On a less superficial level, TEX is a very sophisticated program, first of all because of the ingeniousness of its built-in algorithms for such things as paragraph breaking and make-up of mathematical formulas, and second because of its almost complete programmability.

Victor Eijkhout: *TEX by Topic*, Addison–Wesley Publishers Ltd. 1992.

Kapitola sedmá: Programujeme v TEXu

Petr Olšák

O TEXu se často říká, že je příliš složitý. Příklady, které jsem si připravil do této kapitoly, mohou čtenáře v tomto názoru jenom podpořit. Uvědomme si ale, že ne všichni lidé, kteří s TEXem pracují, potřebují v něm též programovat. Myslím si, že většině uživatelů TEXu stačí zvládnout několik příkazů (např. z LATEXu) a jsou schopni uspokojivým způsobem psát své dokumenty. Mám zkušenosti, že na této úrovni jsou schopny se naučit TEX i administrativní pracovnice, které přitom vykonávají i jinou činnost. Na pracovišti, kde se TEX používá, většinou stačí, když je přítomný jeden člověk, který tomu rozumí i z hlediska tvorby maker (tj. programování). Takováto dělba práce je přímo ideální. Pokud třeba potřebujete zařídit v TEXu něco nestandardního, zavoláte si tohoto „kouzelníka", ten napíše pár řádků maker na začátek Vašeho dokumentu a vše běží podle Vašich představ. Není-li kouzelník po ruce, stačí se poohlédnout po archívech TEXovských maker (stylů). Je velice pravděpodobné, že v nich najdete to, co budete potřebovat.

V této kapitole se tedy budeme zabývat vlastností TEXu, se kterou se asi 90% uživatelů TEXu nikdy nesetká. Čtenář může z tohoto textu získat jakousi ucelenější představu o tom, co je to TEX. Rozhodně nechci nikoho zastrašovat ukázkami, z nichž je patrná poměrně velká složitost TEXu. Myslím si ale, že se najdou čtenáři, kteří mají zkušenosti s jinými programovacími jazyky a uvítají zde možnost srovnání.

Programovací jazyk TEXu je podobný jazyku LISP nebo FORTH. Existuje zde jistá skupina základních kamenů (tzv. TEX-primitivů – je jich přes tři stovky), pomocí nichž se sestavují větší a složitější celky, pomocí nichž se sestavují ještě větší celky atd. Například pro sazbu loga „TEX" je definován příkaz `\TeX` takto:

```
\def\TeX{T\kern-.1667em\lower.5ex\hbox{E}\kern-.125emX}
```

Pokud nyní napíšu slovo `\TeX`, tento zápis se v daném místě rozvine (expanduje) na sekvenci příkazů nižší úrovně: `T\kern-.1667em\lower.5ex\hbox{E}\kern-.125emX`. Pokud by nějaký z těchto příkazů nebyl primitivní, hned by se také rozvinul podle své definice. Tato expanze končí až na úrovni primitivů.

Zastavme se na chvíli u tohoto příkladu. Příkazy (`\kern`, `\lower`, `\hbox`) už jsou primitivní, takže se dále neexpandují. Znamenají toto: Vysadí se písmeno T, za ním se vloží záporná mezera (`\kern`) o velikosti -0.1667 `em`, dále se vysadí box (`\hbox`) obsahující písmeno E, přičemž tento box je snížený (`\lower`) o 0.5 `ex`, pak se zařadí záporná mezera o velikosti -0.125 `em` a nakonec se vysadí písmeno X. Jednotky `em` a `ex` jsou závislé na použité velikosti písma. `1em` znamená velikost stupně písma (například toto písmo je desetibodové) a `1ex` je výška malého písmene `x` v daném písmu. Rozměry jsou uvedeny v těchto jednotkách proto, aby příkaz `\TeX` fungoval pro libovolné písmo v libovolné velikosti.

