Zpravodaj Československého sdružení uživatelů TeXu

Zdeněk Wagner

# K čemu slouží Web?

*Zpravodaj Československého sdružení uživatelů TeXu*, Vol. 3 (1993), No. 4, 162–184

Persistent URL: http://dml.cz/dmlcz/149689

# K čemu slouží WEB?

ZDENĚK WAGNER

K napsání tohoto článku mě vyprovokoval příspěvek Martina Bílého v loňském třetím čísle Zpravodaje. Tam je WEB popsán z hlediska „místního čaroděje“, který musí instalovat TEX ze zdrojových textů. V tomto článku bych se pokusil vysvětlit, jaké výhody přináší WEB programátorům.

Dlouho jsem uvažoval, zda mám vůbec tento článek napsat a zda se hodí do Zpravodaje. Nakonec mě k tomu přiměly dvě skutečnosti:

- Při psaní textu v LATEXu jsem se dostal do problémů, které jsem vyřešil pomocným programem v C++. Tento program by mohl být užitečný, a aby mohl být publikován, je vhodné jej napsat ve WEBu.
- Podobná filosofie programování se dá využít i při psaní rozsáhlých souborů TEXových maker.

M. Bílý označil WEB jako jazyk. Z hlediska matematické lingvistiky je to jistě pravda. Z praktického hlediska bych však raději označil WEB jako programovací nástroj. Není ovšem pravda, že WEB se podobá Pascalu — to vysvětlíme později.

Před časem jsem se ptal v elektronické konferenci `cstex`, zda již existuje oficiální překlad termínu „literate programming“. Nedostal jsem žádnou odpověď, a proto se sám pokusím o překlad. Chtěl bych se na tomto místě omluvit všem, kteří jsou vzdělanější než já. Mnoho věcí jsem studoval z anglických knih a nikdy mě nenapadlo přemýšlet nad tím, jak se výrazy jako „pointer“, „carriage return“, „box“, „glue“ správně přeloží do češtiny. Z toho vznikají mé současné problémy.

Slovo „literate“ lze přeložit jako „gramotný“ nebo „kultivovaný“, zatímco pro opačné slovo „illiterate“ jsem ve slovníku našel pouze jediný ekvivalent: „negramotný“. Raději bych tedy překládal „literate programming“ jako „kultivované programování“.

Základním konceptem kultivovaného programování je zápis zdrojového kódu programu i programové dokumentace do téhož souboru. Obrovská výhoda tohoto přístupu spočívá v tom, že program i dokumentace se udržují v synchronizaci. Dokumentace pak není o několik verzí

pozadu, jak se obvykle stává při „illiterate programming“. Pro předzpracování textu se používají dva překladače: `tangle`, jehož produktem je vstupní soubor pro kompilátor určitého programovacího jazyka, a `weave`, který vytváří formátovaný soubor pro vytištění dokumentace a zdrojového programu, obvykle pomocí TEXu.

Kultivovaný program se skládá ze sekcí. Každá sekce má tři části: část dokumentační, část definiční a část programovou. Kterákoliv z těchto částí může být prázdná. V dokumentační části se uvádí popis dané části programu v jazyce, který je srozumitelný lidským bytostem. Definiční část není nutná při použití jazyka C. Zde je totiž jinými prostředky implementována céčková direktiva `#define`. Programová část obsahuje vlastní program v příslušném programovacím jazyce. Speciálním příkazem lze do programové části vložit jinou sekci. Lze to přirovnat buď k preprocesoru jazyka C, nebo (abych se více držel TEXu) definici makra, v níž je voláno další makro. Programátor tak získává další úroveň strukturovanosti, což umožňuje převést lineární strukturu programu do logičtějšího členění. V každém programu musí být jedna bezejmenná sekce, kterou `tangle` bude považovat za začátek programu. Ostatní sekce mají svá jména.

Před okamžikem bylo uvedeno, že dokumentace se tiskne obvykle TEXem a programová část je psána v příslušném programovacím jazyce. Z toho je vidět, že označení WEB je naprosto nejednoznačné. Existuje více systémů, které implementují stejnou filosofii. Liší se jednak tím, zda podporují více programovacích jazyků, nebo se specializují na jeden z nich (C, Pascal, apod.), a také, jakým programem se pak tiskne dokumentace připravená překladačem `weave`. Původní WEB byl napsán pro Pascal a z toho právě plyne mylná představa, že „jazyk WEB se podobá Pascalu.“

V následujícím textu se omezíme pouze na `CWEB`. Je určen pro jazyk C a ve verzi 3.0 i pro C++. Pro tisk dokumentace slouží plain TEX, od verze 3.0 lze použít i LATEX. Budeme si jej demonstrovat na kompletním programu, který rutinně používám, a z didaktických důvodů si až v závěru vysvětlíme, proč jsem vlastně tento program napsal. Předtím je ovšem vhodné uvést alespoň některé příkazy `CWEBu`. Ve vytištěném textu sice nebudou vidět, ale usnadní nám výklad dalších výhod WEBu.

