

Kn: 177, 361, **Ol:** 191, 194$_{404}$.

\botmark [exp]

Expanduje na obsah poslední značky typu \mark v sestavené straně. Strana je kompletována algoritmem uzavření strany do boxu 255 a při té příležitosti je nastaven \botmark.

Kn: 258, 259–260, 213, 280, 262–263, **Ol:** *258*, 256, 259, 275, 391, 437, 444.

\box ⟨*8-bit number*⟩ [h, v, m]

Vrací box z registru ⟨*8-bit number*⟩ do h/v/m seznamu jako celek a obsah registru vymaže.

Kn: 278, 120–122, 151, 222, 346, 354, 386, 387, **Ol:** *82, 94*, 73–74, 82–85, 115, 119, 128, 143, 226, 238–239, 247–248, 255–257, 265, 272, 274, 319, 342, 399–401, 419, 427–429, 435, 448, 452.

\boxmaxdepth (plain: \maxdimen, tj. 16383.99999 pt) [dimen]

Maximální povolená hloubka boxu.

Při každém kompletování \vboxu TEX kontroluje, zda jeho hloubka není větší než \boxmaxdepth. Pokud má box hloubku větší, je zmenšena na \boxmaxdepth a o stejnou velikost je zvětšena výška, aby celková výška boxu zůstala zachována. Jinými slovy, při velké hloubce boxu se posune jeho účaří poněkud níže.

Kn: 81, 113, 249, 274, 255, 348, **Ol:** 101–102, 262_{303}, 395.

\brace, \brack [plain]

Tak jako makro \choose vytváří dvojici objektů nad sebou v kulatých závorkách, stejně pracují makra \brack (výsledek bude v hranatých závorkách) a \brace (výsledek bude ve složených závorkách). Příklad užití: `$a\brack b$` vede na $\begin{bmatrix} a \\ b \end{bmatrix}$.

```
\def\brack{\atopwithdelims[]}
\def\brace{\atopwithdelims\{\}}
```

Kn: 360, **Ol:** není nikdy použito.

\break [plain]

Vyvolá určitě řádkový nebo stránkový zlom.

```
\def\break{\penalty-10000 }
```

Kn: 94, 97, 106, 114, 193, 353, **Ol:** 75_{47}, 76, 122_{57}, 123_{97}, 211_{2}, 212_{4}, 244_{172}, 245_{194}, 246, 252_{238}, 259_{279}, 272_{419}, 279_{529}, 357_{284}.

\brokenpenalty (plain: 100) [integer]

Pod každý řádek může být automaticky připojena penalta podle registrů \interlinepenalty, \widowpenalty a \clubpenalty. K této penaltě je přičtena \brokenpenalty, pokud řádek obsahuje na konci \discretionary (tj. má rozdělené slovo) Například se nehodí, aby na konci stránky začínalo rozdělené slovo a končilo na další stránce. Tento jev lze potlačit vysokou hodnotou \brokenpenalty.

Upozorňuji, že to je jediný případ, kdy změna v textu odstavce, která nemění počet řádků v odstavci, může změnit stránkový zlom. Kdo to neví, může dlouho bádat, proč jediná korektura v odstavci zcela rozhodila stránkování kapitoly, přestože počet řádků změněného odstavce zůstal zachován.

Kn: 104, 105, 272, 348, **Ol:** 229.

\buildrel ⟨*nad relací*⟩ \over ⟨*relace*⟩ [plain]

V matematickém módu umístí symbol ⟨*relace*⟩ a osově nad ním menším stylem umístí symbol ⟨*nad relací*⟩. Například:

```
$A \buildrel x_i\over \longrightarrow B$
```

vede na $A \buildrel x_i\over \longrightarrow B$.

```
\def\buildrel#1\over#2{%
    \mathrel{\mathop{\kern0pt #2}\limits^{#1}}}
```

Všimneme si, že v makru v konstrukci `\mathop` je na první pohled zbytečný `\kern0pt`. Kdyby tam ale nebyl a současně by `#2` byl jednoznakový parametr, TEX by tento parametr centroval podle matematické osy, protože to je vlastnost atomu typu Op (viz stranu 163).

Kn: 361, 437, **Ol:** 188_{302}, 189_{333}.

`\bye` [plain]

Chová se jako primitiv `\end`. Pomocí `\supereject` navíc vyvolá ve výstupní rutině plainu makro `\dosupereject` (viz stranu 262, řádek 308), které dotiskne všechny zatím nevytištěné inserty třídy `\topins`.

```
208 \outer\def\bye{\par\vfill\supereject\end}
```

Kn: 87–88, 340, 357, **Ol:** 264, 279_{530}, 280_{558}.

`\c` ⟨*znak*⟩ [plain]

Umístí tzv. „cedillu“ pod ⟨*znak*⟩. Například `\c c` vede na ç. Tento „ocásek“ je v CM fontech uložen jako samostatný znak na pozici 24. Makro si nevystačí jen s primitivem `\accent`, protože tento primitiv začne zvedat nebo snižovat akcent podle výšky znaku. Primitiv je použitelný jen pro znaky vysoké 1 ex, kdy se žádný takový posun neprovádí. Jinak makro usadí „cedillu“ pomocí primitivu `\halign`.

```
209 \def\c#1{\setbox0=\hbox{#1}\ifdim\ht0=1ex\accent24 #1%
210     \else{\ooalign{\unhbox0\crcr
211       \hidewidth\char24\hidewidth}}\fi}
```

Kn: 24–25, 52, 356, **Ol:** 188_{299}, 189_{326}, $363_{303–304}$, $375_{394–395}$, $378_{400, 405}$.

`\cal` [plain]

Nastaví `\fam=2`, takže písmena velké abecedy povedou v matematickém módu na kaligrafické znaky, které jsou ve fontech `\tensy` až `\fivesy` nakresleny v místě verzálek: $\mathcal{ABCDEFGHIJKLMNOPQRSTUVWXYZ}$.

```
212 \textfont2=\tensy \scriptfont2=\sevensy
213                   \scriptscriptfont2=\fivesy
214 \def\cal{\fam=2 }
```

Kn: 164, 351, 431, 434, **Ol:** 191.

`\cases {`⟨*případy*⟩`}` [plain]

Vytvoří v matematickém módu svorku, která „větví případy“. Makro je implementováno jako:

```
215 \left\{ \vcenter{\halign{$#$ &#\cr ⟨případy⟩ \crcr}} \right.
```

Svorka tedy přizpůsobí velikost, protože je zpracována primitivem `\left`. Každý ⟨*případ*⟩ má dvě položky. První položka je sázena v matematickém módu a druhá v horizontálním módu.

```
216 \def\cases#1{\left\{\,\vcenter{\normalbaselines\m@th
217     \ialign{$##\hfil$&\quad##\hfil\crcr#1\crcr}}\right.}
```

Kn: 175, 362, **Ol:** 191, 194_{405}.

\catcode ⟨*8-bit number*⟩ (plain: viz strana 20) *restricted* [integer]
Každý znak má svou kategorii, což je číslo z intervalu $\langle 0, 15\rangle$. Kategorie ovlivní činnost token procesoru. Primitiv \catcode umožní čtení nebo změnu kategorie znaku s ASCII kódem ⟨*8-bit number*⟩.

Kn: 271, 134, 214, 305, 39, 343, 380–382, 384, 390–391, 421, 424, **Ol:** *20*, *sekce 1.3*, 11–12, 14–17, 21, 25–29, 41, 46, 73, 80, 123, 133–134, 150, 160, 211, 214, 217, 236, 333–334, 355, 389, 408–409.

\centering [plain]
Registr typu ⟨*glue*⟩. Vložením této mezery vlevo a vpravo od objektu dosáhneme „slabého“ centrování, které přestane pracovat v okamžiku výskytu \hfil nebo \hss. Hodí se pro hodnoty \tabskip v některých případech (viz třeba makro \eqalignno).

```
218 \newskip\centering \centering=0pt plus 1000pt minus 1000pt
```

Kn: 347, 348, 362, **Ol:** $359_{297,\,300}$, $385_{429,\,432}$.

\centerline ⟨*text*⟩ [plain]
⟨*text*⟩ bude centrován na samostatném řádku.

```
219 \def\centerline#1{\line{\hss#1\hss}}
```

Kn: 20, 24, 33, 71, 85, 101, 117, 232, 311, 340, 353, **Ol:** 279_{535}, 280_{546}.

\char ⟨*8-bit number*⟩ [h, m]
Ve vertikálním módu způsobí implicitní přechod do horizontálního módu a tam teprve pracuje. V horizontálním módu vysází znak z pozice ⟨*8-bit number*⟩ z aktuálního fontu. Pak se upraví \spacefactor podle \sfcode⟨*8-bit number*⟩. Jednotlivé dvojice znaků stejného fontu dále mohou podléhat změnám podle ligačního a kerningového programu fontu (strana 305).

Vyskytuje-li se \char v matematickém módu, TEX přepočítá ⟨*8-bit number*⟩ podle tabulky \mathcode a vloží do matematického seznamu odpovídající atom.

Kn: 286, 76, 86, 155, 283, 289, 452, 43–46, 340, 427, **Ol:** $68_{11,\,14}$, 69, 75, 89, 92_{158}, 213_{9}, $214_{11–12,\,20,\,22}$, 218_{50}, 220, 221_{70}, 224, 324, $335_{132,\,136}$, 336, 339_{155}, 345_{211}, 346, 365_{324}, $381_{419,\,421}$, 392, 409_{597}.

\chardef ⟨*control sequence*⟩⟨*equals*⟩⟨*8-bit number*⟩ [a]
Nová ⟨*control sequence*⟩ bude synonymem pro \char⟨*8-bit number*⟩.

Kn: 277, 44, 121, 155, 210, 214, 215, 272, 336, 452, **Ol:** 66, 69_{15}, 73, 75–76, 84, 89, 92_{159}, 220–221, 301, 319, 320_{27}, 324, 326_{55}, $330_{78–82}$, $334_{115–118}$, 336,

337$_{138,\,142–143}$, 348$_{225}$, 349$_{240}$, 363$_{305}$, 365$_{319}$, 367$_{345}$, 374$_{389}$, 381$_{416}$, 395, 399, 400$_{521,\,527,\,529–532}$, 401$_{547}$, 403–404, 408$_{583}$, 409$_{593}$, 422$_{664}$, 437$_{735}$, 441.

\choose [plain]

Sazba binomických koeficientů. `$ {n\choose k} $` vede na $\binom{n}{k}$.

```
\def\choose{\atopwithdelims()}
```

Kn: 139, 143, **Ol:** 344.

\chyph [csplain]

Nastaví české dělení slov a \frenchspacing. Viz soubor `hyphen.lan`.

```
\def\chyph{\language=\czech  \lccode`\'=`\'   \frenchspacing
           \lefthyphenmin=2  \righthyphenmin=3
           \message{Czech hyphenation used.
                    \string\frenchspacing\space is set on.}}
```

\cleaders [v, h, m]

Jako \leaders, navíc centruje; tj. mezera před prvním boxem z \leaders a posledním boxem je stejná.

Kn: 224, 225–226, 357, 374, **Ol:** *117*, 118$_{17}$, 120, 184$_{244,\,247}$, 319, 324, 355$_{276}$, 384, 458.

\cleartabs [plain]

Vymaže tabulátory. Kód makra, viz stranu 126.

Kn: 234, 354, **Ol:** 125$_{118}$, 126$_{127}$.

\closein ⟨*4-bit number*⟩ [h, v, m]

Uzavře vstupní soubor s číslem ⟨*4-bit number*⟩ (programátorské `close`). Na konci činnosti TEXu jsou všechny vstupní i výstupní soubory automaticky uzavřeny, takže tento primitiv není nutno příliš často používat. Viz též \openin, \closeout a \openout.

Kn: 217, 280, **Ol:** 288$_{13}$, 404, 410.

\closeout ⟨*4-bit number*⟩ [h, v, m]

Uzavře výstupní soubor s číslem ⟨*4-bit number*⟩ (programátorské `close`). Pokud chceme v rámci jednoho běhu TEXu nejprve do pomocného souboru něco zapsat (\write) a potom z něj číst (\input), je nutné před zahájením čtení soubor uzavřít pomocí \closeout.

Pozor, všechny operace s pracovním souborem pro zápis (\openout, \write a \closeout) se neprovádějí okamžitě, ale až při činnosti primitivu \shipout. Pokud chceme operaci provést okamžitě, musíme použít prefix \immediate.

Kn: 280, 226–228, 254, 422, **Ol:** 293, 378, 404, 410.

\clqq [csplain]

Vysází levé české uvozovky podle kódu $\mathcal{CS}$-fontů („). Viz soubor `extcode.tex`, resp. `il2code.tex`.

```
\chardef\clqq=254  \sfcode254=0
```

\clubpenalty (plain: 150) [integer]

Hodnotu tohoto registru TEX přičítá k celkové penaltě, která se připojuje za prvním řádkem odstavce. Vysoká hodnota tohoto registru potlačí výskyt tzv. „sirotků“, což je osamělý první řádek odstavce na konci strany. Viz též \widowpenalty, \displaywidowpenalty.

Kn: 104, 113, 272, 317, 348, 419, **Ol:** 229, 243_{145}, 258, 267_{328}, 344, 457.

\cmaccents [csplain]

Ruší změny způsobené makrem \csaccents. Tj. makra \^, \‘, \’, \v a \" dostanou zpět původní význam podle plainu a \r nebude definováno. Po načtení vzorů dělení slov provede csplain návrat k původním významům maker pro akcenty právě pomocí \cmaccents. Tím je zachována kompatibilita s plainem. Viz soubor `extcode.tex`, resp. `il2code.tex`.

```
\def\accentscommands{\string\^, \string\‘, \string\’,
   \string\v, \string\" and \string\r}
\def\cmaccentsmessage{%
   \message{The \accentscommands\space have original
            plainTeX meaning.}}
\def\cmaccents{\cmaccentsmessage
  \def\^##1{{\accent94 ##1}}\let\^^D=\^%
  \def\‘##1{{\accent18 ##1}}%
  \def\’##1{{\accent19 ##1}}%
  \def\v##1{{\accent20 ##1}}\let\^^_=\v%
  \def\"##1{{\accent"7F ##1}}%
  \let\r=\undefined\def\ou{{\accent23u}}}
```

\copy ⟨*8-bit number*⟩ [h, v, m]

Jako \box, ovšem obsah registru nemaže.

Kn: 222, 120, 151, 278, 329, 374, 386, 407, **Ol:** *82*, 83, 119, 127_{142}, 161, 238, 270_{397}, 274_{454}, 319_{17}, 427_{690}, 428–430, 439_{753}.

\copyright [plain]

Vytvoří značku ©. Klade na sebe dva symboly. Jednak malé písmeno c povýšené o 0,07 ex a jednak ◯, což je kroužek z fontu `cmsy10`.

```
\def\copyright{{\ooalign{\hfil\raise.07ex\hbox{c}\hfil
                  \crcr\mathhexbox20D}}}
```

Kn: ii, 308, 339, 356 **Ol:** 4.

\count ⟨*8-bit number*⟩ [integer]
Povel umožní přístup k 256 registrům typu ⟨*number*⟩.

Kn: 271, 276, 118–122, 207–208, 346–347, 379, **Ol:** 72–75, 78–79, 127, 248–249, 251, 255, 257, 281, 310, 319–320, 327, 330, 366, 371, 390, 395, 399–402, 443.

\countdef ⟨*control sequence*⟩⟨*equals*⟩⟨*8-bit number*⟩ [a]
Nová ⟨*control sequence*⟩ bude synonymem pro \count⟨*8-bit number*⟩.

Kn: 277, 119, 121, 210, 215, 271, 346–347, **Ol:** 66, 73_{32}, 74, 326–327, $330_{87,\,98}$, 395, $399_{507,\,519–520}$, 400_{523}, 401_{540}, 414_{620}.

\cr [spec]
Ukončení řádku (sloupce) v \halign (\valign).

Kn: 245, 248, 275, 282, 351, 352, 385–386, 175–177, 190–197, 231–238, 412, 418, 421, **Ol:** *sekce 4.3*, 125–137, 139, 141–143, 180–181, 194, 197, 204–207, 268, 342, 345, 353–354, 358, 408, 411.

\crcr [spec]
Pracuje jako \cr. Pokud ale bylo právě provedeno \cr nebo \crcr nebo \noalign, neprovede tento povel nic.

Kn: 249, 275, 282, 385, 361–362, 412, 421, **Ol:** 127–128, 130, 135, 139, 141, 184, 188–189, 204–205, 339, 342, 345–346, 348, 350, 354, 359, 385, 394, 408–410.

\crqq [csplain]
Vysází pravé české uvozovky (“) podle kódu $\mathcal{CS}$-fontů. Viz soubor extcode.tex, resp. il2code.tex.

```
240 \chardef\crqq=255  \sfcode255=0
```

\csaccents [csplain]
Předefinuje původní významy maker plainu \^, \‘, \’, \v a \". Místo toho, aby expandovaly na primitiv \accent, který znemožní dělení slova, expandují na odpovídající osmibitové kódy podle $\mathcal{CS}$-fontů. Dále je definováno makro \r, které expanduje na písmeno s kroužkem podle $\mathcal{CS}$-fontů. Tj. „\r U“ vede na „Ů“ a „\r u“ vede na „ů“. Viz soubor extcode.tex, resp. il2code.tex.

\csname ⟨*tokens*⟩\endcsname [exp]
Výsledkem expanze je řídicí sekvence, jejíž identifikátor bude sestaven ze znaků, které vzniknou po úplné expanzi ⟨*tokens*⟩. Tento primitiv bývá použit při implementaci řídicích sekvencí, které mohou obsahovat ve svých identifikátorech i jiné znaky, než písmena.

Pokud takto sestavená řídicí sekvence není dříve definována a nejedná se o primitiv, dostává význam \relax.

Pokud se při expanzi ⟨*tokens*⟩ narazí na neexpandovatelnou řídicí sekvenci (například \relax, \kern nebo \pageno), sestavení identifikátoru havaruje s hláškou Missing \endcsname inserted.

```
\expandafter \def \csname !?$ \endcsname {Ahoj}
\def\A{aa\B cc} \def\B{BB} \def\PPaaBBccQQ{lidi!}
\csname hrule\endcsname
\csname !?$ \endcsname \space \csname PP\A QQ\endcsname
\csname Tato sekvence není definována,  to nevadí\endcsname
```

Tento příklad vyprodukuje vodorovnou čáru `\hrule` a pod ní nápis „Ahoj lidi!“. Slovo „Ahoj“ je výsledek expanze řídicí sekvence `!?$␣`. Slovo „lidi!“ je definováno v řídicí sekvenci `PPaaBBccQQ`. Poslední řídicí sekvence s identifikátorem `Tato␣sekvence␣není␣definována,␣to␣nevadí`, se bude chovat v hlavním procesoru jako `\relax` a nezpůsobí chybu.

Všimneme si, že sekvence `!?$␣` obsahuje i mezeru. Pokud ji tam nechceme, musíme správně navázat `\endcsname`.

Specialisty upozorňuji na skutečnost, že pomocí `\csname...\endcsname` může řídicí sekvence změnit svůj význam, pokud byla dříve nedefinovaná:

```
\ifx\xyz\undefined ANO \else NE \fi       % ANO
\ifx\xyz\relax     ANO \else NE \fi       % NE
\csname xyz\endcsname % od toho okamžiku má \xyz nový význam
\ifx\xyz\undefined ANO \else NE \fi       % NE
\ifx\xyz\relax     ANO \else NE \fi       % ANO
```

Toto je jediná situace, kdy lze v TEXu provést jistý druh přiřazení na úrovni expand procesoru. Přitom je toto přiřazení expand procesorem posléze detekovatelné. V nějakých velmi speciálních případech se to možná může někomu hodit.

Kn: 213, 40–41, 348, 375, **Ol:** 37–38, 44–45, 51, 53, 134, 174–176, 207–208, 278, 291–292, 298, 358, 378, 402–403, 438.

`\czech` [csplain]

Číslo jazyka pro češtinu. Viz soubor `hyphen.lan`.

```
\newcount\czech       \global\czech=5
```

`\d` [plain]

Udělá tečku pod písmenem, například `\d a` vede na „ạ“.

```
\def\sh@ft#1{\dimen0=.00#1ex \multiply\dimen0\fontdimen1\font
  \kern-.0156\dimen0} % compensate for slant
\def\d#1{{\lineskiplimit=0pt
  \oalign{\relax#1\crcr\hidewidth\sh@ft{10}.\hidewidth}}}
```

Makro `\sh@ft` se stará o kompenzování polohy tečky podle (případně) nakloněné osy písmene.

Kn: 52–53, 356, **Ol:** 191$_{392–393}$, 330$_{105}$, 378$_{401, 404–405}$.

\dag, \ddag [plain]
Znaky † a ‡, které jsou připraveny ve fontu cmsy10.

```
256 \def\dag{\mathhexbox279}
257 \def\ddag{\mathhexbox27A}
```

Kn: 53, 117, 356, 438–439, **Ol:** 292_{76}.

\day [integer]
Den v měsíci. Údaj je v okamžiku spuštění TEXu načten ze systémové proměnné data a času.

Kn: 273, 349, 406, **Ol:** není nikdy použito.

\deadcycles (iniTEX: 0) *global, restricted* [integer]
Na počátku je \deadcycles=0. Při zahájení výstupní rutiny se zvedne hodnota \deadcycles o jedničku. Po každém povelu \shipout klesá tento registr zpět na nulu. Pokud tedy ve výstupní rutině použijeme \shipout, zůstává \deadcycles=0. Jinak tento registr měří počet „mrtvých“ výstupních rutin, které nerealizovaly výstup do dvi. Viz též \maxdeadcycles.

Kn: 255, 214, 264, 271, 283, 401, **Ol:** 262, 272, 357, 395.

\def ⟨*control sequence*⟩⟨*parameter text*⟩{⟨*balanced text*⟩} [a]
Definice nového makra. Údaj ⟨*parameter text*⟩ je v sekci 2.1 označován volným českým překladem ⟨*maska parametrů*⟩ (strana 36). Závorky obklopující tělo definice ⟨*balanced text*⟩ musí být jedině explicitní. Levá závorka ukončuje ⟨*parameter text*⟩, který je čten bez expanze. Pravá závorka ukončuje ⟨*balanced text*⟩, který je v případě \def a \gdef čten rovněž bez expanze. Na druhé straně při \edef a \xdef je prováděna při čtení ⟨*balanced text*⟩ expanze.

Kn: 444, 136, 199–208, 215, 275–276. **Ol:** *sekce 2.1*, primitiv je použit na velkém množství stránek knihy.

\defaulthyphenchar (plain: ‘\-) [integer]
Hodnota pro \hyphenchar v okamžiku zavedení fontu před případnou změnou tohoto parametru.

Kn: 273, 348, **Ol:** 334, 363_{312}, 374.

\defaultskewchar (plain: −1) [integer]
Hodnota pro \skewchar v okamžiku zavedení fontu před případnou změnou tohoto parametru.

Kn: 273, 348, **Ol:** 433–434.

\delcode ⟨*8-bit number*⟩ (plain: viz stranu 185) *restricted* [integer]
Hodnota určuje, jak se znak s ASCII hodnotou ⟨*8-bit number*⟩ bude chovat, pokud bude použit jako ⟨*delimiter*⟩, tj. za \left a \right. Tato informace je ve tvaru ⟨*24-bit number*⟩. Hexadecimálně např. ABCDEF znamená, že malá

varianta se hledá v rodině fontu A s kódem BC a velká v rodině fontu D s kódem EF. Ve fontu s velkou variantou se předpokládá zřetězení na postupně se zvětšující znaky, končící odkazem na segmenty, z nichž lze poskládat libovolně vysoký symbol.

IniTEX nastavuje `\delcode` všech znaků na −1 (nepovolený ⟨*delimiter*⟩) s výjimkou znaku „.“, který má hodnotu 0 (prázdný delimiter, viz též `\nulldelimiterspace`). Plain nastavuje `\delcode` některých znaků k praktickému použití ve spolupráci s fontem `cmex10` (viz stranu 185).

Kn: 156, 290, 214, 271, 345, **Ol:** *150*, 150$_{29}$, 151$_{31}$, 152–153, 165, 169, 185, 319, 407, 423.

`\delimiter` ⟨*27-bit number*⟩ [m]
Při použití v povelech `\left` a `\right` jako ⟨*delimiter*⟩ se vysází závorka proměnlivé velikosti. V takovém případě se použije nižších 24 bitů k získání analogické informace, jako u `\delcode`. Mimo povel `\left`, `\right` se vysází znak v malé verzi jako atom. Typ atomu je definován (podobně jako třída v `\mathcode`) ve zbylých třech bitech.

Tabulka všech maker, které se opírají o primitiv `\delimiter`, viz stranu 185.

Kn: 156, 289–290, 359, **Ol:** *150*, 151$_{30,\,32}$, 152, 178$_{185–186}$, 185$_{258–281}$, 319, 324, 333.

`\delimiterfactor` (plain: 901) [integer]
Určuje koeficient k poměru minimální výšky natahovací závorky (delimiter) k výšce formule, která je do závorek uzavřená. Platí obvyklý vztah $k =$ `\delimiterfactor`$/1000$.

Kn: 446, 152, 273, 348, **Ol:** *164*, 164$_{83}$, 166.

`\delimitershortfall` (plain: 5 pt) [dimen]
Maximální velikost části matematické formule, které je dovoleno přečnívat přes natahovací závorku.

Kn: 446, 152 274, 348, **Ol:** *164*, 164$_{84}$, 166.

`\dimen` ⟨*8-bit number*⟩ [dimen]
Povel umožní přístup k 256 registrům typu ⟨*dimen*⟩.

Kn: 271, 276, 118–122, 346–347, 349, 360, 363, 395, **Ol:** 73–75, 78, 80, 82, 113, 115, 119, 127, 244–246, 248–249, 251–252, 255, 257, 267–268, 281–283, 330, 335, 339, 350, 366, 390, 399–401, 410–411, 427, 443, 454.

`\dimendef` ⟨*control sequence*⟩⟨*equals*⟩⟨*8-bit number*⟩ [a]
Nová ⟨*control sequence*⟩ bude synonymem pro `\dimen`⟨*8-bit number*⟩.

Kn: 277, 119, 215, 346–347, **Ol:** 66, 73–74, 326–327, 330$_{99–101}$, 400$_{524}$, 401.

`\discretionary {`⟨*pre-break*⟩`}{`⟨*post-break*⟩`}{`⟨*no-break*⟩`}` [h, m]
Místo pro rozdělení slova. V místě použití bude sázeno ⟨*no-break*⟩. Při rozdělení

slova bude ⟨*no-break*⟩ nahrazeno ⟨*pre-break*⟩ před rozdělením a ⟨*post-break*⟩ na začátku dalšího řádku. Například `Zu\discretionary{k-}{k}{ck}er` označuje slovo Zucker, které v případě rozdělení má tvar Zuk-ker.

Všechny parametry ⟨*pre-break*⟩, ⟨*post-break*⟩ a ⟨*no-break*⟩ mohou obsahovat povely pro sazbu v horizontálním módu ve formátu ⟨*balanced text*⟩. Před závorkami může být ⟨*filler*⟩. Parametry nesmí produkovat ⟨*glue*⟩, nesmí mezi nimi být obsažena penalta ani nový `\discretionary` a není možno přepnout do matematického módu pomocí `$`.

Je-li `\discretionary` použit v matematickém módu, jsou přesto parametry ⟨*pre-break*⟩ a ⟨*post-break*⟩ zpracovány v horizontálním módu. V takovém případě navíc ⟨*no-break*⟩ musí být prázdný.

Kn: 287, 292, 95–96, 283, 286, **Ol:** 88–89, 145, 160_{67}, 161, 209–210, 214_{24}, $215_{25–29}$, 216, $217_{32–33}$, $218_{50–51}$, 219–220, 232, 335_{132}, 344, 374.

`\displayindent` [dimen]

Nechť je před rovnicí n řádků odstavce. Registr `\displayindent` obsahuje v display módu velikost odsazení řádku $n+2$ tohoto odstavce, který odpovídá místu s výskytem rovnice. Podle hodnoty tohoto registru se nakonec rovnice do sazby umístí.

Hodnota registru je ve většině případů rovna 0 pt. Od tohoto „standardu“ se hodnota může odchylovat při použití `\parshape` nebo nenulového `\hangindent`.

Kn: 188, 190, 274, 291, 349, **Ol:** *199*, 201, 205.

`\displaylimits` [m]

Povel přiděluje atomu typu Op (který musí být jako poslední v matematickém seznamu) příznak „displaylimits“. Má-li atom typu Op tento příznak, jsou indexy sázeny po straně ve stylu T (a menších) a nad a pod ve stylu D.

Každý nově vzniklý atom typu Op má implicitně tento příznak. Ovšem příznak může být pozměněn primitivy `\limits` a `\nolimits`. Hodnota příznaku bude nakonec odpovídat tomu primitivu, který byl použit za atomem typu Op zcela naposled. Jinými slovy, `\displaylimits` může negovat předchozí nastavení `\limits` nebo `\nolimits` z nějakého makra.

Kn: 292, 144, 159, 443, **Ol:** 386, 406.

`\displaylines` [plain]

Toto makro umožní klást do display módu více rovnic pod sebou zcela samostatně. Mezi rovnicemi je penalta hodnoty `\interdisplaylinepenalty`. Použití makra má tvar:

```
$$\displaylines{⟨první rovnice⟩ \cr
                ⟨druhá rovnice⟩ \cr
                ⟨atd.⟩          \cr
```

```
261                ⟨poslední rovnice⟩ \cr}$$
```

Každá rovnice bude umístěna do řádku jako při `\line{\hfil⟨rovnice⟩\hfil}`, ovšem v matematickém módu. Není použito žádné dělení na části pomocí `&`, jak to známe u maker `\eqalign`, `\eqalignno` a `\leqalignno`.

```
262 \newif\ifdt@p
263 \newcount\interdisplaylinepenalty\interdisplaylinepenalty=100
264 \def\displ@y{\global\dt@ptrue \openup\jot \m@th
265   \everycr={\noalign{\ifdt@p \global\dt@pfalse
266               \ifdim\prevdepth>-1000pt \vskip-\lineskiplimit
267                  \vskip\normallineskiplimit \fi
268             \else \penalty\interdisplaylinepenalty \fi}}}
269 \def\@lign{\tabskip=0pt\everycr={}} % restore inside \displ@y
270 \def\displaylines#1{\displ@y \tabskip=0pt
271    \halign{\hbox to\displaywidth{%
272      $\@lign\hfil\displaystyle##\hfil$}\crcr #1\crcr}}
```

Makro `\displ@y` nastavuje `\everycr` tak, aby mezi rovnicemi bylo vloženo `\interdisplaylinepenalty`. Dále pomocí `\openup\jot` jsou zvětšeny hodnoty `\lineskip`, `\lineskiplimit` a `\baselineskip` o 3 pt. Nad první rovnicí se do `\noalign` vkládá další materiál, který kompenzuje rozdíl mezi původním a změněným `\lineskiplimit`.

Ve vlastním makru `\displaylines` máme v deklaraci řádku tabulky `\halign` řečeno, že rovnice budou mít šířku `\displaywidth` a budou centrovány pomocí `\hfil` z obou stran. Makro `\@lign` vrací pro každou rovnici jednotlivě `\everycr` a `\tabskip` do „normálního stavu“, protože není vyloučeno, že uživatel použije v rovnici své vlastní `\halign`.

Kn: 194, 196, 362, **Ol:** 206, 359$_{296}$, 380–381, 385$_{428}$, 410.

`\displaystyle` [m]

Nastaví styl *D* (displaystyle) pro sestavování matematického seznamu. Jedná se o základní velikost, která je použita jako implicitní v display matematickém módu. Viz též `\textstyle`, `\scriptstyle` a `\scriptscriptstyle`.

Kn: 292, 141–142, 362, **Ol:** *154*, 62$_{338}$, 155$_{47}$, 163, 177, 184$_{231,\,234,\,238,\,240}$, 189$_{330}$, 193$_{397}$, 354$_{272}$, 359$_{294–295,\,298–299}$, 385$_{430–431}$, 393$_{471}$, 429, 441.

`\displaywidowpenalty` (plain: 50) [integer]

Hodnota tohoto registru se přičítá k penaltě před řádkem v odstavci, za kterým následuje rovnice (display mód). Díváme-li se tedy na úsek textu před rovnicí jako na samostatný odstavec, pracuje tento registr analogicky, jako `\widowpenalty` v odstavci bez rovnic.

Kn: 104, 272, 348, **Ol:** 348, 416, 457.

`\displaywidth` [dimen]

Šířka řádku odstavce, který odpovídá místu s výskytem rovnice. Jestliže je

před rovnicí n řádků, jedná se o šířku řádku $n+2$. Na registr se má smysl ptát a měnit jej v display módu.

Obvykle má registr hodnotu \hsize, ovšem od této hodnoty se může lišit při použití \parshape nebo nenulového \hangindent.

Kn: 188, 190, 274, 349, **Ol:** *199*, 200, 202$_{442}$, 203$_{445}$, 205$_{464}$, 206, 354$_{271}$, 359$_{298}$, 385$_{430,\,432–433}$.

\divide ⟨*numeric variable*⟩⟨*optional by*⟩⟨*number*⟩ [a]
Celočíselné dělení registru ⟨*numeric variable*⟩ číslem ⟨*number*⟩. Výsledek je uložen zpět do ⟨*numeric variable*⟩.

Kn: 276, 118–119, 218–219, 391, 397, 398, 417, **Ol:** *81*, 78$_{56}$, 79, 81$_{77,\,82}$, 82$_{92–93}$, 115$_{240}$, 127$_{138}$, 244$_{158,\,166,\,176}$, 270$_{394}$, 274$_{450}$, 317, 327, 433$_{707}$.

\dospecials [plain]
Předpokládá se, že „uživatel“ makra nejprve definuje nějakým způsobem makro \do s jedním parametrem a potom spustí \dospecials. Tím se aplikuje zrovna definované \do postupně na parametry \␣, \\, \{, \}, \$, \&, \#, \^, \^^K, _, \^^A, \% a \~. Například po použití:

```
\def\do#1{\catcode`#1=12 } \dospecials
```

máme pro všechny znaky se speciálními kategoriemi (jak je používá plain) nastavenu kategorii 12.

```
\def\dospecials{\do\ \do\\\do\{\do\}\do\$\do\&%
  \do\#\do\^\do\^^K\do\_\do\^^A\do\%\do\~}
```

Kn: 344, 380, 422–423, **Ol:** 27$_{76}$, 29$_{83}$.

\dotfill [plain]
Vyplní mezeru tečkami. Mezera má pružnost typu \hfill a není vhodná pro použití v obsahu, protože tam je nutno použít namísto primitivu \cleaders primitiv \leaders.

```
\def\dotfill{\cleaders
             \hbox{$\m@th \mkern1.5mu.\mkern1.5mu$}\hfill}
```

Kn: 244, 334–335, 340–341, 357, **Ol:** 120.

\dots [plain]
Vytiskne tři tečky. Použitelné v matematickém i horizontálním módu.

```
\def\dots{\relax\ifmmode\ldots\else$\m@th\ldots\,$\fi}
```

Kn: 173, 365, **Ol:** není nikdy použito.

\doublehyphendemerits (plain: 10 000) [integer]
„Demerits“ za takový zlom řádků, že dva řádky těsně pod sebou končí rozděleným slovem.

Kn: 98, 273, 348, 451, **Ol:** 228–230, 276_{487}.

\downbracefill [plain]
Vodorovná svorka natahovací délky. Kód makra, viz stranu 184.

Kn: 225–226, 331, 357, **Ol:** $184_{237,\,251}$.

\dp ⟨*8-bit number*⟩ *restricted* [dimen]
Umožňuje přístup k hodnotě „hloubka boxu“, která odpovídá boxu z registru ⟨*8-bit number*⟩.

Kn: 388–389, 120, 271, 316, 417, **Ol:** 83, 95_{170}, 115_{239}, 251_{213}, 252_{228}, $255_{256,\,262}$, $269_{387–388}$, 282_{586}, 418_{630}, 427_{689}, 435_{730}, 448_{790}, 452_{805}.

\dump [v]
Chová se jako \end. Navíc uloží obsah paměti do binárního souboru s příponou fmt. Povel je možný jen v iniTEXu a není povolen uvnitř skupiny. Pozdějším načtením souboru fmt před zpracováním dokumentu obnoví TEX rychle znalosti, které udržoval ve své paměti v okamžiku povelu \dump.

Kn: 283, 286, 336, 344, **Ol:** 90, 284–285, 295, 300, 356.

\edef ⟨*control sequence*⟩⟨*parameter text*⟩{⟨*balanced text*⟩} [a]
Jako \def s tím rozdílem, že se tělo definice expanduje před přiřazením nové řídicí sekvenci a nikoli až v okamžiku použití sekvence. Tato expanze je „úplná“, tj. končí až na úrovni neexpandovatelných primitivů nebo tokenů kategorie různé od 13. Při této expanzi se neexpandují pouze:

- Tokeny, před kterými předchází \noexpand.
- Obsah proměnných typu ⟨*tokens*⟩, rozvinutých pomocí \the.

Kn: 275, 215–216, 328, 348, 373–374, **Ol:** *sekce 2.1*, 31–32, 39–40, 54–55, 57–58, 61, 65–66, 71, 236, 251, 258, 290–292, 316, 351, 365, 378, 391, 402–403, 411, 436, 439.

\egroup [plain]
Alternativa k }$_2$. Na rozdíl od přímého (explicitního) použití }$_1$ nemá vliv na párování v ⟨*balanced text*⟩. Viz též \bgroup, \begingroup, \endgroup.

```
279 \let\egroup=}
```

Kn: 363, 407, 421, 269, 351, 382, 385, **Ol:** 27, 40–41, 62, 84, 107, 127–129, 139, 141, 143, 149, 196, 226, 244, 251, 255, 278, 318, 333, 339–340, 412.

\ehyph [csplain]
Nastaví dělení slov podle Americké angličtiny a \nonfrenchspacing. Viz soubor hyphen.lan.

```
280 \def\ehyph{\language=\USenglish \lccode`\'=0
281     \nonfrenchspacing \lefthyphenmin=2 \righthyphenmin=3
282     \message{English hyphenation used.
283             \string\nonfrenchspacing\space is set on.}}
```

\eject [plain]

Ukončí odstavec a vloží penaltu −10 000, což okamžitě vyvolá algoritmus uzavření strany. Bývá obvyklé před \eject psát \vfill.

```
284 \def\eject{\par\break}
```

Kn: 24–25, 105, 109, 189, 353, 418, 419, **Ol:** 14_{20}, 272, 275, $279_{529,\,538,\,542}$, $280_{547,\,551,\,555,\,558}$.

\else [exp]

Součást konstrukce \if⟨*xy*⟩ .. \else .. \fi.

Kn: 213, 207, 210, **Ol:** *sekce 2.3* a dále často tam, kde je nějaký primitiv typu \if....

\emergencystretch (iniTEX: 0 pt) [dimen]

Hodnota roztažení přidaná ke každému řádku při neúspěšném prvním a druhém průchodu algoritmu řádkového zlomu, tj. když startuje třetí (poslední) průchod. Viz též \tolerance, \pretolerance.

Kn: 107, 274, **Ol:** *228*, *sekce 6.4*, 226_{102}, 230–232, 276_{485}.

\empty [plain]

Makro, které expanduje na prázdnou posloupnost tokenů.

```
285 \def\empty{}
```

Kn: 263, 351, 378, **Ol:** 37_{40}, 49_{170}, 58_{281}, $134_{196,\,204}$, 251_{206}, 259.

\end [v]

Ve vnitřním vertikálním módu způsobí \end chybu a ukončení TEXu. V matematickém, resp. vnitřním horizontálním módu, způsobí \end chybové hlášení `missing $`, resp. `} inserted`. Po vložení chybějícího „$“ nebo „}“ se čte povel \end znovu.

V odstavcovém módu vloží \end automaticky před sebe token par. Povel \end je tedy znovu načten po uzavření odstavce.

V hlavním vertikálním módu ukončí \end činnost TEXu, pokud je přípravná oblast i aktuální strana prázdná a současně je \deadcycles=0. Jinak zůstává povel \end ve čtecí frontě a bude čten vstupní bránou hlavního procesoru znova. V takovém případě se vloží do vertikálního seznamu prázdný box šířky \hsize (který nad sebou nemá meziřádkovou mezeru), dále \vfill a konečně \penalty$-2^{30}$, která vyvolá algoritmus uzavření strany. Po ukončení tohoto algoritmu je povel \end čten hlavním procesorem znova. Tím se pokus o ukončení činnosti TEXu opakuje.

Není-li ukončena skupina, objeví se při ukončení TEXu varovné hlášení `\end occured inside a group at level` *n*. Podobné hlášení se objeví, pokud není uzavřena nějaká konstrukce typu \if...\fi.

Kn: 264, 283, 23, 26, 87, 286, 299, 27, 336, 403, **Ol:** $24_{35,\,40}$, 37, $58_{282,\,286–287}$, 64–65, 88, 89_{146}, 90–91, 92_{157}, 123_{113}, 239, 252_{241}, 253–254, 264, 272, 277_{492}, 279_{530}, 284, $287_{3–4}$, 329, 345_{208}, 356.

\endcsname [exp]

Ukončení a kompletování řídicí sekvence zahájené primitivem \csname.

Kn: 213, 40–41, 283, 348, 375, **Ol:** 37–38, 44–45, 51, 53, 134, 174–176, 207–208, 278, 291–292, 298, 349–350, 378, 402–403, 438.

\endgraf [plain]

Alternativní sekvence pro povel \par. Hodí se při předefinování řídicí sekvence par.

```
\let\endgraf=\par
```

Kn: 262, 286, 331, 351, 407, 416, 419, **Ol:** 29_{86}, $84_{119,\,127}$, 85, 91_{151}, 92, 203_{456}, $226_{91,\,102}$, $237_{135–136}$.

\endgroup [h, v, m]

Ukončuje skupinu, která byla otevřena pomocí primitivu \begingroup.

Kn: 279, 21, 249, 262, 380, 407, 419, **Ol:** 25_{53}, $26_{57,\,68}$, 40, 62_{338}, 84_{118}, 87_{142}, 127_{145}, 255_{264}, 269_{383}, 272_{425}, 339–340, 342_{201}, 356.

\endinput [exp]

Po dočtení vstupní fronty tokenů a aktuálního řádku bude ukončeno čtení z aktuálního souboru a další tokeny bude expand procesor čerpat z odložené vstupní fronty a zbytku řádku v době, kdy bylo zahájeno čtení tohoto souboru povelem \input (viz \input).

Je třeba dávat pozor na zbytky:

```
\def\zkusendinput{\endinput abc}
\zkusendinput xyz
další řádek
```

Dřív, než bude ukončeno čtení z aktuálního souboru, budou zpracovány tokeny „abc“ i „xyz“. Teprve „další řádek“ nebude podléhat zpracování.