Zde je snad trochu patrná síla makrojazyka TEXu. Já jako uživatel vůbec nemusím vědět, jak je definován příkaz `\TeX`, mě pouze stačí, že vím, že tento příkaz způsobí sazbu loga TEX. Přece se jako uživatel nebudu zabývat nějakými numery, která způsobí, že písmena v logu na sebe esteticky navazují! Je zajímavé pozorovat, že pokud se o TEXu mluví v článcích, které jsou sázeny jinými DTP systémy, většinou logo TEX dopadne katastrofálně. Je to proto, že tam není toto logo tak precizně a s takovou přesností definované (stačí se podívat do obsahu tohoto časopisu).

Použití TEX-primitivů má přesně definovanou syntaxi, zatímco syntaxe použití nově definovaných příkazů záleží na programátorovi. Tuto syntaxi lze velice pružně přizpůsobit momentální potřebě. Představme si, že máme výstup z databáze, z něhož bychom chtěli tisknout třeba vysvědčení. Nechť například databázový výstup vypadá takto:

```
>třída: 3.A
>jméno: Ferdinand Mravenec
>adresa: Ondřeje Sekory 123, Praha 11
>narozen: 1.2.1934
>známky: 1 1 2 2 3 1 2 1 - 1 1 - 1 1 1 1
... atd.
```

Upozorňuji na to, že formulář vysvědčení jsem naposledy viděl před deseti lety, takže si nepamatuji, které údaje jsou potřeba. Matně si vzpomínám, že jich bude asi víc, ale pro ukázku by nám tyto údaje mohly stačit.

Umíte-li programovací jazyk TEXu, pak jste na nejlepší cestě vyplňování formulářů vysvědčení mechanizovat použitím tohoto programu. Stačí napsat soubor definic příkazů `>jméno`, `>známky` apod. a do nich zanést typ použitého fontu pro danou rubriku, umístění rubriky na stránce, konverzi známek do slovního vyjádření (výborný, chvalitebný, apod.) a případné proškrtnutí neabsolvovaného předmětu. Pokud se jedná o umístění textů do přesně zvoleného místa na stránce, k tomu potřebujete měřítko a ukázkový arch vysvědčení. Údaje získané měřením z tohoto archu zapíšete do definic a pak provedete několik pokusů tisku. TeX samotný pracuje zcela přesně, takže by neměly být žádné problémy, ovšem zřejmě bude záležet na přesnosti vodicí dráhy papíru v tiskárně. V této souvislosti je třeba upozornit, že dokumenty takového významu by neměly být tištěny na jehličkových tiskárnách.

Ukázku těchto definic zde neuvedu, protože sám nemám formulář vysvědčení po ruce a nemohu tedy uvést reálný příklad. Zkoušel jsem si to jen teoreticky a definice vyšly zhruba na třicet řádků. Tyto definice je vhodné mít ve zvláštním souboru, z něhož se pak volá soubor, který je výstupem z databáze a vstupem do TeXu. V těchto definicích se tedy formuluje jednak nastavení režimu čtení souboru tak, aby to bylo v souladu se syntaxí vstupního souboru, jednak konečný vzhled tištěného dokumentu. Vytváří se vlastně jakési rozhraní mezi textovou reprezentací dat v počítači a koncovým tiskem. Podobným příkladem může být databázový výstup obsáhlého slovníku, který chceme TeXem zformátovat do knižní podoby (včetně např. plovoucích záhlaví podle hesel). Viděl jsem anglicko–český slovník středního rozsahu sázený TeXem, který je od starších slovníků sázených horkou sazbou k nerozeznání.

Ukažme si jiný příklad. Píšeme beletrii a naše písařka nerozlišuje uvozovky a píše otevírací i uzavírací uvozovky stejným způsobem, tedy `"takto"`. Na začátek dokumentu přidáme jeden řádek

```
\input czech.sty  \catcode`\"=13  \def"{\begingroup\clqq \def"{\crqq\endgroup}}
```

Tento řádek způsobí, že první výskyt znaku `"` bude sázen jako `\clqq` a druhý výskyt téhož znaku jako `\crqq`. Podívejme se na to podrobněji. Příkaz ``\catcode`\"=13`` způsobí, že znak `"` bude mít kategorii 13, což znamená, že to bude tzv. aktivní znak, který bude TeX interpretovat jako samostatný příkaz. Proto jej vzápětí můžeme definovat příkazem `\def`. Prozkoumejme, co se stane, máme-li například napsáno: `"české" uvozovky`.