Všechny příkazy `CWEBu` začínají znakem „@“. Chceme-li tedy použít tento znak v textu, musíme jej zdvojit, tj. napíšeme „@@“.

Jedním z nejdůležitějších příkazů je definice jména sekce. Jméno sekce píšeme mezi `@<` a `@>`. Očíslování sekcí zařídí `weave`. Programová část

sekce začíná uvedením jména, za nímž následuje znak „=“. Do programové části začleníme jinou sekci tím, že mezi příkazy jazyka C uvedeme její jméno.

Programovou část bezejmenné sekce musíme uvést speciálním příkazem. Je jím `@c`. Kvůli kompatibilitě s původním WEBem pro Pascal je povolen i `@p`.

`Weave` provádí formátování zdrojového textu tak, aby byl čitelnější. Pro tento účel si rozkládá jednotlivé příkazy a různé části pak tiskne různými fonty. Za konec příkazu se považuje středník. Jazyk C je ovšem nedůsledný v tom smyslu, že v určitých příkazech se středník nepíše. Aby překladač `weave` nebyl zmaten, použijeme v takovém případě příkaz `@;`.

Zdrojový text je formátován tak, aby každý programový příkaz začínal na novém řádku. To není vždy žádoucí. K potlačení přechodu na nový řádek slouží příkaz `@+`.

To samozřejmě není úplný výčet toho, co musíte znát, chcete-li použít `CWEB` pro psaní vlastních programů. Podrobnější informace naleznete v případě zájmu v dokumentaci, která je součástí balíku `CWEB` a podobně jako TEX patří mezi volně šiřitelné programy.

Nyní již následuje slíbený program.

**1. PROGRAM PERCENT.** Účelem tohoto programu je usnadnit psaní balíků (LA)TEXových maker dokumentovaných pomocí DOC.STY a DOCSTRIP.TEX. Dokumentace k makrům se zapisuje do komentářů, makra se zapisují mezi `%␣␣␣␣␣\begin{macrocode}` a `%␣␣␣␣␣\end{macrocode}`. Program umožňuje konverzi mezi pohodlnějším zápisem a tvarem pro DOC.STY.

Pro snadnější zápis dokumentace existují následující příkazy, které musí být psány na samostatném řádku od prvního sloupce:

| | |
|---|---|
| `%+` | před všechny následující neprázdné řádky bude přidáno procento a mezera. |
| `%−` | ruší funkci %+. |
| `%%+` | před všechny následující neprázdné řádky budou přidány dva znaky procento a mezera. |
| `%%−` | ruší funkci %%+. |
| `%+++` | expanduje na `%␣␣␣␣␣\begin{macrocode}`. |
| `%−−−` | expanduje na `%␣␣␣␣␣\end{macrocode}`. |

Tato funkce se vyvolá příkazem:

`percent +<`*filename*`>`

kde *filename* je plné jméno souboru. Soubor bude nahrazen konvertovanou verzí, původní soubor bude uschován v souboru s příponou `.bak`.

Opačná konverze se provede příkazem:

`percent −<`*filename*`>`

Zatímco v původním souboru můžete libovolně míchat příkazy %+++ a %−−− s `%␣␣␣␣\begin{macrocode}` a `%␣␣␣␣\end{macrocode}`, při zpětné konverzi budou všechny výskyty `%␣␣␣␣\begin{macrocode}` i `%␣␣␣␣\end{macrocode}` převedeny na %+++ a %−−−.

**2.** Program bude mít následující části.

⟨ Hlavičkové soubory 3 ⟩
⟨ Prototypy 5 ⟩
⟨ Globální proměnné 10 ⟩
⟨ Hlavní program 4 ⟩
⟨ Funkce 6 ⟩

**3.** Jak později uvidíme, budeme potřebovat následující hlavičkové soubory. (Samozřejmě bychom tuto sekci mohli rozdělit na mnoho malých fragmentů a načítat každý hlavičkový soubor vždy, když se použije funkce v něm definovaná. Podle mého názoru to ale k přehlednosti programu nepřispěje.)