Kn: 214, 47, **Ol:** 277_{492}, 284, 379.

\endline [plain]

Alternativa k povelu \cr.

```
\let\endline=\cr
```

Kn: 351, **Ol:** 342_{190}.

\endlinechar (iniTEX: 13, tj. ^^M) [integer]

Input procesor na konec každého řádku připojí \endlinechar. Je-li hodnota tohoto registru mimo interval $\langle 0, 255\rangle$, nepřipojí se nic.

Kn: 48, 348, 273, 331, 390–391, **Ol:** *sekce 1.2*, 11, 13_{3}, 14_{21}, 15–16, 18, 23–24, 38, 438_{747}.

\enskip, \enspace [plain]

Mezery velikosti 0,5 em. Jedna je typu ⟨*glue*⟩ a druhá typu \kern.

```
\def\enskip{\hskip.5em\relax}
\def\enspace{\kern.5em }
```

Kn: 71, 352, 202, 419, **Ol:** 119_{36}.

\eqalign [plain]

Sazba soustav rovnic v display módu. Jednotlivé rovnice mají řádky širší o 3 pt. V každé rovnici můžeme volit místo, které budou mít všechny rovnice společně pod sebou. Toto místo označujeme symbolem &. Ukázka použití \eqalign, viz stranu 204.

```
\def\eqalign#1{\null\,\vcenter{\openup\jot\m@th
  \ialign{\strut\hfil$\displaystyle{##}$%
          &$\displaystyle{{}##}$\hfil\crcr#1\crcr}}\,}
```

Kn: 190–191, 193, 242, 326, 362, **Ol:** *204, 206*, 204_{460}, 205–206, 354, 381.

\eqalignno [plain]

Jako \eqalign, ale sazba každé rovnice může obsahovat třetí sloupec pro značku rovnice, kladenou zcela vpravo. Podobně, jako při použití primitivu \eqno. Ukázka použití makra, viz stranu 205.

```
% \@lign a \displ@y, viz řádky 264 až 269 u \displaylines
\def\eqalignno#1{\displ@y \tabskip=\centering
  \halign to\displaywidth{\hfil$\@lign\displaystyle{##}$%
    \tabskip=0pt &$\@lign\displaystyle{{}##}$\hfil
    \tabskip=\centering &\llap{$\@lign##$}\tabskip=0pt \crcr
    #1\crcr}}
```

Kn: 192–193, 194, 362, **Ol:** *205*, 203, 205_{466}, $206_{468–469}$, 207_{473}, 354, 380–381, 385, 410.

\eqno [m]

Konstrukce $$⟨*display math mode material*⟩ \eqno ⟨*math mode material*⟩$$ umožní umístit vpravo od centrované rovnice např. číslo rovnice.

Kn: 293, 189–191, 193, 375–376, 186–187, **Ol:** *197*, $198_{433, 435}$, 199–201, $203_{446–447}$, 204–205, 207_{478}, 208_{486}.

\errhelp [tokens]

Při \errorstopmode se TEX zastaví při chybě. Na terminálu vidíme chybové hlášení a prompt ve tvaru „?“. Odpovíme-li znakem h nebo H, TEX nám poskytne k základnímu chybovému hlášení dodatečnou informaci. Ke všem chybám, které ošetřují vnitřní algoritmy TEXu, je dodatečná informace implementována. Pokud ale byla chyba ošetřena našimi makry (použili jsme primitiv

\errmessage), musíme se o dodatečnou informaci po uživatelově h postarat sami.

Je-li při \errmessage registr \errhelp prázdný a uživatel použije h, TEX utrousí poznámku o detektivu Poirotovi, který pátrá po všech možných stopách. Není-li v takovém případě \errhelp prázdný, objeví se po h obsah tohoto registru na terminálu. Tím můžeme uživateli lépe vysvětlit situaci, ve které došlo k námi ošetřené chybě. Viz též \newhelp.

Kn: 280, 275, 347, **Ol:** 361, 402_{550}.

\errmessage ⟨*filler*⟩{⟨*balanced text*⟩} [h, v, m]
TEX vypíše hlášení ⟨*balanced text*⟩ jako při \message, ovšem chová se, jako by došlo k chybě. To znamená, že při \errorstopmode se zastaví a čeká na reakci operátora. Přitom vypíše stav vstupních proudů podle \errorcontextlines a před chybovou hlášku ⟨*balanced text*⟩ připojí vykřičník. Viz též \errhelp.

Kn: 279–280, 216, 347, 418, **Ol:** 207_{480}, 208_{492}, 244_{181}, 269_{376}, 318, 361, 400_{539}, 402_{550}.

\errorcontextlines (plain: 5) [integer]
Při chybě se objeví na terminálu řádek, který je zrovna přečten input procesorem. Tento řádek je rozdělen na dva v místě, kde došlo k chybě. Toto místo obvykle souvisí s nějakým makrem. Tělo definice tohoto makra se objeví nad vypsaným řádkem a toto tělo je roztrženo na dva řádky v místě chyby. Tam může být další makro. Tělo definice tohoto makra se vypíše nad dosud vypsanými informacemi a je v místě chyby roztrženo na dva řádky. Tyto výpisy pokračují až po tělo definice toho makra, ve kterém došlo k chybě na úrovni hlavního procesoru. Takový řádek je tedy ve výpisu stavu chyby na terminálu jako první hned pod chybovým hlášením.

Při \errorcontextlines=0 se vypíše jen první a poslední řádek stavu chyby. Tj. řádek, ze kterého je patrna chyba na úrovni hlavního procesoru a dále uživatelův řádek, který je načten input procesorem. Tyto dva řádky jsou ve výpisu stavu chyby vždy. Registr \errorcontextlines tedy určuje, kolik *dalších* řádků bude vypsáno na terminál mezi těmito dvěma důležitými řádky. Jedná se tedy o řádky vypisující stav expanze v místě chyby, které dosti často zajímají programátory maker, ale nikoli uživatele maker. Jsou-li nějaké řádky (kvůli malému \errorcontextlines) vynechány, vidíme ve výpisu stavu chyby tři tečky. Při záporném \errorcontextlines se TEX chová, jako by byl tento registr nulový, ale navíc netiskne zmíněné tři tečky.

Kn: 34, 273, 348, **Ol:** 297.

\errorstopmode [a]
Nastaví se způsob zpracování, při kterém se TEX zastaví při každé chybě a nabídne interaktivní řešení. Na terminálu se objeví otazník. Viz též \batchmode, \scrollmode a \nonstopmode.

Uživatel může na prompt tvaru otazníku reagovat vložením některého z následujících písmen nebo kláves (nezáleží na tom, zda je písmeno malé nebo velké).

- `Enter` — Přeskočení chyby. Na další chybě se TEX zastaví znova.
- `X` — Předčasné ukončení činnosti TEXu.
- `E` — Jako `X`, navíc přechod do editoru, je-li v systému implementováno.
- `H` — Dodatečná informace k chybě, viz \errhelp.
- `S` — Zapíná se způsob zpracování \scrollmode.
- `R` — Zapíná se způsob zpracování \nonstopmode.
- `Q` — Zapíná se způsob zpracování \batchmode.
- `I`⟨*další text*⟩ — Vloží do místa chyby ⟨*další text*⟩.
- ⟨*malé kladné číslo n*⟩ — Odmaže n následujících tokenů ze vstupu.

Při startu TEXu bývá obvykle nastaveno \errorstopmode, ovšem toto nastavení je obvykle možné nějakým způsobem změnit. Například v parametrech příkazového řádku volání TEXu nebo prostřednictvím speciální systémové proměnné.

Kn: 32, 33, 277, 299, **Ol:** 339, 359, 406, 429, 444_{783}.

\escapechar (iniTEX: 92, tj. ‘\\) [integer]
ASCII kód prvního znaku, který se použije při konverzi identifikátoru řídicí sekvence do řady tokenů nebo znaků. Takovou konverzi provádějí primitivy \string, \write, \message, \errmessage. Není-li \escapechar v intervalu $\langle 0, 255\rangle$, netiskne se před identifikátorem řídicí sekvence nic. Tento registr nemusí mít nic společného se znakem, jehož kategorie je nastavena na nulu.

Kn: 228, 348, 40, 213, 273, 308, 377, **Ol:** 25_{41}, $402_{555,\,560}$, 437, $438_{743,\,745}$.

\everycr [tokens]
Seznam tokenů, které expand procesor vloží do vstupní fronty vždy po povelu \cr nebo po takovém povelu \crcr, který skutečně ukončuje vstupní řádek tabulky. Implicitně je registr prázdný.

Kn: 275, 362, **Ol:** $354_{265,\,269}$, 375_{390}.

\everydisplay [tokens]
Seznam tokenů, které expand procesor vloží do vstupní fronty vždy po otevíracím `$$` při zahájení display módu. Implicitně je registr prázdný.

Kn: 287, 179, 275, 326, **Ol:** 167, 199, 202, 203_{443}.

\everyhbox [tokens]
Seznam tokenů, které expand procesor vloží do vstupní fronty vždy po otevírací `{` při startu sestavení horizontálního seznamu v \hbox. Implicitně je registr prázdný.

Kn: 275, 279, **Ol:** 338, 370.

\everyjob [tokens]
Seznam tokenů, které expand procesor vloží do vstupní fronty vždy při spuštění TEXu. Implicitně je registr prázdný. V iniTEXu jej většinou naplníme makry, která tisknou pomocí \message nebo \immediate\write16 informaci o použitém formátu na terminál a do souboru log.

Pokud ⟨*zbytek příkazového řádku*⟩ (viz stranu 284) obsahuje jen název vstupního souboru, pak se \everyjob vykoná až po otevření vstupního souboru. V této situaci je už známo jméno souboru log, takže výpisy z \everyjob se objeví na terminálu i v souboru log. Nezačíná-li ⟨*zbytek příkazového řádku*⟩ názvem vstupního souboru, vykoná se \everyjob okamžitě. Zprávy z \everyjob jsou pak zobrazeny jen na terminálu a nejsou uloženy do souboru log.

Kn: 275, **Ol:** není nikdy použito.

\everymath [tokens]
Seznam tokenů, které expand procesor vloží do vstupní fronty vždy po otevíracím $ při startu vnitřního matematického módu.

Kn: 287, 293, 179, 275, 326, **Ol:** 71, 167, 200, 445.

\everypar [tokens]
Seznam tokenů, které expand procesor vloží do vstupní fronty vždy při přechodu do odstavcového módu.

Kn: 282, 283, 215, 253, 275, 105, 262, 333, 381, 407, 421, **Ol:** *89*, $14_{19,\,21}$, 28, 29_{85}, 53, 73, 90_{149}, 122, $123_{103,\,109–111}$, 124, 417.

\everyvbox [tokens]
Seznam tokenů, které expand procesor vloží do vstupní fronty vždy po otevírací { při startu sestavení vertikálního seznamu ve \vboxu.

Kn: 275, 279, **Ol:** 338, 453.

\exhyphenpenalty (plain: 50) [integer]
Penalta za zlom řádku v místě \discretionary v případě, že ⟨*pre-break*⟩ text je prázdný. Tj. nikoli pro \-. Viz též \hyphenpenalty.

Kn: 96, 272, 262, 348, **Ol:** 216, 226_{101}, 229–230, 276_{486}, 374, 435_{717}.

\expandafter ⟨*token1*⟩⟨*token2*⟩ [exp]
Expand procesor nejprve provede expanzi ⟨*token2*⟩ a pak se vrátí k druhému průchodu, ve kterém zpracuje:

302 ⟨*token1*⟩⟨*výsledek expanze token2*⟩

Je-li ⟨*token2*⟩ neexpandovatelný (tj. nejedná se o aktivní znak a jde-li o řídicí sekvenci, pak pouze ve významu neexpandovatelného primitivu), \expandafter neprovede nic. Totéž platí pro neexpandovatelný ⟨*token1*⟩, protože pak v druhém průchodu expand procesor nemění ⟨*token1*⟩ a začíná

zpracovávat ⟨*výsledek expanze token2*⟩, přičemž tuto činnost by dělal i bez použití `\expandafter`.

⟨*výsledek expanze token2*⟩ je posloupnost tokenů z expanze ⟨*token2*⟩ vzniklá jen expanzí „první hladiny“. Vysvětlíme na příkladě:

```
\def\a{aa} \def\b{bb\a bb} \def\c#1:{{\def\a{CC}#1}}
\expandafter \c \b:
```

Zde do druhého průchodu vstupuje `\c bb\a bb:` a nikoli `\c bbaabb:` Parametr makra `\c` tedy bude obsahovat řídicí sekvenci `\a`, která je v rámci makra `\c` předefinována, takže na výstupu dostaneme bbCCbb.

Kn: 213, 215, 40, 260, 308, 330, 348, 374–380, **Ol:** *celá sekce 2.2*, 37–38, 42–46, 51, 53, 55–56, 59, 62, 65, 72, 80, 134, 160–161, 174–176, 180, 207–208, 217–218, 278, 287, 291–292, 350, 368, 379, 395, 402, 405, 438.

`\extrahyphenchar`, `\extrahyphens` [csplain]

Konstanta `\extrahyphenchar` označuje pozici alternativního znaku divis v $\mathcal{CS}$-fontech. Makro `\extrahyphens` přepíná na tento alternativní divis. Implicitně není alternativní divis použit, aby zůstala kompatibilita s plainem. Viz soubor `extcode.tex`, resp. `il2code.tex`. Viz též stranu 216.

```
\chardef\extrahyphenchar=156
\def\extrahyphens{%
  \hyphenchar\tenrm=\extrahyphenchar
  \hyphenchar\tenbf=\extrahyphenchar
  \hyphenchar\tentt=\extrahyphenchar
  \hyphenchar\tensl=\extrahyphenchar
  \hyphenchar\tenit=\extrahyphenchar
  \defaulthyphenchar=\extrahyphenchar}
```

`\fam` [integer]

Je-li `\fam` $\in \langle 0, 15\rangle$, pak matematické znaky, které mají v `\mathcode` třídu 7, budou sázeny rodinou fontů číslo `\fam`. Je-li `\fam` mimo tento interval, budou zmíněné matematické znaky sázeny podle implicitní rodiny fontů, která je uvedena v `\mathcode`. Při zahájení každého matematického módu nastavuje TEX `\fam=-1`. Vidíme tedy, že beze změny tohoto registru uvnitř matematického módu budou všechny znaky sázeny podle implicitní rodiny fontů.

Kn: 289–290, 154–159, 273, 346–347, 351, 358, 414–415, **Ol:** 130, 147_{12}, 148, 171–172, $174_{112,\,119}$, $175_{132–133}$, 179_{201}, 195_{416}, 336, 340_{160}, 345_{214}, 380_{412}, 397_{499}, 400_{531}, 409_{600}, 427_{683}, 434_{715}, 447_{786}.

`\fi` [exp]

Součást konstrukce `\if`⟨*xy*⟩ `... \else ... \fi`

Kn: 213, 207, 210, **Ol:** *sekce 2.3* a dále všude tam, kde je použit nějaký primitiv typu `\if...`.

\filbreak [plain]

Pokud v místě \filbreak bude proveden stránkový zlom, vyplní se zbylé místo na straně vertikální mezerou \vfil. Ovšem pokud se pod \filbreak vejde na stranu nějaký další text ukončený třeba novým \filbreak, bude první \filbreak ignorován.

```
313 \def\filbreak{\par\vfil\penalty-200\vfilneg}
```

Rozbor makra: Pokud se provede zlom v \penalty-200, pak na konci strany pracuje \vfil a \vfilneg se na další straně neobjeví, protože je to odstranitelný element. Pokud se zlom v místě \penalty-200 neprovede, pak \vfilneg kompenzuje původní \vfil, takže v daném místě není žádná pružná mezera.

Kn: 111, 353, **Ol:** není nikdy použito.

\finalhyphendemerits (plain: 5 000) [integer]

„Demerits“ (kvadrát penalty) za rozdělení slova na konci předposledního řádku odstavce.

Kn: 98, 273, 106, 348, 451, **Ol:** 212–213, 228–230, 276_{487}.

\firstmark [exp]

První značka \mark v sestavené straně. Strana je sestavena do boxu 255 v algoritmu uzavření strany.

Kn: 258, 213, 259–260, 280, **Ol:** *258*, 256, 259_{277}, 260, 391, 437.

\fiverm, \fivei, \fivesy, \fivebf [plain]

Fonty v pětibodové velikosti jsou v plainu použity pro indexy druhé úrovně. Viz též \tenrm, \sevenrm a \scriptscriptfont.

```
314 \font\fiverm=cmr5
315 \font\fivei=cmmi5
316 \font\fivesy=cmsy5
317 \font\fivebf=cmbx5
```

Kn: 153, 350–351, 414–415, **Ol:** $168_{90,\,94}$, 169, 174_{113}, 427_{682}, 430, 440.

\floatingpenalty [integer]

Penalta přidaná za roztržení insertu na více stránek. Hodnotu tohoto registru TEX ukládá do své paměti při ukončení sestavování vertikálního materiálu povelem \insert. Například:

```
318 \insert{⟨vertical material⟩ \floatingpenalty=⟨number⟩}
```

V jiném okamžiku je tato hodnota použita. Viz pravidlo 5 na straně 250. Podobnou vlastnost mají registry \splittopskip a \splitmaxdepth.

Kn: 123–125, 272, 281, 363, **Ol:** *250*, 251_{214}, 255_{261}, 437.

\flqq [csplain]

vysází levé francouzské uvozovky (») podle kódu $\mathcal{CS}$-fontů. Tyto uvozovky

se v němčině a v češtině používají vpravo. Viz soubor `extcode.tex`, resp. `il2code.tex`. V $\mathcal{CS}$-fontech je též tento znak dosažitelný jako ligatura použitím dvojice `<<`.

```
\chardef\flqq=158  \sfcode158=0
```

`\fmtname`, `\fmtversion` [plain]

Název a verze formátu. To je dobrý zvyk použitý i v jiných formátech.

```
\def\fmtname{plain} \def\fmtversion{3.14159}
```

Kn: 364, **Ol:** není nikdy použito.

`\folio` [plain]

Vytiskne stránkovou číslici (absolutní hodnotu registru `\pageno`) arabskými číslicemi nebo jako římské číslo. To záleží na znaménku registru.

```
\def\folio{\ifnum\pageno<0 \romannumeral-\pageno
           \else\number\pageno \fi}
```

Kn: 252–253, 362, 406, 416, **Ol:** 262$_{293,\,296,\,310}$, 263$_{312}$, 366$_{341}$.

`\font` ⟨*control sequence*⟩⟨*equals*⟩⟨*file name*⟩⟨*at clause*⟩ *global* [a]

Je-li primitiv použit ve významu povelu hlavního procesoru, pak zavede do paměti font ze souboru ⟨*filename*⟩ ve zvětšení ⟨*at clause*⟩. Parametr ⟨*control sequence*⟩ se stává přepínačem zavedeného fontu a spadá do syntaktického pravidla ⟨*font*⟩. Je-li font v daném zvětšení už dříve zaveden, není soubor čten znova, ale pouze se přiřadí význam přepínače typu ⟨*font*⟩ pro ⟨*control sequence*⟩.

Primitiv `\font` může být použit též jako parametr podle syntaktického pravidla ⟨*font*⟩. V takovém případě má význam aktuálního textového fontu. Například:

```
\fontname\font        % Název aktuálního fontu (csr10).
\char\hyphenchar\font % Tisk \hyphenchar aktuálního fontu.
\the\fontdimen6\font  % Tisk \fontdimen6 aktuálního fontu.
\textfont2=\font      % \textfont2 nastavíme na akt. font.
\fontname\textfont2   % Překontrolujeme, zda se to povedlo.
\edef\currfont{\the\font} % Zapamatujeme si aktuální font.
\bf bla bla           % Přepneme do jiného fontu.
\currfont             % Vracíme se k původnímu fontu.
```

Kn: 213, 276, 16–17, 60, 210, 214–215, 271, **Ol:** 65–69, 71, 81–82, 90, 104–105, 162, 167–170, 172, 174, 177, 216, 218–219, 230, 276, 284, 317–318, 322, 339, 350, 364, 366, 374, 391, 407, 420–421, 426, 430, 433, 439–441, 447.

`\fontdimen` ⟨*number*⟩⟨*font*⟩ *global, restricted* [dimen]

Každý font má aspoň 7 parametrů typu ⟨*dimen*⟩ (viz stranu 104), které se načítají z metriky nebo jsou upraveny makrem. Uvedený zápis představuje parametr číslo ⟨*number*⟩ fontu ⟨*font*⟩.

Je-li v metrice méně než 7 parametrů, jsou chybějící hodnoty doplněny nulami. Matematické fonty mohou mít více `\fontdimen` než 7 (viz stranu 181).

Je-li ⟨*number*⟩ mimo rozsah parametrů načtených pro ⟨*font*⟩, TEX ohlásí chybu. Existuje ale výjimka: naposledy načtený font primitivem `\font` může být „upraven“ uložením hodnot do nových `\fontdimen`, které jsou mimo načtený rozsah. TEX v tomto případě alokuje další čísla `\fontdimen`, která fyzicky v metrice vůbec nebyla. Například:

```
\font\a=csr10 at 12pt      % font má jen 7 \fontdimen
\font\b=csr10 at 14pt      % font má zatím 7 \fontdimen
\fontdimen8\b=1pt
\fontdimen9\b=3pt          % nyní má font \b 9 \fontdimen
\the\fontdimen9\b          % vytiskne 3pt
\fontdimen8\a=1pt          % TeX ohlásí chybu
```

Kn: 271, 277, 433, 76, 157, 214, 441, 447, 179, 355–356, 375, 390, **Ol:** *104, 181*, 73, 74$_{37,\,39}$, 102, 105$_{184}$, 106$_{185}$, 157, 161–162, 168–169, 172–173, 180$_{207–209}$, 181$_{213–214}$, 203$_{450}$, 303–304, 307, 320, 326$_{58}$, 336, 339$_{151}$, 350$_{252}$, 365$_{325}$, 366, 407, 412, 436, 448.

`\fontname` ⟨*font*⟩ [exp]

Vrátí jméno souboru bez přípony, které odpovídá fontu ⟨*font*⟩. Je-li font zaveden v jiném poměru než 1:1, je připojeno ␣$_{10}$ `at`⟨*dimen*⟩ (v jednotkách pt).

Kn: 213, 214, **Ol:** 82$_{95}$, 365$_{323,\,327}$, 407, 442.

`\footins` [plain]

Třída insertů pro poznámky pod čarou.

```
\newinsert\footins
\skip\footins=\bigskipamount % space added
\count\footins=1000 % footnote magnification factor (1 to 1)
\dimen\footins=8in % maximum footnotes per page
```

Kn: 256,363,396–399,416, **Ol:** 251$_{202–205,\,209}$, 252$_{225,\,229–230,\,232}$, 264, 390$_{456,\,459}$.

`\footline` [plain]

Registr typu ⟨*tokens*⟩, jehož obsah se tiskne do paty strany při standardní výstupní rutině plainu. Registr je použit v kódu výstupní rutiny, viz stranu 262. Například makro `\nopagenumbers` mění výchozí obsah tohoto registru.

```
\newtoks\footline \footline={\hss\tenrm\folio\hss}
```

Kn: 252, 256, 340–341, 362, **Ol:** 243$_{148}$, 262$_{293,\,307,\,310}$, 263$_{312}$, 267$_{334}$, 407$_{575}$.

`\footnote` ⟨*značka*⟩`{`⟨*text poznámky*⟩`}` [plain]

Poznámka pod čarou. Kód makra, viz stranu 251.

Kn: 82, 116, 251, 256, 340, 363, 382, 416, **Ol:** 173$_{111}$, 246, 250, 251$_{206}$, 252$_{240}$, 292$_{60–61}$, 454.

\frenchspacing [plain]

Nastaví mezerování za tečkou, otazníkem, vykřičníkem, dvojtečkou, středníkem a čárkou tak, že tyto mezery nejsou nijak deformovány a mají stejnou velikost jako mezery mezi slovy. Viz též \nonfrenchspacing.

```
342 \def\frenchspacing{\sfcode`\.=1000 \sfcode`\?=1000
343    \sfcode`\!=1000 \sfcode`\:=1000 \sfcode`\;=1000
344    \sfcode`\,=1000 }
```

Kn: 74, 340, 351, 381, 410, **Ol:** 104–105, $347_{221,224}$, 406, $433_{701,704}$.

\frqq [csplain]

Vysází pravé francouzské uvozovky (») podle kódu $\mathcal{CS}$-fontů. Tyto uvozovky se v němčině a v češtině používají vlevo. Viz soubor `extcode.tex`, resp. `il2code.tex`. V $\mathcal{CS}$-fontech je navíc definovaná dvojice `>>` vedoucí na tento znak jako ligatura.

```
345 \chardef\frqq=159  \sfcode159=0
```

\futurelet ⟨*control sequence*⟩⟨*token1*⟩⟨*token2*⟩ [a]

Do ⟨*control sequence*⟩ se přiřadí ⟨*token2*⟩, pak se expanduje ⟨*token1*⟩, přičemž ⟨*token2*⟩ zůstává ve čtecí frontě na svém místě a čeká na expanzi případně jiné zpracování. Přesněji, v prvním průchodu se provede:

```
346 \let ⟨control sequence⟩=␣⟨token2⟩
```

a do druhého průchodu vstupuje k další případné expanzi:

```
347 ⟨token1⟩⟨token2⟩
```

Vidíme, že na rozdíl od přímého \let kombinovaného s \afterassignment zůstává ⟨*token2*⟩ v nezměněné podobě na svém místě ve čtecí frontě. Klasickým příkladem použití \futurelet je případ, kdy chceme v makru zjistit, co následuje a podle toho větvit chování makra. V LaTeXu například dává mnoho maker uživateli možnost použít tzv. nepovinný parametr makra uzavřený v hranatých závorkách (`[...]`), zatímco povinný parametr se uzavírá do množinových závorek (`{...}`). Makro musí zjistit, zda uživatel napsal či nenapsal nepovinný parametr a podle toho se větvit:

```
348 \def\macro{\futurelet \nextchar \doaction}
349 \def\doaction{\ifx [\nextchar  \let\next=\macrowithopt
350                \else           \let\next=\macrowithoutopt
351                \fi \next}
352 \def\macrowithopt[#1]#2{...}
353 \def\macrowithoutopt #1{...}
```

V této ukázce jsme definovali speciální makra pro případ, kdy uživatel napsal či nenapsal nepovinný parametr. Přitom hlavní \macro se expanduje buď na \macrowithopt nebo na \macrowithoutopt podle toho, zda je či není použita hranatá závorka.

V LaTeXu je definováno pracovní makro `\@ifnextchar`, které se zapisuje ve tvaru:

```
\@ifnextchar X{co vykonat, pokud ano}{co vykonat, pokud ne}
```

Makro testuje, zda následuje `X`, a podle toho se expanduje na druhý nebo třetí parametr. Definice makra vypadá takto:

```
\def\@ifnextchar#1#2#3{\let\@tempe #1\def\@tempa{#2}%
    \def\@tempb{#3}\futurelet \@tempc\@ifnch}
\def\@ifnch{\ifx \@tempc \@sptoken \let\@tempd\@xifnch
            \else \ifx \@tempc\@tempe \let\@tempd\@tempa
                  \else \let\@tempd\@tempb
            \fi   \fi \@tempd}
\def\:{\let\@sptoken= } \:  % makes \@sptoken a space token
\def\:{\@xifnch}
\expandafter\def\: {\futurelet\@tempc\@ifnch}
```

V tomto makru není nic nového, kromě poměrně krkolomně řešeného přeskočení případné mezery. Pokud je následující znak mezera, pak se provede opakované `\futurelet`, ovšem mezeru je potřeba přeskočit. Na řádku 363 se vlastně děje toto:

```
def  @xifnch  ␣10  {\futurelet\@tempc\@ifnch}
```

Osobně bych problém řešil přehledněji takto:

```
\def\@ifnextchar#1#2#3{\let\@tempe #1\def\@tempa:{#2}%
    \def\@tempb:{#3}\futurelet \@tempc\@ifnch}
\def\@ifnch{\ifx \@tempc \@sptoken \let\@tempd\@xifnch
            \else \ifx \@tempc\@tempe \let\@tempd\@tempa
                  \else \let\@tempd\@tempb
            \fi   \fi \@tempd:}
\futurelet\@sptoken{ }  % toto taky vkládá space token
\def\@xifnch: {\futurelet\@tempc\@ifnch}
```

LaTeX používá `\@ifnextchar` poměrně často. Například při definování makra `\newcommand`:

```
\def\newcommand#1{\@ifnextchar [{\@argdef#1}{\@argdef#1[0]}}
\long\def\@argdef#1[#2]#3%
                    {\@ifdefinable #1{\@reargdef#1[#2]{#3}}}
```

Jak je dále definováno `\@reargdef`, zde už nebudeme zmiňovat. V LaTeXovém zdrojovém textu je použito mnoho dalších špinavých triků, které si může čtenář sám nastudovat pohledem do příslušných souborů LaTeXu.

Kn: 207, 215, 277, 262, 363, 375–377, 423, **Ol:** 66, 71–72, 126_{128}, 127, $217_{40,\,44}$, 218, 251_{217}, 322, 327, 333_{110}, 338.

`\gdef` ⟨*control sequence*⟩⟨*parameter text*⟩{⟨*balanced text*⟩} *global* [a]
Ekvivalent `\global\def`.

Kn: 206, 215, 275, 352, 407, **Ol:** 26$_{66}$, 27, 29$_{80,\,82}$, 31, 40, 41$_{80–81,\,86}$, 42, 46$_{134}$, 65, 66$_{7}$, 71, 80$_{73}$, 81, 122$_{58}$, 277$_{501–502}$, 333$_{109}$, 351, 402$_{562}$, 403, 408$_{588}$, 411.

`\global` [pre]
Následující definice či jiné přiřazení je globální, tj. přesahuje hranice aktuální skupiny.

Kn: 275, 119, 206, 232, 256, 301, 307, 346, 21, 218, **Ol:** 29, 31, 40, 73, 83–84, 107, 126–128, 143, 180, 203, 207–208, 226, 257, 259–260, 262, 265, 267–274, 276, 292, 334, 337, 342, 350, 354, 382, 400–401, 408–409, 435, 451.

`\globaldefs` (iniTEX: 0) [integer]
Je-li tento parametr kladný, pak všechna přiřazení jsou automaticky globální. Je-li záporný, pak všechna přiřazení jsou automaticky lokální bez závislosti na existenci slova `\global`. Je-li nulový, pak přiřazení je lokální právě tehdy, když není použito slovo `\global`.

Kn: 275, 238, 273, **Ol:** 40, 73, 269$_{379}$, 270.

`\goodbreak` [plain]
Makro naznačuje určitou vhodnost stránkového zlomu v místě použití.

```
376 \def\goodbreak{\par\penalty-500 }
```

Kn: 111, 116, 353, **Ol:** není nikdy použito.

`\H` ⟨*znak*⟩ [plain]
Maďarská přehláska, která je výraznější, než `\"`. Například `\H a` vede na ő.

```
377 \def\H#1{{\accent"7D #1}}
```

Kn: 52–53, 356, 420, **Ol:** není nikdy použito.

`\halign` ⟨*box specification*⟩{⟨*alignment material*⟩} [v, m]
Sestavení řádků tak, aby jednotlivé části řádků lícovaly do sloupců tabulky. Přitom ⟨*alignment material*⟩ obsahuje deklaraci řádku a vlastní data tabulky.

Kn: 282, 291, 117, 190, 193, 194, 235–249, 286, 302, 352, 326, 361–362, 386, 392, **Ol:** *sekce 4.3*, 72, 90, 97, 120, 126, 128–143, 181, 184, 193–194, 197, 204–205, 238, 268, 317, 332, 343, 345, 354, 359, 372, 374–375, 385, 394, 398, 408, 410–411, 435–436, 439, 452.

`\hang` [plain]
Použití `\hang` způsobí, že první řádek odstavce bude mít „normální délku“ a všechny ostatní budou odsazeny o `\parindent` doprava. Viz `\hangindent` a `\hangafter`.

```
378 \def\hang{\hangindent=\parindent}
```

Kn: 355, 419, **Ol:** 380$_{414}$.

\hangafter [integer]
Specifikace těch řádků v odstavci, kterých se týká \hangindent. Kladná hodnota: odsazení se netýká prvních \hangafter řádků; záporná hodnota: odsazení se týká prvních -\hangafter řádků. Po sestavení každého odstavce TEX vrací registr \hangafter na výchozí hodnotu 1.

Kn: 102, 103–104, 273, 348–349, 419, **Ol:** *235*, 227, 229, 236, 416.

\hangindent [dimen]
Velikost odsazení těch řádků v odstavci, které jsou specifikované hodnotou \hangafter. Kladná hodnota: odsazení zleva; záporná hodnota: odsazení zprava. Po sestavení každého odstavce TEX vrací registr \hangindent na hodnotu 0 pt, takže další odstavec bude mít odsazeny řádky jen po výslovné změně registru.

Kn: 102, 274, 349, 262, 407, **Ol:** *235*, 198–199, 227, 229, 236, 353, 355, 369_{378}, 373, 380_{415}, 385, 416, 422, 426.

\hbadness (plain: 1 000) [integer]
Maximální hodnota badness, při níž TEX ještě nepíše varování Underfull \hbox.

Je-li \hbadness > 10 000, nebude TEX oznamovat žádné Underfull \hbox. Na druhé straně hlášení Overfull \hbox nelze potlačit.

Je-li \hbadness < 100, bude TEX oznamovat nejen výskyt podtečených boxů typu „Underfull“, ale též „Tight \hbox“ (stažení mezer v boxu má badness více než \hbadness) a „Loose \hbox“ (roztažení mezer má badness větší než \hbadness, ale menší než 100).

Kn: 302, 29, 272, 401, 348, 387–388, **Ol:** 98–99, 101, 230–231, 233, 276_{485}, 432_{699}, 452.

\hbox ⟨*box specification*⟩{⟨*horizontal material*⟩} [h, v, m]
Sestaví se box s horizontálním seznamem.

Podrobněji: údaj ⟨*box specification*⟩ si TEX uloží do zásobníku a použije jej až při závěrečné kompletaci boxu. Pak v místě závorky „{“ otevře novou skupinu. Hlavní procesor přejde do vnitřního horizontálního módu v němž začne sestavovat horizontální seznam. Nejprve se expanduje \everyhbox. Pak TEX čte povely z ⟨*horizontal material*⟩. Po dosažení koncové závorky „}“, která uzavírá skupinu otevřenou na začátku sestavování boxu, provede TEX kompletaci boxu. Tato kompletace závisí na ⟨*box specification*⟩ a podrobněji o ní viz sekci 3.5. Hlavní procesor se pak vrací do módu, ve kterém byl před zahájením zpracování boxu. V rámci tohoto módu se kompletovaný box klade do příslušného seznamu jako jeden element seznamu. Ve vertikálním módu se před box automaticky připojí meziřádková mezera. Pokud ale předchází \setbox, kompletovaný box se neklade do tiskového seznamu, ale ukládá se do specifikovaného registru.

Kn: 64–67, 77, 86, 93, 151, 159, 163, 175, 179, 185–186, 221, 111, 278, 282, 388–389, **Ol:** *82*, *sekce 3.5*. Průběžně se vyskytuje na mnoha stránkách knihy.

`\headline` [plain]
Registr typu ⟨*tokens*⟩, jehož obsah se tiskne do záhlaví strany při standardní výstupní rutině plainu. Registr je použit v kódu výstupní rutiny, viz stranu 262.

```
379 \newtoks\headline \headline={\hfil} % head is normally blank
```

Kn: 252–253, 255, 362, 406, **Ol:** 259_{278}, $260_{286,\,288}$, $262_{292,\,306}$.

`\hfil` [h, m]
Primitivní ekvivalent k povelu `\hskip 0pt plus 1fil`, tj. vloží do horizontálního seznamu mezeru s hodnotou roztažení prvního řádu. Zatímco při použití `\hskip 0pt plus 1fil` musíme psát za parametrem „bezpečnostní“ `\relax`, abychom ukončili neúplný parametr podle pravidla ⟨*glue*⟩, při použití primitivu `\hfil` nic takového není potřeba.

Kn: 71–72, 194, 235–237, 283, 285, 290, 397, **Ol:** 87, 89, 91, 104, 116–119, 129, 131–135, 139, 141–142, 145, 157, 181, 184, 189, 197, 211–212, 252, 257, 259–260, 262–263, 265, 268, 276, 281, 324, 342, 346, 348, 354, 359, 371, 384–385, 394, 407, 452.

`\hfill` [h]
Primitivní ekvivalent k `\hskip 0pt plus 1fill`, tj. vloží do horizontálního seznamu mezeru s hodnotou roztažení druhého řádu.

Kn: 71–72, 142, 177, 194, 233, 283, 285, 290, **Ol:** 89, 91, 104, 112, 121, 123, 135, 184, 188, 212, 262–263, 265–266, 277, 324, 355, 373, 383–384.

`\hfilneg` [h]
Primitivní ekvivalent k `\hskip 0pt plus -1fil`, tj. stornuje případný `\hfil`.

Kn: 72, 100, 233, 283, 285, 290, 397, **Ol:** 89, 104, 324, 384.

`\hfuzz` (plain: 0,1 pt) [dimen]
Velikost přesazení přes stanovenou šířku `\hbox`u, která se toleruje místo aby se ohlásilo `Overfull \hbox`.

Kn: 302, 30, 274, 348, 387–388, **Ol:** 100–101, 230.

`\hglue` ⟨*glue*⟩ [plain]
Makro vytvoří mezeru v horizontálním seznamu, která nezmizí v řádkovém zlomu. Navíc zachová hodnotu `\spacefactor`. Viz též `\vglue`.

```
380 \def\hglue{\afterassignment\hgl@\skip0=}
381 \def\hgl@{\leavevmode \count255=\spacefactor \vrule width0pt
382     \nobreak\hskip\skip0 \spacefactor=\count255 }
```

Kn: 352, **Ol:** 454.

\hidewidth, \hideskip [plain]

Makro \hidewidth lze použít v tabulkách. Jde o to, aby přirozená šířka boxu byla záporná a neovlivnila vnitřní výpočet šířky sloupce v \halign. Hodnota roztažení mezery přitom kompenzuje „handicap“, že má mezera zápornou přirozenou šířku. Makro je potřeba použít z obou stran položky, aby byla položka centrována.

```
\newskip\hideskip
\hideskip=-1000pt plus 1fill % negative but can grow
\def\hidewidth{\hskip\hideskip}
```

Kn: 243, 245, 247, 325, 347–348, 354, **Ol:** 141, 142_{264}, 143_{269}, $339_{154–155}$, 345_{211}, 350_{255}.

\hoffset (iniTEX: 0 pt) [dimen]

Určuje horizontální usazení sazby při výstupu do dvi souboru. Podrobněji, viz heslo \shipout. Viz též \voffset.

Kn: 251, 274, 342, **Ol:** 281_{577}, 431, 454.

\holdinginserts (iniTEX: 0) [integer]

Pokud je parametr kladný, pak nebudou inserty v algoritmu uzavření strany umisťovány do svých boxů. Výstupní rutina je tedy nemá dostupné ve smluvených boxech. Na druhé straně ale odkazy na inserty zůstávají při kladném \holdinginsert v materiálu boxu 255. Pokud je tedy tento materiál zpětně vrácen do vertikálního seznamu, inserty se budou plnit do míst typu b_n znova.

Je-li parametr \holdinginserts nulový (implicitní nastavení), pak jsou obsahy insertů předány výstupní rutině v odpovídajících boxech, ale odkazy těchto insertů jsou před vyvoláním výstupní rutiny z boxu 255 zrušeny.

Kn: 125, 273, 400, **Ol:** 250, 269_{378}, 271, 283.

\hphantom {⟨*sazba*⟩} [plain]

Vytvoří prázdný box šířky materiálu zapsaného v parametru ⟨*sazba*⟩. Výška s hloubkou boxu je nulová. Je-li ⟨*sazba*⟩ zapsaná jako jeden token, není nutno uvádět závorky kolem. Například \hphantom A vytvoří prázdný box šířky písmene A. Makro pracuje v horizontálním i matematickém módu. Kód makra, viz stranu 418.

Kn: 178, 211, 360, **Ol:** 191, 193, 418_{623}, 435.

\hrule ⟨*rule specification*⟩ [v]

Obdélník, který má implicitní šířku závislou na výsledné šířce \vboxu v němž je použit. Implicitní výška je 0.4pt a hloubka 0pt. Všechny tři údaje se dají explicitně formulovat pomocí řídicích slov width, height a depth v ⟨*rule specification*⟩.

Kn: 281–282, 64, 85, 221–225, 286, 24, 246, 357, 420, 421, **Ol:** *327*, 87, 90–91, 95, 101, 111, 113, 116–117, 121, 123, 129, 137–139, 141, 143, 156, 183, 188, 252, 257, 259, 270, 276, 278, 281, 317, 319, 328, 334, 373, 383, 421, 454.

`\hrulefill` [plain]
Makro vyplní horizontální mezeru linkou tloušťky 0,4 pt položenou na účaří. Mezera má pružnost `\hfill`.

```
386 \def\hrulefill{\leaders\hrule\hfill}
```

Kn: 244, 252, 357, 412, **Ol:** 111_{221}, $112_{224–225}$, 120, $139_{226–227, 236}$, 140, $141_{246–247, 256}$.

`\hsize` (plain: 6,5in, csplain: 159,2mm) [dimen]
Šířka jednotlivých `\hbox`ů, které se sestavují při algoritmu řádkového zlomu. Tj. šířka řádků právě sestavovaného odstavce. Výjimka — viz `\hangindent` a `\parshape`.

V implicitním nastavení (plain, csplain) se počítá s okraji širokými po obou stranách 1 in. Nastavení z plainu je vhodné pro papíry formátu letter (šířka 8,5 in, americký formát papíru), zatímco nastavení csplainu je vhodné pro papíry formátu A4 (šířka 159,2 mm + 2 in).

Kn: 102, 188, 237, 251, 274, 413, 415, 26–27, 60, 257, 340–341, 348, 387, 406, 407, 417, **Ol:** 73, 91, 97–98, 118, 123, 125–127, 132, 140–141, 143, 198–200, 227, 229–230, 234, 236, 244–245, 265, 269–273, 276–278, 280–282, 320, 323, 327, 355, 357, 385–386, 390, 395, 426, 432, 452.

`\hskip` ⟨*glue*⟩ [h]
Vloží horizontální mezeru typu ⟨*glue*⟩ požadované hodnoty. Je-li povel použit ve vertikálním módu, nejprve přejde implicitně do odstavcového módu.