První výskyt symbolu `"` se expanduje na sekvenci příkazů `\begingroup\clqq \def"{\crqq\endgroup}`, tj. otevře se skupina, v níž budou všechna přiřazení lokální (`\begingroup`), vysází se `\clqq` a předefinuje se symbol `"`. Proto se druhý výskyt symbolu `"` expanduje podle nové definice na `\crqq\endgroup`, což znamená, že se vysází `\crqq` a uzavře se skupina (`\endgroup`), takže od této chvíle jsou všechny definice, realizované uvnitř skupiny zapomenuté. Proto nyní znak `"` má původní význam, jaký měl před otevřením skupiny a další výskyt tohoto znaku způsobí tutéž akci, jako při prvním výskytu.

Příkazy `\clqq` a `\crqq` jsou definovány ve stylu `czech.sty` a způsobí tisk české levé a pravé dvojité uvozovky. Zde je patrná jistá „strukturovanost“ našeho programování. Nemusíme totiž při řešení našeho problému zkoumat, jak jsou tyto příkazy definovány. Tyto definice udělal někdo jiný a my je jako programátoři vyšší úrovně pouze použijeme. Definice těchto příkazů nejsou zcela jednoduché, protože se v nich mimo jiné testuje, zda je povoleno dělení slov uzavřených do uvozovek či nikoli a podle toho se jistým způsobem ovlivní chování algoritmu pro dělení slov.

Uvedené řešení uvozovek má ještě jednu výhodu. Pokud není v dokumentu sudý počet uvozovek, TeX vypíše varování, že jeho činnost končí uvnitř skupiny. Navíc skupina otevřená příkazem `\begingroup` se nesmí křížit s běžnými skupinami označenými znaky `{` a `}`, takže se dá velmi často překlep související s uvozovkami odhalit.

Ukažme si nyní, jak se v TeXu realizuje větvení. Názorná je třeba definice příkazu `\today`, kterou najdete ve stylu `czech.sty`. Tento příkaz vypíše aktuální datum a je definován takto:

```
\def\today{\number\day.~\ifcase\month\or ledna\or února\or března\or dubna\or května\or
  června\or ... \or listopadu\or prosince\fi \space\number\year}
```

Primitivní registry `\day`, `\month` a `\year` obsahují současný den, měsíc a rok. Tyto údaje načte TeX ze systémových proměnných při svém startu. Příkaz `\number\day` způsobí sazbu čísla `\day` v desítkové soustavě. Pak následuje tečka a za ní je mezera, v níž není povolen řádkový zlom (`~`). Dále se sazba větví na dvanáct možností podle hodnoty registru `\month` (viz `\ifcase`). Poslední alternativa je ukončena příkazem `\fi`. Nakonec se vysází mezera (`\space`) a za ní rok (`\number\year`).

Chcete sázet rok ve zkrácené formě (tj. nikoli 1993 ale jen 93) ? Pak definujte

```
\def\ignorujprvnidruhou#1#2{}
\def\rok{\expandafter\ignorujprvnidruhou\number\year}
```

Zde příkaz `\ignorujprvnidruhou` „spolkne“ první dva objekty, které jsou za ním napsány. Tyto objekty budou totiž interpretovány jako parametry příkazu (viz `#1` a `#2`), ovšem příkaz samotný tyto parametry nikdy nepoužije (je definován jako prázdný příkaz se dvěma parametry). Dále se věnujme definici vlastního příkazu `\rok`. Bez použití primitivního příkazu `\expandafter` by příkaz `\ignorujprvnidruhou` „spolkl“ objekty `\number` a `\year`, takže bychom na výstupu nedostali nic. Použití příkazu `\expandafter` způsobí, že se nejprve provede primitiv `\number`, který si vezme parametr `\year` a vrátí řadu číslic: `1993` a poté teprve příkaz `\ignorujprvnidruhou` „spolkne“ číslici `1` a `9` a výsledkem příkazu `\rok` je tedy dvojčíslí `93`. Nyní už asi tušíte, jak byste si předefinovali příkaz `\today` tak, aby pracoval s nově definovaným příkazem `\rok`.