⟨ Hlavičkové soubory 3 ⟩ ≡

```
#include <dir.h>
#include <io.h>
#include <fcntl.h>
#include <string.h>
#include <fstream.h>
#include <ibmanip.h>
#include <useful.h>
#include <stdlib.h>
#include <ctype.h>
```

Tento kód je použit v sekci 2.

**4.** Hlavní program musí přečíst parametr z příkazového řádku. Všimněte si, že mezi znakem + či − a jménem souboru není mezera. Vše se tedy přečte jako jediný parametr, takže *argc* musí mít hodnotu 2. První znak parametru rozhodne o tom, jakou funkci máme provést, zbytek se pošle dál jako jméno souboru.

```
⟨ Hlavní program 4 ⟩ ≡
  void main(int argc, const char *argv[])
  {
    if (argc ≠ 2) {
      cout ≪ "Nesprávný␣počet␣parametrů\n";
      return;
    }
    switch (*argv[1]) {
    case '+': AddPerCent(argv[1] + 1); break;
    case '-': RemovePerCent(argv[1] + 1); break;
    default: cout ≪ "Neznámá␣funkce;␣zvol␣+␣nebo␣-.\n";
    }
  }
```

Tento kód je použit v sekci 2.

**5.** Použité funkce musí mít své prototypy.

```
⟨ Prototypy 5 ⟩ ≡
  void AddPerCent(const char *);
  void RemovePerCent(const char *);
  int MakeBakFile(const char *);
```

Tento kód je použit v sekci 2.

**6.** V programu budeme potřebovat tři funkce (tu třetí jsme předčasně uvedli v sekci prototypů).

```
⟨ Funkce 6 ⟩ ≡
  ⟨ Přidej procento 8 ⟩
  ⟨ Odstraň procento 26 ⟩
  ⟨ Udělej záložní soubor 7 ⟩
```

Tento kód je použit v sekci 2.

**7.** Nejprve si vytvoříme posledně jmenovanou funkci. Ta vezme jméno souboru jako parametr a rozdělí jej na jednotlivé části. Pak vytvoří jméno, které má příponu `.bak`. Původní soubor s příponou `.bak` se zruší a zdrojový soubor je přejmenován. Tento soubor je pak otevřen a funkce vrátí hodnotu rukojeti souboru (file handle).

```
⟨ Udělej záložní soubor 7 ⟩ ≡
  int MakeBakFile(const char *fn)
  {
    char myfn[MAXPATH], fullfn[MAXPATH], dr[MAXDRIVE], dir[MAXDIR],
         fname[MAXFILE], ext[MAXEXT];
    _fullpath(fullfn, fn, sizeof fullfn);
    fnsplit(fullfn, dr, dir, fname, ext);
    fnmerge(myfn, dr, dir, fname, ".bak");
    unlink(myfn);
    if (rename(fullfn, myfn)) {
      cout ≪ "chyba␣vstupu/výstupu,␣pravděpodobně␣nee\
          xistující␣soubor\n";
      exit(EXIT_FAILURE);
    }
    return open(myfn, O_RDONLY | O_TEXT);
  }
```

Tento kód je použit v sekci 6.

**8.** Následující funkce bude konvertovat soubor do formátu pro DOC.STY, tj. stručně řečeno, bude přidávat procento na začátek řádků.

```
⟨ Přidej procento 8 ⟩ ≡
  void AddPerCent(const char *fn)
  {
    ⟨ Otevři soubory 9 ⟩
    ⟨ Add: inicializuj lokální proměnné 12 ⟩
    ⟨ Add: zpracuj soubor 13 ⟩
    ⟨ Zavři soubory 11 ⟩
  }
```

Tento kód je použit v sekci 6.

**9.** Při otvírání vstupního souboru použijeme vlastní buffer pro zrychlení čtení a zápisu. Jako první parametr v konstruktoru použijeme rukojeť souboru, kterou vrátí funkce *MakeBakFile*.

⟨ Otevři soubory 9 ⟩ ≡
```
ifstream inf(MakeBakFile(fn), inbuf, sizeof inbuf);
ofstream outf(fn);
```
Tento kód je použit v sekcích 8 a 26.

**10.** Buffer musíme deklarovat jako globální proměnnou.

⟨ Globální proměnné 10 ⟩ ≡
```
char inbuf[16384];
```
Viz též sekce 23, 24 a 25.