Kn: 285, 71, 86, 168, 283, 290, 314, **Ol:** 30, 70, 77–78, 84, 87, 89, 97, 104–105, 116, 134, 145, 157–158, 202–203, 209, 212, 226–227, 274, 279, 324, 359, 371–372, 384, 416, 423.

`\hss` [h]
Primitivní ekvivalent k `\hskip 0pt plus 1fil minus 1fil`, tj. vloží mezeru s hodnotou stažení i roztažení prvního řádu. Protože má mezera hodnotu stažení prvního řádu, nelze ji použít v odstavcovém módu. Algoritmus řádkového zlomu by totiž nebyl nucen hledat za touto mezerou žádný další řádkový zlom.

Kn: 71, 233, 283, 285, 290, 442, 82–83, **Ol:** 82, 87, 89, 104, 119, 121, 123, 126, 128, 145, 156, 203, 245, 262, 324, 346, 366, 381, 384–385, 387, 395, 409, 426–427.

`\ht` ⟨*8-bit number*⟩ *restricted* [dimen]
Umožňuje přístup k hodnotě „výška boxu“, která odpovídá boxu z registru ⟨*8-bit number*⟩.

Kn: 388–389, 120, 271, 417, **Ol:** 73, 83, 95, 115, 184, 244, 251, 255, 257, 265, 267–269, 274, 282, 335, 342, 345, 409, 418, 427, 435, 452.

\hyphenation ⟨*filler*⟩{⟨*balanced text*⟩} [a]

Uživatelská definice výjimek z pravidel dělení. Slo-va se pí-šou tak-to a od-dě-lu-jí se me-ze-ra-mi.

Kn: 452–453, 277, 455, 419, **Ol:** *221*, 221$_{73}$, 222, 223$_{75,\,77}$, 227, 230, 318, 336, 374, 382.

\hyphenchar ⟨*font*⟩ *restricted* [integer]

Povel umožní přístup k registru „hyphenchar“ daného fontu ⟨*font*⟩. Tato hodnota určuje kód znaku, který má dva významy: (1) Znak uvnitř slova způsobí, že se za něj automaticky vloží `\discretionary{}{}{}`. (2) Při dělení slov pomocí `\-` (nebo při automatickém dělení slov podle `\patterns` a `\hyphenation`) se znak „hyphenchar“ použije na konci slova jako divis.

Je-li `\hyphenchar`⟨*font*⟩ mimo interval $\langle 0, 255\rangle$, je potlačeno dělení slov ve fontu ⟨*font*⟩.

Při zavedení nového fontu pomocí `\font` je nastaveno `\hyphenchar` tohoto fontu na hodnotu podle `\defaulthyphenchar`. Později je možno `\hyphenchar` fontu změnit.

Kn: 271, 277, 95, 214, 273, 286, 351, 395, 454, 455, 414, **Ol:** *216*, 65, 216$_{30–31}$, 218$_{49–50}$, 219$_{53}$, 220$_{54}$, 221$_{63}$, 226, 334, 351, 363$_{307–311}$, 365$_{324}$, 445.

\hyphenpenalty (plain: 50) [integer]

Penalta za rozdělení slova při neprázdném ⟨*pre-break*⟩ (viz `\discretionary`). Jedná se tedy o penaltu za běžné rozdělení slova, protože v ⟨*pre-break*⟩ je v těchto případech divis. Viz též `\exhyphenpenalty`.

Kn: 96, 101, 272, 348, 451, **Ol:** 216, 226$_{94}$, 229–230, 362.

\i [plain]

Beztečkové i z pozice 16 CM fontu. Používáme-li pouze CM fonty a sestavujeme tedy akcenty pomocí `\'` a `\v`, je potřeba nezapomenout na to, že `\'i` dá „í“, zatímco správně je třeba psát `\'\i`, což dá kýžené í. V novém LaTeXu a po použití `\csaccents` je makro `\'` vybaveno vyšší inteligencí a dokáže rozpoznat, zda nenásleduje „i“. Pokud ano, pak správně sází „í“ a nikdy ne „í“. To je výhodné, protože třeba

```
387 \uppercase{p\v r\'i\v sern\'y} dá správně "PŘÍŠERNÝ", ale
388 \uppercase{p\v r\'\i\v sern\'y} dá chybné "PŘíŠERNÝ".
```

Makro `\i` je definováno jednoduše:

```
389 \chardef\i="10
```

Kn: 52–53, 356, **Ol:** není nikdy použito.

\ialign [plain]

V makrech občas potřebujeme mít jistotu, že před použitím `\halign` je registr `\tabskip` nulový. V takových situacích použijeme `\ialign`.

```
390 \def\ialign{\everycr={}\tabskip=0pt \halign}
```

Kn: 354, **Ol:** 128$_{148}$, 132$_{173}$, 184$_{229,\,232,\,236,\,239}$, 188$_{303}$, 189$_{325}$, 342$_{191}$, 346$_{217}$, 359$_{294}$, 394$_{476}$, 408$_{586}$.

`\if` ⟨*token1*⟩⟨*token2*⟩⟨*true text*⟩ [`\else` ⟨*false text*⟩] `\fi` [exp]
Je-li ASCII hodnota ⟨*token1*⟩ rovna ASCII hodnotě ⟨*token2*⟩, provede se ⟨*true text*⟩, jinak se [případně] provede ⟨*false text*⟩. Podmínka ⟨*token1*⟩ ⟨*token2*⟩ se sestaví až po úplné expanzi. Řídicí sekvence, které přijaly význam nějakého znaku (nebo sekvence) pomocí `\let` se považují za shodné s tímto znakem (nebo sekvencí). Ostatní (neexpandovatelné) řídicí sekvence mají hypotetickou ASCII hodnotu 256. Kategorie ⟨*token1*⟩ a ⟨*token2*⟩ se při vyhodnocení podmínky ignorují.

```
391 \if \kern\relax Ano to je pravda, protože 256=256.\fi
392 \let\hvezda=*
393 \if *\hvezda Ano, to je také pravda.\fi
394 \def\c{Aa}
395 \if \c Neplatí, protože ASCII "A" není rovno ASCII "a".\fi
```

Viz též ostatní primitivy typu `\if...`

Kn: 209, 210–211, 307, 377, 379, **Ol:** *sekce 2.3*, 30, 45–50, 52–53, 62, 72, 107, 126, 128, 218, 255, 357, 387, 402, 411.

`\ifcase` ⟨*number*⟩⟨*case* 0⟩`\or` ...`\or` ⟨*case n*⟩ [`\else` ⟨*others*⟩] `\fi` [exp]
Větvení zpracování podle ⟨*number*⟩ na $n+1$ větví. ⟨*case i*⟩ jsou ⟨*tokens*⟩, která se provedou v případě, že ⟨*number*⟩=i. Mezi ⟨*case* 0⟩ až ⟨*case n*⟩ je použito n separátorů `\or`. Není-li ⟨*number*⟩ v intervalu $0 \ldots n$, provede se [případně] ⟨*others*⟩.

Chceme-li interval podmínek posunout někam jinam, můžeme třeba psát:

```
396 \newcount\tempnum
397 \def\Vetvi #1{\tempnum=#1 \advance\tempnum by 1
398   \ifcase\tempnum #1 je $-1$\or #1 je 0\or #1 je 1\else
399   #1 není v intervalu $-1,\ldots,1$\fi}
```

Viz též ostatní `\if...`

Kn: 210, 349, 373, 390, 406, **Ol:** *sekce 2.3*, 46–47, 178$_{183}$, 292$_{75}$, 391$_{467}$.

`\ifcat` ⟨*token1*⟩⟨*token2*⟩⟨*true text*⟩ [`\else` ⟨*false text*⟩] `\fi` [exp]
Je-li kategorie ⟨*token1*⟩ rovna kategorii ⟨*token2*⟩, provede se ⟨*true text*⟩, jinak se [případně] provede ⟨*false text*⟩. Podmínka ⟨*token1*⟩ ⟨*token2*⟩ se sestaví až po úplné expanzi. U řídicích sekvencí, které přijaly význam nějakého znaku (nebo sekvence) pomocí `\let` se uvažuje kategorie shodná s kategorií tohoto znaku (nebo sekvence) v době provedení `\let`. Ostatní (neexpandovatelné) řídicí sekvence mají hypotetickou kategorii 16. ASCII kódy porovnávaných ⟨*token1*⟩ a ⟨*token2*⟩ se ignorují. Viz též ostatní `\if...`

Kn: 209, 210, 307, 377, **Ol:** *sekce 2.3*, 46, 48$_{164–165}$, 107$_{196–197}$, 251$_{218}$, 337$_{145}$.

\ifdim ⟨*dimen1*⟩⟨*relation*⟩⟨*dimen2*⟩⟨*true text*⟩ [\else ⟨*false text*⟩] \fi [exp]
Porovnávají se hodnoty typu ⟨*dimen*⟩. Přitom ⟨*relation*⟩ může být jeden ze tří tokenů: <$_{12}$, >$_{12}$ nebo =$_{12}$. Vlastnost ≤ a ≥ je možné implementovat přes ⟨*false text*⟩. Viz též ostatní primitivy \if...

Kn: 209, 353, 387, 417, **Ol:** *sekce 2.3*, 46, 119$_{33}$, 162$_{76}$, 236$_{121}$, 244$_{169, 171, 179}$, 245$_{198}$, 255$_{258}$, 265$_{321}$, 267$_{341, 345}$, 268$_{355, 357}$, 270$_{392}$, 282$_{588}$, 342$_{182}$, 345$_{209}$, 354$_{266}$, 396$_{492}$, 409$_{596}$, 422$_{662}$, 425$_{677}$, 435$_{719}$, 450$_{796}$.

\ifeof ⟨*4-bit number*⟩⟨*true text*⟩ [\else ⟨*false text*⟩] \fi [exp]
Test, zda vstupní soubor s číslem ⟨*4-bit number*⟩ není načten až do konce. Soubor je přiřazen povelem \openin a čten povelem \read. Pokud se nepovede pomocí \openin otevřít soubor (protože například neexistuje), TEX nehavaruje a hodnota \ifeof je hned nastavena na ⟨*true*⟩. Tím je možné implementovat algoritmus „načti soubor, jen když existuje“, viz makro \softinput na straně 288. Viz též ostatní \if...

Kn: 210, 217, **Ol:** *sekce 2.3*, 46, 288$_{12}$.

\iffalse ⟨*toto se přeskakuje*⟩ [\else ⟨*toto se vykoná*⟩] \fi [exp]
Test dopadne vždy „false“. Použití pro implementaci logického typu boolean a nových testů — naše makro může expandovat do \iftrue nebo \iffalse podle dalších podmínek. Viz též ostatní \if...

Kn: 210, 211, 260–261, 348, 385–386, **Ol:** *sekce 2.3*, 46, 50$_{183–184, 190}$, 51$_{193, 195}$, 52, 53$_{219}$, 318$_{14}$, 377, 402$_{559}$, 403, 411$_{612}$.

\ifhbox ⟨*8-bit number*⟩⟨*true text*⟩ [\else ⟨*false text*⟩] \fi [exp]
Test, zda \box⟨*8-bit number*⟩ je naplněn jako \hbox. Viz též ostatní \if...

Kn: 210, 392, 399, **Ol:** *sekce 2.3*, 46, 83, 84$_{122}$, 226$_{97}$, 265$_{318}$.

\ifhmode ⟨*true text*⟩ [\else ⟨*false text*⟩] \fi [exp]
Test, zda je hlavní procesor v horizontálním módu. Tj. v odstavcovém nebo vnitřním horizontálním módu. Viz též ostatní \if...

Kn: 209, 363, **Ol:** *sekce 2.3*, 45$_{115, 124}$, 46, 84$_{116}$, 91$_{151}$, 107$_{204}$, 203$_{452}$, 217$_{39}$, 237$_{133}$, 251$_{207}$.

\ifinner ⟨*true text*⟩ [\else ⟨*false text*⟩] \fi [exp]
Test, zda je hlavní procesor ve vnitřním módu. Tj. ve vnitřním horizontálním, vertikálním nebo matematickém módu. Nikoli v odstavcovém, hlavním vertikálním a display matematickém módu. Viz též ostatní \if...

Kn: 209, **Ol:** *sekce 2.3*, 45$_{116, 125}$, 46, 207$_{478}$, 208$_{486}$, 269$_{376}$.

\ifmmode ⟨*true text*⟩ [\else ⟨*false text*⟩] \fi [exp]
Test, zda je hlavní procesor v matematickém módu. Tj. ve vnitřním matematickém nebo display matematickém módu. Viz též ostatní \if...

Kn: 209, 215, 240, 353, 356, 360, 423, **Ol:** *sekce 2.3*, 46, 158$_{58}$, 192$_{394}$, 196$_{419}$, 355$_{278}$, 418$_{625}$, 435$_{725}$, 439$_{753}$.

`\ifnum` ⟨*number1*⟩⟨*relation*⟩⟨*number2*⟩⟨*true text*⟩ [`\else` ⟨*false text*⟩] `\fi` [exp]
Porovnávají se hodnoty typu ⟨*number*⟩. Přitom ⟨*relation*⟩ může být jeden ze tří tokenů: `<`$_{12}$, `>`$_{12}$ nebo `=`$_{12}$. Vlastnost $\leq$ a $\geq$ je možné implementovat přes ⟨*false text*⟩. Viz též ostatní `\if...`

Kn: 209, 208, 218–219, **Ol:** *sekce 2.3*, 37, 46, 51, 53, 59, 61–63, 106, 121–123, 127, 180, 195, 218, 236–237, 245, 262, 269, 271–272, 274, 282, 293, 337, 365, 398, 400.

`\ifodd` ⟨*number*⟩⟨*true text*⟩ [`\else` ⟨*false text*⟩] `\fi` [exp]
Test, zda číslo je liché. Vhodné pro implementaci různého chování TEXu na lichých a sudých stranách ve výstupní rutině. Viz též ostatní `\if...`

Kn: 209, 207, 416, 418–419, **Ol:** *sekce 2.3*, 46, 259$_{278}$, 260$_{289}$, 262$_{310}$, 263$_{312}$, 266$_{323}$, 267$_{336}$, 276$_{465}$, 281$_{565}$.

`\iftrue` ⟨*toto se vykoná*⟩ [`\else` ⟨*toto se přeskakuje*⟩] `\fi` [exp]
Test dopadne vždy „true“. Použití pro implementaci logického typu boolean a nových testů — naše makro může expandovat do `\iftrue` nebo `\iffalse` podle dalších podmínek. Viz též ostatní `\if...`

Kn: 210, 211, 260–261, 348, **Ol:** *sekce 2.3*, 46, 50$_{182}$, 51$_{192}$, 52, 376, 402$_{557}$, 403$_{567}$.

`\ifvbox` ⟨*8-bit number*⟩⟨*true text*⟩ [`\else` ⟨*false text*⟩] `\fi` [exp]
Test, zda `\box`⟨*8-bit number*⟩ je naplněn jako `\vbox`. Viz též ostatní `\if...`

Kn: 210, **Ol:** *sekce 2.3*, 46, 83, 274$_{452}$.

`\ifvmode` ⟨*true text*⟩ [`\else` ⟨*false text*⟩] `\fi` [exp]
Test, zda je hlavní procesor ve vertikálním módu. Tj. v hlavním vertikálním nebo vnitřním vertikálním módu. Viz též ostatní `\if...`

Kn: 209, **Ol:** *sekce 2.3*, 46, 214$_{18}$, 282$_{580}$.

`\ifvoid` ⟨*8-bit number*⟩⟨*true text*⟩ [`\else` ⟨*false text*⟩] `\fi` [exp]
Test, zda `\box`⟨*8-bit number*⟩ je prázdný. Viz též ostatní `\if...`

Kn: 210, 256, **Ol:** *sekce 2.3*, 46, 83, 128$_{154}$, 244$_{181}$, 245$_{197}$, 252$_{227, 229}$, 272$_{412}$, 273$_{428}$, 277$_{499}$, 281$_{571}$, 282$_{585}$.

`\ifx` ⟨*token1*⟩⟨*token2*⟩⟨*true text*⟩ [`\else` ⟨*false text*⟩] `\fi` [exp]
Jediný `\if`, kde se ⟨*token1*⟩ a ⟨*token2*⟩ pro zpracování podmínky berou v neexpandovaném tvaru. Test, zda jsou makra ⟨*token1*⟩ a ⟨*token2*⟩ zcela stejně definovaná (tj. makra mají stejnou masku parametrů a jejich těla obsahují stejnou posloupnost tokenů a makra jsou definována se stejným prefixem `\long` a `\outer`), případně test na shodu ASCII kódu i kategorie znaku současně,

případně test na naprostou shodu významu neexpandovatelné řídicí sekvence. Řídicí sekvence zavedené pomocí \let se považují za shodné se svými vzory.

Nedefinované řídicí sekvence vytvořené pomocí \csname…\endcsname se považují za shodné s \relax, ostatní nedefinované řídicí sekvence se považují za shodné s \undefined (pokud samozřejmě není sekvence \undefined definovaná, což nebývá zvykem).

```
\def\a#1.{a} \def\b#1:{a} \long\def\c#1:{a}
\def\d{a} \edef\e{a} \let\f=\kern \def\g{\kern}
\ifx\kern\f Ano, to je pravda.\fi
\ifx\kern\g To není pravda.\fi
\ifx\d\e To je pravda.\fi
% ale ostatní \a, \b, \c, \d se vzájemně z pohledu \ifx liší
\ifx\kern\hbox To tedy pravda není.\fi
\ifx AA To je fakt.\fi
```

Viz též ostatní \if...

Kn: 210, 384, 215, 307, 375–377, 418, **Ol:** *sekce 2.3*, 29, 37–38, 41, 46, 48–49, 58–60, 72, 126, 134, 174, 207–208, 217, 292, 333, 350, 367–368, 423.

\ignorespaces [h, v, m]

Hlavní procesor se přepne do stavu, ve kterém jsou ignorovány všechny následující mezery až po první povel, který není mezerou. Od této chvíle už jsou v horizontálním módu zase všechny mezery významné. Mimo horizontální mód lze primitiv také použít, ale nemá tam žádnou funkci.

Kn: 279, 333, 355, 424, **Ol:** 25_{49}, $26_{56,67}$, 441_{771}.

\immediate [pre]

Pokud toto slovo předchází (jako prefix) před \write, \openout a \closeout, provede se výstupní operace okamžitě, tedy nikoli až v okamžiku akce \shipout.

Kn: 280, 226–228, 422, 423, **Ol:** 18_{29}, 57_{263}, $208_{485,495}$, 289_{25}, $292_{71,73}$, $293_{84–85}$, 347, 362, 410, $457_{811–812}$.

\indent [h, v, m]

Vynucený start odstavcového módu, přičemž se vloží nejprve prázdný box o šířce \parindent. Uvnitř odstavcového módu se pouze vloží zmíněný prázdný box.

Kn: 282, 286, 291, 89, 94, 101, 263, 355, **Ol:** *88*, 89, 92, 282_{580}, 380_{415}, 384, 416, 441_{771}, 448.

\input ⟨*file name*⟩ [exp]

Přesměrování vstupu na specifikovaný soubor. Po přečtení tohoto souboru se pokračuje dále ve čtení aktuálního souboru. Povel \input může být použit i v souboru čteném pomocí \input atd. Je tedy možné vnořené čtení souborů.

Je-li ⟨*file name*⟩ uvedeno bez přípony (v názvu není tečka), doplní se přípona `.tex`. Pokud je přípona uvedena, je vyhledáván soubor se specifikovanou příponou. Podrobněji o způsobu zápisu ⟨*file name*⟩ viz stranu 321. O způsobu vyhledávání souborů v systému viz stranu 286.

Není-li soubor ⟨*file name*⟩ nalezen, tj. nelze-li jej otevřít ke čtení, začne TEX komunikovat s uživatelem: `I can't find file '`⟨*file name*⟩`'. Please type another input file name`. Pokud uživatel neodpoví jménem platného souboru v systému, TEX tvrdošíjně otázku opakuje. To může být pro uživatele frustrující. Doporučuje se zkusit odpovědět souborem `nul` nebo `null`, případně `/dev/null`. Často je užitečné mít pro tento účel instalován v systému TEXovských vstupů prázdný soubor `null.tex`. Pokud nechceme uživatele naštvat, používejme v makrech myšlenku z makra `\softinput`, viz stranu 288.

Přesnější popis činnosti povelu `\input` po úspěšném otevření souboru: Expand procesor přeruší zpracování vstupní fronty, uloží případný nezpracovaný zbytek do paměti a další tokeny bude čerpat z jednotlivých řádků nově otevřeného souboru. Například při:

```
\def\zkusinput{\input pokus ABC}
\zkusinput XYZ
```

je přesměrován text na soubor `pokus.tex` a po ukončení načítání tohoto souboru (buď pomocí `\endinput` nebo dosažením jeho konce) se expand procesor vrátí k tokenům ABC a token procesor pak bude zpracovávat znaky XYZ ze zbytku řádku a pak další řádek textu.

Protože je `\input` záležitostí expand procesoru, lze dělat i takové triky:

```
\def\makro #1\par{...}
\expandafter \makro \input pokus
```

V tomto příkladě se do parametru `#1` načte text prvního odstavce ze souboru `pokus.tex`. Bohužel neexistují v TEXu prostředky, jak načíst do parametru text souboru až po konec souboru, protože token procesor nevytváří na konci souboru žádný speciální token a expand procesor při dosažení konce souboru a neukončení čtení parametru nám vynadá `File ended while scanning use of \macro`. Ani prefix `\long` nepomůže.

Kn: 214,7,9,47,199,217,382–383,25–27,380,422, **Ol:** *sekce 7.1*, 14_{22}, 123_{112}, 277_{492}, 278_{505}, 284–286, $287_{6–7,\,9}$, 288_{13}, 293, 302, 321–322, 347, 358, 406.

`\inputlineno` *read-only* [integer]

Číslo zrovna čtené řádky ze zrovna čteného souboru.

Kn: 214, 271, **Ol:** není nikdy použito.

`\insert` ⟨*8-bit number*⟩`{`⟨*vertical material*⟩`}` [h, v, m]

Do separátní paměti uloží výsledek sazby ⟨*vertical material*⟩ v podobě vertikálního seznamu a do zrovna vytvářeného seznamu vloží odkaz typu insert na

tuto separátní paměť. Tento odkaz může po ukončení odstavce z horizontálního seznamu přejít do vertikálního (podobně jako \vadjust). Přítomnost tohoto odkazu v hlavním vertikálním módu ovlivní algoritmus plnění strany.

Kn: 122–125, 95, 259, 280–281, 416, 424, 454, 363, **Ol:** *247*, *sekce 6.7*, 87–88, 145, 247_{200}, 248–249, 251_{209}, 252_{225}, 255_{260}, $282_{582,590}$, 364_{318}, 401_{548}, 452, 454.

\interdisplaylinepenalty, \interfootnotelinepenalty [plain]
Registry použité v \displaylines, \eqalignno, \leqalignno a \vfootnote.

Kn: 193, 349, 362, 363, **Ol:** 353, $354_{263,268}$, $251_{201,210}$.

\insertpenalties *global, restricted* [integer]
Součet všech penalt za roztržení všech objektů typu \insert na aktuální stránce. Tato hodnota ovlivní vyhodnocení ceny zlomu a tím ovlivní algoritmus stránkového zlomu.

Kn: 123–125, 111, 114, 214, 254, 271, 256, **Ol:** *250*, *sekce 6.7*, 241, 251_{214}, 253, 262_{308}, 264.

\interlinepenalty (plain: 0) [integer]
Přidaná penalta mezi řádky uvnitř odstavce.

Kn: 104, 272, 363, 406, 419, **Ol:** 229, 243_{146}, 251_{210}, 267_{329}, 344.

\it [plain]
Makro \it je přepínač fontu do kurzívy. V matematickém módu se uplatní nastavení registru \fam na hodnotu \itfam a mimo matematický mód se uplatní přepínač \tenit deklarovaný přímo primitivem \font.

```
412 \newfam\itfam \def\it{\fam\itfam\tenit} % \it is family 4
413 \textfont\itfam=\tenit
```

Kn: 13–14, 165, 231–232, 332, 351, 409, 414–415, 419, 428, **Ol:** $14_{15–17}$, 31_{1}, 173, 174_{119}, $194_{406–407}$, $195_{408–410}$, 212_{8}, 259_{277}, 260_{284}, 276_{471}, 278_{513}, 427.

\item ⟨*znak*⟩, \itemitem ⟨*znak*⟩ [plain]
Makro \item zahájí sazbu odstavce speciálním způsobem. Především nastaví \hang, takže od druhého řádku budou všechny řádky odsazeny o velikost \parindent. První řádek bude také odsazen o tuto velikost (to je ovšem obvyklé i u ostatních odstavců). Navíc ale ⟨*znak*⟩ bude sázen vlevo do místa odsazení v prvním řádku. Makro \itemitem se chová analogicky, ovšem odsazení je dvojnásobné.

```
414 \def\item{\par\hang\textindent}
415 \def\itemitem{\par\indent\hangindent2\parindent \textindent}
```

Kn: 102–103, 117, 340–342, 355, 416, 419, **Ol:** není nikdy použito.

\j [plain]
Beztečkové j z pozice 17 CM fontu.

```
416 \chardef\j="11
```

Kn: 52, 356, **Ol:** není nikdy použito.

`\jobname` [exp]

Vrací název hlavního souboru zpracování.

Kn: 213, 214, 336, **Ol:** *285*, 208$_{494–495}$, 284, 287$_{5}$, 289$_{25}$, 290–291, 292$_{70–71}$, 293$_{81, 83–84}$, 442.

`\joinrel` [plain]

Makro se používá ke spojení dvou atomů typu Rel v jeden celek. Mezi dva atomy Rel se vloží `\joinrel`. Co se stane? (1) Připojí se atomy Rel k sobě, protože je tam záporná mezera. (2) Uvnitř vzniklé trojice atomů Rel Rel Rel nebude žádná automaticky přidaná mezera. (3) Tato trojice se jako celek z hlediska automatického mezerování chová jako jediný atom typu Rel a může se před ní a za ní vyskytnout mezera, pokud navazuje atom odpovídajícího typu.

```
417 \def\joinrel{\mathrel{\mkern-3mu}}
```

Kn: 358, **Ol:** 188$_{297–298, 308–311}$, 189$_{312–313, 317, 319, 334}$.

`\jot` [plain]

Hodnota, o kterou se zvětší řádkový proklad u většího počtu rovnic v jednom display módu. Použito v makrech `\displaylines`, `\eqalign`, `\eqalignno` a `\leqalignno`.

```
418 \newdimen\jot \jot=3pt
```

Kn: 194, 242, 349, 362, **Ol:** 354$_{264}$, 359$_{293}$.

`\kern` ⟨*dimen*⟩ [h, v, m]

Vloží netisknoucí výplněk dané šířky [h, m] nebo výšky [v]. Velikost výplňku může být i záporná (záporná mezera).

Kn: 280, 10, 40, 75, 87, 168, 454–455, 66, 256, 263, 306, 389, 394–395, 416, 424, **Ol:** na mnoha stránkách v knize, viz například heslo `\TeX`.

`\l`, `\L` [plain]

Polské ł nebo Ł, konstruované z písmen l nebo L a znaku z pozice 32 v CM fontech.

```
419 \def\l{\char32l}
420 \def\L{\leavevmode\setbox0=\hbox{L}%
421    \hbox to\wd0{\hss\char32L}}
```

Kn: 52–53, 356, **Ol:** není nikdy použito.

`\language` (iniTEX: 0) [integer]

Registr rozlišuje jednotlivé tabulky vzorů dělení. Jedná se o celé číslo v intervalu ⟨0, 255⟩, tj. maximální množství tabulek vzorů dělení je 256. Je-li

\language mimo tento interval, jako by bylo \language rovno nule, což je též implicitní hodnota tohoto registru.

Změna registru \language ve vertikálním módu ovlivní číslo zrovna čtené tabulky \patterns (při iniTEXu) a tabulky \hyphenation (i v produkční verzi TEXu).

Při vstupu do odstavcového módu (z vertikálního) TEX vloží na začátek horizontálního seznamu značku \setlanguage⟨*hodnota registru* \language⟩. Tato značka se na začátek seznamu nevkládá, pokud je \language rovno nule. Trvalé nastavení registru \language třeba na hodnotu 5 tedy způsobí automatické vložení značky \setlanguage5 na začátek každého odstavce. Všechny odstavce pak budou mít dělení slov podle páté tabulky \patterns a \hyphenation.

Každá změna registru \language v odstavcovém módu je ekvivalentní vložení značky pomocí primitivu \setlanguage⟨*nová hodnota* \language⟩. Ovšem tato akce se provede se zpožděním až v okamžiku, kdy TEX vkládá do seznamu další písmeno. Značka se vloží před toto písmeno. O významu značky \setlanguage viz primitiv \setlanguage.

Kn: 455, 273, 346, **Ol:** *222*, 220$_{54}$, 221, 223$_{76,\,78–80}$, 225, 230, 347$_{221}$, 356$_{280}$, 384, 399$_{517}$, 400$_{532}$, 430, 433$_{701}$.

\lastbox [h, v, m]

Je-li posledním elementem v aktuálním seznamu \hbox nebo \vbox, je ze seznamu odebrán a primitiv \lastbox se chová jako tento box. Například:

```
422 \vbox{...\par\hbox{abc}\global\setbox0=\lastbox}
```

Zde byl naposledy do aktuálního seznamu vložen \hbox{abc}. Primitiv \lastbox jej odebere ze seznamu. Protože je použit v kontextu \setbox, je odebraný vertikální box vložen do registru 0 a bude možné jej použít později.

Není-li posledním elementem v seznamu box, primitiv \lastbox se chová jako prázdný box a neodebírá ze seznamu nic.

Primitivem \lastbox nelze odebrat box z hlavního vertikálního seznamu, pokud tento box již přešel z přípravné oblasti do aktuální strany. To se stává téměř vždy po vložení boxu do hlavního vertikálního seznamu. Výjimkou je použití \lastbox za \unvbox, viz heslo \unvbox.

Kn: 278, 222, 354, 392, 398, 399, **Ol:** 84$_{121}$, 85, 90, 127$_{145}$, 128$_{153}$, 129, 143$_{279–280}$, 226$_{96}$, 265$_{315}$, 266, 269$_{386}$, 270, 319, 342$_{195–196}$, 343, 450$_{799}$.

\lastkern *read-only* [dimen]

Obsahuje velikost poslední mezery typu \kern v aktuálním seznamu. Pokud není posledním elementem mezera typu \kern, je \lastkern rovno 0 pt.

Primitiv \lastkern (na rozdíl od \lastbox) neodebírá ze seznamu měřený element. Pro odebrání posledního kernu ze seznamu je potřeba použít primitiv \unkern.

Kn: 214, 271, **Ol:** není nikdy použito.

`\lastpenalty` *read-only* [integer]
Udává hodnotu penalty, je-li tato posledním objektem v právě zpracovávaném seznamu. Jinak je `\lastpenalty` rovno nule.

Primitiv `\lastpenalty` (na rozdíl od `\lastbox`) neodebírá ze seznamu měřený element. Pro odebrání poslední penalty ze seznamu je potřeba použít primitiv `\unpenalty`.

Kn: 214, 271, **Ol:** není nikdy použito.

`\lastskip` *read-only* [glue]
Obsahuje velikost poslední mezery typu ⟨*glue*⟩ v aktuálním seznamu. Pokud není posledním elementem mezera typu ⟨*glue*⟩, je `\lastskip` rovno 0 pt plus 0 pt minus 0 pt.

Primitiv `\lastskip` (na rozdíl od `\lastbox`) neodebírá ze seznamu měřený element. Pro odebrání poslední mezery typu ⟨*glue*⟩ ze seznamu je potřeba použít primitiv `\unskip`.

Kn: 214, 271, 223, 392, **Ol:** 265$_{314–317}$, 266, 342$_{182}$, 396$_{492}$, 422$_{662}$, 425$_{677–678}$, 435$_{719}$, 450.

`\lccode` ⟨*8-bit number*⟩ *restricted* [integer]
Jedná se o kód malé alternativy znaku s ASCII kódem ⟨*8-bit number*⟩. Tím lze pro každý znak definovat chování algoritmu `\lowercase`. Rovněž je tento kód důležitý při vyhodnocování dělení slov.

Malá písmena anglické abecedy mají v iniTEXu `\lccode` nastaveno přímo ASCII hodnotu znaku, takže malá alternativa malého písmene je písmeno samo. Velká písmena anglické abecedy mají `\lccode` rovno ASCII hodnotě znaku plus 32 což odpovídá uspořádání písmen velké/malé v ASCII tabulce.

V csplainu je dále nastaveno `\lccode` akcentovaných minuskulí (malých písmen) shodně s pozicí znaku v $\mathcal{CS}$-fontu a majuskule (velká písmena) mají `\lccode` shodné s pozicí odpovídající minuskule. Viz soubor `extcode.tex`, resp. `il2code.tex`.

Kn: 41, 214, 271, 345, 452–454, **Ol:** 150, 218$_{52}$, 219–222, 224–225, 226$_{103–104}$, 227$_{105}$, 317, 347$_{221}$, 356$_{280}$, 389, 433$_{701}$, 447.

`\leaders` ⟨*box or rule*⟩⟨*glue specification*⟩ [v, h, m]
Jedná se o zobecněnou mezeru typu ⟨*glue*⟩, která je vyplněna opakovaně definovaným boxem, nebo natahovacím obdélníkem (linkou) v příslušném směru. Například:

```
423 \leaders \hrule  \hfill  % linka pružnosti \hfill
424 \leaders \hbox{x} \hfill % opakovaný box v mezeře \hfill
```

Ve vertikálním módu musí být ⟨*glue specification*⟩ vertikální mezerou typu ⟨*glue*⟩. Jsou dovoleny tyto varianty: \vskip⟨*glue*⟩, \vfil, \vfill, \vss, \vfilneg. V horizontálním módu musí tento parametr specifikovat mezeru horizontální: \hskip⟨*glue*⟩, \hfil, \hfill, \hss, \hfilneg. Jinak TEX ohlásí chybu. V matematickém módu může být ⟨*glue specification*⟩ buď horizontální nebo matematickou mezerou (\hskip a přátelé nebo \mskip).

Mezi parametry ⟨*box or rule*⟩ a ⟨*glue specification*⟩ může být mezera, která je ignorována (viz například řádek 424). Způsob usazení jednotlivých boxů do mezery závisí na použité alternativě primitivu (\leaders, \cleaders nebo \xleaders) a podrobně je vyložen v sekci 4.1.

Kn: 224, 95, 110, 223, 225, 357, 392–394, **Ol:** *116*, *sekce 4.1*, 87, 111_{218}, 112_{223}, $116_{1–6,\,8–11}$, $117_{12–13}$, $118_{15,\,23}$, 119_{34}, 120, $184_{252–253,\,255–256}$, 185, 188_{306}, 259_{276}, 319, 324, 347, 355, 373_{386}, 384.

\leavevmode [plain]

Makro, které přepíná z vertikálního módu do horizontálního (tj. zahajuje odstavec jako \indent). Uvnitř horizontálního módu nebo matematického módu neudělá makro vůbec nic. Makro se opírá o schopnost primitivu \unhbox zahájit odstavec, pokud je TEX ve vertikálním módu. Navíc makro do seznamu vkládá obsah prázdného boxu \voidb@x, takže vlastně nevkládá vůbec nic.

```
425 \newbox\voidb@x % permanently void box register
426 \def\leavevmode{\unhbox\voidb@x}
```

Kn: 313, 333, 356, 408, 420, **Ol:** 29_{86}, 92, 93_{161}, 123_{87}, 192_{394}, 195_{417}, 196_{420}, 214_{18}, 226_{102}, $289_{19,\,21,\,27}$, 334_{119}, 335_{134}, 338_{147}, 371_{381}, 381_{420}, 393_{469}, 408_{585}.

\left ⟨*delimiter*⟩ [m]

Parametr ⟨*delimiter*⟩ se použije ve smyslu otevírací závorky pružné velikosti. Viz též primitiv \right.

Kn: 292, 155–157, 196, 437, 148–150, 171, **Ol:** *150*, *164*, 134, 144–146, 150–153, 158, 165, 169, 178, 185–186, 192–194, 197, 319, 341–343, 345–346, 351–352, 407, 412, 419, 426, 464.

\leftarrowfill [plain]

Šipka natahovací délky. Kód makra, viz stranu 184.

Kn: 357, **Ol:** $184_{233,\,246}$.

\lefthyphenmin (plain: 2) [integer]

Minimální počet znaků v části slova před rozdělením. Formát csplain nastavuje pro češtinu (\language=5) hodnotu 2, tj. je možné ta-kové dě-lení slov, ale nikoli o-bludné dělení slov. Viz též \righthyphenmin, \setlanguage.

Kn: 454, 273, 455, 364, **Ol:** 221, 223, 225, 347_{222}, 356_{281}, 426, 430, 433_{702}.

\leftline {⟨*text*⟩} [plain]
Parametr ⟨*text*⟩ bude usazen vlevo do řádku. Vpravo může ⟨*text*⟩ přečnívat nebo naopak nemusí zaplnit řádek celý.

```
427 \def\leftline#1{\line{#1\hss}}
```

Kn: 257, 259–260, 326, 353, 101, **Ol:** 340$_{159}$, 395$_{484}$.

\leftskip (iniTEX: 0 pt) [glue]
Mezera, která je umístěna na levé straně každého řádku při sestavování odstavce. Tato mezera zůstává uvnitř boxu s řádkem, který má celý šířku \hsize (nebo je jeho šířka určena pomocí \hangindent nebo \parshape). Viz též \rightskip.

Kn: 100, 274, 317, 407, 419, **Ol:** *234*, 25$_{48}$, 26$_{55,66}$, 143$_{267}$, 211, 227, 229, 234$_{111}$, 235$_{113}$, 251$_{215}$, 278$_{512}$, 398$_{504}$, 426.

\leqalignno [plain]
Sazba soustavy rovnic v rámci jednoho vstupu do display módu. Jednotlivé řádky jsou širší o 3 pt a obsahují levou stranu a pravou stranu rovnice oddělené pomocí &. Třetí (nepovinné) pole oddělené pomocí & může specifikovat značku rovnice, která se umístí vlevo od příslušné rovnice. Podobně pracuje makro \eqalignno.

```
428 % \@lign a \displ@y, viz řádky 264 až 269 u \displaylines
429 \def\leqalignno#1{\displ@y \tabskip=\centering
430   \halign to\displaywidth{\hfil$\@lign \displaystyle{##}$%
431    \tabskip=\z@skip &$\@lign \displaystyle{{}##}$\hfil
432    \tabskip=\centering&\kern-\displaywidth\rlap{$\@lign##$}%
433    \tabskip\displaywidth\crcr
434    #1\crcr}}
```

Kn: 192, 194, 362, **Ol:** *206*, 354, 380–381, 410.

\leqno [m]
Jako \eqno, ovšem text za tímto slovem se umístí vlevo od centrované rovnice.

Kn: 293, 189, 375–376, 187, **Ol:** *197*, 198$_{434}$, 199–201, 203$_{448–449}$, 204–205.

\let ⟨*control sequence*⟩⟨*equals*⟩⟨*one optional space*⟩⟨*token*⟩ [a]
Daná ⟨*control sequence*⟩ dostane stejný význam jako ⟨*token*⟩. Tento ⟨*token*⟩ může být cokoli, tedy token typu dvojice, nebo řídicí sekvence. Význam tokenu je do ⟨*control sequence*⟩ uložen v okamžiku provedení \let. Pokud je například ⟨*token*⟩ řídicí sekvence nebo aktivní znak a později je jeho význam změněn, ⟨*control sequence*⟩ zůstává u původního významu.

Shodnost mezi ⟨*control sequence*⟩ a ⟨*token*⟩ po přiřazení pomocí \let není absolutní. Při načítání parametru makra a rozlišování separátoru parametru si TEX všímá v ⟨*control sequence*⟩ jen jejího identifikátoru a nepátrá po jejím

významu. Podobná situace nastává, pokud ⟨*control sequence*⟩ vystupuje jako parametr v povelu hlavního procesoru.

Protože za ⟨*equals*⟩ následuje ⟨*one optional space*⟩ (neboli jedna nepovinná mezera), je potřeba v makru mezeru fyzicky napsat, kdykoli chceme, aby se pomocí `\let` zavedla do ⟨*control sequence*⟩ i uživatelova mezera. Vyzkoušejte si:

```
\def\ukazznak{\message{\meaning\znak}}
\def\sejmitoken:{\afterassignment\ukazznak \let\znak }
% nebo
\def\sejmitoken:{\afterassignment\ukazznak \let\znak=}
% pak v obou případech po
\sejmitoken: a
% zjistíme, že byl do \znak zaveden token "a".
% zatímco při
\def\sejmitoken:{\afterassignment\ukazznak \let\znak= }
\sejmitoken: a
% se zavede do \znak mezera před "a".
```

Kn: 277, 206–207, 215, 307, 309, 352, 376, **Ol:** , *sekce 3.2* a dále na mnoha stránkách v knize.

`\limits` [m]

Pokud je posledním elementem seznamu atom typu Op, nastaví se mu příznak „limits“, který ovlivní usazování indexů. Indexy i exponenty budou sázeny nahoře a dole na osu základu atomu typu Op a nikoli vpravo nahoře a vpravo dole.

Implicitní příznak každého atomu typu Op je „displaylimits“, viz též primitiv `\displaylimits` a `\nolimits`.

Kn: 292, 144, 159, 443, 359, **Ol:** *163*, 163_{80}, 164_{82}, $184_{238,\,242}$, 189_{333}, 343, 344_{207}, 353, 406.

`\line` [plain]

Zkratka za `\hbox to\hsize`, což v makrech můžeme často využít. Upozorňuji, že LaTeX definuje makro `\line` zcela jiným způsobem, takže používání této řídicí sekvence v našich makrech může způsobit, že naše práce nebude kompatibilní s LaTeXem.

```
\def\line{\hbox to\hsize}
```

Kn: 72, 77, 101, 224, 232, 252, 255–257, 353, 412, **Ol:** 112_{225}, 118_{23}, $119_{36,\,38,\,40}$, 243_{148}, 245_{189}, 257_{266}, $262_{306–307,\,309}$, 267_{334}, 281_{566}, 346_{219}, 385_{427}, 395_{484}, 426_{679}, 452_{803}.

`\linepenalty` (plain: 10) [integer]

Tato hodnota vystupuje ve vzorci pro výpočet „demerits“ jednotlivého řádku v algoritmu řádkového zlomu. Viz stranu 228.

Kn: 98, 272, 314, 316, 348, **Ol:** *228*, 212, 226$_{94}$, 230.

`\lineskip` (plain: 1 pt) [glue]
Vertikální mezera mezi spodní částí jednoho a horní částí druhého řádku v případě, že nelze použít `\baselineskip`. Viz též `\lineskiplimit`.