Zde je vidět další rys tohoto programování. Změníte-li některý příkaz na nižší úrovni (například `\today`), pak se nemusíte starat o příkazy vyšší, které tento příkaz používají (například příkaz pro tisk hlavičky dopisu). Vaše změna se do všech těchto vyšších příkazů automaticky promítne.

A na závěr ještě jeden malinko rozsáhlejší příklad. Máte tiskárnu, kterou jste ještě nenaučili česky? Místo akcentovaných písmen sype v textovém režimu jen nějaké záhadné znaky? Podařilo se Vám nakonfigurovat CSTeX tak, aby s touto tiskárnou spolupracoval? Pak už nemáte žádné problémy. Tisk textových souborů můžete realizovat prostřednictvím TeXu (v grafice, tj. na jehličkových tiskárnách pomaleji). Přiznávám, že uvedený problém lze řešit i v mnoha velmi jednoduchých editorech, takže použití TeXu pro tyto účely se zdá poněkud přehnané, ale na druhé straně si na tomto příkladě ukážeme některé další možnosti programovacího jazyka TeXu. Navíc je zde zahrnuta jedna funkce, kterou v obyčejných editorech nenajdete. Jde o to, že při výpisu listingů programů nás často irituje, že je stránka zlomená ve velmi nevhodném místě. Listing máme většinou rozčleněný prázdnými řádky na jednotlivé celky a tyto celky se nám podle zákona schválnosti rozpadnou do více stránek. V naší ukázce je proto proveden test, zda náhodou netiskneme prázdný řádek. Pokud ano, pak se v rozsahu plus 3 řádky provede pokus o preventivní stránkový zlom. Prázdný řádek se samozřejmě nikdy netiskne na začátek nové stránky.

Následuje text souboru `print.tex`, který obsahuje potřebné makro.

```
% Tisk zvoleného souboru jako "verbatim". Použijte plainTeX.

\newread\inputstream \newlinechar=`\@

\def\listing#1{\begingroup \def\par{\endline\begline}%
  \frenchspacing \raggedbottom \setverb \obeylines \tt
  \begline \input#1 \egroup \endgroup}
\def\begline{\setbox0=\hbox\bgroup}
\def\endline{\egroup
  \ifdim\wd0=0pt \vskip3\baselineskip \penalty-200 \vskip-2\baselineskip
  \else \box0 \fi}
\def\setverb{\def\do##1{\catcode`##1=12}\dospecials\obeyspaces}

\def\readfilename{\message{Napište název souboru pro tisk: }
  \catcode`\\=12 \read-1 to\file \catcode`\\=0
  \immediate\openin\inputstream=\file
  \ifeof\inputstream \message{Soubor \file\space neexistuje !!@}\let\next=\readfilename
  \else \immediate\closein\inputstream \let\next=\relax \fi
  \next}

\readfilename   \listing\file
\end
```

Chceme-li nyní vytisknout na tiskárně třeba soubor `myprog.cpp`, pak stačí vyvolat plainTeX se vstupním souborem `print.tex`. Na terminálu se objeví:

```
Napište název souboru pro tisk: ␣
```

Napíšeme třeba `myprog.cpp` a odešleme. TeX zpracuje tento soubor jako „verbatim" (tj. tak jak leží a běží) v písmu `\tt` (strojopis). Vznikne soubor `print.dvi`, který pak vytiskneme na tiskárně.

Nyní se podíváme na to, jak to pracuje. Povšimněme si nejdříve předposledního řádku ukázky. Tam se příkazem `\readfilename` realizuje komunikace s uživatelem, jejíž výsledkem je text (název souboru) uložený v proměnné `\file`. Pak se příkazem `\listing\file` provede vlastní tisk souboru ve „verbatim" režimu. Zaměříme se nejprve na komunikaci s uživatelem, tj. na definici příkazu `\readfilename` (viz `\def\readfilename`).