Kn: 78–80, 104, 194, 274, 281, 349, 351–352, **Ol:** *sekce 3.7*, 108, 109$_{208–209}$, 110, 138, 189$_{324}$, 264, 266, 354, 387, 407$_{579}$, 408$_{586}$, 409$_{595}$, 410$_{605}$, 421.

`\lineskiplimit` (plain: 0 pt) [dimen]
Je-li vertikální mezera mezi spodní částí jednoho a horní částí druhého řádku menší než `\lineskiplimit` při použití `\baselineskip`, pak se druhý řádek vertikálně usadí nikoli podle `\baselineskip`, ale podle `\lineskip`.

Kn: 78–80, 104, 194, 274, 281, 349, 351–352, 362, **Ol:** *sekce 3.7*, 108, 109$_{208, 210}$, 110, 112, 115, 143$_{266}$, 189$_{324}$, 190$_{378}$, 264, 330$_{105}$, 339$_{153}$, 350$_{254}$, 354$_{266}$, 387, 407$_{581}$, 408, 409$_{595}$, 410$_{601, 603}$, 411$_{607}$.

`\llap {`⟨*text*⟩`}` [plain]
Do seznamu se vloží prázdný box nulové šířky, z něhož ⟨*text*⟩ vyčnívá směrem doleva. Pravý okraj ⟨*textu*⟩ se kryje s polohou boxu. Viz též `\rlap`.

```
447 \def\llap#1{\hbox to0pt{\hss#1}}
```

Kn: 82–83, 189, 340–341, 353, 355, 381, 416–417, 422, **Ol:** 25$_{49}$, 26$_{56, 67}$, 29$_{85}$, 84$_{123}$, 85, 201, 359$_{300}$, 427, 441$_{771}$, 452.

`\long` [pre]
Je-li toto slovo uvedeno jako prefix povelu `\def` (resp. `\edef` apod.), je dovoleno, aby parametr takto definovaného makra obsahoval token `par`. Jinak TEX v takovém případě hlásí z bezpečnostních důvodů chybu.

Kn: 205–206, 210, 275, 331, 375, 378, 382, **Ol:** *34*, 31, 34$_{26}$, 40, 368$_{374}$, 377, 378$_{400}$, 379.

`\loop` ⟨*tokens1*⟩ `\if...` ⟨*tokens2*⟩ `\repeat` [plain]
V místě `\repeat` se zpracování vrací do místa `\loop`, pokud je podmínka `\if...` splněna. Není-li podmínka `\if...` splněna, přeskočí se ⟨*tokens2*⟩, sekvence `\repeat` uzavře `\if...` a cyklus se ukončí. Podmínkou `\if...` může být jakýkoli primitiv pro podmínky (viz stranu 46). Podmínka *nesmí* mít mezi ⟨*tokens2*⟩ své `\else` ani `\fi`. Na sekvenci `\repeat` je možno z tohoto hlediska pohlížet jako na `\fi` od podmínky `\if...` Vložené podmínky uvnitř ⟨*tokens1*⟩ a ⟨*tokens2*⟩ jsou možné.

Makro `\loop` je definováno:

```
448 \def\loop#1\repeat{\def\body{#1}\iterate}
449 \def\iterate{\body \let\next\iterate
450    \else \let\next\relax \fi \next}
451 \let\repeat=\fi % this makes \loop...\if...\repeat skippable
```

Při expanzi \body se předpokládá, že tam je jedna neukončená podmínka, na kterou navazuje \else a \fi na řádku 450. Proto se nakonec provede \next jako \relax nebo jako \iterate podle toho, zda podmínka je splněna či nikoli. Zavedení \let\repeat=\fi umožní správné přeskočení celé konstrukce \loop v rámci nějaké vnější podmínky \if.

Vidíme též, že vnoření cyklů \loop je možné pouze v případě, že vnitřní \loop bude uzavřen ve skupině {...}. Pak totiž vnější \loop nabere do parametru #1 vše až po *svůj* separátor \repeat. Také se nebude křížit dvojí definice makra \body, protože po opuštění vnitřního cyklu se uzavře skupina a \body bude mít znova význam těla celého vnějšího cyklu.

Kn: 352, 217–219, 373–374, 387, 417, **Ol:** 48, 84_{121}, 85, 106_{188}, 121_{53}, 122_{56}, $123_{93,\,96,\,99}$, 127_{137}, 226_{96}, $236_{124,\,126}$, 245_{184}, 265_{315}, $274_{442,\,451}$, 398_{502}.

\looseness [integer]

Počet řádků, o kolik má být odstavec delší (kladná hodnota) nebo kratší (záporná hodnota), než představuje ideální řádkový zlom bez tohoto doplňujícího požadavku.

Přesněji: nechť algoritmus řádkového zlomu najde řešení v k-tém průchodu (pro $k = 1, 2, 3$) a toto řešení má při požadavku na minimální „demerits“ n řádků. Pak TEX místo tohoto řešení hledá řešení s $n+$\looseness řádků. V ideálním případě takové řešení existuje v i-tém průchodu pro $i = k, \ldots, 3$. TEX pak volí řešení, které za podmínky stanoveného počtu řádků má nejmenší „demerits“ a nejmenší i. Pokud požadované řešení neexistuje pro žádné i, TEX vyhledá v posledním průchodu taková řešení, která mají počet řádků nejblíže požadovanému počtu $n+$\looseness. Mezi nimi vybere řešení s nejmenším „demerits“. Při tomto rozdílu mezi požadavkem uživatele a skutečným výsledkem bohužel TEX nehlásí žádnou zprávu.

Po sestavení každého odstavce vrací TEX hodnotu parametru \looseness zpět na nulu. Nenulové nastavení \looseness v nějakém odstavci tedy chápeme jako „lokální“ nastavení týkající se jediného odstavce.

Pokud je odstavec rozdělen do více částí display módem, pak se každá část počítá podle aktuálního stavu registru \looseness, ale registr se nuluje až po úplném ukončení odstavce. Nechť máme například odstavec rozdělen na dvě části jedním display módem. Uvedeme-li nenulové \looseness v první části odstavce, bude se týkat obou částí. Napíšeme-li nenulové \looseness až v druhé části, bude se týkat jen druhé části.

Práce s tímto registrem je důležitá při závěrečných úpravách knihy, kdy chceme z důvodu požadavku na vyrovnaný počet řádků na jednotlivých stranách bez parchantů přesázet některé odstavce, aby byly o řádek delší nebo kratší. Bohužel, takovéto úpravy je nutno zanášet přímo do textu a jsou závislé na konečné volbě formátu knihy. Jestliže příště volíme jiný formát knihy, můžeme manipulaci s registrem \looseness v textu ignorovat pomocí triku

```
452 \newcount\looseness % zakryje přístup k primitivu
```

Pokud ale i v novém formátu budeme chtít s touto vlastností pracovat, nezbyde nám, než editorem vyhledat všechny výskyty `\looseness` v textu, vymazat je a nahradit novými volbami.

Kn: 103–104, 109, 273, 349, 342, **Ol:** 228–229, 258, 416.

`\lower` ⟨*dimen*⟩⟨*box*⟩ [h]
Box se umístí o ⟨*dimen*⟩ níže, než by odpovídalo sazbě boxu bez použití tohoto primitivu.

Kn: 285, 290, 80, 151, 179, 66, **Ol:** 10, 87_{139}, 91_{152}, 95–96, 189_{323}, 289_{18}, 440_{770}.

`\lowercase` ⟨*filler*⟩`{`⟨*balanced text*⟩`}` [h, v, m]
Konvertuje ⟨*balanced text*⟩ do malých písmen podle hodnot `\lccode`.

Přesněji: první token parametru musí (po případné expanzi) být „`{`“. Následující řadu tokenů čte TEX bez expanze až po odpovídající „`}`“. Při tomto čtení konvertuje všechny tokeny typu „uspořádaná dvojice“ (ASCII kód, kategorie). Zaměří se na ASCII kódy a kategorie ponechá beze změny. Všem takovým tokenům s nenulovým `\lccode` změní ASCII kód podle jejich `\lccode`. Pak TEX odstraní obklopující závorky `{...}` a vrátí se na začátek (případně upraveného) ⟨*balanced text*⟩ znovu. V druhém průchodu již TEX provádí obvyklou úplnou expanzi s následným zpracováním. Viz též `\uppercase`.

Konverze na malá písmena se netýká pouze tokenů typu „řídicí sekvence“ a dále tokenů typu „uspořádaná dvojice“, které mají nulový `\lccode`. Beze změny zůstávají též tokeny, jejichž `\lccode` je přímo rovna ASCII kódu (jedná se o malá písmena). Všechny ostatní tokeny budou změněny. Například po zápise:

```
453 \catcode`A=13 \def A{xx}  \lowercase{A}
```

dostaneme chybové hlášení `Undefined control sequence a`. Co se stalo? Primitiv `\lowercase` konvertoval token $\boxed{\texttt{A}}_{13}$ na token $\boxed{\texttt{a}}_{13}$ a ve druhém průchodu se ukázalo, že token $\boxed{\texttt{a}}_{13}$ nebyl ještě definován.

Kn: 279, 41, 215, 307, 345, **Ol:** 71, 318, 451.

`\mag` (iniTEX: 1 000) [integer]
1 000×koeficient zvětšení celého dokumentu. Údaj se ukládá do hlavičky `dvi` souboru při první akci `\shipout`. Odtud může být údaj použit příslušným `dvi` ovladačem. Tím je míněno, že `dvi` ovladač vynásobí všechny rozměrové údaje v `dvi` požadovaným koeficientem zvětšení. Jednotlivé rozměrové údaje tedy nejsou v `dvi` souboru měněny. Viz stranu 309, údaj *magnification* a viz též makro `\magnification`. Výchozí hodnota registru `\mag` je 1 000, tj. žádné zvětšení.

Pokud změníme `\mag` až po provedení prvního povelu `\shipout`, nebude mít tato změna vliv. Pouze se objeví varovné hlášení.

Maximální možná hodnota registru `\mag` je 32 768 (tj. zhruba 32 násobné zvětšení) a minimální hodnota je 1 (tj. tisícinásobné zmenšení). V obou mezních případech se může stát, že ovladače budou mít problémy.

Jednotky s předponou `true` jsou v TEXu násobeny převrácenou hodnotou koeficientu zvětšení f = `\mag`/1 000. Je-li například `\mag=1500` ($f = 1{,}5$), pak jednotka `1truecm` vlastně označuje velikost 1/1,5 cm a s touto velikostí TEX pracuje. Pokud `dvi` ovladač při tisku použije zvětšení podle použitého údaje (v našem příkladě $f = 1{,}5$), pak jednotka `1cm` bude na výstupu veliká 1,5 cm, zatímco jednotka `1truecm` bude mít na výstupu skutečně velikost 1 cm. Proto se tyto jednotky označují „true“, neboli „pravdivé“. Jakmile použijeme jednotku s předponou `true`, není již možné změnit hodnotu registru `\mag`.

Kn: 60, 270, 273, 348, **Ol:** 77, 309, 320, 390$_{455,\,457}$, 431.

`\magnification` ⟨*equals*⟩⟨*number*⟩ [plain]

Číslo ⟨*number*⟩ se uloží do registru `\mag`, takže `dvi` ovladač použije dodatečné zvětšení sazby podle koeficientu ⟨*number*⟩/1 000. V rámci tohoto zvětšení zůstávají zachovány původní rozměry zrcadla strany, jak je definuje plain. Jednoduše řečeno, velikost písma se změní ale zrcadlo sazby nikoli. Sazba bude tedy formátována do obecně jiného počtu řádků.

```
\def\magnification{\afterassignment\m@g\count255 }
\def\m@g{\mag=\count255
  \hsize=6.5truein \vsize=8.9truein \dimen\footins=8truein}
```

Formát csplain pracuje s jiným implicitním formátem zrcadla sazby. Požadavkem je, aby zůstaly okraje jako v plainu (1 in), ale pracuje se s formátem A4, nikoli s americkým formátem papíru. Proto csplain v souboru `plaina4.tex` předefinovává pomocné makro `\m@g`:

```
\def\m@g{\mag=\count255
   \hsize=159.2truemm \vsize=239.2truemm
   \dimen\footins8truein}
```

Kn: 17, 59–60, 349, 403–404, **Ol:** 389.

`\magstep` ⟨*number*⟩, `\magstephalf` [plain]

Je-li koeficient zvětšení f vyjádřen pomocí malé mocniny čísla 1,2, je možno použít tato makra:

`\magstep0`	($f = 1{,}0$)	`\magstephalf`	($f = \sqrt{1{,}2} = 1{,}095$)
`\magstep1`	($f = 1{,}2$)		
`\magstep2`	($f = 1{,}2^2 = 1{,}44$)		
`\magstep3`	($f = 1{,}2^3 = 1{,}728$)		
`\magstep4`	($f = 1{,}2^4 \doteq 2{,}074$)		
`\magstep5`	($f = 1{,}2^5 \doteq 2{,}488$)		

Takto odstupňované zvětšení fontů je v typografii dosti časté. Například, pro nadpisy nejnižší úrovně použijeme zvětšení `\magstep1` vzhledem k základnímu

písmu. Pro nadpisy vyšší úrovně volíme zvětšení `\magstep2` vzhledem k základnímu písmu, tj. zvětšení `\magstep1` vzhledem k nadpisům první úrovně.

Uvedená makra expandují na příslušný tisícinásobek koeficientu f, takže jsou použitelná v kontextu `\font...scaled\magstep`⟨n⟩ nebo třeba při deklaraci koeficientu zvětšení celého dokumentu `\magnification=\magstep`⟨n⟩.

```
\def\magstephalf{1095}
\def\magstep#1{\ifcase#1 1000\or 1200\or 1440\or 1728\or
  2074\or 2488\fi\relax}
```

Kn: 17, 59–60, 349, 403–404, **Ol:** 65$_{6}$, 66, 69$_{16}$, 90$_{147}$, 175$_{137–142}$, 421$_{652–654}$.

`\makeheadline`, `\makefootline` [plain]
Makra k vytvoření záhlaví a paty strany ve výstupní rutině. Kód maker, viz stranu 262.

Kn: 255–257, 364, **Ol:** 262$_{300, 305}$, 263.

`\mark` ⟨*filler*⟩`{`⟨*balanced text*⟩`}` [a]
Na příslušný ⟨*balanced text*⟩ budou po činnosti algoritmu uzavření strany expandovat sekvence `\botmark`, `\topmark` a `\firstmark` (případně po `\vsplit` analogické sekvence `\splitfirstmark` a `\splitbotmark`).

V okamžiku načtení primitivu `\mark` je ⟨*balanced text*⟩ expandován analogicky jako při `\edef`. Do zpracovávaného seznamu se pak vloží bezrozměrná značka odkazující na expandovaný ⟨*balanced text*⟩. Jestliže byla značka vložena do horizontálního seznamu v odstavcovém módu, je po kompletování odstavce přesunuta do odpovídajícího vnějšího vertikálního módu bezprostředně pod řádek, ve kterém značka byla umístěna původně (jako při `\vadjust`). Při kompletování boxu 255 v algoritmu uzavření strany jsou významné jen ty značky, které jsou přítomny ve vnějším vertikálním seznamu. Značky, které zůstaly „schovány v boxech“ jsou ignorovány. Totéž platí pro alternativy `\split...` při činnosti primitivu `\vsplit`.

Kn: 280, 95, 157, 216, 258–263, 417, 454, **Ol:** 87–88, 101, 145, 256, 258$_{271}$, 259$_{275}$, 260$_{290}$, 266, 275, 318, 343, 364, 437, 444, 452, 456.

`\mathaccent` ⟨*15-bit number*⟩⟨*math field*⟩ [m]
Vytvoří atom typu Acc s akcentem ⟨*15-bit number*⟩ a se základem ⟨*math field*⟩. Při sazbě atomu typu Acc se umístí akcent nad základ. Vlastnost akcentu (třída math objektu, příslušnost k rodině fontů, kód ve fontu) je specifikována v ⟨*15-bit number*⟩ stejně jako v `\mathchar`. Údaj o třídě objektu je v tomto čísle ignorován. Podrobnější algoritmus, viz `\skewchar`.

Kn: 291, 443, 157, 170, 359, **Ol:** 152, 167, 183$_{215–226}$, 333, 433.

`\mathbin` ⟨*math field*⟩ [m]
Příslušný ⟨*math field*⟩ bude základem atomu typu Bin.

Kn: 155, 291, 361, **Ol:** 149, 156$_{49}$, 190$_{364}$, 197$_{427–428}$.

\mathchar ⟨*15-bit number*⟩ [m]
Vysází se symbol jako matematický objekt. Třída tohoto objektu, příslušnost k rodině fontů a kód ve fontu jsou specifikovány v ⟨*15-bit number*⟩. Např. hexadecimální vyjádření tohoto čísla 6ABC značí objekt třídy 6 sázený z rodiny fontu A z pozice BC. O třídách matematického objektu, viz stranu 147.

Kn: 155, 289, **Ol:** 147$_{14–15}$, 148, 151$_{32}$, 152, 162$_{75}$, 188$_{292, 296}$, 324, 392, 393$_{470}$.

\mathchardef ⟨*control sequence*⟩⟨*equals*⟩⟨*15-bit number*⟩ [m]
Nová ⟨*control sequence*⟩ bude synonymem pro \mathchar⟨*15-bit number*⟩.

Kn: 155, 199, 214, 215, 272, 277, 289, 336, 394, **Ol:** 66, 75$_{46}$, 148$_{17}$, 157, 160$_{68}$, 164$_{81}$, 171, 173, 178$_{191–192}$, 179$_{194–200}$, 184$_{249–250}$, 186, 272$_{413}$, 319–320, 324, 326$_{56}$, 330$_{83–86}$, 333, 400$_{522}$, 441.

\mathchoice {*D*}{*T*}{*S*}{*SS*} [m]
Větvení zpracování matematického seznamu do čtyř větví podle toho, zda je sazba ve stylu *D*, *T*, *S* nebo *SS*. Do matematického seznamu se vloží všechny čtyři alternativy vedle sebe. Teprve při konverzi matematického seznamu do horizontálního TEX ví, v jakém stylu bude zpracováno místo, ve kterém je \mathchoice použito. V tomto okamžiku se vybere z matematického seznamu odpovídající alternativa a ostatní alternativy se ignorují.

O stylech matematického seznamu viz stranu 154.

Kn: 151, 157, 292, **Ol:** 156$_{53}$, 189$_{330}$, 195$_{414}$, 196–197, 393$_{471}$, 428.

\mathclose ⟨*math field*⟩ [m]
Příslušný ⟨*math field*⟩ bude základem atomu typu Close.

Kn: 155, 291, 322, 359, **Ol:** 149, 341$_{174, 176, 178, 180}$.

\mathcode ⟨*8-bit number*⟩ (iniTEX a plain: viz stranu 182) *restricted* [integer]
Každý znak s ASCII hodnotou ⟨*8-bit number*⟩ má svůj \mathcode, což je 15 bitové číslo. Toto číslo obsahuje stejné údaje pro matematický objekt jako v povelu \mathchar. Odpovídající matematický objekt bude vytvořen vždy při výskytu znaku s ASCII kódem ⟨*8-bit number*⟩ nebo při použití \char⟨*8-bit number*⟩ v matematickém módu.

Znak může mít přiřazen i \mathcode hodnoty "8000, což způsobí chování znaku v matematickém módu, jako by byl aktivní. Mimo matematický mód nemá toto nastavení žádný vliv. Viz heslo ’$_{12}$ a příklad na straně 160.

Kn: 154–155, 134, 214, 271, 289, 319, 326, 344, **Ol:** 62$_{337}$, 148$_{16}$, 150, 151$_{33}$, 160$_{68}$, 161$_{69}$, 171–173, 178$_{188–189}$, 182–183, 194, 324, 333, 346, 352, 363, 392.

\mathhexbox ⟨*hexadecimální zápis rodiny a pozice*⟩ [plain]
Makro vytvoří box obsahující stejný znak jako při použití \mathchar. Na rozdíl od \mathchar nepíšeme údaj o třídě. Argument makra musí obsahovat tři tokeny: hexadecimální číslici udávající rodinu a dvě hexadecimální číslice pro

pozici. Výsledná sazba je vždy ve stylu T. Protože je v boxu, je sazba použitelná v libovolném módu.

```
469 \def\mathhexbox#1#2#3{\leavevmode
470   \hbox{$\m@th \mathchar"#1#2#3$}}
```

Kn: 356, **Ol:** 348$_{239}$, 351$_{256–257}$, 413$_{619}$, 428$_{691}$.

\mathinner ⟨*math field*⟩ [m]
Příslušný ⟨*math field*⟩ bude základem atomu typu Inner. Ten se chová jako typ Ord až na jiné mezerování.

Kn: 155, 171, 199, 291, 359, **Ol:** 149, 190$_{375–376,\,380}$.

\mathop ⟨*math field*⟩ [m]
Příslušný ⟨*math field*⟩ bude základem atomu typu Op. O sazbě tohoto atomu, viz stranu 163.

Kn: 155, 178, 291, 324–325, 361, **Ol:** 149, 150$_{25}$, 184$_{235,\,239}$, 189$_{333,\,338–345}$, 190$_{346–362,\,368–374}$, 343, 344$_{207}$.

\mathopen ⟨*math field*⟩ [m]
Příslušný ⟨*math field*⟩ bude základem atomu typu Open. Ten se chová jako typ Ord až na jiné mezerování.

Kn: 155, 291, 322, 359, **Ol:** 149, 341$_{174,\,176,\,178,\,180}$.

\mathord ⟨*math field*⟩ [m]
Příslušný ⟨*math field*⟩ bude základem atomu typu Ord. Případný exponent nebo index bude vpravo od základu. Podrobněji viz stranu 162. Vkládání automatických mezer mezi atomy, viz stranu 158.

Kn: 88–89, 455, 291, **Ol:** 149–150, 184$_{245–246}$.

\mathpalette ⟨*macro*⟩{⟨*text*⟩} [plain]
Například použití `\mathpalette\macro{text}` v matematickém módu pracuje jako `\macro`⟨*style primitive*⟩`{text}`, kde ⟨*style primitive*⟩ je jeden z tokenů \displaystyle, \textstyle, \scriptstyle nebo \scriptscriptstyle podle toho, v jakém stylu je `\mathpalette` použito. Je potřeba nejprve definovat `\macro` se dvěma parametry tak, aby se podle prvního parametru například větvilo a provádělo tím činnost závislou na matematickém stylu.

```
471 \def\mathpalette#1#2{\mathchoice{#1\displaystyle{#2}}%
472   {#1\textstyle{#2}}{#1\scriptstyle{#2}}%
473   {#1\scriptscriptstyle{#2}}}
```

Kn: 151, 360, **Ol:** 188$_{295,\,299,\,301}$, 189$_{330}$, 197$_{427–428}$, 418$_{625}$, 419, 427$_{687}$, 428, 435$_{725}$.

\mathpunct ⟨*math field*⟩ [m]
Příslušný ⟨*math field*⟩ bude základem atomu typu Punct. Ten se chová jako typ Ord až na jiné mezerování.

Kn: 155, 291, **Ol:** 149.

\mathrel ⟨*math field*⟩ [m]
Příslušný ⟨*math field*⟩ bude základem atomu typu Rel. Ten se chová jako typ Ord až na jiné mezerování.

Kn: 155, 291, 359–361, **Ol:** 149, 162, 188–189, 341, 344, 381, 425.

\mathstrut [plain]
Podpěra velikosti kulaté závorky. V desetibodovém písmu fontu CM má tato podpěra celkovou výšku 10 pt. Výška je 7,5 pt a hloubka 2,5 pt. \mathstrut lze použít ve všech módech, tedy nejen v matematickém módu. Srovnejte například \strut, který má celkovou výšku 12 pt.

```
474 \def\mathstrut{\vphantom(}
```

Kn: 131, 178, 360, **Ol:** 191, 193, 394$_{477–478}$.

\mathsurround (plain: 0 pt) [dimen]
Velikost netisknoucího výplňku před a za každou matematickou formulí vytvořenou ve vnitřním matematickém módu. Hodnota tohoto registru je rozhodující v okamžiku konverze z matematického do horizontálního seznamu, neboli v okamžiku dosažení závěrečného znaku $. Je tedy možné uvnitř skupiny $...$ nastavit tento registr lokálně jen pro sazbu jednoho konkrétního vzorečku v matematickém módu.

Kn: 97, 274, 305, 314, 323, 447, 162, 353, **Ol:** 167, 184$_{257}$, 189$_{337}$, 195$_{417}$, 196$_{420}$, 197$_{429}$, 198, 331$_{107}$, 419.

\matrix {⟨*řádky matice*⟩} [plain]
Vytvoří matici bez závorek kolem. Viz též \pmatrix. Ukázka je na straně 193.

```
475 \def\matrix#1{\null\,\vcenter{\normalbaselines\m@th
476   \ialign{\hfil$##$\hfil&&\quad\hfil$##$\hfil\crcr
477     \mathstrut\crcr\noalign{\kern-\baselineskip}
478     #1\crcr\mathstrut\crcr\noalign{\kern-\baselineskip}}}\,}
```

Makro se zhruba chová jako

```
479 \vcenter{\halign{\hfil$##$\hfil&&\quad\hfil$##$\hfil\crcr
480           ⟨řádky matice⟩ \crcr}}
```

První uživatelův řádek matice je kladen na prázdný řádek obsahující \mathstrut. To se chová jako podpěra v prvním řádku. Analogicky je řešen poslední řádek matice.

Kn: 176–178, 182, 325, 361, **Ol:** 137$_{220,222}$, 191, 193, 194$_{402}$, 419$_{632}$.

\maxdeadcycles (IniTEX: 25) [integer]
Maximální povolený počet opakování výstupní rutiny bez použití povelu \shipout.

Před vyvoláním výstupní rutiny a před zvětšením \deadcycles o jedničku TEX zkontroluje, zda náhodou \deadcycles ≥ \maxdeadcycles. Pokud ano, ohlásí TEX chybu.

Kn: 255, 273, 384, **Ol:** 262, 269_{378}, 351.

\maxdepth (plain: 4 pt) [dimen]
Maximální povolená hloubka stránkového boxu.

Po každém vložení boxu nebo linky do aktuální strany TEX kontroluje, zda hloubka tohoto elementu není větší než \maxdepth. Hloubka posledního elementu je uložena v \pagedepth. Je-li tedy \pagedepth > \maxdepth, TEX zmenší \pagedepth na hodnotu \maxdepth a o stejnou velikost zvětší \pagetotal, aby celková výška strany zůstala zachována.

Smysl tohoto algoritmu: Má-li poslední řádek na straně výrazně větší hloubku, je z estetického hlediska lepší porušit pravidlo požadující stejnou polohu účaří takového řádku. Je vhodnější posunout řádek poněkud výše, aby jeho spodní hrana zhruba odpovídala spodním hranám textu na ostatních stránkách.

Analogicky se TEX chová při kompletování každého \vboxu, viz registr \boxmaxdepth.

Kn: 123–125, 274, 400, 112–114, 255, 262–263, 348, 415, **Ol:** 101, 262_{303}, 413, 437.

\maxdimen [plain]
Hodnota maximální možné velikosti registru typu ⟨*dimen*⟩.

```
481 \newdimen\maxdimen \maxdimen=16383.99999pt
```

\meaning ⟨*token*⟩ [exp]
Vrací význam následujícího tokenu. Tento primitiv se dá použít na zjištění, jak je definováno makro. Makro je vypsáno ve formátu

```
482 macro:⟨maska parametrů⟩->⟨tělo definice⟩
```

Je-li zkoumaná řídicí sekvence primitiv, pak povel \meaning pracuje stejně jako \string. Je-li zkoumaná sekvence deklarovaná pomocí \chardef, \countdef apod., vypíše se číslo použitého registru. Má-li zkoumaný token v hlavním procesoru význam povelu pro sazbu znaku, primitiv expanduje na text „the letter *X*“. Příklad:

```
483 %                           %  expanduje na:
484 \meaning\leftline           %  macro:#1->\line␣{#1\hss␣}
485 \meaning\meaning            %  \meaning
486 \meaning\hsize              %  \hsize
487 \meaning\pageno             %  \count0
488 \meaning A                  %  the␣letter␣A
489 \expandafter\meaning\space %  blank␣space␣␣
490 \meaning\tenrm              %  select␣font␣csr10
```

```
\meaning\bla                    %  undefined
```

Výsledkem expanze je posloupnost tokenů, kde *všechny* tokeny mají kategorii 12, včetně speciálních znaků „\“, „#“ a včetně písmen. Výjimkou jsou jen mezery, které zůstávají u své kategorie 10. Mezery jsem v ukázce vyznačil vaničkami.

Výsledek expanze primitivu `\meaning` můžeme zpracovat dalším makrem a podle kvality testovaného tokenu se můžeme v makru větvit. V jednodušším případě používáme tento primitiv jen pro ladicí účely.

Kn: 213–215, 336, 382, **Ol:** 65, 72, 386_{435}, 431, 442.

`\medbreak` [plain]

Jako `\bigbreak`, ale výsledná mezera i penalta je poloviční.

```
\def\medbreak{\par\ifdim\lastskip<\medskipamount
  \removelastskip\penalty-100\medskip\fi}
```

Kn: 111, 113, 353, 355, 419, 422, **Ol:** 422_{660}.

`\medmuskip` (plain: 4 mu plus 2 mu minus 4 mu) [muglue]

Hodnota střední matematické mezery, např. mezi atomy typu Ord a Bin.

Kn: 167–168, 274, 349, 446, **Ol:** *158*, 145, 158_{61}, 160–161, $190_{363,365}$, 191, 335_{129}.

`\medskip`, `\medskipamount` [plain]

Makro `\medskip` vloží do seznamu mezeru velikosti poloviny výšky řádku. Viz též `\smallskip` a `\bigskip`.

```
\newskip\medskipamount
\medskipamount=6pt plus 2pt minus 2pt
\def\medskip{\vskip\medskipamount}
```

Kn: 79, 102, 109, 111, 352, 410–412, **Ol:** 57_{267}, 64_{1}, 114, $119_{35–36}$, 175_{153}, 176_{154}, 180_{211}, 277_{504}, $290_{32,44}$, 342, 396_{493}, 422_{663}, 435.

`\message` ⟨*filler*⟩`{`⟨*balanced text*⟩`}` [h, v, m]

Příslušný ⟨*balanced text*⟩ se po expanzi vypíše na terminál a do souboru `log`. Před výpisem se přejde na terminálu nebo v souboru `log` na nový řádek pouze tehdy, je-li délka zprávy delší, než zbytek místa na aktuálním řádku. Jinak je text vypsán na stejný řádek oddělen pouze jednou mezerou. Při `\message` nelze použít znaku `\newlinechar` k řízenému přechodu na nový řádek. Je to velmi užitečný primitiv pro ladění maker.

Kn: 279, 216, 227–228, 328, 217–218, 308, 343–344, 355, **Ol:** 18, 25, 35, 41, 49–50, 52–53, 59–60, 81, 95, 103, 137, 208, 242, 268, 277, 282, 286, 288, 292–293, 318, 340, 347–348, 356, 360–362, 386, 433, 457.

\midinsert ⟨*vertical material*⟩ \endinsert [plain]
Plovoucí objekt (insert). Má tendenci být umístěn v místě použití. Kód makra s výkladem, viz stranu 254.

Kn: 116, 340–341, 363, **Ol:** 247, 254_{244}, 255_{251}.

\mit [plain]
Přepínač na matematickou kurzívu (cmmi) v matematickém módu. Makro nastavuje registr \fam na jedničku, takže sazba matematických objektů třídy 7 bude realizovaná z rodiny fontů 1. O třídách matematických objektů viz stranu 146.

```
497 \textfont1=\teni \scriptfont1=\seveni
498                 \scriptscriptfont1=\fivei
499 \def\mit{\fam1 }
```

Kn: 164, 351, 430, 434, **Ol:** 191.

\mkern ⟨*mudimen*⟩ [m]
Jako \kern, ovšem měřeno v „math. units“, tj. v jednotce mu, jejíž velikost je závislá na stylu (D, T, S nebo SS). Jednotka mu, viz strany 78, 157, 325.

Kn: 280, 442, 168, **Ol:** 145, $156_{50,\,52}$, 157, $184_{243–248}$, $188_{296,\,304–306}$, $189_{327,\,334}$, $190_{363,\,365,\,367,\,380–383}$, 195_{418}, 196_{421}, 355_{277}, 381_{417}, 427_{690}, $433_{708–710}$.

\month [integer]
Číslo měsíce. Údaj je při startu načten TEXem ze systémové proměnné data a času.

Kn: 273, 349, 406, **Ol:** není nikdy použito.

\moveleft ⟨*dimen*⟩⟨*box*⟩ [v]
Box je vysázen o ⟨*dimen*⟩ vlevo, než by odpovídalo sazbě téhož boxu bez použití tohoto primitivu. Poloha levého okraje vnějšího \vboxu není touto operací změněna.

Kn: 282, 80–81, 287, **Ol:** 95, 100.

\moveright ⟨*dimen*⟩⟨*box*⟩ [v]
Box je vysázen o ⟨*dimen*⟩ vpravo, než by odpovídalo sazbě téhož boxu bez použití tohoto primitivu. Poloha levého okraje vnějšího \vboxu není touto operací změněna. Poloha pravého okraje vnějšího \vboxu je ovšem určena pravým okrajem takového boxu nebo linky, jež má ze všech elementů tento okraj nejvíce vpravo. Tato poloha tedy může být operací \moveright nebo \moveleft ovlivněna.

Kn: 282, 80–81, 221, **Ol:** 95, 100, 270_{396}.

\mskip ⟨*muglue*⟩ [m]
Jako \hskip, ovšem měřeno v „math. units“, tj. v jednotce mu, jejíž velikost je závislá na stylu (D, T, S nebo SS). Jednotka mu, viz strany 78, 157, 325.

Kn: 290, 168, 442, **Ol:** 145, 157, 158_{58}, $190_{363,\,365}$, 192_{394}, 324, $335_{128–131}$, 384.

\multiply ⟨*numeric variable*⟩⟨*optional by*⟩⟨*number*⟩ [a]
Registr ⟨*numeric variable*⟩ je vynásoben hodnotou ⟨*number*⟩ a výsledek je zpět uložen do registru ⟨*numeric variable*⟩.

Kn: 276, 349, 118–119, 218, 391, 398, **Ol:** *81*, 115, 244, 293, 317, 327, 339, 350.

\multispan *n* [plain]
Například \multispan3 je ekvivalent k \omit\span\omit\span\omit.

Obecně zápis \multispan *n* na začátku položky v \halign nebo \valign nahrazuje $n-1$ tabelátorů typu „&“ (pomocí \span) a ruší platnost *n* schémat položek v deklaraci řádku (pomocí \omit). Je-li *n* dvouciferné číslo, nesmíme zapomenout psát závorky, například \multispan{13}.

```
\newcount\mscount
\def\multispan#1{\omit \mscount=#1\relax
  \loop\ifnum\mscount>1 \sp@n\repeat}
\def\sp@n{\span\omit\advance\mscount by-1 }
```

Kn: 243, 246–247, 334, 354, **Ol:** 135, $139_{227–228,\,234,\,236}$, 140, $141_{246–248,\,254,\,256}$, 142_{264}.

\muskip ⟨*8-bit number*⟩ [m]
Povel umožní přístup k 256 registrům typu ⟨*muglue*⟩.

Kn: 271, 276, 118, 168, **Ol:** 73–74, 78_{51}, 325_{51}, 399_{511}, 400_{526}, $433_{707–710}$.

\muskipdef ⟨*control sequence*⟩⟨*equals*⟩⟨*8-bit number*⟩ [a]
Nová ⟨*control sequence*⟩ bude synonymem pro \muskip⟨*8-bit number*⟩.

Kn: 277, 119, 215, **Ol:** 66, 73, 327, 400_{526}, 403.

\narrower [plain]
Zúžení sazby. Přesněji: rozšíření okrajů po obou stranách sazby o velikost \parindent. Makro je možno použít například v prostředích pro citace. Je vhodné nejprve zahájit skupinu, pak použít makro \narrower, pak zpracovat vlastní text. Za textem nejprve uzavřít odstavec (pomocí \par) a teprve potom uzavřít skupinu. Makro je možno použít opakovaně i ve vnořených konstrukcích.

```
\def\narrower{\advance\leftskip by\parindent
  \advance\rightskip by\parindent}
```

Kn: 100, 340–341, 355, **Ol:** není nikdy použito.

\negthinspace [plain]
Malá záporná mezera typu \kern.

```
\def\negthinspace{\kern-.16667em }
```

Kn: 332, 352, **Ol:** není nikdy použito.

\newbox ⟨*control sequence*⟩ [plain]

Makro „deklaruje konstantu ⟨*control sequence*⟩ jako číslo registru typu ⟨*box*⟩“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat stejně jako s číslem registru typu ⟨*box*⟩.

Makro alokuje čísla 10 až 254 pomocí \chardef. Kód makra, viz \newcount.

Kn: 121, 346–347, 353, 394, 417, **Ol:** 74, 84_{114}, 126_{126}, 161, $269_{368–370}$, 271_{403}, 272_{417}, 276_{477}, 330_{92}, 384_{425}, 399, 400_{527}, 427_{684}, 439_{749}.

\newcount ⟨*control sequence*⟩ [plain]

Makro \newcount „deklaruje proměnnou ⟨*control sequence*⟩ typu ⟨*number*⟩“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat stejně jako s registrem typu ⟨*number*⟩.

Předchozí interpretace makra \newcount je při jeho používání dostačující. Většinou se totiž nemusíme starat, jaká konkrétní „adresa“ je námi požadované ⟨*control sequence*⟩ přidělena. Pokud ale chceme činnost makra pochopit, pak musíme vědět, že makro alokuje další z řady registrů \count23 až \count254. Vybraný registr se ztotožní s uvedenou ⟨*control sequence*⟩ pomocí

```
\countdef ⟨control sequence⟩=⟨nějaké číslo mezi 23 až 254⟩
```

Přitom ⟨*nějaké číslo mezi 23 až 254*⟩ je při prvním volání makra \newcount rovno číslu 23, při dalším volání je rovno číslu 24, pak 25 atd. Naposledy alokované číslo je uloženo v pracovním registru \count10.

Makro \newcount je do plainu implementováno společně s dalšími alokačními makry podobných vlastností \newdimen, \newskip, \newmuskip, \newbox, \newtoks, \newread, \newwrite, \newfam, \newlanguage a \newinsert. Proto uvedeme společný kód pro všechna makra a podrobně jej rozebereme.

```
\count10=22 % allocates \count registers 23, 24, ...
\count11=9 % allocates \dimen registers 10, 11, ...
\count12=9 % allocates \skip registers 10, 11, ...
\count13=9 % allocates \muskip registers 10, 11, ...
\count14=9 % allocates \box registers 10, 11, ...
\count15=9 % allocates \toks registers 10, 11, ...
\count16=-1 % allocates input streams 0, 1, ...
\count17=-1 % allocates output streams 0, 1, ...
\count18=3 % allocates math families 4, 5, ...
\count19=0 % allocates \language codes 1, 2, ...
\count20=255 % allocates insertions 254, 253, ...
\countdef\insc@unt=20 % the insertion counter
\countdef\allocationnumber=21 % the most recent allocation
```

Zde vidíme způsob použití pracovních registrů \count10 až \count21. Tyto registry vesměs ukládají číslo naposledy alokovaného registru příslušného typu.

Definice maker \newcount atd. mají příznak \outer, takže je možno tato makra použít jen samostatně a nikoli v těle definice dalšího makra. Programátor maker je tedy nucen nejprve deklarovat „své proměnné“ a teprve pak je může v makrech používat. Přitom deklarování proměnných se nesmí odehrát jako vedlejší efekt jeho vlastních maker.

```
\chardef\sixt@@n=16      % zkratka pro konstantu 16
\mathchardef\@cclvi=256  % zkratka pro konstantu 256
\outer\def\newcount{\alloc@0\count\countdef\insc@unt}
\outer\def\newdimen{\alloc@1\dimen\dimendef\insc@unt}
\outer\def\newskip{\alloc@2\skip\skipdef\insc@unt}
\outer\def\newmuskip{\alloc@3\muskip\muskipdef\@cclvi}
\outer\def\newbox{\alloc@4\box\chardef\insc@unt}
\outer\def\newtoks{\alloc@5\toks\toksdef\@cclvi}
\outer\def\newread{\alloc@6\read\chardef\sixt@@n}
\outer\def\newwrite{\alloc@7\write\chardef\sixt@@n}
\outer\def\newfam{\alloc@8\fam\chardef\sixt@@n}
\outer\def\newlanguage{\alloc@9\language\chardef\@cclvi}
```

Makra společně volají pomocné makro \alloc@, které má pět parametrů s těmito významy:

- #1 ... *druhá číslice odpovídajícího čísla pracovního registru* \count1?
- #2 ... *primitiv odpovídající danému typu alokace*
- #3 ... *primitiv, pomocí kterého se má provést požadované ztotožnění*
- #4 ... *maximální dovolené alokované číslo plus jedna*
- #5 ... *uživatelem deklarovaná* ⟨*control sequence*⟩

Pozastavíme se u parametru #4 (maximální dovolené alokované číslo). U některých maker se jedná o pevnou konstantu (16 nebo 256), zatímco u jiných se jedná o hodnotu registru \insc@unt. Jde o to, že makro \newinsert alokuje postupně třídy insertů odzadu. Tyto třídy nárokují registry typu \box, \count, \dimen a \skip. První \newinsert použije číslo 254, pak číslo 253 atd. Toto makro vždy zmenší hodnotu \insc@unt o jedničku. Přitom je potřeba, aby alokace samostatných registrů \box, \count, \dimen a \skip, která probíhá od menších čísel k větším, nepřekryla alokovanou třídu insert. Proto je zde maximální dovolené alokované číslo rovno proměnnému registru \insc@unt.

Uvedeme nyní definici pomocného makra \alloc@, které dělá hlavní práci.

```
\def\alloc@#1#2#3#4#5{\global\advance\count1#1by1
  \ch@ck#1#4#2% make sure there's still room
  \allocationnumber=\count1#1%
  \global#3#5=\allocationnumber
  \wlog{\string#5=\string#2\the\allocationnumber}}
\def\ch@ck#1#2#3{\ifnum\count1#1<#2%
  \else\errmessage{No room for a new #3}\fi}
```

Řádek 536 se například při použití makra `\newcount` expanduje na:

```
\global\countdef ⟨control sequence⟩=\allocationnumber
```

Konečně makro `\newinsert` je definováno samostatně:

```
\outer\def\newinsert#1{\global\advance\insc@unt by-1
  \ch@ck0\insc@unt\count
  \ch@ck1\insc@unt\dimen
  \ch@ck2\insc@unt\skip
  \ch@ck4\insc@unt\box
  \allocationnumber=\insc@unt
  \global\chardef#1=\allocationnumber
  \wlog{\string#1=\string\insert\the\allocationnumber}}
```

Kn: 121, 218, 346–347, 349, 418, **Ol:** 29$_{81}$, 37$_{45}$, 41$_{78}$, 52$_{205}$, 53$_{225}$, 57$_{261}$, 58$_{277}$, 61$_{310,\,315–316}$, 73–74, 79$_{64}$, 81$_{78}$, 82$_{88}$, 84$_{114}$, 106$_{187}$, 121$_{45}$, 123$_{84}$, 180$_{205}$, 207$_{475}$, 208$_{482}$, 236$_{118}$, 237$_{132}$, 244$_{155}$, 251$_{201}$, 271$_{400–401}$, 289$_{23}$, 291$_{55–56}$, 293$_{77}$, 350$_{251}$, 354$_{263}$, 375$_{396}$, 389$_{452}$, 398$_{500}$, 435$_{718}$, 451$_{801}$.

`\newdimen` ⟨*control sequence*⟩ [plain]

Makro „deklaruje proměnnou ⟨*control sequence*⟩ typu ⟨*dimen*⟩“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat stejně jako s registrem typu ⟨*dimen*⟩.

Makro alokuje registry `\dimen10` až `\dimen254` pomocí `\dimendef`. Kód makra, viz heslo `\newcount`.

Kn: 121, 346–347, 353, 394, 417, **Ol:** 40$_{71}$, 41, 74, 103$_{179}$, 107$_{190}$, 134$_{194}$, 203$_{451}$, 214$_{11–14}$, 236$_{114,\,119}$, 244$_{154}$, 269$_{371–373}$, 271$_{402}$, 272$_{417}$, 273$_{433}$, 276$_{478–480}$, 281$_{563}$, 328$_{75}$, 330$_{89–90}$, 381$_{418}$, 395$_{481}$, 399, 400$_{524}$, 407$_{578}$.

`\newfam` ⟨*control sequence*⟩ [plain]

Makro „deklaruje konstantu ⟨*control sequence*⟩ jako číslo nové rodiny fontu“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat stejně jako s číslem nové rodiny fontu.

Makro alokuje čísla 4 až 15 pomocí `\chardef`. Kód makra, viz `\newcount`.

Kn: 121, 157, 346–347, 351, **Ol:** 74, 170$_{98–99}$, 171$_{100–101}$, 177$_{168}$, 178$_{181–182}$, 340$_{160}$, 380$_{412}$, 399, 400$_{531}$, 434$_{715}$, 447$_{786}$.

`\newhelp` ⟨*control sequence*⟩ [plain]

Makro ukládá do `poolu` text nápovědy, která se dá vyvolat na terminálu pomocí uživatelova `H` při chybě. Protože není text uložen do hlavní paměti jako posloupnost tokenů, šetří se tím 3 až 7 B na jeden znak textu v TEXovské paměti.

Použití makra vypadá například takto:

```
\newhelp\helpA{Tady je text nápovědy pro chybu A}
...\if⟨něco se stane⟩ \errhelp=\helpA \errmessage{Chyba A}\fi
```

Makro je implementováno tak, že text se ukládá do paměti jako identifikátor řídicí sekvence.

```
\let\newtoks=\relax % to allow plain.tex to be read in twice
\outer\def\newhelp#1#2{\newtoks#1%
   #1=\expandafter{\csname#2\endcsname}}
\outer\def\newtoks{\alloc@5\toks\toksdef\@cclvi}
```

Je sice pravda, že toto řešení šetří pamětí, ale má jednu drobnou nevýhodu. Před takový text nápovědy je na terminálu připojen znak „\“, což odpovídá momentální hodnotě registru `\escapechar`. Volíme-li `\escapechar=-1`, pak sice není znak „\“ zobrazen, ale tento znak také chybí ve výpisu chyby, což je dosti podstatný nedostatek. Dá se to obejít pomocí použití primitivu `\string` v době, kdy je `\escapechar=-1`. Častěji se ale setkáváme s tím, že nápovědy nejsou vůbec v makru použity. Když už použity jsou, pak jsou uchovány klasickým způsobem jako posloupnost tokenů (například v LaTeXu). Můžeme tedy makro `\newhelp` chápat jako Knuthův experiment, který neměl velkou životnost.

Kn: 346–347, **Ol:** 74, 360.

`\newif` ⟨*control sequence*⟩ [plain]

Alokátor nové „logické proměnné“. Identifikátor ⟨*control sequence*⟩ musí začínat na písmena `if`. Na rozdíl od ostatních alokačních maker typu `\new..` se nealokuje číslo ani registr, ale definují se dvě makra. Při `\newif\ifxxx` se definuje makro `\xxxtrue` s významem `\let\ifxxx=\iftrue` a makro `\xxxfalse` s významem `\let\ifxxx=\iffalse`.

```
\outer\def\newif#1{\count255=\escapechar \escapechar=-1
  \expandafter\expandafter\expandafter
   \edef\@if#1{true}{\let\noexpand#1=\noexpand\iftrue}%
  \expandafter\expandafter\expandafter
   \edef\@if#1{false}{\let\noexpand#1=\noexpand\iffalse}%
  \@if#1{false}\escapechar=\count255 }
\def\@if#1#2{\csname\expandafter\if@\string#1#2\endcsname}
{\uccode‘1=‘i \uccode‘2=‘f \uppercase{\gdef\if@12{}}}
```

Po zapamatování původní hodnoty `\escapechar` a nastavení na -1 se provedou jednotlivé průchody `\expandafter`. Primitiv `\string` nám tedy nebude vracet token $\boxed{\backslash}_{12}$. Pro lepší přehled budeme v každém průchodu podtrhávat co se bude vykonávat a primitiv `\expandafter` budeme značit zkráceně jako `\ex`.

```
  \ex \ex \ex \edef \@if \ifxxx {true}
  \ex \edef \csname \ex \if@ \string \ifxxx true\endcsname
  \ex \edef \csname \if@ i12 f12 x12 x12 x12true\endcsname
```

```
566  \ex \edef \csname xxxtrue\endcsname
567  \edef \xxxtrue {\let\noexpand\ifxxx=\noexpand\iftrue}
```

Všimněme si, že `\if@` je definováno na řádku 562 pomocí `\gdef` se separátorem i$_{12}$ f$_{12}$. Tento separátor byl vyprodukován prostřednictvím `\uppercase`. Nelze přímo psát `\def\if@ if{}`, protože bychom obdrželi separátor i$_{11}$ f$_{11}$ a nikoli i$_{12}$ f$_{12}$.

Podobně pracuje i `\edef\xxxfalse`. Konečně se na řádku 560 nastaví výchozí hodnota `\ifxxx` na `\iffalse`, aby použití této sekvence nezlobilo při přeskakování ve vložených konstrukcích typu `\if...` před prvním použitím `\xxxtrue` nebo `\xxxfalse`.

Kn: 348, 211, 218, 354, 357, 416, 423, **Ol:** 51$_{198–199}$, 53$_{218}$, 60$_{304}$, 61$_{311–314}$, 74, 126$_{125}$, 162$_{72}$, 255$_{249}$, 276$_{481}$, 354$_{262}$, 418$_{621}$, 423$_{669}$.

`\newinsert` ⟨*control sequence*⟩ [plain]
Makro „deklaruje konstantu ⟨*control sequence*⟩ jako číslo nové třídy insertů“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat stejně jako s číslem třídy insertů.

Makro alokuje čísla 254 až 23 pomocí `\chardef`. Kód makra, viz `\newcount`.

Kn: 121–122, 346–347, 363, 415, **Ol:** 74, 251$_{202}$, 255$_{245}$, 281$_{559}$, 366$_{337}$, 399–400, 401$_{541}$, 443$_{775}$.

`\newlanguage` ⟨*control sequence*⟩ [plain]
Makro „deklaruje konstantu ⟨*control sequence*⟩ jako číslo nové tabulky vzorů dělení slov“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat stejně jako s číslem nové tabulky vzorů dělení slov.

Makro alokuje čísla 1 až 255 pomocí `\chardef`. Kód makra, viz `\newcount`.

Kn: 121, 157, 346–347, 351, **Ol:** 74, 399, 400$_{532}$.

`\newlinechar` (plain: −1) [integer]
Znak s tímto ASCII kódem se při `\write` konvertuje do znaku pro konec řádku. Je-li hodnota registru mimo $\langle 0, 255\rangle$ (viz nastavení plainu), žádný znak se nekonvertuje na konec řádku. LaTeX používá pro tyto účely znak `^^J`.

Kn: 228, 273, 348, **Ol:** 18$_{29}$, 396, 457.

`\newmuskip` ⟨*control sequence*⟩ [plain]
Makro „deklaruje proměnnou ⟨*control sequence*⟩ typu ⟨*muglue*⟩“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat stejně jako s registrem typu ⟨*muglue*⟩.

Makro alokuje registry `\muskip10` až `\muskip255` pomocí `\muskipdef`. Kód makra, viz heslo `\newcount`.

Kn: 121, 346–347, 353, 394, 417, **Ol:** 74, 399, 400$_{526}$.

\newread ⟨*control sequence*⟩ [plain]
Makro „deklaruje konstantu ⟨*control sequence*⟩ jako číslo nového souboru ke čtení“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat v souvislosti s primitivy \openin, \closein a \read stejně jako s číslem souboru ke čtení.

Makro alokuje čísla 0 až 15 pomocí \chardef. Kód makra, viz \newcount.

Kn: 121, 216, 346–347, **Ol:** 74, 76, 288_{10}, 399, 400_{529}.

\newskip ⟨*control sequence*⟩ [plain]
Makro „deklaruje proměnnou ⟨*control sequence*⟩ typu ⟨*glue*⟩“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat stejně jako s registrem typu ⟨*glue*⟩.

Makro alokuje registry \skip10 až \skip254 pomocí \skipdef. Kód makra, viz heslo \newcount.

Kn: 121, 346–347, 349, 394, 414, **Ol:** 74, $267_{330–331}$, 330_{91}, 342_{184}, 346_{218}, 372_{383}, 396_{494}, 399, 400_{525}, $407_{576–577}$, 435_{721}.

\newtoks ⟨*control sequence*⟩ [plain]
Makro „deklaruje proměnnou ⟨*control sequence*⟩ typu ⟨*tokens*⟩“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat stejně jako s registrem typu ⟨*tokens*⟩.

Makro alokuje registry \toks10 až \toks255 pomocí \toksdef. Kód makra, viz heslo \newcount.

Kn: 121, 212, 262, 346–347, 401, **Ol:** 54_{235}, 55_{248}, 57_{260}, $58_{273, 276}$, 61_{317}, 74, 134_{194}, $262_{292–293}$, 289_{22}, 290_{41}, 366_{341}, 371_{379}, 399, 400_{528}, $402_{551–552, 554}$.

\newwrite ⟨*control sequence*⟩ [plain]
Makro „deklaruje konstantu ⟨*control sequence*⟩ jako číslo nového souboru k zápisu“. V uživatelských makrech pak můžeme s ⟨*control sequence*⟩ pracovat v souvislosti s primitivy \openout, \closeout a \write stejně jako s číslem souboru k zápisu.

Makro alokuje čísla 0 až 15 pomocí \chardef. Kód makra, viz \newcount.

Kn: 121, 227, 346–347, 422–423, **Ol:** 57_{262}, 74, 76, 208_{483}, 289_{24}, 292_{69}, 293_{80}, 399, 400_{530}, 457.

\noalign {⟨*vertical (horizontal) material*⟩} [spec]
Specifikace vertikálního (horizontálního) materiálu, který se má vložit mezi řádky (sloupce) v \halign (\valign).

Kn: 237, 282, 285, 176, 191, 249, 286, 193, 246, **Ol:** *sekce 4.3*, 129_{160}, 130–131, 136, 138, $139_{232, 243}$, $141_{253, 263}$, 142, 143_{282}, $184_{230, 233, 236–237, 241–242}$, 188_{305}, 206_{470}, 268_{364}, $342_{189, 193}$, 349, 354_{265}, $394_{477–478}$.

\noboundary [h]

Potlačení sazby ligatur a implicitních kernů s hraničním znakem, který může být definován v některých fontech a je pak přítomen před a za každým slovem. Podrobněji o pojmu hraniční znak, viz stranu 305.

Kn: 286, 283, 290, **Ol:** 89.

\nobreak [plain]

Penalta, která zakáže řádkový nebo stránkový zlom.

```
568 \def\nobreak{\penalty10000 }
```

Kn: 97, 109, 174, 193, 335, 353, 394, 407, **Ol:** 33, 35, 57, 64, 110, 112–115, 175–176, 211–212, 245, 255, 259, 262, 277, 289–290, 340, 371, 454.

\noexpand ⟨*token*⟩ [exp]

Následující ⟨*token*⟩ je označen příznakem „neexpanduj“. Jakmile se expand procesor dostane k jakémukoli zpracování tokenu s takovým příznakem (pokus o expanzi nebo zavedení do parametru), odejme mu tento příznak, ale nebude jej expandovat. Pokud takový token projde až do hlavního procesoru, a přitom tam nemá žádný smysluplný význam (například nejedná se o znak nebo primitiv), chová se tam stejně jako \relax.

Obvykle je pořadí zpracování posloupnosti \noexpand ⟨*token*⟩ přímé, tj. hned po zpracování \noexpand přichází na řadu zpracovat ⟨*token*⟩. Výsledkem je tedy ⟨*token*⟩, který v této fázi expanze zůstává nezměněn. Situace se však stává komplikovanější, pokud se mění pořadí zpracování. Například:

```
569 \def\a {ABC} \expandafter \a \noexpand \bla
```

Zde se v prvním průchodu označí \bla příznakem „neexpanduj“, pak se expanduje \a na ABC a pak znovu přichází na řadu \bla. Ten má ovšem příznak „neexpanduj“, proto zůstane nezměněn a hlavní procesor se s ním vypořádá, jako by se jednalo o \relax. Nedojde tedy k žádné chybě i když \bla není dříve definováno. Na druhé straně

```
570 \def\a #1{ABC#1} \expandafter \a \noexpand \bla
```

způsobí chybu „`Undefined control sequence`“, protože v okamžiku, kdy expand procesor zavádí token \bla do parametru #1, odejme mu příznak „neexpanduj“, takže v místě použití parametru #1 TEX ohlásí chybu.

Kn: 209, 213, 215, 216, 348, 377, 424, **Ol:** 46$_{137,\,139}$, 48$_{165}$, 49, 50$_{180}$, 55$_{245,\,250}$, 57$_{268}$, 58$_{282–283}$, 59, 72, 107$_{194–197}$, 160$_{66}$, 251$_{218}$, 288–289, 290$_{33–34,\,37,\,46}$, 316$_{1}$, 356, 402$_{557,\,559}$, 403$_{567}$.

\noindent [h, v, m]

Vynucený start odstavce, přičemž odstavec nemá na začátku odsazení. Uvnitř odstavcového módu tento primitiv neudělá nic.

\nointerlineskip [plain]

Makro zařídí, aby následující box byl do sazby vložen zcela bez meziřádkové mezery nad boxem. Na druhé straně makro \offinterlineskip nastavuje nulovou meziřádkovou mezeru mezi všemi následujícími boxy.

```
571 \def\nointerlineskip{\prevdepth=-1000pt }
```


\nolimits [m]

Pokud je posledním elementem seznamu atom typu Op, nastaví se mu příznak „nolimits“, který ovlivní usazování indexů. Indexy i exponenty budou sázeny vpravo od základu a nikoli nahoru a dolu.

Implicitní příznak každého atomu typu Op je „displaylimits“, viz primitiv \displaylimits. Viz též \limits.

\nonfrenchspacing [plain]

Nastaví mezerování za tečkou, otazníkem, vykřičníkem, dvojtečkou, středníkem a čárkou tak, že tyto mezery jsou poněkud větší než ostatní mezery mezi slovy. Plain inicializuje toto nastavení, zatímco csplain toto nastavení ruší pomocí \frenchspacing, ovšem až po použití maker \chyph nebo \shyph.

```
572 \def\nonfrenchspacing{\sfcode`\.=3000 \sfcode`\?=3000
573       \sfcode`\!=3000 \sfcode`\:=2000 \sfcode`\;=1500
574       \sfcode`\,1250 }
```


\nonscript [m]

Následující mezera typu ⟨*glue*⟩ bude použita jen ve stylech D, D', T a T'. Nikoli však ve stylech S, S', SS, SS'. Povel je třeba použít bezprostředně před povelem, který vkládá mezeru typu ⟨*glue*⟩ do matematického seznamu.

\nonstopmode [a]

Nastaví se způsob zpracování, při němž TEX přeskakuje zcela všechny chyby. Jedná se jednak o „běžné“ chyby a jednak o chyby spočívající v nenalezení vstupního souboru. Není-li vstupní soubor nalezen, je povel \input při tomto způsobu zpracování ignorován. Viz též \errorstopmode, \batchmode a \scrollmode.

\nopagenumbers [plain]
Potlačí tisk stránkové číslice v patě strany.

```
\def\nopagenumbers{\footline={\hfil}}
```

Kn: 251–252, 362, 406, 409, **Ol:** 123$_{83}$, 262–263, 366.

\normalbaselines,
\normalbaselineskip, \normallineskip, \normallineskiplimit [plain]
Plain deklaruje vedle primitivů \baselineskip, \lineskip a \lineskiplimit alternativní registry, které ukládají hodnoty z těchto primitivů k trvalejšímu použití. Makro \normalbaselines pak může vrátit třeba pozměněné primitivní registry do původního stavu.

```
\newskip\normalbaselineskip \normalbaselineskip=12pt
\newskip\normallineskip \normallineskip=1pt
\newdimen\normallineskiplimit \normallineskiplimit=0pt
\def\normalbaselines{\lineskip=\normallineskip
                     \baselineskip=\normalbaselineskip
                     \lineskiplimit=\normallineskiplimit}
```

Kn: 325, 349, 351, 414–415, **Ol:** 346$_{216}$, 394$_{475}$.

\normalbottom [plain]
Vrátí do původního stavu nastavení z \raggedbottom. Kód makra, viz stranu 423.

Kn: 363, **Ol:** 423$_{671}$.

\null [plain]
Užitečná zkratka pro prázdný horizontální box.

```
\def\null{\hbox{}}
```

Kn: 311, 332, 351, **Ol:** 111$_{221}$, 112$_{223–224}$, 126$_{127–128}$, 127$_{143}$, 342$_{201}$, 359$_{293}$, 394$_{475}$, 418$_{629}$.

\nulldelimiterspace (plain: 1,2 pt) [dimen]
Šířka prázdného boxu, který se vkládá do horizontálního seznamu při konverzi matematického seznamu v místě prázdné závorky pružné velikosti. Jedná se o závorku konstruovanou jako \left⟨*delimiter*⟩ nebo \right⟨*delimiter*⟩, kde ⟨*delimiter*⟩ má \delcode rovno nule. Prázdné závorky TEX klade automaticky též kolem konstrukcí z \over, \atop a \above.

Kn: 442, 150, 274, 348, **Ol:** 165–166, 341$_{173}$, 352, 411.

\nullfont [font]
Jedná se o řídicí sekvenci typu ⟨*font*⟩, která je známá už před prvním použitím povelu \font (v iniTEXu). Reprezentuje prázdný font, který neobsahuje žádný znak. Zápis \fontname\nullfont expanduje na „nullfont“ a font má sedm nulových parametrů \fontdimen.

Kn: 14, 153, 271, 443, **Ol:** 322.

\number ⟨*number*⟩ [exp]
Expanduje na desítkovou reprezentaci čísla. Jednotlivé tokeny mají kategorii 12. Naprosto stejně pracuje \the⟨*number*⟩.

Kn: 213, 40–41, 214, 252, 406, 424, **Ol:** 262$_{297}$, 365$_{322}$, 442.

\o, \O [plain]
Makra vysázejí znaky ø a Ø z CM fontů.

```
583 \chardef\o="1C  \chardef\O="1F
```

Kn: 356, **Ol:** není nikdy použito.

\oalign {⟨*řádky tabulky*⟩} [plain]
Makro expanduje (zhruba řečeno) na:

```
584 \vtop{\halign{#\crcr ⟨řádky tabulky⟩\crcr}}
```

takže se vytvoří \vbox s řádky tabulky, přičemž tento box bude do horizontálního seznamu umístěn podle účaří prvního řádku.

Jednotlivé ⟨*řádky tabulky*⟩ budou sázeny při nulovém \lineskiplimit ve vzdálenosti 0,25 ex od sebe. Při \lineskiplimit=-\maxdimen (viz \ooalign) budou tyto řádky sázeny přes sebe na společné účaří. Proto je nulováno \baselineskip.

```
585 \def\oalign#1{\leavevmode\vtop{\baselineskip=0pt
586   \lineskip=.25ex \ialign{##\crcr#1\crcr}}}
```

Kn: 356, **Ol:** 330$_{105}$, 339$_{154}$, 350$_{255}$, 410$_{603}$.

\obeylines [plain]
Každý konec řádku ve vstupním souboru má význam konce odstavce. Vhodné pro psaní veršů.

Makro nastavuje znak konce řádku ^^M jako aktivní znak (kategorie 13). Dále prohlásí \let^^M=\par. Makro je nutno definovat uvnitř skupiny, kde už je znak ^^M aktivní (srovnej například s problémem makra \begitems na straně 26).

```
587 {\catcode`\^^M=\active   % these lines must end with %
588   \gdef\obeylines{\catcode`\^^M=\active \let^^M=\par}%
589   \global\let^^M=\par} % this is in case ^^M in a \write
```

V kódu plainu je komentář, že bylo v \obeylines vhodnější použít \let namísto \def^^M{\par}, protože uživatel může s výhodou psát například

```
590 {\let\par=\cr \obeylines \halign{...
```

Kn: 94, 249, 262, 342, 352, 380–382, 407, 419, **Ol:** 27$_{77}$, 28, 29$_{87}$, 122$_{58}$, 123$_{101}$.

\obeyspaces [plain]

Makro uvádí mezeru jako aktivní znak. Je-li mezera aktivní, expanduje shodně, jako makro \space. Díky tomu nebude v token procesoru ignorováno více mezer vedle sebe. Každá mezera napsaná na vstupu se nakonec (po expanzi) realizuje jako $\boxed{\sqcup}_{10}$, což vytvoří jednu mezeru.

```
591 \def\obeyspaces{\catcode`\ =\active}
592 {\obeyspaces\global\let =\space}
```

Kn: 254, 308, 342, 352, 380–381, 394, 421, **Ol:** 27_{77}, 28, $29_{80,82}$, 183.

\oe, \OE [plain]

Makra tisknou ligatury œ a Œ z CM fontů.

```
593 \chardef\oe="1B  \chardef\OE="1E
```

Kn: 52–53, 356, **Ol:** není nikdy použito.

\offinterlineskip [plain]

Makro způsobí, že všechny boxy budou ve vertikálním módu sázeny pod sebe s nulovou meziřádkovou mezerou. Vzdálenost účaří dvou za sebou jdoucích boxů bude tedy závislá na výšce kladeného boxu a hloubce předchozího boxu.

```
594 \def\offinterlineskip{\baselineskip=-1000pt
595   \lineskip=0pt \lineskiplimit=\maxdimen}
```

Kn: 245–247, 312, 352, 416, **Ol:** 87_{133}, 109_{207}, 121_{44}, 123_{83}, 134_{207}, 138, 139_{224}, 141_{244}, 264, 269_{377}, 406.

\ogonek [csplain]

Vysází polský akcent pod znakem. Například „\ogonek a“ vede na „ą“. Akcent je brán podle kódu $\mathcal{CS}$-fontů. Viz soubor extcode.tex, resp. il2code.tex. Tento akcent je odlišný od tzv. „cedilla“ (například \c a vede na „ą“).

```
596 \def\ogonek #1{\setbox0\hbox{#1}\ifdim\ht0=1ex\accent157 #1%
597    \else{\ooalign{\unhbox0\crcr\hss\char157}}\fi}
```

\oldstyle [plain]

Makro se chová jako přepínač do fontů rodiny 1 (cmmi*). V matematickém módu zabere nastavení registru \fam na hodnotu 1 a v ostatních módech pracuje přepínač \teni. Jméno makra je odvozeno od tvaru číslic 0123456789, jejichž kresby jsou umístěny ve fontu cmmi10. Přístup k těmto číslicím je tedy umožněn přepínačem \oldstyle což můžeme číst jako „číslice starodávného stylu“.

```
598 \textfont1=\teni \scriptfont1=\seveni
599                 \scriptscriptfont1=\fivei
600 \def\oldstyle{\fam1 \teni}
```

Kn: 351, **Ol:** není nikdy použito.

\omit [spec]

Ignorovat odpovídající vzor tabulky v \halign (\valign). Primitiv se musí vyskytnout (po expanzi) jako první nemezerový token položky.

Kn: 240, 243–244, 282, 246–247, **Ol:** *sekce 4.3*, 135$_{211}$, 136, 139$_{226}$, 140, 342$_{193–194}$, 343, 398$_{501,\,503}$.

\ooalign [plain]

Zápis \ooalign{⟨*řádky tabulky*⟩} expanduje (zhruba řečeno) na:

```
\vtop{\lineskiplimit=-\maxdimen \baselineskip=0pt
   \halign{#\crcr ⟨řádky tabulky⟩\crcr}}
```

takže budou ⟨*řádky tabulky*⟩ kladeny přes sebe na společné účaří. Makro se opírá o jiné makro \oalign, které jsme v tomto slovníku uvedli dříve.

```
\def\ooalign{\lineskiplimit=-\maxdimen \oalign}
```

Kn: 356, **Ol:** 189$_{327–328}$, 197$_{426,\,431}$, 345$_{210}$, 348$_{238}$, 408, 409$_{597}$.

\openin ⟨*4-bit number*⟩⟨*equals*⟩⟨*filename*⟩ [h, v, m]

Otevře soubor pro čtení. Se souborem ⟨*filename*⟩ bude dále možno pracovat v povelech \read a \closein. Soubor je v těchto povelech charakterizován svým číslem ⟨*4-bit number*⟩, kterému je zde přiřazeno konkrétní ⟨*filename*⟩. TEX může mít ke čtení otevřeno současně nejvýše 16 souborů. Čísla těchto souborů jsou nezávislá na číslech souborů pro zápis.

Kn: 216–217, 280, **Ol:** 76, 287, 288$_{11}$, 376, 404, 424, 431.

\openout ⟨*4-bit number*⟩⟨*equals*⟩⟨*filename*⟩ [h, v, m]

Otevře soubor pro zápis. Se souborem ⟨*filename*⟩ bude dále možno pracovat v povelech \write a \closeout. Soubor je v těchto povelech charakterizován číslem ⟨*4-bit number*⟩, kterému je zde přiřazeno konkrétní ⟨*filename*⟩. Jakmile je povel \openout úspěšně proveden, je (případně) původní obsah souboru vymazán. TEX může mít pro zápis otevřeno současně nejvýše 16 souborů. Čísla těchto souborů jsou nezávislá na číslech souborů pro čtení.

Povel \openout se implicitně neprovádí okamžitě, ale až při akci \shipout. Pokud chceme povel realizovat okamžitě, pišme před primitiv \openout prefix \immediate.

Kn: 280, 226–228, 254, 422, 423, **Ol:** 57$_{263}$, 76, 208$_{495}$, 289$_{25}$, 292$_{71}$, 293$_{84}$, 378, 404, 431, 457.

\openup ⟨*dimen*⟩ [plain]

Toto makro zvětší registry \lineskip, \baselineskip a \lineskiplimit o hodnotu ⟨*dimen*⟩. Použito pro zvětšení řádkování například v makrech \displaylines, \eqalignno a \leqalignno.

```
\def\openup{\afterassignment\@penup\dimen0=}
\def\@penup{\advance\lineskip by\dimen0
```

```
606    \advance\baselineskip by\dimen0
607    \advance\lineskiplimit by\dimen0 }
```

\or [exp]

Separátor údajů v konstrukci \ifcase.

Kn: 213, 210, 406, **Ol:** 47, 178$_{183–184}$, 292$_{76}$, 375$_{398}$, 391$_{467–468}$.

\outer [prefix]

Prefix pro \def, \edef, \gdef a \xdef. Způsobí, že volání takto definovaného makra nesmí být dále použito v těle jiné definice, v deklaraci řádku u \halign, \valign, mezi tokeny, které jsou přeskakovány v konstrukcích typu \if, a uvnitř textů aktuálních parametrů. Například:

```
608 \outer\def\zakázáno{...}
609 \def\a{...\zakázáno...}       % způsobí chybu
610 \def\b#1{...} \b{\zakázáno}   % způsobí chybu
611 \halign{\zakázáno #\cr a\cr}  % způsobí chybu
612 \iffalse \zakázáno \fi        % způsobí chybu
```

Kn: 206, 210, 275, 354, 357, 418–419, 422, **Ol:** 31, 40–41, 127$_{134–135}$, 179, 340$_{156}$, 345$_{208}$, 377, 400$_{523–532}$, 401$_{541}$, 402$_{552, 554–555}$, 422$_{660}$.

\output [tokens]

Primitivní registr typu ⟨*tokens*⟩ definující výstupní rutinu, která se vyvolá vždy z algoritmu uzavření strany. Plain nastavuje:

```
613 \output={\plainoutput}
```

IniTEX vychází z prázdného registru \output. Je-li tento registr prázdný, TEX použije vestavěnou výstupní rutinu, která je tvaru „\shipout\box255“.

Kn: 253, 125, 275, 370, 254–257, 364, 417, **Ol:** *sekce 6.8, 6.9*, 54, 71, 238, 256, 257$_{265}$, 261$_{291}$, 262$_{299}$, 266$_{323}$, 267$_{336}$, 269$_{379, 381}$, 272$_{419, 424}$, 273$_{437}$, 274$_{438}$, 276$_{469}$, 277$_{498}$, 282$_{578}$, 290$_{39}$, 292$_{72}$, 411, 419, 445.

\outputpenalty [integer]

Je-li stránkový zlom proveden v penaltě, je v registru \outputpenalty hodnota této penalty. Jinak tam je hodnota 10 000. S tímto registrem může pracovat výstupní rutina a například se větvit podle hodnoty tohoto registru.

Kn: 125, 273, 349, 400, 254–255, 417, **Ol:** 256, 261, 262$_{302}$, 263–264, 269$_{379}$, 272$_{415}$, 274$_{438}$, 282$_{578}$, 411.

\over [m]

Zlomek. Konstrukce {⟨*čitatel*⟩\over⟨*jmenovatel*⟩} vytvoří příslušný zlomek jako základ atomu typu Ord. Vlevo a vpravo od zlomku je vložen prázdný box velikosti \nulldelimiterspace.

Vymezující závorky {...} v konstrukci zlomku mohou být:

- Tokeny kategorie 1 a 2.

- Zástupné řídicí sekvence tokenů kat. 1 a 2 (\bgroup, \egroup).
- \left⟨*delimiter*⟩ a \right⟨*delimiter*⟩.
- Vnější hranice matematického módu (např. $1\over2$).

Příklady:

```
$ {a+b \over c} + d $           % Zlomek plus d
$ a+b \over c $, $1\over2$      % Zlomky vymezené $...$
$ {a+b \over {c \over a}} + d $ % (a+b / (c/a)) + d
$ \left(a \over b\right) + d $  % Vymezují \left, \right
$ {a \over b \over c} $         % Způsobí chybu
```

Konstrukce typu {1\over2\over3} není povolena. Ovšem je možno psát například {1\over{2\over3}}.

Viz též \above, \atop a alternativy \..withdelims.

Kn: 152, 292, 437, 444–445, 139–141, 148, **Ol:** *151*, 144$_{2,4}$, 152$_{36–38,41}$, 155$_{44–48}$, 166, 188$_{302}$, 189$_{333}$, 191$_{392}$, 193$_{398–399}$, 335, 344$_{205–206}$, 407, 412.

\overbrace [plain]

Svorka nad textem. Kód makra, viz stranu 184.

Kn: 176, 225, 359, **Ol:** 183$_{228}$, 184$_{235}$.

\overfullrule (plain: 5 pt, LaTeX: 0 pt) [dimen]

Šířka černých obdélníčků (slimáků) na konci nesprávně zalomených řádků při Overfull \hbox.

Kn: 274, 307, 348, **Ol:** 98, 100.

\overleftarrow [plain]

Šipka nad textem. Kód makra, viz stranu 184.

Kn: 359, **Ol:** 183$_{227}$, 184$_{232}$.

\overline ⟨*math field*⟩ [m]

Vytvoří nový atom třídy Over obsahující v základu ⟨*math field*⟩. Tento atom bude sázen s čarou nad základem. Čára bude mít tloušťku t = \fontdimen8 fontu rodiny 3 odpovídajícího stylu. Nad čarou je ještě mezera velikosti t a mezi čarou a základem atomu je mezera velikosti $3t$. Viz též \underline.

Kn: 141, 170, 291, 443, 130–131, **Ol:** 149, 167$_{86}$, 181, 448.

\overrightarrow [plain]

Šipka nad textem. Kód makra je uveden na straně 184.

Kn: 226, 359, **Ol:** 183$_{227}$, 184$_{229}$.

\overwithdelims ⟨*delim1*⟩⟨*delim2*⟩ [m]

Jako \over. Navíc jsou po stranách (místo prázdných boxů) závorky pružné velikosti. Závorka vlevo je určena pomocí ⟨*delim1*⟩ a vpravo pomocí ⟨*delim2*⟩.

Kn: 292, 444–445, 152, **Ol:** 152$_{41}$.

\P [plain]
Sazba znaku ¶, který se používá zvláště v příručkách ke komerčním sázecím programům. Znak se bere z pozice "7B fontu rodiny 2 (cmsy10).

```
619 \def\P{\mathhexbox27B}
```

Kn: 53, 117, 356, 438–439, **Ol:** není nikdy použito.

\pagebody, \pagecontents [plain]
Makra na usazení textu strany ve výstupní rutině. Kód makra \pagebody je uveden na straně 262 a kód makra \pagecontents na straně 252.

Kn: 255–257, 354, **Ol:** 243$_{147}$, 262$_{301,303}$, 263, 267$_{333}$.

\pagedepth *global, restricted* [dimen]
Hloubka boxu reprezentujícího tiskový materiál aktuální strany.

Je-li aktuální strana prázdná, je \pagedepth=0pt. Pokud do aktuální strany vstoupí box nebo linka, bude \pagedepth rovno hloubce tohoto boxu nebo linky. Pokud ale je hloubka tohoto boxu větší než \maxdepth, bude upravena jednak hodnota registru \pagetotal a jednak hodnota \pagedepth (viz heslo \maxdepth).

Pokud do strany vstoupí \kern nebo mezera typu ⟨*glue*⟩, je znovu nastavena hloubka strany \pagedepth=0pt.

Kn: 114, 123, 214, 271, **Ol:** 249, 253, 395, 415.

\pagefilllstretch *global, restricted* [dimen]
Součet hodnot roztažení řádu 3 vertikálních mezer na aktuální straně.

Je-li aktuální strana prázdná, je \pagefilllstretch=0pt. Jakmile do aktuální strany vstupuje vertikální mezera typu ⟨*glue*⟩ s nenulovou hodnotou roztažení řádu 3 (jednotka filll), je tato hodnota přičtena k registru \pagefilllstretch. Počet jednotek filll se do registru promítne jako počet jednotek pt. Přirozená velikost mezery se přičítá k \pagetotal.

Je-li do aktuální strany vkládán první insert třídy n a odpovídající mezera \skip$n$ má nenulovou hodnotu roztažení řádu 3, je tato hodnota rovněž přičtena k registru \pagefilllstretch. Přirozená velikost mezery \skip$n$ se v takovém případě nepřičítá k \pagetotal, ale odečítá se od \pagegoal. Obsahuje-li insert samotný nějakou vertikální mezeru s hodnotou roztažení, není tato hodnota zaznamenána do žádného z registrů pro stránkový zlom (registry začínající na slovo \page...).

Kn: 114, 214, 271, **Ol:** není nikdy použito.

\pagefillstretch *global, restricted* [dimen]
Součet hodnot roztažení řádu 2 vertikálních mezer na aktuální straně. Podrobněji, srovnejte s \pagefilllstretch.

Kn: 114, 214, 271, **Ol:** není nikdy použito.

\pagefilstretch *global, restricted* [dimen]
Součet hodnot roztažení řádu 1 vertikálních mezer na aktuální straně. Podrobněji, srovnejte s \pagefilllstretch.

Kn: 114, 214, 271, **Ol:** není nikdy použito.

\pagegoal *global, restricted* [dimen]
Tento registr určuje požadovanou výšku strany, na kterou je potřeba zaplnit aktuální stranu. Obecně je tento registr roven \vsize, ale při vstupu insertů se jeho hodnota může změnit.

Je-li strana prázdná, je \pagegoal=\maxdimen. Před vstupem prvního boxu nebo linky nebo insertu do aktuální strany se nastaví \pagegoal=\vsize. Při vstupu insertu do aktuální strany je \pagegoal zmenšen o celkovou výšku obsahu insertu násobenou jistým koeficientem f. Podrobněji, viz stranu 249, pravidlo 3 a 4. Viz též \pagetotal.

Kn: 114, 123, 214, 271, **Ol:** 240_{137}, 241, 242_{140}, $244_{169–170}$, 246, 248–250, 253–254, 255_{258}, 257–258, 270_{390}, 272, $281_{561–562}$, 283, 413, 415.

\pageinsert ⟨*vertical material*⟩ \endinsert [plain]
Plovoucí objekt. Má tendenci zaplnit celou příští stranu. Kód makra s vysvětlením, viz stranu 254.

Kn: 115, 363, **Ol:** 254_{243}, 255_{252}.

\pageno [plain]
Ekvivalent ke \count0. Číslo strany.

```
620  \countdef\pageno=0 \pageno=1 % first page is number 1
```

Kn: 252, 256, 340, 362, 406, **Ol:** 39, 56–57, 65, 73–74, 257, 259–260, 262–263, 266–267, 276, 279, 281, 289–290, 310, 321, 323, 326, 337, 365, 395.

\pageshrink *global, restricted* [dimen]
Součet hodnot stažení řádu 0 vertikálních mezer na aktuální straně.

Je-li aktuální strana prázdná, je \pageshrink=0pt. Jakmile do aktuální strany vstupuje vertikální mezera typu ⟨*glue*⟩ s nenulovou hodnotou stažení řádu 0, je tato hodnota přičtena k registru \pageshrink. Přirozená velikost mezery se přičítá k \pagetotal. Případné hodnoty roztažení se přičítají k registru \pagestretch nebo \pagefil(ll)stretch.

Má-li vertikální mezera typu ⟨*glue*⟩ vstupující do aktuální strany nenulovou hodnotu stažení řádu 1, 2 nebo 3, je tato hodnota ignorována a TEX ohlásí chybu. Algoritmus stránkového zlomu nemůže ve straně trpět žádnou mezeru s nekonečnou hodnotou stažení. To by nikdy nevznikla potřeba vytvořit ve vertikálním materiálu jediný stránkový zlom. Z analogických důvodů je zakázána horizontální mezera s nekonečným stažením v odstavcovém módu.

Je-li do aktuální strany vkládán první insert třídy n a odpovídající mezera \skip$n$ má nenulovou hodnotu stažení řádu 0, je tato hodnota rovněž přičtena k registru \pageshrink. Přirozená velikost mezery \skip$n$ se v takovém případě nepřičítá k \pagetotal, ale odečítá se od \pagegoal. Obsahuje-li insert samotný nějakou vertikální mezeru s hodnotou stažení, není tato hodnota zaznamenána do žádného z registrů pro stránkový zlom (registry začínající na slovo \page...).

Kn: 114, 123, 214, 271, **Ol:** 249, 255_{257}.

\pagestretch *global, restricted* [dimen]

Součet hodnot roztažení řádu 0 vertikálních mezer na aktuální straně. Podrobněji, srovnejte s \pagefilllstretch.

Kn: 114, 214, 271, **Ol:** 249, 414.

\pagetotal *global, restricted* [dimen]

Výška zaplněného textu na aktuální straně až po účaří posledního boxu (bez hodnot stažení a roztažení).

Je-li aktuální strana prázdná, je \pagetotal=0pt.

Vstoupí-li do aktuální strany box nebo linka, je registr \pagetotal zvětšen o výšku tohoto boxu nebo linky a o hloubku předchozího elementu z \pagedepth.

Vstoupí-li do aktuální strany \kern nebo mezera typu ⟨*glue*⟩, je registr \pagetotal zvětšen o přirozenou velikost této mezery a dále o hloubku předchozího elementu z \pagedepth.

Poznamenejme, že při vyvolání výstupní rutiny nemusí registry \pagetotal a \pagegoal odpovídat skutečné výšce materiálu v boxu 255. Stránkový zlom může být nakonec nalezen před naposledy vloženým boxem v aktuální straně. Přitom registry typu \page... se zpětně neupravují podle konečné polohy stránkového zlomu. Hodnotu \pagegoal odpovídající skutečnému stránkovému zlomu a skutečnou výšku strany je možno ve výstupní rutině změřit přímo z výšky boxu 255 a jeho materiálu uvnitř. Viz stranu 257.

Kn: 114, 123, 214, 271, **Ol:** 240–241, 242_{140}, 244_{170}, 245, 249, 253, 255_{257}, 257, $270_{390–391}$, 282_{587}, 395, 413–415, 463.

\par [h, v]

Ukončení odstavcového módu a formátování horizontálního seznamu do jednotlivých řádků odstavce.

Povel \par se ve čtecí frontě hlavního procesoru objeví z některého z následujících důvodů:

- Token procesor vytváří token par v místě prázdného řádku.
- Token par má implicitně význam povelu \par.
- Nějaké makro expanduje na \par nebo na zástupnou řídicí sekvenci.

- Token par je vložen při implicitním ukončení odstavce (strana 90).
- Povel \par se provede při vynuceném ukončení odstavce (strana 91).

Ve vertikálním módu a ve vnitřním horizontálním módu povel \par neprovede nic. V matematickém módu způsobí tento povel chybu Missing $ inserted a povel vstupuje po ukončení matematického módu na scénu znovu.

V odstavcovém módu TEX při povelu \par nejprve zjišťuje, zda je příslušný horizontální seznam prázdný. To se může stát jedině při zahájení odstavce pomocí \noindent a nevložení žádného elementu do seznamu. Například při dvojici \noindent\par. V takovém případě TEX přejde zpět do vertikálního módu a nevkládá do vertikálního seznamu žádný výsledek zlomu. Pouze pronuluje \hangindent a \looseness, nastaví \hangafter=1 a zruší případné \parshape.