Nejprve se příkazem `\message` vypíše na terminál příslušná zpráva. Pak se příkazem `\read-1 to\file` uloží do proměnné `\file` text, který odpověděl uživatel. Zde ovšem se věc malinko komplikuje. Pokud uživatel například odpoví `d:\work\myprog.cpp`, pak při pozdější expanzi tohoto textu TeX prohlásí, že příkaz `\work` ani `\myprog` není definován. Tento problém můžeme obejít tak, že vysvětlíme uživateli, aby při komunikaci s TeXem použil pro separátory mezi adresáři znak `/` a nikoli znak `\`, který má v TeXu implicite speciální funkci. Ovšem uživatel DOSu, pokud se zrovna neučí UNIX, nebude chtít takovou věc akceptovat. Proto je v makru při čtení uživatelské odpovědi na chvíli vypnutá řídicí funkce znaku `\`. Příkaz `\catcode`\\=12` znamená, že backslash bude mít kategorii 12, což je kategorie pro obyčejné znaky. Hned po načtení vstupního řetězce vrátíme znaku backslash původní kategorii (0). V tuto chvíli je už vstupní řetězec (např. `d:\work...`) interpretován tak, jak potřebujeme.

Dále se příkazem `\immediate\openin\inputstream=\file` testuje, zda soubor `\file` lze otevřít ke čtení. Příkaz znamená zhruba toto: přiřaď do proměnné `\inputstream` odkaz na fyzický soubor `\file` a otevři tento soubor ke čtení. Tuto akci proveď okamžitě (`\immediate`), tj. nečekej až na zalomení další stránky. Všimněte si, že proměnná `\inputstream` je nejprve „deklarovaná“ jako proměnná typu „vstupní soubor“ (příkaz `\newread`). Pokud je příslušný soubor prázdný, nebo nejde otevřít ke čtení, pak test `\ifeof\inputstream` vrací pravdivou hodnotu. V takovém případě se vypíše odpovídající hlášení (příkaz `\message`) a do příkazu `\next` se dosadí význam příkazu `\readfilename` (viz `\let`). V opačném případě (soubor existuje) se provedou příkazy mezi `\else` a `\fi`. Zde se praví toto: zavři soubor `\inputstream` a do příkazu `\next` ulož význam příkazu `\relax` (nic nedělej, relaxuj). Nakonec se provede vlastní příkaz `\next`, tj. v případě neúspěšného otevření souboru se znovu opakuje příkaz `\readfilename`, zatímco v případě úspěšného otevření souboru se neprovede nic a „cyklus“ se ukončí.

Toto je nejběžnější mechanismus, jakým lze programovat cyklus typu `repeat...until...` Vypadá to spíš jako rekurze (ta je samozřejmě v TeXu taky možná), ale v tomto případě se jedná skutečně o cyklus. Dále se můžete pozastavit nad tím, proč je hned po úspěšném otevření souboru tento soubor zavřen. My ho totiž dále nebudeme načítat primitivem `\read`, ale použijeme příkaz `\input`, který tento soubor znovu otevře ke čtení. Mohli jsme si makro podstatně zjednodušit a načítat požadovaný soubor rovnou příkazem `\input`. Tím by ale nešel provést test na existenci otevíraného souboru a čtenář by přišel o názornou ukázku techniky programování cyklu.

Podívejme se nyní, jak je definován příkaz `\listing`. Tento příkaz má jeden parametr `#1` (viz `\def\listing#1`), do nějž je v okamžiku použití příkazu dosazena hodnota `\file` (viz volání příkazu `\listing\file`). Na začátku příkaz otevře skupinu (`\begingroup`), přičemž všechny definice a přiřazení budou lokální v rámci této skupiny. Přeskočme na chvíli lokální re-definici příkazu `\par` a zaměřme se na další příkazy. `\frenchspacing` nastaví rovnoměrné mezerování (to budeme při tisku strojopisem potřebovat). `\raggedbottom` potlačí problémy, při nichž se řádky pod sebou nevejdou přesně do tiskového zrcadla se stanovenou výškou (dole nechá případně malé prázdné místo). `\setverb` nastaví kategorie speciálních znaků na 12, tj. všechny znaky budou brány jako obyčejné, žádný z nich nebude mít speciální význam. Vše se bude pouze přepisovat do výstupu. Konečně příkaz `\obeylines` způsobí, že se konec každého řádku bude interpretovat jako `\par`.