Jinak (při neprázdném horizontálním seznamu) TEX vykoná následující činnosti:

- Provede \unskip, tj. vymaže případné poslední ⟨*glue*⟩ ze seznamu.
- Připojí do seznamu \penalty10000\hskip\parfillskip.
- Najde řádkový zlom podle sekce 6.4.
- Vrátí se k vertikálnímu módu, ve kterém byl před sestavením odstavce.
- Vloží do vertikálního seznamu výsledek řádkového zlomu.
- Je-li v hlavním vertikálním módu, vyvolá algoritmus plnění strany.
- Provede \hangindent=0pt, \hangafter=1, \looseness=0.
- Zruší případné nastavení \parshape.

Pokud je odstavec přerušen display módem, provede se jakési „neúplné \par“, které odpovídá prvním šesti činnostem (až po případné vyvolání algoritmu plnění strany). V této situaci TEX klade za předposlední řádek penaltu podle \displaywidowpenalty místo obvyklého \widowpenalty.

Kn: 47, 283, 286, 86–87, 100, 135, 202, 249, 262, 351, 340, 380–381, **Ol:** *sekce 3.4, [povely]*. Primitiv \par se vyskytuje na velkém množství stránek v této knize.

\parfillskip (plain: 0pt plus 1fil) [glue]
Mezera, která se vloží na konec horizontálního seznamu bezprostředně před vyvoláním algoritmu řádkového zlomu.

Kn: 286, 100, 188, 274, 307, 332, 315, 348, 394, 419, **Ol:** 121, 143_{268}, 198–199, 212, 227, 229, 233, $234_{106–110,\,112}$, 278_{513}, 320_{25}, 416, 432_{697}.

\parindent (plain: 20 pt) [dimen]
Šířka odstavcové zarážky. Při implicitním zahájení odstavcového módu nebo při \indent se do horizontálního seznamu vloží prázdný box šířky \parindent.

Kn: 86, 100, 101–102, 105, 274, 282, 286, 291, 262, 342, 348, 355, 394, 406, 415, **Ol:** 66_{7}, 89, 90_{149}, 92, 121_{46}, 123_{86}, 143_{265}, 202_{442}, 203_{444}, 229, 233, $234_{108,\,112}$, 235_{113}, 276_{484}, 369_{378}, 380_{415}, $398_{504–505}$.

\parshape ⟨*equals*⟩⟨n⟩⟨i_1⟩⟨l_1⟩...⟨i_n⟩⟨l_n⟩ [h, v]
Povel pro nastavení obecného tvaru odstavce. n: počet řádků, kterých se nastavení týká (počítáno od prvního), i_j: odsazení j-tého řádku od levého okraje (může být i záporná hodnota), l_j: délka j-tého řádku. Údaj n je ve tvaru ⟨*number*⟩ a ostatní údaje mají formát ⟨*dimen*⟩. Při \prevgraf $= m > 0$ je prvních m dvojic i_k, l_k ignorováno, jako první se použije dvojice i_{m+1}, l_{m+1}.

Má-li odstavec více než n řádků, pak zbylé řádky mají stejnou délku i odsazení jako n-tý. Povel \parshape má lokální platnost pouze pro jeden odstavec. Tímto povelem uvedeným před ukončením odstavce lze nastavit nejrůzněji kroucené tvary odstavců „obtékající“ například libovolný tvar obrázku.

Kn: 101–102, 214, 271, 277, 283, 349, 374, 315, **Ol:** *235*, 198–199, 227, 229, $236_{115,131}$, 237_{134}, 353, 355, 373, 385, 416, 422, 426.

\parskip (plain: 0 pt plus 1 pt) [glue]
Vertikální mezera, která se přidá do vertikálního seznamu vždy při zahájení odstavcového módu, tj. nad začátek odstavce. Do prázdného vertikálního seznamu se tato mezera nevkládá. Velikost této mezery před odstavcem závisí na stavu registru \parskip v okamžiku zahájení odstavce a nikoli v době jeho ukončení.

Bezprostředně po vložení mezery z \parskip TEX přeruší činnost hlavního procesoru a vyvolá algoritmus plnění strany. Po ukončení tohoto algoritmu teprve začíná hlavní procesor zpracovávat případné \everypar ze zahájení odstavcového módu a další povely odstavcového módu.

Kn: 79, 104–105, 262, 274, 282, 342, 348, 355, 406, 417, **Ol:** $74_{33,35}$, 113–114, 121_{46}, 123_{86}, 212, 238–239, 241–242, 266, 276_{488}, 340_{158}.

\patterns ⟨*filler*⟩{⟨*balanced text*⟩} [a]
Definuje se tabulka vzorů dělení s číslem, odpovídajícím momentální hodnotě parametru \language. Parametr ⟨*balanced text*⟩ se zapisuje podle speciálních pravidel, viz sekci 6.3. Povel \patterns je možno použít pouze v iniTEXu a je aditivní. Je tedy možné uvést více \patterns stejného čísla \language a data se sloučí do jedné tabulky.

Kn: 453, 277, 455, **Ol:** *223*, 218, 222, 224_{82}, 225–227, 230, 318, 336, 374, 382.

\pausing (iniTEX: 0) [integer]
Kladná hodnota způsobí, že se TEX bude zastavovat po přečtení každého řádku ve vstupním souboru.

Kn: 303, 273, **Ol:** není nikdy použito.

\penalty ⟨*number*⟩ [h, v, m]
Vloží do aktuálního seznamu penaltu dané hodnoty ⟨*number*⟩. Jedná se o bezrozměrné místo možného řádkového nebo stránkového zlomu. Hodnota penalty udává (ve srovnatelných jednotkách jako badness) vhodnost či nevhodnost zlomu v tomto místě.

Je-li ⟨*number*⟩ nulové, je místo zlomu v penaltě stejně vhodné jako zlom v mezeře. Záporné ⟨*number*⟩ označuje jistou „bonifikaci“. Algoritmus zlomu bude mít tendenci v tomto místě zlomit, i když se třeba dopustí srovnatelně velké badness souvisejících řádků nebo strany, jako absolutní hodnota ⟨*number*⟩. Kladné číslo ⟨*number*⟩ označuje „trest“ za zlom v tomto místě. Je srovnatelný s trestem, který je vyhodnocen za zlom v mezeře, při kterém vzniká stejně velká badness řádku nebo strany. Přesněji o tom, jak se promítne hodnota penalty do algoritmu řádkového nebo stránkového zlomu, viz sekci 6.4 a 6.6.

Je-li ⟨*number*⟩ $\geq 10\,000$, jedná se o penaltu, ve které není dovolen vůbec zlom. Takovou penaltu většinou použijeme před mezerou, ve které je tím zabráněn řádkový zlom. V mezeře totiž TEX smí zlomit jen tehdy, nepředchází-li odstranitelný element (viz sekci 6.1).

Je-li ⟨*number*⟩ $\leq -10\,000$, jedná se o penaltu, ve které je algoritmus zlomu přinucen zlomit materiál v každém případě (ať to stojí, co to stojí). Pomocí této penalty většinou ukončujeme stranu nebo (výjimečně) ukončujeme řádek v odstavci a přecházíme na nový řádek. V obou těchto případech je vhodné vložit před penaltu pružnou mezeru, aby TEX neměl problémy s badness řádku nebo strany.

Kn: 280, 79, 97, 110–111, 174, 353, **Ol:** *210, sekce 6.1*. Primitiv je použit na mnoha stránkách v této knize.

`\phantom {`⟨*sazba*⟩`}` [plain]

Makro vytvoří `\hbox`, který je prázdný, ale má stejné rozměry jako teoretický `\hbox{`⟨*sazba*⟩`}`. Parametr ⟨*sazba*⟩ se tedy ve výstupu neobjeví, ale jeho rozměry ovlivní velikost prázdného místa, které se rozprostírá v místě použití makra `\phantom`. Obsahuje-li ⟨*sazba*⟩ jen jeden token, není nutno psát závorky. Například `\phantom A` vytvoří prázdný box se stejnými rozměry jako `\hbox{A}`. Makro je možné použít v horizontálním i matematickém módu.

Kód makra je společný pro tři uživatelská makra:

- `\phantom`,
- `\vphantom` (jako `\phantom`, ale šířka výsledného boxu bude nulová),
- `\hphantom` (jako `\phantom`, ale výška s hloubkou boxu bude nulová).

```
\newif\ifv@ \newif\ifh@
\def\vphantom{\v@true\h@false\ph@nt}
\def\hphantom{\v@false\h@true\ph@nt}
\def\phantom{\v@true\h@true\ph@nt}
\def\ph@nt{\ifmmode\def\next{\mathpalette\mathph@nt}%
  \else\let\next\makeph@nt\fi\next}
\def\makeph@nt#1{\setbox0=\hbox{#1}\finph@nt}
\def\mathph@nt#1#2{\setbox0=\hbox{$\m@th #1{#2}$}\finph@nt}
\def\finph@nt{\setbox2=\null
  \ifv@ \ht2=\ht0 \dp2=\dp0 \fi
```

```
631   \ifh@ \wd2=\wd0 \fi \box2 }
```

Logická proměnná `\ifv@` oznamuje, že uživatel použil `\vphantom` nebo `\phantom`, tj. chce nenulovou výšku a hloubku výsledného boxu. Podobně logická proměnná `\ifh@` značí požadavek na nenulovou šířku výsledného boxu. Vlastní práci provádí makro `\ph@nt`, které se větví podle toho, zda je či není makro použito v matematickém módu.

Trasujme nejprve činnost makra mimo matematický mód (to bude jednodušší). Tam se provede `\makeph@nt`, což uloží do boxu 0 `\hbox{`⟨*sazba*⟩`}` a vyvolá finální rutinu `\finph@nt`. Zde se vytvoří prázdný box 2 a podle logických proměnných diskutovaných na začátku se tomuto prázdnému boxu nastaví nenulové rozměry podle boxu 0. Nakonec makro „položí“ do sazby výsledný prázdný box 2.

V matematickém módu se využije vlastnost makra `\mathpalette`, které v tomto případě vede na `\mathph@nt`⟨*style primitive*⟩`{`⟨*sazba*⟩`}`, kde parametr ⟨*style primitive*⟩ označuje odpovídající přepínač stylu. Makro `\mathph@nt` si vezme do parametru `#1` přepínač ⟨*style primitive*⟩ a v `#2` bude ⟨*sazba*⟩. Toto makro vytvoří box 0 se sazbou v matematickém módu. Přitom nejprve je nastaven styl podle ⟨*style primitive*⟩. Poznamenejme, že makro `\m@th` pouze nuluje `\mathsurround`, které může být ve vnějším prostředí nenulové. Nakonec se vyvolá rutina `\finph@nt`, kterou už jsme popsali výše.

Kn: 131, 178, 211, 360, 412, **Ol:** 191, 193, 435.

`\plainoutput` [plain]
Plain nastavuje `\output={\plainoutput}`, takže se jedná o výstupní rutinu plainu. Kód makra, viz stranu 262.

Kn: 255, 364, **Ol:** $262_{299–300}$, 263, 266_{323}, 267_{337}, 272_{410}, 273_{432}, 277_{498}, 290_{40}, 292_{72}, 411_{613}.

`\pmatrix` [plain]
Jako `\matrix`, ale navíc kolem přidá kulaté závorky pružné velikosti.

```
632   \def\pmatrix#1{\left(\matrix{#1}\right)}
```

Kn: 176, 323, 362, **Ol:** 137_{220}, 191, 193, 194_{403}, 394.

`\postdisplaypenalty` (iniTEX: 0) [integer]
Penalta vložená těsně za rovnicí z display módu.

Kn: 189–190, 272, **Ol:** 201, 202_{441}, 205, 451.

`\predisplaypenalty` (plain: 10 000) [integer]
Penalta vložená pod část odstavce před display módem.

Kn: 189–190, 272, 348, **Ol:** 201, 202_{440}, 205.

`\predisplaysize` [dimen]
Šířka posledního řádku odstavce před display módem zvětšená o 2 em. Tento

registr dává uvedenou kvantitu jen v display módu. Jinde je jeho hodnota nulová. Změna tohoto registru v display módu může ovlivnit konečné umístění rovnice do sazby, viz sekci 5.6.

Kn: 188, 190, 274, 349, **Ol:** *199*, 201, 203$_{452–454}$.

\preloaded [plain]

Toto slovo je použito při zavádění fontů, které nemají ve formátu plain svůj samostatný identifikátor typu ⟨*font*⟩. Jedná se o fonty zavedené v době iniTEXu do paměti „do zásoby“. Fonty tam čekají na případné použití.

Záměr autora plainu byl následující. V době iniTEXu zavedeme rozsáhlejší skupinu fontů do paměti TEXu. Ne všechny budeme bezpodmínečně potřebovat. Že si na to musíme chvíli počkat, to v případě iniTEXu nevadí. Pokud si při práci s takto vytvořeným formátem vzpomeneme, že budeme potřebovat třeba font `cmr9`, napíšeme prostě

```
633 \font\ninerm=cmr9
```

a TEX se nebude zdržovat novým zavedením fontu. Pouze přiřadí identifikátor `\ninerm` už dříve zavedenému fontu `cmr9`. Nebudeme tedy nuceni na tento úkon dlouho čekat. TEX totiž při zavádění nového fontu musí otevřít soubor s metrikou, načíst jeho obsah do paměti a přepočítat všechny rozměrové údaje na své jednotky.

Vidíme, že popsaný záměr preferuje rychlost při práci s vygenerovaným formátem před šetřením pamětí. Tyto dvě veličiny jsou, jak jistě víme, v neustálé opozici. V současné době jsou rychlosti počítačů natolik dostatečné, že se jeví jako výhodnější preferovat šetření pamětí před rychlostí. Alespoň v našem případě fontů. Proto v csplainu *nejsou* žádné fonty, uvedené níže jako `\preloaded`, zavedeny. Z uživatelského pohledu v tom není žádný rozdíl, protože uživatel stejně musí pro přístup k `\preloaded` fontům použít konstrukci podle řádku 633. V dnešní době si ani nevšimne, že tento povel v csplainu trošičku déle trvá. Navíc tím umožňujeme zavést do paměti vyhrazené pro fonty zcela jiné fonty, než CM (nebo $\mathcal{CS}$-fonty), protože paměť zůstává relativně volná.

Plain zavádí do své paměti „do zásoby“ tyto fonty:

```
634 \font\preloaded=cmr9       \font\preloaded=cmr8
635 \font\preloaded=cmr6       \font\preloaded=cmmi9
636 \font\preloaded=cmmi8      \font\preloaded=cmmi6
637 \font\preloaded=cmsy9      \font\preloaded=cmsy8
638 \font\preloaded=cmsy6      \font\preloaded=cmss10
639 \font\preloaded=cmssq8     \font\preloaded=cmssi10
640 \font\preloaded=cmssqi8    \font\preloaded=cmbx9
641 \font\preloaded=cmbx8      \font\preloaded=cmbx6
642 \font\preloaded=cmtt9      \font\preloaded=cmtt8
643 \font\preloaded=cmsltt10   \font\preloaded=cmsl9
644 \font\preloaded=cmsl8      \font\preloaded=cmti9
```

```
\font\preloaded=cmti8     \font\preloaded=cmti7
\font\preloaded=cmu10    % unslanted text italic
\font\preloaded=cmmib10  % bold math italic
\font\preloaded=cmbsy10  % bold math symbols
\font\preloaded=cmcsc10  % caps and small caps
\font\preloaded=cmssbx10 % sans serif bold extended
\font\preloaded=cmdunh10 % Dunhill style
\font\preloaded=cmr7 scaled \magstep4      % for titles
\font\preloaded=cmtt10 scaled \magstep2
\font\preloaded=cmssbx10 scaled \magstep2
\font\preloaded=manfnt   % METAFONT logo and dragon curve
% preloaded fonts must be declared anew later.
\let\preloaded=\undefined
```

Kn: 350, 413, **Ol:** 299.

`\pretolerance` (plain: 100) [integer]
Maximální povolená hodnota badness pro všechny řádky při prvním průchodu algoritmu řádkového zlomu.

Kn: 96, 107, 272, 317, 348, 364, 394, 451, **Ol:** *sekce 6.4*, 226$_{94}$, 227, 230, 232, 276$_{486}$, 432$_{699}$.

`\prevdepth` *global, restricted* [dimen]
Hloubka naposledy přidaného boxu do vertikálního seznamu. Tento parametr ovlivní vložení případné mezířádkové mezery před další box.

Je-li vertikální seznam prázdný, je `\prevdepth` nastaveno na magickou hodnotu −1000 pt. Tato hodnota znamená, že nebude před box vložena žádná mezířádková mezera z `\baselineskip` ani z `\lineskip`. Makro může na tuto hodnotu nastavit registr `\prevdepth` záměrně, aby potlačilo mezířádkovou mezeru (viz `\nointerlineskip`).

Vstupuje-li do vertikálního seznamu box, je provedena následující činnost:

- Podle `\prevdepth` a výšky boxu se vloží mezířádková mezera (sekce 3.7).
- Dále se do seznamu vloží samotný box.
- Nastaví se nové `\prevdepth`, které bude rovno hloubce vloženého boxu.

Vstupuje-li do vertikálního seznamu linka (`\hrule`), je automaticky nastaveno `\prevdepth=-1000pt`, takže následující box bude bez mezířádkové mezery.

Vstupuje-li do vertikálního seznamu jiný materiál (například `\kern`, mezera typu ⟨*glue*⟩), není hodnota `\prevdepth` měněna. Následující box tedy bude mít před sebou mezířádkovou mezeru podle hloubky předchozího boxu bez závislosti na tom, že jsou mezi nimi i jiné mezery.

Kn: 282, 79–80, 89, 271, 281, **Ol:** 110, 113$_{227–228}$, 339, 354$_{266}$, 406$_{571}$, 449, 450$_{796–797}$, 452, 454$_{807–808}$.

\prevgraf *global, restricted* [integer]

Počet řádků odstavce, které byly zahrnuty do vertikálního seznamu.

Při každém zahájení odstavcového módu je nastaveno `\prevgraf=0`. Po formátování odstavce (nebo jeho části před display módem) je tento registr zvětšen o výsledný počet řádků odstavce (nebo jeho části). Po ukončení display módu je registr zvětšen o tři, takže rovnice se počítá jakoby za tři řádky odstavce. Pokud neměníme v průběhu odstavcového (nebo display) módu hodnotu `\prevgraf`, máme po ukončení odstavcového módu (povelem `\par`) v `\prevgraf` údaj o počtu řádků naposledy formátovaného odstavce.

Chceme-li mít v registru `\prevgraf` součet všech řádků *všech* odstavců, musíme obejít skutečnost, že při zahájení každého odstavce se registr nuluje. Viz například ukázka na straně 237.

Změna registru `\prevgraf` v odstavcovém módu může ovlivnit konečné formátování odstavce podle `\parshape` a `\hangindent`. Při formátování každé části odstavce TEX totiž předpokládá, že už bylo dříve naformátováno n řádků (kde $n =$ `\prevgraf`) a údaje pro tyto řádky jsou ignorovány.

Kn: 103, 188, 190, 214, 271, **Ol:** 84_{119}, 202, $237_{134–136}$, 417.

\proclaim [plain]

Jednoduché makro na sazbu matematické věty v odborném článku. Použití makra může vypadat například takto:

```
\proclaim Věta 28. \TeX{} je otevřený systém.
```

Takový pokus dopadne následovně:

Věta 28. *TEX je otevřený systém.*

Vidíme, že záhlaví před první tečkou je sázeno polotučně a vlastní text věty (po první prázdný řádek, neboli konec odstavce) je sázen skloněným písmem.

```
\outer\def\proclaim #1. #2\par{\medbreak
  \noindent{\bf#1.\enspace}{\sl#2\par}%
  \ifdim\lastskip<\medskipamount \removelastskip
  \penalty55\medskip\fi}
```

Kn: 202–203, 206, 340–341, 355, **Ol:** není nikdy použito.

\promile [csplain]

Vysází znak (‰). Je použit kód podle $\mathcal{CS}$-fontů. Viz soubor `extcode.tex`, resp. `il2code.tex`.

```
\chardef\promile=141
```

\quad, \qquad [plain]

Mezera velikosti 1 em a 2 em.

```
\def\quad{\hskip1em\relax}
\def\qquad{\hskip2em\relax}
```

Kn: 94, 166-167, 185, 232–233, 352, **Ol:** 29, 119, 129, 133, 139, 141, 157–158, 163, 181, 183, 191, 268, 342, 346, 394, 452.

`\r` ⟨*znak*⟩ [csplain]

Makro se aktivuje po použití `\csaccents`. `\r`⟨*znak*⟩ vede na ⟨*znak s kroužkem*⟩. Pokud je ⟨*znak*⟩ roven znaku „u“ nebo „U“, provede se expanze na kód podle $\mathcal{CS}$-fontů. Jinak se použije `\accent23`. Viz soubor `extcode.tex`, resp. `il2code.tex`.

```
\def\r#1{\ifx u#1^^f9\else
         \ifx U#1^^d9\else {\accent23 #1}\fi\fi}
```

`\radical` ⟨*27-bit number*⟩⟨*math field*⟩ [m]

Sazba odmocniny. TEX vloží ⟨*math field*⟩ do základu atomu typu Rad a k tomuto atomu připojí informaci ve formě ⟨*24-bit number*⟩. Z původního ⟨*27-bit number*⟩ jsou první tři bity ignorovány. Údaj ⟨*24-bit number*⟩ je interpretován podobně, jako u `\delcode`.

Při konverzi matematického seznamu do horizontálního TEX v případě atomu typu Rad změří výšku sazby základu atomu a vlevo od této sazby připojí symbol odmocniny podle ⟨*24-bit number*⟩. Volí takovou velikost v řadě následníků, aby měl symbol odmocniny celkovou výšku větší než celková výška základu. Dále TEX připojí nad sazbu základu linku. Tloušťka této linky je určena výškou (ne hloubkou) symbolu odmocniny a linka na tento symbol přesně navazuje.

Kn: 157–159, 291, 443, **Ol:** *152, 166*, 152, 427, 437_{734}.

`\raggedbottom` [plain]

Makro zapíná režim, při kterém nepožadujeme, aby nutně poslední řádky textu na každé straně byly ve stejném místě. Pod posledním řádkem textu může být vynechána mezera. Jedná se o pravoúhlou analogii k `\raggedright`.

```
\newif\ifr@ggedbottom
\def\raggedbottom{\topskip=10pt plus60pt \r@ggedbottomtrue}
\def\normalbottom{\topskip=10pt \r@ggedbottomfalse}
```

Makro spolupracuje s výstupní rutinou plainu, která při `\r@ggedbottomtrue` vkládá pod box 255 mezeru `\vfil`. Viz definici `\pagecontents`, která je součástí výstupní rutiny plainu a je uvedena na straně 252. Makro `\raggedbottom` navíc vkládá do `\topskip` hodnotu roztažení. Předpokládá se totiž, že uživatel makra zruší ze všech vertikálních výplňků pružnost. Potom pružnost z `\topskip` dává při vyhledávání stránkového zlomu TEXu určitý manévrovací prostor. Ve výstupní rutině nebude tato pružnost pracovat, protože je materiál boxu 255 podepřen zespoda mezerou `\vfil`.

Makro `\normalbottom` vrací režim stránkového zlomu do původního stavu.

Kn: 111, 253, 363, 406, **Ol:** 114$_{232}$, 407.

\raggedright [plain]

Zapíná sazbu odstavce na pravý praporek.

```
672 \def\raggedright{\rightskip=0pt plus2em \spaceskip=.3333em
673         \xspaceskip=.5em \relax}
```

Pravá mezera každého řádku dostává pružnost `plus2em` a dále je mezislovní mezera nastavena pomocí `\spaceskip` na hodnotu bez pružnosti. Mezislovní mezery tedy nebudou podléhat pružným deformacím.

Proč není voleno `\rightskip=0pt plus 1fil`? Takové nastavení by způsobilo, že TEX nebude nikdy dělit slova. Algoritmus řádkového zlomu totiž najde řešení vždy v prvním průchodu. Jsou-li ale slova poměrně dost dlouhá (řádově 4 em), chtěli bychom, aby byla dělena. Při hodnotě roztažení 2 em pro `\rightskip` nutíme algoritmus řádkového zlomu, aby v opodstatněných situacích přešel do druhého průchodu a rozdělil dlouhá slova.

Kn: 29–30, 76, 101, 107, 115, 262, 356, 396, 407, **Ol:** 106, 234, 265, 266$_{324}$, 282$_{582}$, 423, 447, 452$_{804}$.

\raise ⟨*dimen*⟩⟨*box*⟩ [h]

Box se umístí o ⟨*dimen*⟩ výše, než by odpovídalo sazbě boxu bez použití tohoto primitivu.

Kn: 285, 290, 80, 151, 179, 66–67, 193, 408, **Ol:** 95–96, 112$_{225}$, 189$_{328}$, 190$_{381–383}$, 278$_{510}$, 289$_{17}$, 335$_{136}$, 348$_{238}$, 427$_{690}$.

\read ⟨*number*⟩to⟨*control sequence*⟩ [h, v, m]

je ekvivalentem k `\def`⟨*control sequence*⟩`{`⟨*balanced text*⟩`}`, přičemž tělo definice ⟨*balanced text*⟩ je přečteno a token procesorem zpracováno ze vstupního souboru určeného číslem ⟨*number*⟩.

Číslo ⟨*number*⟩ by mělo být přiřazeno k fyzickému souboru systému pomocí povelu `\openin`. Říkáme, že soubor je otevřen ke čtení. Není-li ⟨*number*⟩ přiřazeno, probíhá čtení z terminálu.

Při každém jednotlivém `\read` proběhne obvykle načtení ⟨*balanced text*⟩ z jediného řádku souboru. Výjimky existují pouze tehdy, není-li na řádku text balancovaný, tj. neodpovídají-li si explicitní závorky typu „`{`“ a „`}`“. Pokud se na čteném řádku vyskytne koncová závorka, která nemá odpovídající otevírací, pak je přečtena jen část řádku po tuto koncovou závorku mimo ni. Zbytek řádku je ignorován. Pokud po dosažení konce řádku není stále text balancovaný, pokračuje čtení dalším řádkem (dalšími řádky) až do doby, než se podaří načíst všechny odpovídající zavírací závorky.

Kn: 217–218, 215, 276, 346, 401, **Ol:** *288*, 66, 71, 76, 284, 286, 302, 317, 376, 400$_{529}$, 404, 410.

\relax [h, v, m]

Nic nedělej, relaxuj. Povel je vhodný pro ukončení neúplného parametru předchozího povelu.

Kn: 279, 23, 25, 71, 240, 276, 307, 353, **Ol:** 41, 45, 48–49, 52–53, 61, 63, 70, 74, 82, 107, 114, 123, 126–127, 137, 157–158, 174, 196, 203, 207–208, 226, 276, 278, 281, 288–290, 292, 313, 319, 321–322, 338–339, 350, 355, 359, 371, 375, 378, 387, 391, 398, 402, 423–424, 435, 439, 447.

\relbar, \Relbar [plain]

Pomocné makro k sazbě elementu, který prodlužuje šipky \longrightarrow a \Longrightarrow.

```
\def\relbar{\mathrel{\smash-}}
                % \smash, because - has the same height as +
\def\Relbar{\mathrel=}
```

Kn: 358, **Ol:** 188_{311}, $189_{313,\,335}$.

\relpenalty (plain: 500) [integer]

Penalta vložená za každý atom typu Rel. V této penaltě může být zlomena formule. Viz též \binoppenalty.

Existují výjimky, kdy penalta podle registrů \relpenalty a \binoppenalty není vložena do seznamu. Jedná se o tyto případy:

- V display módu.
- Za atomem typu Rel, pokud následuje další atom typu Rel.
- Následuje uživatelem stanovená penalta.
- Příslušný atom typu Rel nebo Bin je posledním atomem seznamu.

Připomeňme, že v horizontálním seznamu, který byl vytvořen konverzí z matematického, není dovolen zlom v mezerách. Zlom je povolen jedině v penaltách. Registry \relpenalty a \binoppenalty a dále explicitně vložené penalty dávají TEXu jedinou příležitost, kde zlomit matematickou formuli.

Kn: 446, 101, 174, 272, 322, 348, **Ol:** 160_{64}, 210, 342.

\removelastskip [plain]

Makro kompenzuje naposledy vloženou mezeru typu ⟨*glue*⟩ ve vertikálním seznamu, takže se to chová, jako by tam byla nulová mezera. Na rozdíl od použití primitivu \unskip mezera jako potenciální místo zlomu v seznamu zůstává.

```
\def\removelastskip{\ifdim\lastskip=0pt
                                \else\vskip-\lastskip\fi}
```

Makro je vhodné použít v hlavním vertikálním módu, kde nemusí vždy fungovat \unskip.

Kn: 353, **Ol:** 342_{183}, 396_{493}, 422_{662}, 435_{720}, 449.

\restorefont [csplain]
Vrátí význam primitivu \font do původního stavu. Význam tohoto primitivu je při generování formátu csplain předefinován tak, aby načtením originálního plain.tex byly místo CM fontů zavedeny $\mathcal{CS}$-fonty. Viz soubor csfonts.tex.

\right ⟨*delimiter*⟩ [m]
Následující ⟨*delimiter*⟩ se použije jako zavírací závorka pružné velikosti. Jednotlivé dvojice \left⟨*delimiter1*⟩...\right⟨*delimiter2*⟩ musí párovat. Tím se myslí, že ke každému \left odpovídá právě jedno \right. Tyto konstrukce navíc otevírají a zavírají skupinu a nesmí se křížit s jiným způsobem otevření a zavření skupiny.

TEX změří celkovou výšku sazby mezi \left a \right a obklopí ji závorkami odpovídající velikosti podle údajů ⟨*delimiter1*⟩ a ⟨*delimiter2*⟩. Způsob sazby těchto závorek je přesně vyložen na straně 164.

Kn: 292, 155–157, 196, 437, 148–150, 171, **Ol:** *150, 164*, 135, 144–146, 150–153, 158, 165, 169, 178, 185–186, 192–194, 197, 319, 341–343, 345–346, 351–352, 384, 407, 412, 419, 464.

\rightarrowfill [plain]
Šipka natahovací délky. Kód makra, viz stranu 184.

Kn: 226, 357, **Ol:** $184_{230,\,243}$.

\righthyphenmin (plain: 3) [integer]
Minimální počet znaků v části slova za rozdělením. Implicitně angličtina i čeština: 3. Vzory dělení pro češtinu jsou v csplainu snesou i změnu \righthyphenmin na hodnotu 2, tj. je mož-né tako-vé děle-ní slov. To se využije v sazbě do úzkých sloupců. Obecně to ale není v souladu s typografickými doporučeními pro českou sazbu. Viz též \lefthyphenmin, \setlanguage.

Kn: 454, 273, 455, 364, **Ol:** 218, 220_{54}, 221, $223_{79–80}$, 225–226, 276_{483}, 347_{222}, 356_{281}, 384, 430, 433_{702}.

\rightline {⟨*text*⟩} [plain]
⟨*text*⟩ bude umístěn do samostatného řádku na pravou zarážku. Jeho pravý okraj bude tedy lícovat s pravým okrajem řádku. Levý okraj může klidně i přečnívat doleva přes zrcadlo sazby.

```
679 \def\rightline#1{\line{\hss#1}}
```

Kn: 101, 317, 340–341, 353, **Ol:** není nikdy použito.

\rightskip (iniTEX: 0 pt) [glue]
Mezera, která je umístěna na pravé straně každého řádku při sestavování odstavce. Tato mezera zůstává uvnitř boxu s řádkem, který má celý šířku \hsize (nebo je jeho šířka určena pomocí \hangindent nebo \parshape). Viz též \leftskip.

Kn: 100–101, 274, 393, 317, 356, 421, **Ol:** *234*, 143$_{267}$, 211, 226$_{93}$, 227, 229, 234$_{111}$, 251$_{215}$, 265, 278$_{512}$, 385, 398$_{505}$, 424$_{672}$, 447$_{788}$.

`\rlap {`⟨*text*⟩`}` [plain]

Do seznamu se vloží prázdný box nulové šířky, z něhož ⟨*text*⟩ vyčnívá směrem doprava. Levý okraj ⟨*textu*⟩ se kryje s polohou boxu. Viz též `\llap`.

```
\def\rlap#1{\hbox to0pt{#1\hss}}
```

Kn: 82–83, 189, 247, 319, 353, 389, 416, **Ol:** 201, 335$_{136}$, 385$_{432}$, 387, 452$_{803}$.

`\rm` [plain]

Přepínač fontu do základního antikvového řezu. V matematickém módu se uplatní nastavení registru `\fam` na hodnotu 0 a mimo matematický mód se uplatní přepínač `\tenrm` deklarovaný přímo primitivem `\font`.

```
\textfont0=\tenrm \scriptfont0=\sevenrm
                  \scriptscriptfont0=\fiverm
\def\rm{\fam0 \tenrm}
```

Viz též `\it`, `\bf`, `\tt` a `\sl`.

Kn: 13–15, 154, 163, 320, 351, 364, 409, 414–415, 419, 427, **Ol:** 144$_{2}$, 147$_{13}$, 150$_{25}$, 172$_{103}$, 173, 174$_{112,\,119}$, 175$_{133}$, 176$_{156,\,159,\,161}$, 189$_{338–345}$, 190$_{346–362,\,364,\,367–374}$, 191$_{384–385,\,391–392}$, 198$_{435}$, 205$_{466–467}$, 257$_{266}$, 276$_{470–471}$, 278$_{510}$, 289$_{16}$, 336$_{137}$.

`\romannumeral` ⟨*number*⟩ [exp]

Expanduje na zápis čísla ⟨*number*⟩ v římské soustavě. Jednotlivé „číslice“ jsou písmena malé abecedy. Např. `\romannumeral 1993` expanduje na `mcmxciii`.

Jednotlivé tokeny mají po expanzi vždy kategorii 12 (stejně jako po `\the`). Je-li ⟨*number*⟩ nekladné, pak je výsledkem expanze prázdná posloupnost tokenů.

Kn: 40–41, 213, 214, 252, **Ol:** 43$_{96–97,\,99}$, 44$_{100}$, 262$_{296}$, 365$_{321}$, 442.

`\root` ⟨*mark*⟩`\of`⟨*math field*⟩ [plain]

Makro na sazbu n-té odmocniny. Na primitivní úrovni lze pouze sázet odmocninu libovolného výrazu, ovšem bez čísla odmocniny (viz `\radical`).

Použití makra: například `$\root n+1\of{2x}$` vede na $\root n+1\of{2x}$.

```
\newbox\rootbox
\def\root#1\of{\setbox\rootbox=
   \hbox{$\m@th\scriptscriptstyle{#1}$}
  \mathpalette\r@@t}
\def\r@@t#1#2{\setbox0=\hbox{$\m@th#1\sqrt{#2}$}
  \dimen0=\ht0 \advance\dimen0 by-\dp0
  \mkern5mu \raise.6\dimen0\copy\rootbox \mkern-10mu \box0 }
```

Makro si nejprve schová do boxu `\rootbox` sazbu čísla odmocniny ⟨*mark*⟩ ve stylu *SS*. Použití makra `\mathpalette\r@@t`⟨*math field*⟩ vede podle aktuálního matematického stylu na `\r@@t`⟨*style primitive*⟩⟨*math field*⟩. Jde tedy o to definovat `\r@@t` se dvěma parametry.

Makro `\r@@t` schová do boxu 0 sazbu odmocniny z ⟨*math field*⟩, přičemž pomocí ⟨*style primitive*⟩ byl nastaven odpovídající styl. Do registru `\dimen0` pak uložíme výšku odmocniny zmenšenou o její hloubku. Praxe ukazuje, že pronásobíme-li tento údaj koeficientem 0,6, dostaneme vhodnou vertikální polohu pro ⟨*mark*⟩. Proto nejprve umístíme značku odmocniny do výšky 0,6 × `\dimen0` a potom následuje vlastní odmocnina jako `\box0`.

Proč je `\rootbox` kladen do sazby pomocí `\copy` a ne pomocí `\box`? Je třeba si uvědomit, že makro `\r@@t` bude pro každou odmocninu pracovat čtyřikrát. Postupně pro styly *D*, *T*, *S* a *SS*. To je důsledek použití makra `\mathpalette`, které se opírá o primitiv `\mathchoice`.

Kn: 130–131, 179, 325, 360, **Ol:** 166, 437.

`\S` [plain]

Vysází znak § podle kódu CM fontů, což ocení hlavně právníci.

```
691  \def\S{\mathhexbox278}
```

Kn: 53, 117, 356, 438–439, **Ol:** 292_{76}.

`\scriptfont` ⟨*4-bit number*⟩ [font]

Registr označuje ⟨*font*⟩ rodiny ⟨*4-bit number*⟩ pro matematický styl *S*. Například `\scriptfont1` je v plainu font `cmmi7`. Jedná se o font rodiny 1 pro sazbu matematickou kurzívou v první indexové velikosti. Nastavení všech registrů `\scriptfont` v plainu, viz tabulku na straně 169 a 170. Viz též `\textfont` a `\scriptscriptfont`.

Kn: 153, 168, 213, 271, 321, 351, 441–442, 414–415, **Ol:** *sekce 5.3*, 167, 168_{93}, 169, 170_{97}, $174_{113–117,\,130}$, $177_{174,\,176}$, 180_{208}, 181_{213}, 195_{415}, 340_{161}, 345_{212}, 397_{497}, 409_{598}, 427_{681}, 428, 440.

`\scriptscriptfont` ⟨*4-bit number*⟩ [font]

Registr označuje ⟨*font*⟩ rodiny ⟨*4-bit number*⟩ pro matematický styl *SS*. Například `\scriptscriptfont1` je v plainu font `cmmi5`. Jedná se o font rodiny 1 pro sazbu matematickou kurzívou v druhé indexové velikosti. Nastavení všech registrů `\scriptscriptfont` v plainu, viz tabulku na straně 169 a 170. Viz též `\textfont` a `\scriptfont`.

Kn: 153, 168, 213, 271, 351, 441–442, 414–415, **Ol:** *sekce 5.3*, 167, 168_{94}, 169, 170_{97}, 171, $174_{113–116,\,118}$, 175_{131}, $177_{175,\,177}$, 180_{209}, 181_{214}, 195_{415}, 322, 340_{162}, 345_{213}, 364, 397_{498}, 409_{599}, 427_{682}, 428, 440.

\scriptscriptstyle [m]
Nastaví styl SS pro sestavování matematického seznamu. Jedná se o index druhé úrovně. Viz též \displaystyle, \textstyle a \scriptstyle.

Kn: 292, 141–142, 179, **Ol:** *154*, *sekce 5.2*, 155, 169_{95}, 189_{332}, 354, 393_{473}, 427_{686}, 429, 441.

\scriptspace (plain: 0,5 pt) [dimen]
Přidaná mezera za exponentem anebo indexem.

Má-li v matematické sazbě nějaký atom neprázdný index nebo exponent sázený vpravo od základu, je vhodné za takovým atomem vložit nepatrnou mezeru. Například \$ab\$ je sázeno bez mezery jako ab, zatímco $a_i b$ má mezi indexem i a písmenem b mezeru velikosti \scriptspace. Tato mezera je stejná bez závislosti na matematickém stylu.

Kn: 274, 348, 445–446, **Ol:** 162.

\scriptstyle [m]
Nastaví styl S pro sestavování matematického seznamu. Jedná se o index první úrovně. Viz též \displaystyle, \textstyle a \scriptscriptstyle.

Kn: 292, 141–142, 145, 179, **Ol:** *154*, *sekce 5.2*, 84_{123}, 145, 155_{48}, 188_{303}, 189_{331}, 354, 393_{472}, 429, 441.

\scrollmode [a]
Nastaví se způsob zpracování, při němž TEX přeskakuje chyby a hlášení o chybách na obrazovce roluje. Na rozdíl od \nonstopmode při nenalezení souboru v systému se zastaví a ptá na jiný soubor pro nahrazení. Viz též \errorstopmode.

Kn: 32, 277, **Ol:** 360–361, 406.

\setbox ⟨*8-bit number*⟩⟨*equals*⟩⟨*box*⟩ [a]
Přiřadí ⟨*box*⟩ do registru typu ⟨*box*⟩ s číslem ⟨*8-bit number*⟩. Takový box se dá později použít v sazbě pomocí \box, \copy, \unhbox, \unhcopy, \unvbox, \unvcopy, případně číst a měnit jeho vnější rozměrové parametry pomocí \ht, \dp, \wd.

Je-li v místě argumentu ⟨*box*⟩ použit primitiv \hbox, \vbox nebo \vtop následovaný případným ⟨*box specification*⟩ a závorkami pro vymezení skupiny, TEX postupně provádí tyto činnosti:

- Uloží do zásobníku informaci \setbox⟨*8-bit number*⟩.
- Uloží do zásobníku případné ⟨*box specification*⟩.
- Otevře skupinu a v ní vytvoří horizontální resp. vertikální seznam.
- Kompletuje box podle ⟨*box specification*⟩.
- Výsledný box uloží do registru ⟨*8-bit number*⟩.

Je-li v místě ⟨*box*⟩ použit primitiv \copy nebo \box, TEX má usnadněnu práci. Při \box⟨*jiný registr*⟩ pouze ztotožní ukazatel registru ⟨*8-bit number*⟩

s ukazatelem ⟨*jiný registr*⟩ a ukazatel ⟨*jiného registru*⟩ vynuluje. V případě \copy⟨*jiný registr*⟩ TEX navíc musí duplikovat ve své paměti data boxu a ukazatel registru ⟨*8-bit number*⟩ nasměruje na nové místo s daty boxu v paměti.

Kn: 120, 66–67, 77, 81, 276, 386–392, **Ol:** *82*, 83–84, 95, 103, 107, 115, 126–128, 143, 161–162, 184, 214, 226, 244–245, 255, 257, 265, 267–274, 277, 335, 338, 342, 345, 381–382, 409, 418, 427, 432, 435, 439, 448, 450, 452, 455.

\setlanguage ⟨*number*⟩ [h]

Do aktuálního seznamu se vloží stejnojmenná značka \setlanguage obsahující číslo ⟨*number*⟩ a stávající hodnoty registrů \lefthyphenmin a \righthyphenmin. Číslo ⟨*number*⟩ označuje číslo použitého vzoru dělení slov (\language) tabulek \patterns a \hyphenation.

V okamžiku, kdy TEX vyhledává v už hotovém horizontálním seznamu místa dělení slov (druhý průchod algoritmu řádkového zlomu), přepíná mezi jednotlivými vzory dělení, které jsou určeny číslem tabulky (\language) a hodnotami registrů \lefthyphenmin a \righthyphenmin. Na začátku odstavce TEX použije implicitní číslo tabulky nula s hodnotami \lefthyphenmin a \righthyphenmin, jaké měly tyto registry při zahájení odstavcového módu. Při dosažení značky \setlanguage v seznamu, TEX změní uvedené tři údaje podle hodnot značky. Tím může TEX (1) přepnout do jiného vzoru dělení než vzor nula, (2) přepínat mezi vzory dělení uvnitř odstavce. Viz též primitivní registr \language.

Kn: 455, 287, **Ol:** 223, 382, 384, 426.

\settabs [plain]

Nastaví tabulátory. Kód makra, viz stranu 126.

Kn: 231–234, 354–355, **Ol:** $125_{119–120}$, $126_{123,\,128}$.

\sevenrm, \seveni, \sevensy, \sevenbf [plain]

Fonty v sedmibodové velikosti jsou v plainu použity pro indexy první úrovně. Viz též \tenrm a \fiverm.

```
692 \font\sevenrm=cmr7
693 \font\seveni=cmmi7
694 \font\sevensy=cmsy7
695 \font\sevenbf=cmbx7
```

Kn: 15, 153, 350–351, 414–415, **Ol:** 29_{85}, $168_{89,\,93}$, 169, 195_{412}, 364, 427_{681}, 440.

\sfcode ⟨*8-bit number*⟩ *restricted* [integer]

Každý znak s ASCII hodnotou ⟨*8-bit number*⟩ má svůj \sfcode, což je celé číslo z intervalu ⟨0, 32 767⟩. Je-li sázen znak s nenulovým \sfcode do horizontálního seznamu, může tento kód ovlivnit registr \spacefactor, který zase později může ovlivnit velikost mezislovní mezery.

IniTEX nastavuje `\sfcode` všech znaků na hodnotu 1000 s výjimkou velkých písmen anglické abecedy, která mají `\sfcode` rovno 999. Plain dále nastavuje

```
696 \sfcode‘\)=0 \sfcode‘\’=0 \sfcode‘\]=0
```

aby tyto znaky neovlivnily hodnotu registru `\spacefactor` a tím velikost případné následující mezery. Kromě toho formát plain aktivuje makro `\nonfrenchspacing` (viz výše), což způsobí větší mezerování za tečkou, čárkou a další interpunkcí.

Jakým způsobem ovlivňují tyto kódy registr `\spacefactor` a jak tento registr ovlivní velikost mezislovní mezery je popsáno na straně 104.

Kn: 76, 214, 271, 286, 321, 345, 351, **Ol:** 104–105, $106_{186,\,188}$, 346, 348_{225}, 349_{240}, 365_{319}, $367_{342–345}$, $406_{572–574}$, 436.

`\shipout` ⟨*box*⟩ [h, v, m]

Uloží tiskový materiál z ⟨*box*⟩u jako další stránku do `dvi` souboru. Povel se obvykle používá ve výstupní rutině, ačkoli to není explicitně nutné.

Každá stránka má svůj referenční bod, který je umístěn 1 in od levého a 1 in od horního okraje papíru. Je-li `\hoffset` a `\voffset` v okamžiku akce `\shipout` rovno nule, kryje se horní levý roh ukládaného boxu s referenčním bodem strany. Obecně, levý horní roh ukládaného boxu je posunut od referenčního bodu strany doprava o `\hoffset` a dolů o `\voffset`. Tyto registry mohou být i záporné, což způsobí posunutí doleva nebo nahoru.

Při akci `\shipout` se provedou všechny pozdržené výstupní operace (povely `\write`, `\openout` a `\openin`), které jsou ve formě značek uloženy kdekoli v materiálu boxu. Tyto operace se provádějí ve stejném pořadí, v jakém jsou umístěny jejich značky v tiskovém materiálu.

Při prvním `\shipout` TEX založí hlavičku `dvi` souboru. K tomu použije hodnotu registru `\mag`.

Kn: 227, 253–254, 279, 300, 302, **Ol:** 72, 239, 256, 257_{265}, 261, 262_{300}, 263, 271, 274_{457}, 276_{460}, 281_{564}, 284, 290, 292, 297, 311, 347, 351, 372, 378, 389, 394, 410, 436, 445–446, 454, 457.

`\show` ⟨*token*⟩ [h, v, m]

Zobrazí význam parametru ⟨*token*⟩ na terminálu. Formát zobrazení je podobný, jako u primitivu `\meaning`.

Kn: 215, 279, 299, 10, **Ol:** 65, 71.

`\showbox` ⟨*8-bit number*⟩ [h, v, m]

Zobrazí obsah boxu do souboru `log`. Obsah popisuje veškerý tiskový materiál boxu v množství, které je nastaveno v `\showboxbreadth` a hloubce vnořených boxů podle `\showboxdepth`.

Kn: 121, 234, 279, 66–67, **Ol:** 83, 100, 265_{322}, 266, 432.

\showboxbreadth (plain: 5) [integer]
Počet zobrazených objektů tiskového materiálu vedle sebe při požadavku na zobrazení boxu do souboru log. Toto zobrazení se týká primitivů \showbox, \showlists, \tracingoutput a hlášení typu Overfull a Underfull.

Nastavení plainu je pro běžnou práci (výpisy typu Underfull) dostačující, ale pro ladění maker nevhodné. Viz též \showboxdepth.

Kn: 302, 273, 303, 348, **Ol:** 109$_{214}$, 145$_{5}$, 265$_{322}$, 431–432, 444$_{782}$, 445.

\showboxdepth (plain: 3) [integer]
Počet úrovní pro zobrazení vnořených boxů při požadavku na zobrazení boxu do souboru log. Viz též primitiv \showboxbreadth.

Kn: 302, 79, 273, 303, 348, **Ol:** 265$_{322}$, 431, 432$_{700}$, 444$_{783}$, 445.

\showhyphens {⟨*text*⟩} [plain]
Makro zobrazí na terminál a do souboru log všechna slova v parametru ⟨*text*⟩ roz-dě-le-na tak-to. Můžeme tím ověřit, jak pracuje algoritmus pro vyhledávání míst dělení slov. Zajímavější řešení téhož problému, viz stranu 226.

```
\def\showhyphens#1{\setbox0=\vbox{\parfillskip=0pt
  \hsize=\maxdimen \tenrm
  \pretolerance=-1 \tolerance=-1 \hbadness=0
  \showboxdepth=0 \ #1}}
```

Makro využívá toho, že při hlášení Underfull \hbox se text zobrazí se slovy rozdělenými. Proto jsou nastaveny odpovídající parametry tak, aby ⟨*text*⟩ způsobil vždy Underfull \hbox.

\showlists [h, v, m]
Zapíše do log souboru obsah všech otevřených seznamů v daném okamžiku. Hlavní vertikální seznam je rozdělen na dvě části: recent contributions (přípravná oblast) a current page (aktuální strana). Každá z těchto částí může být prázdná. Jsou-li nad hlavním vertikálním módem momentálně otevřeny další módy hlavního procesoru, \showlists zobrazí jejich rozpracované seznamy tiskového materiálu a údaj, na kterém řádku vstupního souboru byl příslušný mód zahájen.

Údaje ze \showlists jsou v souboru log uspořádány v pořadí od naposledy otevřeného seznamu po hlavní vertikální seznam, který je vypsán až na konci. Pokud nemáme nastaveno dostatečně velké \showboxbreadth, vidíme většinou velmi málo informací.

Kn: 112, 125, 279, 293, 88–89, 158–159, **Ol:** 30, 100, 103$_{178}$, 109$_{214}$, 144, 145$_{6}$, 153, 161, 220, 432, 463.

\showthe ⟨*internal quantity*⟩ [h, v, m]
Jako \the ale s výstupem na terminál.

Kn: 121, 215, 279, **Ol:** 442.

\shyph [csplain]

Nastaví slovenské dělení slov a \frenchspacing. Viz soubor hyphen.lan.

```
\def\shyph{\language=\slovak  \lccode`\'=`\'  \frenchspacing
           \lefthyphenmin=2   \righthyphenmin=3
           \message{Slovakian hyphenation used.
                    \string\frenchspacing\space is set on.}}
```

\skew [plain]

Vyrovná druhý akcent nad už akcentovaným znakem. Použití

```
\hat{\hat X}        % vytvoří nehezké:  X̂̂
\skew4\hat{\hat X}  % korekce druhého akcentu o 4mu: X̂̂
```

Proč dopadlo prosté dvojí akcentování na řádku 705 tak nehezky? První akcent je umístěn nad jednoznakový základ a jeho poloha je individuálně korigována (viz heslo \skewchar). Na druhé straně, pokud je základ víceznakový, umístí se akcent nad základ prostě na společnou vertikální osu se základem. Jak vidíme z řádku 705, to může být nevyhovující.

```
\def\skew#1#2#3{{\muskip0=#1mu \divide\muskip0 by2
    \mkern\muskip0
    #2{\mkern-\muskip0{#3}\mkern\muskip0}%
    \mkern-\muskip0}{}}
```

Makro zařídí, aby byl akcent posunut vzhledem k ose základu o #1/2 mu. K pochopení tohoto makra je nejlépe si na kus papíru nakreslit obdélníčky.

Kn: 136, 359, **Ol:** 434.

\skewchar ⟨*font*⟩ *restricted* [integer]

Každý font má jeden rezervovaný znak, který je označen jako \skewchar. Tento znak má speciální funkci při usazování matematického akcentu (viz níže). Při zavedení nového fontu (pomocí \font) je nastaven \skewchar tohoto fontu na hodnotu \defaultskewchar.

Sazba matematického akcentu (viz \mathaccent) je v TEXu poněkud odlišná, než sazba textového akcentu (viz \accent). Algoritmus usazení matematického akcentu nyní podrobně popíšeme.

Základ atomu typu Acc (nad ním bude akcent) je uložen do boxu společně se svou italickou korekcí. V řadě následníků akcentu je volen největší takový, který má šířku ještě menší nebo rovnu šířce boxu se základem atomu. Často nemá akcent žádného následníka. Pak není co řešit a tento znak bude přímo použit pro sazbu akcentu. Vertikální usazení akcentu: Je-li box se základem vyšší než 1 ex, je akcent posunut nahoru o rozdíl mezi výškou základu a velikostí 1 ex. Horizontální usazení akcentu: Akcent je umístěn na společnou vertikální osu s boxem základu. Pokud je ale základ jednoznakový a má v tabulce kerningových párů vazbu na znak \skewchar, je akcent od osy vychýlen doprava o hodnotu implicitního kernu mezi základem a znakem \skewchar.

Vidíme tedy, že horizontální usazení akcentu není počítáno podle (případně skloněné) osy znaku, jako to známe z textového \accent. Místo toho se pracuje s osou vždy kolmou na účaří. Tvůrce matematického fontu volí individuálně pro každý znak možné vychýlení akcentu tak, aby to co nejlépe odpovídalo (třeba nesouměrné) kresbě znaku. Tvůrce přitom musí vybrat ve fontu jeden znak \skewchar, který patrně nepoužije pro hladkou sazbu. Všechny znaky, které mají mít vychýlený akcent, budou mít zanesenu hodnotu vychýlení v tabulce kerningových párů.

Příklad. Ve fontu cmmi10 je mezi znakem X a znakem s kódem 127 (což je \skewchar tohoto fontu) implicitní kern velikosti 0,833 pt. Proto je o tuto velikost posunut matematický akcent nad písmenem X doprava. Všimněte si na řádku 705 (heslo \skew), že první akcent je posunut doprava, zatímco druhý je přesně na ose boxu se základem. Kdyby nebyl mezi znakem X a \skewchar žádný kern, byly by oba akcenty přesně na vertikální ose.

Plain nastavuje \skewchar pro fonty rodiny 1 (cmmi*) a 2 (cmsy*) takto:

```
711 \skewchar\teni='177    \skewchar\seveni='177
712 \skewchar\fivei='177
713 \skewchar\tensy='60    \skewchar\sevensy='60
714 \skewchar\fivesy='60
```

Ostatní fonty se znakem \skewchar nepracují, protože je \defaultskewchar rovno −1.

Kn: 214, 271, 273, 277, 351, **Ol:** 152, 167, 351, 391, 433.

\skip ⟨*8-bit number*⟩ [glue]
Povel umožní přístup k 256 registrům typu ⟨*glue*⟩.

Kn: 271, 276, 118–122, 346–347, 349, 352, 363, 394, **Ol:** 73–74, 78–79, 113, 248–249, 251–252, 255, 265–266, 281, 330, 366, 371, 399–401, 413, 415, 443, 454.

\skipdef ⟨*control sequence*⟩⟨*equals*⟩⟨*8-bit number*⟩ [a]
Nová ⟨*control sequence*⟩ bude synonymem pro \skip⟨*8-bit number*⟩.

Kn: 277, 119, 215, 346–347, **Ol:** 66, 73, 326–327, 330_{102}, 400_{525}, 404.

\sl [plain]
Přepínač fontu do *skloněného řezu*. V matematickém módu se uplatní nastavení registru \fam na hodnotu \slfam a mimo matematický mód se uplatní přepínač \tensl deklarovaný přímo primitivem \font. Všimneme si, že rodina \slfam není v plainu úplná. Chybí indexové velikosti.

```
715 \newfam\slfam \def\sl{\fam\slfam\tensl} % \sl is family 5
716 \textfont\slfam=\tensl
```

Kn: 13–15, 165, 351, 409, 414–415, 419, **Ol:** 174, 422_{661}, 427.

`\slash` [plain]

Sazba znaku „/“ s možností řádkového zlomu za tímto znakem.

```
\def\slash{/\penalty\exhyphenpenalty}
```

Kn: 93, 353, **Ol:** není nikdy použito.

`\slovak` [csplain]

Číslo jazyka pro slovenštinu. Viz soubor `hyphen.lan`.

```
\newcount\slovak      \global\slovak=6
```

`\smallbreak` [plain]

Jako `\bigbreak`, ale výsledná mezera má velikost `\smallskipamount` a penalta hodnotu −50.

```
\def\smallbreak{\par\ifdim\lastskip<\smallskipamount
  \removelastskip\penalty-50\smallskip\fi}
```

Kn: 111, 353, 421, **Ol:** není nikdy použito.

`\smallskip`, `\smallskipamount` [plain]

Makro `\smallskip` vloží do seznamu mezeru velikosti čtvrtiny výšky řádku. Viz též `\medskip` a `\bigskip`.

```
\newskip\smallskipamount
\smallskipamount=3pt plus 1pt minus 1pt
\def\smallskip{\vskip\smallskipamount}
```

Kn: 70, 78, 100, 109, 111, 181, 340–341, 352, 355, 410–412, **Ol:** 129$_{160}$, 340$_{159}$, 342, 396, 435$_{720}$.

`\smash {`⟨*sazba*⟩`}` [plain]

Makro pracuje podobně jako `\hphantom`, ale výsledný box není prázdný, nýbrž obsahuje sazbu parametru ⟨*sazba*⟩. Jde tedy pouze o to pronulovat výšku a hloubku výsledného boxu. Vlastní ⟨*sazba*⟩ pak může vyčnívat nahoru nebo dolu přes rozměry boxu.

Makro má analogický kód, jako u makra `\phantom` (strana 418).

```
\def\smash{\relax % in case this comes first in \halign
  \ifmmode\def\next{\mathpalette\mathsm@sh}%
  \else\let\next=\makesm@sh
  \fi\next}
\def\makesm@sh#1{\setbox0=\hbox{#1}\finsm@sh}
\def\mathsm@sh#1#2{\setbox0=\hbox{$\m@th#1{#2}$}\finsm@sh}
\def\finsm@sh{\ht0=0pt \dp0=0pt \box0 }
```

Kn: 131, 178, 327, 360, **Ol:** 184$_{243–244,\,247–248}$, 189$_{335}$, 191, 425$_{674–675}$.

`\softd`, `\softt`, `\softl`, `\softL` [csplain]

expandují po řadě na písmena ď, ť, ľ a Ľ. Při použití `\csaccents` vede tato

expanze na příslušný kód podle $\mathcal{CS}$-fontů. Bez použití \csaccents tyto sekvence dají chybný výsledek.

```
731 \def\softd{\v{d}}\def\softt{\v{t}}\def\ou{\r{u}}%
732 \def\softl{\v{l}}\def\softL{\v{L}}
```

Viz soubor extcode.tex, resp. il2code.tex

\space [plain]

Makro expanduje na ␣$_{10}$.

```
733 \def\space{ }
```

Kn: 254, 272, 306, 351, 380, 406, **Ol:** 14$_{18}$, 28–29, 82$_{95}$, 107$_{195}$, 292$_{63}$, 316$_{1,3}$, 322$_{29}$, 347$_{224}$, 348$_{229}$, 350$_{244}$, 356$_{283}$, 395$_{489}$, 409$_{592}$, 433$_{704}$.

\spacefactor *global, restricted* [integer]

1 000 krát hodnota koeficientu, podle nějž se deformují horizontální mezery. Při sazbě každého znaku se hodnota tohoto parametru může měnit podle \sfcode tohoto znaku. Podrobnější výklad, viz stranu 104.

Kn: 76, 271, 285, 433, 363, **Ol:** *104*, 72, 105–106, 251$_{207}$, 327$_{59–61}$, 334, 336, 346, 371$_{381–382}$, 430–431, 448.

\spaceskip (iniTEX: 0 pt) [glue]

Je-li aspoň jedna komponenta tohoto registru nenulová, pak se \spaceskip použije jako hodnota mezery mezi slovy. Při nulovém \spaceskip se údaje pro mezislovní mezeru čtou z použitého fontu (registry \fontdimen 2, 3 a 4). Viz též \xspaceskip.

Kn: 76, 274, 429, 317, 356, **Ol:** 106, 230, 251$_{216}$, 424$_{672}$, 436, 447.

\span [h, v, m]

Je-li použito v deklaraci řádku tabulky \halign (\valign), bude následující token expandován už v době čtení deklarace řádku. Vyskytne-li se \span v datové části tabulky, nahrazuje tento primitiv formálně zápis oddělovače „&“ a navíc obsah položky bude společný pro sousední sloupce. Je-li \span použito mimo tabulku, způsobí chybu.

Kn: 243, 245, 215, 238, 248, 249, 282, 385, 244, 330, **Ol:** *132, 135*, 72, 132, 135$_{210–211}$, 136, 140–142, 398$_{503}$.

\special ⟨*filler*⟩{⟨*balanced text*⟩} [h, v, m]

TEX provede úplnou expanzi ⟨*balanced text*⟩ (jako při \edef). Do tiskového materiálu uloží bezrozměrnou značku odkazující na výsledek expanze ⟨*balaced text*⟩. Ten bude dále interpretován jako textová informace ve formě „vzkazu“. Při akci \shipout se pak tento „vzkaz“ uloží v místě příslušné značky do dvi souboru bez žádné interpretace, která by ovlivnila TEXovskou sazbu. Jednotlivé dvi ovladače bývají naprogramovány tak, aby podle smluvených vzkazů v dvi souboru vykonaly nějakou činnost. Viz dokumentace k dvi ovladačům.

Tato vlastnost se používá například pro zařazení obrázků do publikace. Obrázky mohou být v libovolném grafickém formátu, který umí zpracovat použitý `dvi` ovladač.

Kn: 228–229, 216, 226, 280, **Ol:** 117, 123_{85}, 278_{510}, 313–314, 318.

`\splitbotmark` [exp]
Poslední značka `\mark` v boxu rozděleném pomocí operace `\vsplit`. Srovnej `\botmark`.

Kn: 259, 213, 280, **Ol:** 275, 391, 456.

`\splitfirstmark` [exp]
První značka `\mark` v boxu rozděleném pomocí operace `\vsplit`. Srovnej `\firstmark`.

Kn: 259, 213, 280, **Ol:** 391, 456.

`\splitmaxdepth` (plain: 4 pt) [dimen]
Maximální hloubka boxu rozděleného pomocí operace `\vsplit`. Srovnej `\maxdepth`. Při rozdělení insertů platí analogická vlastnost, jako pro registry `\floatingpenalty` a `\splittopskip`.

Kn: 124, 274, 281, 348, 363, 417, **Ol:** 251_{213}, 255_{261}, 364, 437, 456.

`\splittopskip` (plain: 10 pt) [glue]
Jako `\topskip`, ovšem uplatní se na zbytek materiálu po operaci `\vsplit`. Pro případ insertů platí analogie s `\floatingpenalty` a `\splitmaxdepth`.

Kn: 124, 274, 281, 348, 363, 397, 417, **Ol:** 113, $244_{156,\,159–160}$, 245_{188}, 246, 249, $251_{212,\,223}$, 255_{261}, 277_{490}, 364, 437, 456.

`\sqrt` ⟨*math field*⟩ [plain]
Vysází odmocninu z výrazu ⟨*math field*⟩. Chcete-li n-tou odmocninu, viz heslo `\root`.

```
734  \def\sqrt{\radical"270370 }
```

Kn: 130–131, 141, 145, 157, 169–170, 360, 443, **Ol:** 191_{387}, $193_{400–401}$, 204_{461}, 205_{467}, 206_{471}, 207_{474}, 427_{688}.

`\ss` [plain]
Německé ostré es (ß) podle kódu CM fontů.

```
735  \chardef\ss="19
```

Kn: 52, 356, **Ol:** není nikdy použito.

`\string` ⟨*token*⟩ [exp]
Expanduje na text identifikátoru řídicí sekvence uvozený znakem podle registru `\escapechar`. Například `\string\Pokus` expanduje na:

$\boxed{\backslash}_{12}\ \boxed{\mathtt{P}}_{12}\ \boxed{\mathtt{o}}_{12}\ \boxed{\mathtt{k}}_{12}\ \boxed{\mathtt{u}}_{12}\ \boxed{\mathtt{s}}_{12}$

Nyní popíšeme vlastnost primitivu přesněji. Je-li ⟨*token*⟩ typu uspořádaná dvojice, primitiv `\string` mu nastaví kategorii 12 bez závislosti na tom, jaká byla původní kategorie tokenu. Tím činnost primitivu končí.

Je-li ⟨*token*⟩ typu řídicí sekvence, `\string` expanduje na posloupnost tokenů. Prvním tokenem posloupnosti je token kategorie 12 s ASCII kódem `\escapechar`. Tento token chybí právě tehdy, když je hodnota registru `\escapechar` mimo interval $\langle 0, 255\rangle$. Následující posloupnost tokenů je shodná s textem identifikátoru. Je-li v textu identifikátoru mezera, vytvoří se token $\boxed{\sqcup}_{10}$. Všechny ostatní tokeny mají kategorii 12.

Příklady:

```
%                                  % expanduje na:
\string A                          % A
\string {                          % {
\expandafter
\string\csname #$ x\endcsname      % \ # $ ␣ x
\string\                           % \ ␣
\string\string                     % \ s t r i n g
{\escapechar=-1
 \string\string                    % s t r i n g
 \escapechar=`\@
 \string\string }                  % @ s t r i n g
\endlinechar=-1
\string\
```

Poslední příklad expanduje na:

$$\boxed{\backslash}_{12}\,\boxed{c}_{12}\,\boxed{s}_{12}\,\boxed{n}_{12}\,\boxed{a}_{12}\,\boxed{m}_{12}\,\boxed{e}_{12}\,\boxed{\backslash}_{12}\,\boxed{e}_{12}\,\boxed{n}_{12}\,\boxed{d}_{12}\,\boxed{c}_{12}\,\boxed{s}_{12}\,\boxed{n}_{12}\,\boxed{a}_{12}\,\boxed{m}_{12}\,\boxed{e}_{12}$$

což si zaslouží naši pozornost. Pomocí znaku „\“ zcela na konci řádku se nám podařilo při `\endlinechar` $= -1$ vytvořit token typu řídicí sekvence se zcela prázdným identifikátorem. Stejný token vytvoříme též pomocí zápisu `\csname\endcsname`. Primitiv `\string` v případě prázdného identifikátoru expanduje specifickým způsobem na právě zmíněný zápis.

Ještě ve verzi TEXu 3.14 byla poměrně podstatná chyba. Například při použití `\string\^^X` primitiv chybně expandoval na posloupnost tokenů $\boxed{\backslash}_{12}\,\boxed{\hat{}}_{12}\,\boxed{\hat{}}_{12}\,\boxed{X}_{12}$, zatímco v současné verzi TEXu 3.14159 už expanduje správně na posloupnost $\boxed{\backslash}_{12}\,\boxed{\hat{}\hat{}X}_{12}$. Řídicí sekvence `\^^X` má totiž jednoznakový identifikátor (viz stranu 22, dvojitá stříška v token procesoru).

Kn: 213–214, 40–41, 215, 348, 377, **Ol:** 25_{43}, 26, 50_{181}, $53_{215–216}$, 56_{259}, $57_{268,\,271}$, 72, 208_{485}, $290_{34,\,36,\,46}$, $292_{67,\,73}$, 293_{85}, 347_{224}, $348_{226–227}$, 356_{283}, 361, 400_{537}, 401_{548}, $402_{561,\,564}$, 433_{704}, 442, 457.

\strut [plain]

Vloží do horizontálního nebo matematického seznamu neviditelnou podpěru nulové šířky a celkové výšky 12 pt.

```
\newbox\strutbox
\setbox\strutbox=
               \hbox{\vrule height8.5pt depth3.5pt width0pt}
\def\strut{\relax % aby \strut fungoval i v položce \halign
          \ifmmode\copy\strutbox \else\unhcopy\strutbox \fi}
```

Kn: 82, 142, 178, 240, 246–247, 316, 329, 333, 353, 396, 400, 421, **Ol:** 87$_{136}$, 115, 135$_{208}$, 138, 139$_{229,\,231,\,234}$, 141$_{249,\,252}$, 143$_{270–271}$, 191–192, 193$_{398–399}$, 251$_{222}$, 257$_{266}$, 268$_{361}$, 342$_{193–194}$, 343, 359$_{294}$, 394, 450$_{792}$, 452$_{805}$.

\supereject [plain]

Při použití \supereject se vyvolá makro \dosupereject ve výstupní rutině plainu. Kód makra \dosupereject, viz stranu 262. Rozbor činnosti makra, viz stranu 264.

```
\def\supereject{\par\penalty-20000 }
```

Kn: 116, 254, 256–257, 353, 407, **Ol:** *264*, 262$_{309}$, 345$_{208}$.

\t ⟨*znak*⟩ [plain]

Vysází akcent tvaru obloučku. Levý okraj obloučku je na ose znaku a kresba obloučku „vyčnívá“ ze sazby znaku směrem doprava. Například \t T vede na „T͡“. V některých jazycích se tento akcent používá k vyznačení spojení znaku s následujícím. V TEXbooku je příklad „o͡o“ a dále jméno „Sergeĭ Ĭur'ev“.

```
\def\t#1{{\edef\next{\the\font}%
    \the\textfont1\accent"7F\next#1}}
```

Protože obloučeк popsaných vlastností je k dispozici jen ve fontu cmmi*, slouží nám toto makro jako praktický příklad využití vlastnosti primitivu \accent zmíněné na řádku 137.

Kn: 52–53, 356, **Ol:** 128$_{148–150}$, 336.

\tabalign [plain]

Řádek řízený tabulátory. Kód makra je uveden na straně 124.

Kn: 354–355, **Ol:** *124*, 126, 127$_{134–135}$, 179.

\tabskip (iniTEX: 0 pt) [glue]

Velikost horizontální (vertikální) mezery mezi sloupci (řádky) v \halign (\valign).

Kn: 237–239, 215, 274, 282, 285, 244, 247, 354, **Ol:** 72, 128, 129$_{158}$, 131, 132$_{172,\,174–175}$, 133, 135$_{208–210}$, 136, 140, 141$_{250–252}$, 142, 184, 205, 346, 354$_{269–270}$, 359$_{297,\,299–300}$, 374, 375$_{390}$, 385$_{429,\,431–433}$.

\tenrm, \teni, \tensy, \tenex, \tenbf, \tentt, \tensl, \tenit [plain]

Fonty v desetibodové velikosti jsou v plainu použity pro základní textovou sazbu a dále sazbu ve stylech D a T v matematickém módu. Viz též \fiverm a \sevenrm.

```
\font\tenrm=cmr10  % roman text
\font\teni=cmmi10  % math italic
\font\tensy=cmsy10 % math symbols
\font\tenex=cmex10 % math extension
\font\tenbf=cmbx10 % boldface extended
\font\tentt=cmtt10 % typewriter
\font\tensl=cmsl10 % slanted roman
\font\tenit=cmti10 % text italic
```

Formát csplain zavádí pro textovou sazbu místo CM fontů $\mathcal{CS}$-fonty. Zavedení fontů v csplainu vypadá tedy takto:

```
\font\tenrm=csr10  % základní řez, antikva
\font\tenbf=csbx10 % polotučná varianta
\font\tentt=cstt10 % písmo psacího stroje
\font\tensl=cssl10 % geometricky nakloněné písmo
\font\tenit=csti10 % textová kurzíva
```

Podobná změna je realizovaná pro fonty „\seven...“ a „\five...“. Fonty určené pro matematickou sazbu (cmmi*, cmsy* a cmex10) zůstávají nezměněny. Podrobněji viz soubor csfonts.tex.

Kn: 153, 350–351, 414–415, **Ol:** 65, 74, 106, 168–169, 172–173, 175, 262, 326, 336, 363–364, 366, 395, 427, 430, 432.

\TeX [plain]

Makro vysází logo „TEX“. Je to ukázkové cvičení na práci s primitivy \lower a \kern.

```
\def\TeX{T\kern-.1667em\lower.5ex\hbox{E}\kern-.125emX}
```

Kn: 8–10, 19, 66–67, 204, 225, 340–341, 356, 418–419, **Ol:** 10, $24_{32,\,37}$, 31_{1}, $119_{25–26}$, 290_{39}, 381, 422_{658}.

\textfont ⟨*4-bit number*⟩ [font]

Registr označuje ⟨*font*⟩ rodiny ⟨*4-bit number*⟩ pro matematický styl T a D. Například \textfont1 je v plainu font cmmi10. Jedná se o font rodiny 1 pro sazbu matematickou kurzívou v základní velikosti. Nastavení registrů \textfont v plainu, viz tabulku na straně 169 a 170. Viz též \scriptfont a \scriptscriptfont.

Ačkoli se přiřazení do registru realizuje prostřednictvím přepínače typu ⟨*font*⟩, (například `\textfont1=\teni`) pozdější změna tohoto přepínače hodnotu registru neovlivní. Pokud třeba v době přiřazení měla řídicí sekvence `\teni` význam fontu `cmmi10` a později napíšeme kupříkladu `\font\teni=jiny10`, bude `\textfont1` stále znamenat font `cmmi10`.

Kn: 351, 153, 168, 188, 213, 271, 441–442, 414–415, **Ol:** *sekce 5.3*, 162, 167–171, 173–174, 177, 180–181, 195, 200, 203, 276, 322, 335, 340, 345, 365, 380, 397, 409, 427–428, 434, 439, 447.

`\textindent {`⟨*text*⟩`}` [plain]

Zahájí odstavec s odstavcovou zarážkou a do prostoru této zarážky vloží ⟨*text*⟩ tak, aby mezi pravým okrajem sazby ⟨*text*⟩ a pravým okrajem zarážky byla vzdálenost rovna 0,5 em. Následující sazba bude těsně navazovat na odstavcovou zarážku a bude tedy od pravého okraje ⟨*text*⟩ vzdálena o zmíněnou mezeru. Levý okraj ⟨*text*⟩ může utéci zcela mimo zrcadlo sazby.

```
771 \def\textindent#1{\indent\llap{#1\enspace}\ignorespaces}
```

Kn: 117, 135, **Ol:** 251_{217}, $380_{414–415}$.

`\textstyle` [m]

Nastaví styl T pro sestavování matematického seznamu. Jedná se o základní velikost matematické sazby. Do tohoto stylu se sazba matematiky zapíná implicitně ve vnitřním matematickém módu. Viz též `\displaystyle`, `\scriptstyle` a `\scriptscriptstyle`.

Kn: 292, 141–142, 326, **Ol:** *154*, *sekce 5.2*, $155_{45–46}$, 163_{79}, 177, 188_{302}, 189_{331}, 354, 393_{472}, 429.

`\the` ⟨*internal quantity*⟩ [exp]

Expanduje na posloupnost tokenů, která vypisuje obsah ⟨*internal quantity*⟩. Například `\the\pageno` expanduje na této stránce na posloupnost

$\boxed{4}_{12}\,\boxed{4}_{12}\,\boxed{1}_{12}$.

Jednotlivé tokeny mají vždy kategorii 12. Výjimkou je pouze mezera (ASCII 32), která vytváří token $\boxed{\sqcup}_{10}$.

Parametr ⟨*internal quantity*⟩ označuje nějaký registr TEXu typu ⟨*number*⟩, ⟨*dimen*⟩, ⟨*glue*⟩, ⟨*muglue*⟩, ⟨*tokens*⟩, nebo ⟨*font*⟩. Příklady: `\the\lineskip`, `\the\dimen0`, `\the\fontdimen2\textfont0`, `\the\catcode‘\\`, `\the\sum`, `\the\%`. Poslední dva případy ukazují použití tokenů, dříve definovaných pomocí `\mathchardef` a `\chardef`.

Je-li ⟨*internal quantity*⟩ typu ⟨*number*⟩, je výsledkem dekadická reprezentace čísla. Jedná-li se o token z `\chardef` a `\mathchardef`, je výsledkem kód ve formě desetinného čísla. Například `\the\%` expanduje na „`37`“ a `\the\sum` na „`4944`“.

Je-li ⟨*internal quality*⟩ typu ⟨*dimen*⟩, je výsledkem číslo s desetinnou tečkou a s jednotkou `pt`. Například `\the\hsize` pro tento dokument expanduje na „`360.0pt`“.

Je-li ⟨*internal quantity*⟩ typu ⟨*glue*⟩ nebo ⟨*muglue*⟩, jsou připojena slova `plus` a `minus` jen v případě, že jsou příslušné hodnoty nenulové. Například `\the\parfillskip` expanduje na „`0.0pt plus 1.0fil`“.

Je-li ⟨*internal quantity*⟩ typu ⟨*tokens*⟩, je výsledkem expanze posloupnost tokenů uložená ve vyšetřovaném registru. Kategorie těchto tokenů nejsou měněny, takže zde je jediný případ, kdy expanze `\the` může vést na tokeny jiných kategorií než 12 nebo 10. Například `\the\output` expanduje na `plainoutput`.

Parametr ⟨*internal quantity*⟩ může být typu ⟨*font*⟩. Pak `\the` expanduje na řídicí sekvenci, která byla použita při zavedení fontu jako přepínač fontu. Příklad: `\the\tenrm`, `\the\textfont0`, `\the\font` expandují na společný výsledek, kterým je token `tenrm`.

Ostatní typy (například ⟨*box*⟩) nelze po `\the` použít.

Primitiv `\showthe` pracuje shodně jako `\the`, ovšem výsledek je vytištěn na terminál. Viz též primitivy `\number`, `\romannumeral`, `\meaning`, `\string`, `\jobname` a `\fontname`.

Kn: 214–215, 216, 373, 375, 422, **Ol:** 25, 29, 37–39, 41, 52–63, 65, 75–76, 79–81, 84, 95, 103, 121–123, 134–135, 162, 180, 195, 207–208, 242–243, 251, 257, 259–260, 262, 267, 276, 281, 289–293, 322, 335, 356, 365–366, 400–401, 408, 427, 439.

`\thickmuskip` (plain: 5 mu plus 5 mu) [muglue]
Hodnota velké matematické mezery, např. mezi atomy typu Ord a Rel.

Kn: 167–168, 274, 446, 349, **Ol:** *158*, 134_{187}, 158_{60}, 159–160, 162_{74}, 191, 325_{52}, 335_{130}.

`\thinmuskip` (plain: 3 mu) [muglue]
Hodnota malé matematické mezery, například mezi atomy Punct a Ord.

Kn: 167–168, 274, 446, 349, **Ol:** *158*, 62_{337}, 73, 158_{62}, 159, 191, 192_{394}, $335_{128,\,131}$.

`\thinspace` [plain]
Vloží do sazby malý výplněk. Vhodné například před fyzikální jednotky. „Délka rovníku je zhruba 40 000 km“ zapíšeme takto:

```
772 Délka rovníku je zhruba 40\thinspace000\thinspace km
```

Je vhodné pro tento výplněk definovat zkratku, třeba `\def\,{\thinspace}`, nebo ještě lépe, viz ukázku na straně 192. Pak lze délku rovníku zapisovat takto: `40\,000\,km`.

```
773 \def\thinspace{\kern .16667em }
```

Kn: 5, 10, 305, 311, 352, 409, **Ol:** 192$_{394}$, 335$_{132}$, 342$_{192}$.

`\time` [integer]
Počet minut po půlnoci. Údaj je v okamžiku spuštění TEXu načten ze systémové proměnné data a času.

Kn: 273, 349, **Ol:** není nikdy použito.

`\toks` ⟨*8-bit number*⟩ [tokens]
Povel umožní přístup k 256 registrům typu ⟨*tokens*⟩.

Kn: 212, 215, 262, 276, **Ol:** 54$_{233–234}$, 71, 73–74, 330$_{103}$, 399$_{513}$, 400$_{528}$, 402$_{554}$.

`\toksdef` ⟨*control sequence*⟩⟨*equals*⟩⟨*8-bit number*⟩ [a]
Nová ⟨*control sequence*⟩ bude synonymem pro `\toks`⟨*8-bit number*⟩.

Kn: 212, 215, 277, 347, 378, **Ol:** 54, 66, 73, 327, 330$_{103}$, 400$_{528}$, 402$_{554}$, 404.

`\tolerance` (plain: 200) [integer]
Maximální povolená hodnota badness pro všechny řádky při druhém průchodu algoritmu řádkového zlomu, při němž se sestavuje odstavec s možným dělením slov. Viz též `\pretolerance`, `\emergencystretch`.

Kn: 96, 91, 94, 107, 272, 29–30, 317, 333, 342, 348, 364, 451, **Ol:** *sekce 6.4*, 73, 227–228, 230, 232, 276$_{486}$, 432$_{699}$.

`\topglue` [plain]
Jako `\vglue`, ale vhodné pro začátek strany, kde je potřeba kompenzovat mezeru z `\topskip`.

```
774 \def\topglue{\nointerlineskip\vglue-\topskip\vglue}
```

Kn: 340, 352, **Ol:** 113, 454.

`\topins` [plain]
Třída insertů pro plovoucí obrázky a tabulky.

```
775 \newinsert\topins
776 \skip\topins=0pt % no space added
777 \count\topins=1000 % magnification factor (1 to 1)
778 \dimen\topins=\maxdimen % no limit per page
```

Kn: 256, 363–364, **Ol:** 252$_{227}$, 255$_{245–248, 260}$, 264.

`\topinsert` ⟨*vertical material*⟩ `\endinsert` [plain]
Plovoucí objekt má tendenci být umístěn nahoru na stránce. Kód makra s výkladem, viz stranu 254.

Kn: 115–116, 251, 363, **Ol:** 254$_{242}$, 255$_{250}$.

`\topmark` [exp]
Algoritmus uzavření strany vždy přiřadí sekvenci `\topmark` význam původní

sekvence \botmark. Proto \topmark expanduje na poslední značku typu \mark z předposlední kompletované strany.

Kn: 258, 213, 259–260, 280, **Ol:** *258*, 256, 259–260, 391.

\topskip (plain: 10 pt) [glue]

Jakmile vstupuje do aktuální strany první box nebo linka, je před tento element vložena mezera podle \topskip. Hodnota stažení nebo roztažení této mezery se přímo přebírá z odpovídajících hodnot v \topskip. Přirozená velikost mezery je rovna:

$$\max(0\,\text{pt},\ \textit{přirozená velikost z}\ \texttt{\topskip} - \textit{výška elementu})$$

Obvykle tedy vychází vzdálenost účaří prvního elementu od horního okraje strany rovna přirozené hodnotě \topskip.

Kn: 113–114, 124, 256, 274, 348, **Ol:** *112*, 113–114, 240, 242_{142}, 243_{143}, $244_{159–160}$, 246, 252_{237}, 258, 262_{309}, 263, $267_{325–326,\,330,\,343,\,347,\,351}$, 268_{358}, 269_{378}, 277_{490}, $282_{583,\,587}$, $423_{670–671}$, 437, 443_{774}, 456.

\tracingall [plain]

Nastaví nejvíce upovídané trasující informace, které TEX zapisuje do souboru log a na terminál. Pro jen trochu delší dokument to je naprosto nevhodné.

```
779 \def\tracingall{\tracingonline=1 \tracingcommands=2
780   \tracingstats=2 \tracingpages=1 \tracingoutput=1
781   \tracinglostchars=1 \tracingmacros=2 \tracingparagraphs=1
782   \tracingrestores=1 \showboxbreadth=\maxdimen
783   \showboxdepth=\maxdimen \errorstopmode}
```

Kn: 121, 303, 364, **Ol:** není nikdy použito.

\tracingcommands (iniTEX: 0) [integer]

Je-li tento registr roven jedné, TEX uloží do souboru log veškeré povely hlavního procesoru, které při sazbě vykonává. Je-li tento registr roven aspoň dvěma, jsou navíc ukládány informace o vyhodnocení podmínek u primitivů typu \if... Je-li registr nulový nebo záporný, žádná zpráva tohoto charakteru se nevypíše. To odpovídá implicitnímu nastavení.

Pro stručnost nejsou povely pro sazbu znaku a povel $\boxed{\sqcup}_{10}$ uváděny všechny, ale v souvislé řadě takových povelů vždy jen ten první.

Kn: 212, 88–89, 273, 299, **Ol:** 30, 444_{779}.

\tracinglostchars (plain: 1) [integer]

Je-li tento registr kladný, TEX uloží do souboru log hlášení o nenalezených písmenech v použitém fontu. Kdy k tomu dojde? Zkuste si fontem logo10, které obsahuje jen pár písmen (M, E, T, A, F, O, N, v nových verzích ještě P a S) vysázet třeba písmeno „č“ a podívejte se do souboru log.

Při nulovém nebo záporném registru nebudou informace o chybějících znacích v použitém fontu nikam ventilovány a TEX pouze mlčky nahradí chybějící znak prázdným znakem s nulovými rozměry.

Příklad. Srovnejme tyto dvě definice loga METAFONT:

```
784 \def\mf{{\mflogo METAFONT}}
785 \def\mf{{\mflogo META}\-{\mflogo FONT}}
```

Použijeme-li první variantu (z řádku 784), pak po použití makra `\mf` v odstavci, ve kterém se algoritmus řádkového zlomu dostal do druhého průchodu, můžeme v souboru `log` číst:

```
Missing character: There is no - in font logo10!
```

Toto hlášení se objeví i v případě, že nakonec není slovo METAFONT rozděleno, protože druhý průchod řádkového zlomu bezpodmínečně potřebuje metrické údaje znaku `\hyphenchar`. Vidíme tedy, že jediná správná definice loga METAFONT je uvedena na řádku 785.

Kn: 301, 273, 348, 401, **Ol:** 444_{781}.

`\tracingmacros` (iniTEX: 0) [integer]
Je-li registr kladný, TEX uloží do souboru `log` informaci o způsobu expanze všech použitých maker, včetně hodnot jejich parametrů. Je-li hodnota registru aspoň dvě, bude navíc trasována expanze maker z vestavěných registrů typu ⟨*tokens*⟩ (`\output`, `\everymath` atd.). V některých implementacích TEXu (emTEX) není mezi jedničkou a dvojkou rozdíl.