Jak je tento konec řádku (`\par`) definován? Spustí se příkaz `\endline` (ukončení řádku) a vzápětí příkaz `\begline` (start nového řádku). Příkaz `\begline` nastavuje sazbu obsahu řádku nikoli do výstupu, ale do boxu s číslem 0 (viz `\setbox0=\hbox\bgroup`). Sestavení tohoto boxu je ukončeno příkazem `\egroup` (viz `\endline`). Dále se testuje, zda výsledný box nemá nulovou šířku, což odpovídá na vstupu prázdnému řádku. Pokud ano (test `\ifdim\wd0=0pt` vrátí pravdivou hodnotu), pak se provedou příkazy `\vskip...`. Zde se vloží vertikální mezera odpovídající výšce tří řádků a místo se označí jako vhodné pro stránkový zlom (`\penalty-200`). V případě, že se skutečně stránkový zlom uskuteční, pak následující `\vskip` nemá funkci. Jinak tento následující `\vskip` srovná výšku prázdné mezery na výšku jednoho řádku (tři bez dvou je jedna). Druhá možnost (neprázdný řádek – alternativa `\else`) vysází obsah boxu (`\box0`) a tím se řádek dostane do sazby.

Konečně se nastaví typ písma (`\tt`) a provede se vlastní `\input` souboru `#1`, tedy souboru `\file`. Současně probíhá sazba obsahu tohoto souboru podle speciálního režimu, který byl předtím nastaven a který jsme popsali v předchozích odstavcích. Po ukončení zpracování tohoto souboru se ukončí skupina (`\endgroup`), takže všechna lokální speciální nastavení přestanou platit a režim práce TeXu se vrací do původních kolejí. Příkaz `\listing` se tedy dá použít i uvnitř dokumentu pro přechodné přepnutí do režimu „verbatim“.

Definici příkazu `\setverb` (viz `\def\setverb`), který přiřadí všem speciálním znakům kategorii obyčejného znaku (12), nebudu pro stručnost rozebírat.

Poslední nevysvětlenou záležitostí v naší ukázce zůstává příkaz `\newlinechar=`\@`. Tím se nastaví znak `@` tak, že v příkazu `\message` pracuje jako konec řádku, takže např. `\message{...@}` pracuje jako Pascalské `writeln`.

Náš program můžeme dále vylepšovat. Samozřejmostí by mělo být nastavení výšky tiskového zrcadla a tím počtu řádků na stránce, práce s tabelátory, možnost umístit stránkovou číslici podle potřeby, tisk dalších informací apod.

Pro větší představu si na závěr uvedeme, které z použitých příkazů jsou TeX-primitivní a které jsou definovány dodatečně. TeX-primitivní příkazy (případně parametry) jsou:

```
\newlinechar, \def, \begingroup, \endgroup, \par, \catcode, \input,
\hbox, \setbox, \box, \wd, \vskip, \penalty \baselineskip, \message, \read,
\immediate, \openin, \closein, \ifeof, \else, \fi, \ifdim, \let, \end.
```

PlainTeX definuje tyto příkazy:

```
\newread, \frenchspacing, \obeylines, \obeyspaces, \tt, \dospecials, \space, \bgroup, \egroup.
```

Poznamenejme nakonec, že uvedené ukázky zdaleka nevyčerpávají všechny možnosti programovacího jazyka TeXu. Srovnal bych to zhruba s látkou „první třídy školy programování v TeXu“. Obrovský zdroj inspirace najdete ve zdrojovém textu k LaTeXu, o němž jsme si už v dřívějších kapitolách řekli, že představuje zhruba 10000 řádků napsaných v tomto programovacím jazyku.