Implicitní nastavení registru je 0, tj. tato informace nebude vypisována.

Kn: 205, 212, 273, 329, **Ol:** 36_{36}, 444_{781}.

`\tracingonline` (iniTEX: 0) [integer]
Je-li registr kladný, TEX zobrazí všechny trasovací informace, které jsou důsledkem ostatních kladných registrů typu `\tracing...`, také na terminál. Implicitní hodnota registru je 0, takže případné trasovací informace se zapisují jen do souboru `log`.

Kn: 303, 121, 212, 273, **Ol:** 444_{779}.

`\tracingoutput` (iniTEX: 0) [integer]
Je-li registr kladný, TEX uloží do souboru `log` informaci o obsahu boxů, které jsou ukládány do výstupního `dvi` souboru pomocí primitivu `\shipout`. Viz ale `\showboxbreadth` a `\showboxdepth`.

Implicitní nastavení registru je 0, tj. tato informace nebude vypisována.

Kn: 254, 273, 301–302, **Ol:** 252_{236}, 432, 444_{780}.

`\tracingpages` (iniTEX: 0) [integer]
Je-li registr kladný, TEX uloží do souboru `log` trasovací údaje o průběhu algoritmu plnění strany. V každém potenciálním místě zlomu vidíme zápis o všech důležitých registrech algoritmu stránkového zlomu.

Implicitní nastavení registru je 0, tj. tato informace nebude vypisována.

Kn: 303, 124, 273, 112–114, **Ol:** 241, 252_{236}, 444_{780}.

`\tracingparagraphs` (iniTEX: 0) [integer]
Je-li registr kladný, TEX uloží do souboru `log` trasovací údaje o průběhu algoritmu řádkového zlomu. Jak tyto údaje vypadají, vidíme například na straně 231 v této knize.

Implicitní nastavení registru je 0, tj. tato informace nebude vypisována.

Kn: 303, 98–99, 273, **Ol:** 231, 444_{781}.

`\tracingrestores` (iniTEX: 0) [integer]
Je-li registr kladný, TEX uloží do souboru `log` informaci o všech hodnotách registrů a maker, které se obnovují po uzavření skupiny.

Implicitní nastavení registru je 0, tj. tato informace nebude vypisována.

Kn: 301, 303, 273, **Ol:** 299, 444_{782}.

`\tracingstats` (iniTEX: 0) [integer]
Je-li registr kladný, TEX uloží na konci práce do souboru `log` informaci o alokaci jednotlivých paměťových polí, jak bylo vysvětleno v sekci 7.2. Je-li hodnota tohoto registru aspoň dvě, vypisuje TEX navíc informaci o zaplnění hlavní paměti TEXu (limit *mem_max*) při každém provedení primitivu `\shipout`. Tato informace je rozdělena znakem `&` do dvou částí, které je nutno sečíst. Jednotlivé části referují o zaplnění hlavní paměti podle typu uložené informace. Vlevo je (zhruba řečeno) údaj o zaplnění datovými informacemi boxů a vpravo údaj o zaplnění definicemi maker a pomocnými údaji. Například text:

```
Memory usage before: 1794&7397; after: 116&6054;
                                 still untouched: 56108
```

oznamuje, že před provedením `\shipout` byla paměť zaplněna $1794 + 7397$ paměťovými místy (jednotka m, viz stranu 296) a po provedení `\shipout` si TEX „ulevil“ a jeho zaplnění je jenom $(116 + 6054)\,m$. V tento okamžik zbývá (pro další stranu) $56\,108\,m$ volného místa v hlavní paměti.

Mohou existovat instalace TEXu, které z důvodu požadavku na vyšší rychlost nemají v programu TEX zakompilovány všechny části. Chybějí části označené ve WEBovském zdrojovém textu závorkami **stat . . . tats**. Důsledkem toho je, že například `\tracingstats=2` nemusí fungovat. S takovou instalací jsem se ale ještě nesetkal.

Kn: 300, 303, 273, 383, **Ol:** 295–296, 300, 444_{780}.

\tt [plain]

Přepínač fontu do strojopisu. V matematickém módu se uplatní nastavení registru \fam na hodnotu \ttfam a mimo matematický mód se uplatní přepínač \tentt deklarovaný přímo primitivem \font. Všimneme si, že rodina \ttfam není v plainu úplná. Chybí indexové velikosti.

```
\newfam\ttfam \def\tt{\fam\ttfam\tentt} % \tt is family 7
\textfont\ttfam=\tentt
```

Kn: 13, 53, 113, 165, 340–341, 351, 380–382, 414–415, 429, **Ol:** 68_{13}, 97_{173}, 173, 174_{119}, 216, 221_{63}, 427, 447_{788}.

\ttraggedright [plain]

Jako \raggedright, ale pro písmo neproporcionálního řezu (strojopis). Toto písmo nemá možnost stažení ani roztažení mezislovní mezery, a proto není nutno (na rozdíl od \raggedright) pracovat s registrem \spaceskip.

```
\def\ttraggedright{\tt \rightskip=0pt plus2em\relax}
```

Kn: 356, **Ol:** není nikdy použito.

\u [plain]

Akcent tvaru obloučku. Například \u a vede na „ă“.

```
\def\u#1{{\accent21 #1}}
```

Kn: 52–53, 356, **Ol:** není nikdy použito.

\uccode ⟨*8-bit number*⟩ *restricted* [integer]

Jedná se o kód velké alternativy znaku s kódem ⟨*8-bit number*⟩. Tím lze pro každý znak definovat chování algoritmu \uppercase.

Velká písmena anglické abecedy mají v iniTEXu \uccode rovno přímo ASCII hodnotě znaku a malá písmena anglické abecedy mají \uccode rovno ASCII hodnotě znaku minus 32. Ostatní znaky mají \uccode nulové.

V csplainu je dále nastaveno \uccode akcentovaných majuskulí (velkých písmen) shodně s pozicí znaku v $\mathcal{CS}$-fontu a minuskule (malá písmena) mají \uccode shodné s pozicí odpovídající majuskule. Viz soubor extcode.tex, resp. il2code.tex.

Kn: 41, 241, 271, 345, 348, 377, 394, **Ol:** 27_{70}, 150, 402_{562}, 451.

\uchyph (plain: 1) [integer]

Je-li registr kladný, pak je dovoleno dělit slova začínající velkým písmenem. Velké písmeno je znak, pro který je \lccode různý od nuly a od ASCII hodnoty znaku.

Kn: 454, 273, 348, **Ol:** 220.

\underbar {⟨*text*⟩} [plain]

Podtrhne ⟨*text*⟩ pomocí primitivu \underline. Makro pracuje pouze mimo

matematický mód a čára, která podtrhává text, je v konstantní vzdálenosti pod účařím bez závislosti na tom, jaká je hloubka boxu se sazbou parametru ⟨*text*⟩. Vypadá to například takto.

```
790 \def\underbar#1{$\setbox0=\hbox{#1}\dp0=0pt
791   \m@th \underline{\box0}$}
```

Kn: 244, 323, 353, **Ol:** není nikdy použito.

\underbrace [plain]

Svorka pod textem. Kód makra, viz stranu 184.

Kn: 176, 225–226, 359, **Ol:** 183_{228}, 184_{239}.

\underline ⟨*math field*⟩ [m]

Vytvoří nový atom třídy Under, obsahující v základu ⟨*math field*⟩. Tento atom bude sázen s čarou pod základem. Čára bude mít tloušťku $t =$ \fontdimen8 fontu rodiny 3 odpovídajícího stylu. Pod čarou je ještě mezera velikosti t a mezi čarou a spodním okrajem základu atomu je mezera velikosti $3t$. Viz též \overline.

Kn: 443, 141, 291, 130–131, **Ol:** 149, 167_{86}, 181, 412, 447, 448_{791}.

\unhbox ⟨*8-bit number*⟩ [h, v]

Do zpracovávaného seznamu neuloží box z registru ⟨*8-bit number*⟩ jako celek, ale jeho „vnitřek“. Vkládá tedy horizontální seznam tiskového materiálu, který je obsažen v \hboxu uloženém v registru ⟨*8-bit number*⟩.

Přesněji. Je-li primitiv použit ve vertikálním módu, zůstane ve čtecí frontě a hlavní procesor zahájí odstavcový mód, jako při \indent. Pak je primitiv čten znova. Je-li primitiv v matematickém módu, TEX ohlásí chybu.

Následuje popis činnosti primitivu \unhbox v horizontálním módu. Je-li registr ⟨*8-bit number*⟩ prázdný, pak \unhbox nedělá nic. Rovněž se nic nestane, pokud je registru ⟨*8-bit number*⟩ přiřazen \vbox. V tomto případě TEX navíc vypíše chybové hlášení (vertikální seznam nelze vkládat do horizontálního). Je-li v registru ⟨*8-bit number*⟩ neprázdný \hbox, pak se jeho obsah připojí k aktuálnímu seznamu a registr ⟨*8-bit number*⟩ se vynuluje (při \unhcopy se registr nenuluje). Hodnota \spacefactor se při vložení materiálu vůbec nemění.

Kn: 285, 120, 283, 354, 356, 361, 399, **Ol:** 83, $84_{124,\,127}$, 85, 89, 96, $127_{139,\,146}$, $128_{152,\,155–156}$, $226_{98,\,102}$, $245_{185,\,189}$, 247, 265_{318}, 272_{412}, 274_{444}, 342_{196}, 345_{210}, 384_{426}, 409_{597}, 451.

\unhcopy ⟨*8-bit number*⟩ [h, v, m]

Jako \unhbox, ovšem nemaže se obsah použitého registru ⟨*8-bit number*⟩.

Kn: 285, 120, 283, 353, **Ol:** 83, 89, 439_{753}, 448.

\unkern [h, v, m]

Pokud je posledním objektem ve zpracovávaném seznamu \kern, je vymazán.

Není-li posledním objektem seznamu \kern, nic se nestane. Je-li přípravná oblast prázdná a povel \unkern je použit v hlavním vertikálním módu, TEX ohlásí chybu.

Kn: 280, **Ol:** 382.

\unpenalty [h, v, m]

Pokud je posledním objektem ve zpracovávaném seznamu penalta, je vymazána. Není-li posledním objektem seznamu penalta, nic se nestane. Je-li přípravná oblast prázdná a povel \unpenalty je použit v hlavním vertikálním módu, TEX ohlásí chybu. Nelze totiž odebrat penaltu, která už vstoupila do aktuální strany.

Kn: 280, **Ol:** 84$_{121,\,127}$, 226$_{96}$, 265$_{317}$, 266, 383.

\unskip [h, v, m]

Pokud je posledním objektem ve zpracovávaném seznamu mezera typu ⟨*glue*⟩, je vymazána. Není-li posledním objektem seznamu taková mezera, nic se nestane. Je-li přípravná oblast prázdná a povel \unskip je použit v hlavním vertikálním módu, TEX ohlásí chybu.

Končí-li například odstavec display módem (\$\$...\$\$\par), je pod rovnicí mezera z \belowdisplay(short)skip, kterou nelze pomocí \unskip odebrat, protože už stačila přejít do aktuální strany. V takovém případě lze použít trik použitý v makru \removelastskip.

Kn: 280, 222–223, 286, 313, 392, 418–419, **Ol:** 84$_{121}$, 91$_{152}$, 212$_{6}$, 226$_{96,\,98}$, 227, 245$_{189}$, 265$_{314–317}$, 266, 277$_{493}$, 338$_{148}$, 342$_{196}$, 383, 416, 425.

\unvbox ⟨*8-bit number*⟩ [h, v]

Do zpracovávaného seznamu neuloží box z registru ⟨*8-bit number*⟩ jako celek, ale jeho „vnitřek“. Vkládá tedy vertikální seznam tiskového materiálu, který je obsažen ve \vboxu uloženém v registru ⟨*8-bit number*⟩.

Přesněji. Je-li primitiv použit v odstavcovém módu, zůstane ve čtecí frontě a před něj se vloží token par. To (obvykle) ukončí odstavcový mód a povel \unvbox je čten znova ve vertikálním módu. Je-li \unvbox použit v matematickém, resp. vnitřním horizontálním módu, TEX ohlásí chybu Missing \$, resp. }, inserted. Po ukončení příslušného módu je tedy povel \unvbox čten znova.

Následuje popis činnosti primitivu \unvbox ve vertikálním módu. Je-li registr ⟨*8-bit number*⟩ prázdný, pak \unvbox nedělá nic. Rovněž se nic nestane, pokud je registru ⟨*8-bit number*⟩ přiřazen \hbox. V tomto případě TEX navíc vypíše chybové hlášení (horizontální seznam nelze vkládat do vertikálního). Je-li v registru ⟨*8-bit number*⟩ neprázdný \vbox, pak se jeho obsah připojí k aktuálnímu seznamu a registr ⟨*8-bit number*⟩ se vynuluje (při \unvcopy se registr nenuluje). Hodnota \prevdepth se při vložení materiálu vůbec nemění.

Protože není měněno \prevdepth, vznikají dva problémy:

- Před materiálem z \unvbox není žádná mezeřádková mezera.
- Před boxem, který následuje za \unvbox, je chybná mezeřádková mezera.

První problém obvykle řešíme vložením \strut do prvního boxu, který je obsažen v materiálu \unvbox. Druhý problém obvykle řešíme trikem s \lastskip. Příklad:

```
\setbox0=\vbox{\strut⟨několik řádků textu⟩}
% Na jiném místě je použití:
⟨nějaké řádky textu⟩
\par \penalty0
\ifdim\prevdepth>-1000pt
   \kern-\prevdepth \kern3.5pt \fi
\unvbox0
\nointerlineskip \lastbox
⟨další řádky textu⟩
```

Nejprve popíšeme, jak jsme se vypořádali s mezeřádkovou mezerou nad materiálem z \unvbox. Pomocí testu na \prevdepth zjišťujeme hloubku naposledy vloženého boxu. Záporným kernem se vracíme na účaří tohoto boxu a vkládáme kern odpovídající hloubce podpěry \strut. Pak naváže první box z materiálu \unvbox, ale ten obsahuje \strut, takže řádkování je zachováno.

Zajímavější je trik s \lastskip, který řeší mezeru za materiálem vloženým z \unvbox. Trik předpokládá, že posledním elementem tohoto materiálu je box. Před ním je správná mezeřádková mezera z doby, kdy byl tento materiál budován. Primitiv \lastbox odebere tento box a znovu jej do seznamu vloží na stejné místo. Aby před tímto boxem nevznikla nová mezeřádková mezera, je použito \nointerlineskip. V okamžiku zpětného vložení tohoto zrovna odebraného boxu do vertikálního seznamu se nastaví správné \prevdepth, takže následující řádek bude připojen se správnou mezeřádkovou mezerou. Odebrání boxu pomocí \lastbox je v tomto případě možné i v hlavním vertikálním seznamu, protože po provedení operace \unvbox není vyvolán algoritmus plnění strany. Veškerý materiál je tedy zatím v přípravné oblasti a odtud je odebíratelný.

Jestliže materiál z \unvbox obsahuje jediný box (například jediný řádek textu), nepišme \nointerlineskip a použijme jen \lastbox. To řeší jednak problém mezery nad \unvbox i pod \unvbox. Viz například stranu 127, řádek 145.

Kn: 282, 120, 254, 286, 354, 361, 363–364, 392, 399, 417, **Ol:** 83, 90, 96, 127, 129, 243, 247, 252, 255, 257, 261, 265, 267–270, 272–274, 276–277, 281–282, 319, 342, 382.

\unvcopy ⟨*8-bit number*⟩ [h, v, m]

Jako \unvbox, ovšem nemaže obsah použitého registru ⟨*8-bit number*⟩.

Kn: 282, 120, 286, 361, **Ol:** 83, 90, 274_{449}, 342_{195}, 449.

\upbracefill [plain]
Vodorovná svorka natahovací délky. Kód makra, viz stranu 184.

Kn: 225–226, 357, **Ol:** $184_{242,\,254}$.

\uppercase ⟨*filler*⟩{⟨*balanced text*⟩} [h, v, m]
Konvertuje ⟨*balanced text*⟩ do velkých písmen podle hodnot \uccode.

Přesněji: první token parametru musí (po případné expanzi) být „{“. Následující řadu tokenů čte TEX bez expanze až po odpovídající „}“. Při tomto čtení konvertuje všechny tokeny typu „uspořádaná dvojice“ (ASCII kód, kategorie). Zaměří se na ASCII kódy a kategorie ponechá beze změny. Všem takovým tokenům s nenulovým \uccode změní ASCII kód podle jejich \uccode. Pak TEX odstraní obklopující závorky {...} a vrátí se na začátek (případně upraveného) ⟨*balanced text*⟩ znovu. V druhém průchodu již TEX provádí obvyklou úplnou expanzi. Viz též \lowercase.

Kn: 279, 41, 215, 217, 307, 345, 348, 374, 377, 394, **Ol:** 27_{71}, $43_{96–99}$, 44_{100}, 71, 318, $374_{387–388}$, 389, 402_{562}, 403.

\USenglish [csplain]
Číslo jazyka pro Americkou angličtinu. Viz soubor hyphen.lan.

```
801 \newcount\USenglish  \global\USenglish=0
```

\v ⟨*znak*⟩ [plain]
Umístí nad ⟨*znak*⟩ háček. Například \v n vede na „ň“.

```
802 \def\v#1{{\accent20 #1}}
```

V csplainu lze po použití makra \csaccents význam \v a dalších řídicích sekvencí pro akcenty předefinovat tak, že dvojice \v ⟨*znak*⟩ expanduje na odpovídající osmibitový kód akcentovaného znaku podle $\mathcal{CS}$-fontů.

Kn: 52, 356, **Ol:** 225, 336, $348_{227,\,235}$, 349, $374_{387–388}$, $418_{622–624}$, $436_{731–732}$, 451.

\vadjust ⟨*filler*⟩{⟨*vertical material*⟩} [h, m]
Do horizontálního seznamu se uloží odkaz na ⟨*vertical material*⟩, který se zařadí později do vnějšího vertikálního seznamu. Příslušný vertikální materiál se v okamžiku sestavení odstavce připojuje bezprostředně pod řádek, ve kterém byl odkaz zanesen. Odkaz musí být v řádku na „vnější úrovni“ horizontálního seznamu. Pokud je odkaz vložen do explicitně napsaného \hboxu, sazba ⟨*vertical material*⟩ se mlčky ztrácí. Nemusí se ztratit, pokud je příslušný \hbox „odpouzdřen“ v odstavcovém módu pomocí \unhbox.

Ve vertikálním módu způsobí povel \vadjust chybu. Při použití \vadjust v display módu se ⟨*vertical material*⟩ vloží těsně pod kompletovanou rovnici před \postdisplaypenalty. Standardní je použití \vadjust v odstavcovém nebo vnitřním matematickém módu. Pak se ⟨*vertical material*⟩ dostane pod ten řádek odstavce, ve kterém byl primitiv \vadjust použit.

V okamžiku, kdy se ⟨*vertical material*⟩ připojuje do vnějšího vertikálního seznamu pod řádek, ve kterém byla značka použita, není nijak měněno \prevdepth a tudíž tento materiál neovlivní mezi řádkovou mezeru. Nechť například je při sestavování odstavce do vertikálního seznamu zařazen řádek R_1 s hloubkou 2 pt a v něm je \vadjust obsahující třeba nějaký box. Tento box se připojí bez mezery pod řádek R_1. Pak následuje řádek R_2, před který se připojí mezi řádková mezera, jako by předcházel box s hloubkou 2 pt.

Objeví-li se více odkazů typu \vadjust v jednom řádku, jsou příslušné vertikální materiály připojeny pod řádek postupně pod sebou ve stejném pořadí, v jakém jsou odkazy v řádku čteny zleva doprava. Tyto značky se dále mohou kombinovat s odkazy typu \insert a \mark, které se chovají analogicky, jako odkazy na materiál z \vadjust.

Uvedeme příklad, ve kterém využijeme \vadjust pro jednoduchou poznámku na okraj.

```
803 \def\okraj#1{\setbox0=\line{\hfil\rlap{%
804       \quad\vtop{\hsize=2cm\noindent\raggedright#1}}}%
805    \ht0=-3.5pt\dp0=3.5pt \strut\vadjust{\box0}}
```

ix Uživatel napíše někde v odstavci třeba \okraj{ix} a na okraji se objeví obsah argumentu, jako to vidíme v tomto odstavci. Okrajová poznámka je ve stejné výšce, jako řádek, ve kterém bylo makro \okraj použito. Pokud chceme mít poznámku na levém okraji (jako v tomto odstavci), použijme \llap místo \rlap, tj.: \line{\llap{\vtop{...}\quad}\hfil}.

V makru jsme vytvořili box šířky \hsize z něhož doprava vyčnívá text poznámky (\rlap). Tento box bude vložen pod aktuální řádek pomocí \vadjust. Aby ve vnějším vertikálním seznamu nepřekážel, upravili jsme tomuto boxu výšku a hloubku tak, aby celková výška byla nulová. Do místa použití makra vkládáme \strut hloubky 3,5 pt, abychom měli jistotu, že řádek bude mít uvedenou hloubku. Pak stačí „vyzvednout“ box s poznámkou o stejnou velikost nahoru, aby účaří řádku a poznámky bylo společné.

Kn: 281, 95, 105, 109–110, 117, 259, 317, 393, 454, **Ol:** 88, 145, 229, 258, 280, 282_{581}, 380, 391.

\valign ⟨*box specification*⟩{⟨*alignment material*⟩} [h]
Transponovaná \halign. Sestavení sloupců (které jsou ze zdrojového textu čteny po řádcích) tak, aby jednotlivé části těchto sloupců lícovaly do řádků podle úvodní specifikace.

Kn: 285–286, 249, 283, 302, 335, 397, **Ol:** *142*, 72, 89, 101, $143_{270,278}$, 317, 332, 398, 411, 436.

\vbadness (plain: 1000) [integer]
Maximální hodnota badness, při níž TEX ještě nekřičí Underfull \vbox. Viz též \hbadness.

Kn: 272, 348, 397, 417, **Ol:** 101, 257.

`\vbox` ⟨*box specification*⟩`{`⟨*vertical material*⟩`}` [h, v, m]
Sestaví se box s vertikálním seznamem.

Podrobněji: údaj ⟨*box specification*⟩ si TEX uloží do zásobníku a použije jej až při závěrečné kompletaci boxu. Pak v místě závorky „{“ otevře novou skupinu. Hlavní procesor přejde do vnitřního vertikálního módu v němž začne sestavovat vertikální seznam. Nejprve se expanduje `\everyvbox`. Pak TEX čte povely z ⟨*vertical material*⟩. Po dosažení koncové závorky „}“, která uzavírá skupinu otevřenou na začátku sestavování boxu, provede TEX kompletaci boxu. Tato kompletace závisí na ⟨*box specification*⟩ a podrobněji o ní viz sekci 3.5. Hlavní procesor se pak vrací do módu, ve kterém byl před zahájením zpracování boxu. V rámci tohoto módu se kompletovaný box klade do příslušného seznamu jako jeden element seznamu. Ve vertikálním módu se před box automaticky připojí meziřádková mezera. Pokud ale předchází `\setbox`, kompletovaný box se neklade do tiskového seznamu, ale ukládá se do specifikovaného registru.

Kn: 65, 80–82, 103, 151, 193, 222, 278, 388–389, **Ol:** *82*, *sekce 3.5*. Průběžně se vyskytuje na mnoha stránkách knihy.

`\vcenter` ⟨*box specification*⟩`{`⟨*vertical material*⟩`}` [m]
Vytvoří se atom typu Vcent, v jehož základu je:

`\vbox` ⟨*box specification*⟩`{`⟨*vertical material*⟩`}`.

Při konverzi do horizontálního seznamu se výška a hloubka uvedeného `\vbox`u upraví následovně: (1) celková výška boxu zůstane zachována, (2) výsledný box bude vertikálně centrován podle matematické osy v rámci své celkové výšky.

Kn: 290, 443, 150–151, 159, 170, 193, 222, 242, 361, **Ol:** 91, 93, 100, 102, 134_{207}, 146, 153, 189_{328}, 193–194, 204_{457}, 206, 342_{199}, 343, 345_{215}, 346_{216}, 359_{293}, $394_{475,\,479}$.

`\vfil` [v]
Primitivní ekvivalent k povelu `\vskip 0pt plus 1fil`, tj. vloží do vertikálního seznamu mezeru s hodnotou roztažení prvního řádu.

Kn: 72, 71, 111, 256, 281, 286, 417, **Ol:** 14_{20}, 87, 90, 115_{243}, 116_{8}, 117_{12}, 122_{57}, 123_{97}, $143_{270–271}$, $244_{167,\,172}$, 245_{194}, 246, 252_{233}, 270_{398}, 271, $272_{419,\,425}$, 274_{457}, 276_{462}, 277_{493}, 279_{529}, 324, 364_{313}, 384, 423.

`\vfill` [v]
Primitivní ekvivalent k `\vskip 0pt plus 1fill`, tj. vloží do horizontálního seznamu mezeru s hodnotou roztažení druhého řádu.

Kn: 72, 24–25, 71, 256–257, 281, 286, **Ol:** 90, 239, 253–254, 259_{279}, 262_{309}, 264, 324, 345_{208}, 357, 384.

`\vfilneg` [v]
Primitivní ekvivalent k `\vskip 0pt plus -1fil`, tj. stornuje případný `\vfil`.

Kn: 72, 111, 281, 286, **Ol:** 90, 324, 364$_{313}$, 384.

\vfootnote ⟨*značka*⟩{⟨*text poznámky*⟩} [plain]
Poznámka pod čarou bez sazby značky. Makro se hodí, pokud klademe značku do vnitřního boxu, kde nelze přímo použít makro \footnote, protože tam nefunguje primitiv \insert. Do boxu tedy zapíšeme jenom značku a těsně za box napíšeme \vfootnote, což je makro, které expanduje na primitiv \insert. Kód makra je uveden na straně 251.

Kn: 117, 363, **Ol:** 251$_{208–209}$, 380.

\vfuzz (plain: 0,1 pt) [dimen]
Velikost přesazení přes stanovenou výšku \vboxu, která se toleruje, místo aby se ohlásilo `Overfull \vbox`.

Kn: 274, 348, **Ol:** 101.

\vglue ⟨*glue*⟩ [plain]
Makro vytvoří mezeru ve vertikálním seznamu, která nezmizí ve stránkovém zlomu. Navíc nemění hodnotu \prevdepth, takže meziřádková mezera zůstane zachována. Viz též \hglue a \topglue.

```
\def\vglue{\afterassignment\vgl@\skip0=}
\def\vgl@{\par \dimen0=\prevdepth \hrule height0pt
  \nobreak\vskip\skip0 \prevdepth\dimen0}
```

Kn: 352, 408, **Ol:** 112, 113$_{226}$, 279$_{534}$, 371, 443$_{774}$.

\voffset (iniTEX: 0 pt) [dimen]
Určuje vertikální usazení sazby při výstupu do `dvi` souboru. Podrobněji, viz heslo \shipout. Viz též \hoffset.

Kn: 251, 274, 342, **Ol:** 123$_{85}$, 372, 431.

\vphantom {⟨*sazba*⟩} [plain]
Vytvoří prázdný box nulové šířky, který má výšku a hloubku stejnou, jako výška a hloubka materiálu v parametru ⟨*sazba*⟩. Je-li ⟨*sazba*⟩ zapsaná jako jeden token, není nutno uvádět závorky kolem. Například `\vphantom g` vytvoří prázdný box nulové šířky s výškou a hloubkou podle písmene g. Makro pracuje v horizontálním i matematickém módu. Kód makra, viz stranu 418.

Kn: 178–179, 211, 321, 360, **Ol:** 191, 193$_{401}$, 394$_{474}$, 418$_{622}$.

\vrule ⟨*rule specification*⟩ [h]
Obdélník, který má implicitní výšku a hloubku závislou na výsledných rozměrech \hboxu, v němž je použit. Implicitní šířka je 0,4 pt. Všechny tři údaje se dají explicitně formulovat pomocí řídicích slov `width`, `height` a `depth` v ⟨*rule specification*⟩.

`\vsize` (plain: 8,9 in, csplain: 239,2 mm) [dimen]
Výška tiskového zrcadla. Na začátku každé stránky se tato hodnota kopíruje do `\pagegoal` a ovlivní algoritmus stránkového zlomu S touto hodnotou pracuje obvykle též výstupní rutina. Proto by uživatel neměl v průběhu sazby dokumentu tuto hodnotu měnit, aniž by přesně věděl, co dělá.

`\vskip` ⟨*glue*⟩ [v]
Vloží vertikální mezeru typu ⟨*glue*⟩ požadované velikosti. Je-li povel použit v odstavcovém módu, nejprve se vloží token `par` a pak je povel čten znova. V matematickém a vnitřním horizontálním módu způsobí povel chybu.

`\vsplit` ⟨*8-bit number*⟩`to`⟨*dimen*⟩ [h, v, m]
Vrátí `\vbox` výšky ⟨*dimen*⟩, který obsahuje horní část `\vbox`u uloženého v ⟨*8-bit number*⟩. Přenesenou část z výchozího boxu vymaže. Do výšky se samozřejmě nezapočítává hloubka posledního elementu. Často se povel `\vsplit` používá v kontextu:

```
809  \setbox⟨newbox⟩=\vsplit⟨oldbox⟩to⟨dimen⟩
```

Hledá se tedy stránkový zlom uvnitř boxů. Box ⟨*newbox*⟩ bude obsahovat počáteční úsek materiálu z ⟨*oldbox*⟩, který nazveme „odlomený materiál“. Původní ⟨*oldbox*⟩ bude mít materiál zmenšen o „odlomený materiál“ a tento zbylý materiál pojmenujeme slovem „zbytek“.

Povel `\vsplit` hledá v materiálu boxu ⟨*8-bit number*⟩ (tj. ⟨*oldbox*⟩) takové místo stránkového zlomu, aby „odlomený materiál“ měl při požadavku na výšku stránky ⟨*dimen*⟩ nejmenší cenu zlomu. Pracuje shodně jako algoritmus stránkového zlomu (sekce 6.6), ovšem bez manipulace s inserty (sekce 6.7).

„Odlomený materiál“ i „zbytek“ mohou po operaci zůstat prázdné. Například není-li v původním materiálu ⟨*oldbox*⟩ žádné povolené místo zlomu, povel zařadí vše do „odlomeného materiálu“ a „zbytek“ zůstává prázdný. Druhým extrémem je případ, kdy nejmenší cena zlomu vychází hned pro první element v původním materiálu. Pak je „odlomený materiál“ prázdný a „zbytek“ už první element neobsahuje.

„Odlomený materiál“ může na začátku obsahovat odstranitelné elementy. Zde je tedy rozdíl proti mechanismu vkládání elementů do aktuální strany a ignorování odstranitelných elementů na začátku. Výsledný \vbox je kompletován jako

810 \vbox to⟨*dimen*⟩{⟨*odlomený materiál*⟩}

To může při malých či dokonce nulových hodnotách stažení nebo roztažení přítomných mezer a při neúspěšném nalezení místa zlomu přesně podle požadovaného parametru ⟨*dimen*⟩ způsobit varování typu `Underfull \vbox` nebo `Overfull \vbox`.

Po provedené operaci \vsplit TEX ještě upraví „zbytek“ následujícím způsobem: (1) vymaže ze začátku „zbytku“ všechny odstranitelné elementy až po první neodstranitelný element. Zůstane-li „zbytek“ prázdný, ukončí úpravy. (2) Přeskočí všechny elementy ve „zbytku“, které nejsou linkami nebo boxy (například \mark). Dosáhne-li tímto způsobem konec materiálu, ukončí úpravy. (3) V předchozím bodu TEX dosáhl prvního boxu nebo linky ve „zbytku“. Před tento element nyní vloží mezeru typu ⟨*glue*⟩ z registru \splittopskip. Postupuje analogicky, jako při vkládání mezery na začátek aktuální strany z registru \topskip. Mezera má tedy přirozenou velikost zmenšenu o výšku následujícího boxu nebo linky. Pokud ale vychází tato velikost záporná, je vložena mezera nulové velikosti.

Povel \vsplit ohlásí chybu, pokud není ⟨*8-bit number*⟩ \vbox (tedy box s *vertikálním* materiálem. Analogický primitiv \hsplit v TEXu neexistuje. Viz též \splitbotmark, \splitfirstmark a \splitmaxdepth.

Kn: 124, 222, 259, 278, 397, 417, **Ol:** 113, 209, 244_{167}, 245_{185}, 249–250, $251_{212–213}$, 253, 256, 273, 274_{444}, 275, $277_{494–495}$, 317, 319_{17}, 391, 437.

\vss [v]

Primitivní ekvivalent k `\vskip 0pt plus 1fil minus 1fil`, tj. vloží mezeru s hodnotou stažení i roztažení prvního řádu. Protože má mezera hodnotu stažení prvního řádu, nelze ji použít v hlavním vertikálním módu, který podléhá stránkovému zlomu.

Kn: 72, 71, 281, 286, 255, **Ol:** $82_{99–100}$, 87, 90, 92_{158}, 115_{243}, 116, 139_{235}, 141_{255}, 243_{147}, 262_{306}, 267_{333}, 281_{565}, 324, 339_{155}, 384.

\vtop ⟨*box specification*⟩{⟨*vertical material*⟩} [h, v, m]

Sestaví vertikální box jako \vbox, ale referenční bod je na účaří prvního řádku boxu.

Kn: 81–82, 151, 222, 278, 333, **Ol:** 87–88, 91, 93, 100, 102, 184_{239}, 277_{497}, 279_{536}, $280_{549, 554}$, $281_{568–569, 574–575}$, 319_{18}, $408_{584–585}$, 410_{601}, 429, 452_{804}.

\wd ⟨*8-bit number*⟩ *restricted* [dimen]

Umožňuje přístup k hodnotě „šířka boxu“, která odpovídá boxu z registru ⟨*8-bit number*⟩.

Kn: 388–389, 120, 271, 391, 417, **Ol:** 82$_{90–91}$, 83, 84$_{110, 112}$, 95$_{170}$, 103$_{180, 182}$, 107$_{202}$, 128$_{154–155}$, 162$_{76}$, 214$_{11–14}$, 270$_{393–394}$, 342$_{187, 197}$, 381$_{421}$, 419$_{631}$.

\widowpenalty (plain: 150) [integer]
Hodnotu tohoto registru TEX přičítá k celkové penaltě, která se připojuje za předposledním řádkem odstavce. Vysoká hodnota tohoto registru potlačí výskyt tzv. „vdov“, což je osamělý poslední řádek odstavce na začátku strany. Viz též \clubpenalty, \displaywidowpenalty.

Kn: 104, 113, 272, 348, **Ol:** 229, 243$_{144}$, 258, 265, 267$_{327}$, 344, 348, 354, 416.

\wlog {⟨*text*⟩} [plain]
Zapíše ⟨*text*⟩ (po expanzi) do souboru log, ale nikoli na terminál.

```
811  \def\wlog{\immediate\write-1 } % write on log file (only)
```

Kn: 347, **Ol:** 400$_{537}$, 401$_{548}$, 457.

\write ⟨*number*⟩⟨*filler*⟩{⟨*balanced text*⟩} [p,exp]
Do paměti uloží v neexpandovaném tvaru ⟨*balanced text*⟩ a do sazby vloží neviditelnou značku s odkazem na místo v paměti. Teprve později, když povel \shipout ukládá stranu sazby do dvi a objeví se tam zmíněná značka, provede se zápis do souboru způsobem popsaným níže. Pokud nechceme mezi použitím \write a skutečným zápisem do souboru využít toto zdržení, pišme před primitiv \write prefix \immediate. Pak se provede zápis okamžitě a ne až při činnosti \shipout.

Před zápisem by měl být číslu ⟨*number*⟩ přiřazen fyzický soubor systému pomocí \openout (říkáme, že soubor je otevřen k zápisu). Číslo otevřeného souboru musí být v intervalu ⟨0, 15⟩. Je-li ⟨*number*⟩ mimo tento interval nebo pokud není otevření souboru uskutečněno nebo neproběhlo-li úspěšně, zápis se provede do souboru log a při kladném ⟨*number*⟩ též na terminál. Pro alokaci čísel ⟨*number*⟩ se obvykle používá makro \newwrite.

Každý zápis do souboru začíná na novém řádku. Při zápisu se ⟨*balanced text*⟩ zcela expanduje až na případné řídicí sekvence, které již nelze expandovat. Jména těchto řídicích sekvencí se zapíšou ve stejném formátu, jako by byl před nimi použit primitiv \string. Objeví-li se při zápisu znak s ASCII hodnotou \newlinechar, zápis pokračuje na dalším řádku.

Například makro LaTeXu \typeout:

```
812  \def\typeout{\immediate\write16 }
```

zapíše zprávu na terminál a do souboru log podobně, jako to dělá \message. Rozdíl je pouze v tom, že \typeout navíc zahajuje každou zprávu na novém řádku a pracuje s hodnotou \newlinechar. Srovnejte též makro \wlog.

Kn: 280, 215–216, 226–228, 254, 346, 377, 422, 424, **Ol:** *56, 288*, *sekce 7.1*, 18, 56–57, 71–72, 76, 87–88, 145, 208, 266, 280, 284, 288–293, 318, 347, 361–362, 378, 400, 404, 408, 410, 431, 457.

\xdef ⟨*control sequence*⟩⟨*parameter text*⟩{⟨*balanced text*⟩} *global* [a]
Synonymum pro \global\edef.

Kn: 275, 215–216, 373, 418, 424, **Ol:** 31–32, 40, 65–66, 71, 207_{477}, 260_{282}, $287_{2,\,4–5,\,8}$, 351, 411.

\xleaders [h, v, m]
Jako \cleaders, ovšem vyplňující mezera mezi opakovanými boxy se nevkládá jen před první a za poslední box, ale rozpočítá se stejným dílem a vloží i mezi jednotlivé opakované boxy.

Kn: 224, **Ol:** *117*, 118_{19}, 319, 324, 384.

\xspaceskip (iniTEX: 0 pt) [glue]
Mezislovní mezera, která se použije právě tehdy, když je tento parametr nenulový a současně je \spacefactor větší nebo roven 2 000. Je-li tento parametr nulový (implicitní hodnota), pak se při \spacefactor $\geq$ 2 000 použije pro mezislovní mezeru hodnota definovaná v použitém fontu.

Kn: 76, 274, 356, 429, 433, 317, **Ol:** 106, 230, 251_{216}, 424_{673}, 436.

\year [integer]
Rok. Údaj je v okamžiku spuštění TEXu načten ze systémové proměnné data a času.

Kn: 41, 273, 349, 406, **Ol:** není nikdy použito.

4. Seznam příkladů použitých v knize

V knize jsou na číslovaných řádcích příklady dvou druhů. Za prvé se jedná o příklady bez velkého praktického významu, které především ilustrují určitý algoritmus TEXu. Za druhé jsou zde příklady, které navíc mají praktickou použitelnost. Ty druhé jsou v tomto rejstříku abecedně seřazeny podle slova, které je nejvíc charakterizuje. Pokud si například čtenář po přečtení části A matně vzpomíná, že tam viděl příklad na zpětné reference, ale už neví, v které části knihy ho hledat, může použít tento rejstřík. Hledaný příklad najde pod heslem *Reference.*

Kódy maker plainu jsou k vyhledání ve slovníku primitivů a maker. Proto na ně nejsou v tomto rejstříku odkazy. Zbývají zde tedy ukázky, které jsem téměř ve všech případech navrhoval sám. Připouštím, že jiná a lepší řešení jsou jistě možná...

5. Literatura

Zde jsou uvedeny jen tituly, na které někde odkazujeme v textu. V závorce na konci citace je proto uvedeno, na které straně se o dané knížce mluví. Podstatně rozsáhlejší seznam literatury je obsažen například v [7].

[1] Stephan von Bechtolsheim. *TEX in practice*. Volume I–IV. Springer Verlag, 1993 (264).

[2] Michael Doob. *A Gentle Introduction to TEX*, 1990. Text je volně přístupný v CTAN archívu v souboru `gentle.tex` (462).

[3] Michael Doob. *Jemný úvod do TEXu*, CSTUG, Praha 1993. Třetí (upravené) vydání českého překladu [2] (5, 144).

[4] Donald E. Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-13447-0 (5, 10, 136, 149, 174, 181, 206, 212, 264, 283, 316, 333, 439, 463).

[5] Donald E. Knuth. *TEX: The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-13437-3 (301).

[6] Michaela Lichá — Oldřich Ulrych. AMS-TEX verze 2.1. CSTUG, Praha 1992 (144).

[7] Petr Olšák. *Typografický systém TEX*, CSTUG 1995. ISBN 80-901950-0-8 (5, 17, 234, 280, 303, 462).

[8] Jiří Rybička. *LATEX pro začátečníky*. Konvoj, Brno 1995 (144).

[9] Stanislav Brabec. Lunisolární výpočty v TEXu. *Zpravodaj CSTUGu*, strany 16–18, No. 1–4/1995 (331).

6. Rejstřík

Tento rejstřík je jednak skutečně rejstříkem v tom smyslu, že hesla odkazují na jednotlivé stránky, kde je pojem vysvětlen, ale navíc to je anglicko-český i česko-anglický slovníček dohromady. Když si například pomocí `\showlists` necháte vypsat jednotlivé tiskové materiály a TEX prozradí, že něco má třeba v „current page“, můžete v rejstříku vyhledat pod písmenem C heslo „(current page) aktuální strana“. Pojem v závorce je skutečně použit v TEXbooku a v protokolech programu TEX, zatímco termín, který není v závorce, je použit v této knize. Pokud naopak čtete ve slovníku primitivů výklad hesla `\pagetotal` a narazíte tam na neznámý pojem „aktuální strana“, najdete v rejstříku pod písmenem A termín „aktuální strana (current page)“. V obou případech stránky odkazují nejčastěji do části A, kde je pojem vysvětlen (matematik by řekl, kde je pojem definován). V místě definice pojmu je v textu termín vysázen kurzívou, zatímco kdekoli jinde je už sázen běžným antikvovým řezem. Jestliže používáme v knize stejný termín, jako v TEXbooku (token, atom), najdete samozřejmě v rejstříku pojem jen jednou a bez závorky.

Petr Olšák
TEXbook naruby

Verze textu: 7. 12. 2000

Z formátu PDF se nedoporučuje tisknout

Grafická úprava, sazba, kresba na obálce — Petr Olšák
Sazba z písma Computer Modern ve variantě $\mathcal{CS}$-font

Adresa nakladatelství:
KONVOJ, spol. s r. o., Berkova 22, 612 00 Brno.
http://www.konvoj.cz *e-mail:* konvoj@konvoj.cz

ISBN 80-7302-007-6 (2. vyd.)
ISBN 80-85615-64-9 (1. vyd.)