Tento kód je použit v sekci 2.

**11.** Zavírání souborů nepotřebuje bližší vysvětlení.

⟨ Zavři soubory 11 ⟩ ≡
```
outf.close(); inf.close();
```
Tento kód je použit v sekcích 8 a 26.

**12.** V této funkci budeme především potřebovat proměnnou, do níž budeme načítat vstupní řádky. Mlčky budeme předpokládat, že řádek nebude mít více než 255 znaků. Kromě toho potřebujeme logické proměnné s následujícím významem:

| | |
|---|---|
| **one** | zpracovává se text za příkazem %+. |
| **two** | zpracovává se text za příkazem %%+. |
| **macro** | zpracovává se "macrocode". V určitém okamžiku se dočasně využívá i pro jiné účely, což bude na příslušném místě dokumentováno. |
| **empty** | poslední zpracovaný řádek byl prázdný. |

Všechny logické proměnné se na začátku naplní hodnotou 0.

⟨ Add: inicializuj lokální proměnné 12 ⟩ ≡
```
char Line[256];
int one, two, macro, empty;
one ← two ← macro ← empty ← 0;
```
Tento kód je použit v sekci 8.

**13.** Dokud jsou data ve vstupním souboru a do výstupního souboru lze psát, budeme číst při každém průchodu vždy jeden řádek, zjistíme, zda je prázdný, a dále provedeme podrobnější analýzu a zpracování.

```
⟨ Add: zpracuj soubor 13 ⟩ ≡
  while (inf ∧ outf ∧ ¬inf.eof()) {
    ⟨ Add: přečti řádek 14 ⟩
    ⟨ Add: zpracuj prázdný řádek 15 ⟩
    ⟨ Add: analyzuj a zpracuj řádek 16 ⟩
  }
```

Tento kód je použit v sekci 8.

**14.** Některé editory nechávají na konci řádků mezery, což by znemožnilo správné rozeznávání prázdných řádků. Proto po načtení řádku a přeskočení všech následujících znaků až do `\n` včetně zavoláme funkci, která odstraní koncové mezery.

```
⟨ Add: přečti řádek 14 ⟩ ≡
  inf.get(Line, sizeof Line); inf ≫ endl; CutStg(Line);
```

Tento kód je použit v sekci 13.

**15.** V TEXu prázdný řádek znamená konec odstavce. Dva či více po sobě následující prázdné řádky nemají žádný smysl, a proto je odstraníme. Jestliže najdeme prázdný řádek a předchozí řádek byl také prázdný, skočíme rovnou na konec smyčky.

```
⟨ Add: zpracuj prázdný řádek 15 ⟩ ≡
  if (¬∗Line) {
    if (¬empty) outf ≪ Line ≪ endl;
    empty ← 1;
    continue;
  }
  else empty ← 0;
```

Tento kód je použit v sekci 13.

**16.** Při analýze a zpracování řádku budeme nejprve hledat začátek prostředí macrocode. Je-li nastaven přepínač *macro*, budeme hledat konec prostředí macrocode. V opačném případě zjistíme typ příkazu a provedeme výstup.

```
⟨ Add: analyzuj a zpracuj řádek 16 ⟩ ≡
  ⟨ Hledej begin macrocode 17 ⟩
  if (macro) {
    ⟨ Hledej end macrocode 18 ⟩
  }
  else {
    ⟨ Zjisti typ příkazu 19 ⟩
    ⟨ Zapiš upravený řádek 20 ⟩
  }
```

Tento kód je použit v sekci 13.

**17.** Zde případně expandujeme %+++. Pokud objevíme začátek prostředí macrocode, nastavíme přepínač *macro*. Současně si nadefinujeme *beginmac* pro pohodlnější programování.

```
#define beginmac "%␣␣␣␣\\begin{macrocode}"
⟨ Hledej begin macrocode 17 ⟩ ≡
  if (¬macro ∧ ¬strcmp(Line, "%+++")) {
    strcpy(Line, beginmac); macro ← 1;
  }
  if (¬macro ∧ ¬strcmp(Line, beginmac)) macro ← 1;
```

Tento kód je použit v sekci 16.

**18.** Při zpracování prostředí macrocode musíme nejprve otestovat a případně expandovat %−−−. Poté zapíšeme řádek do výstupního souboru, a pokud prostředí končí, vynulujeme *macro*. I zde použijeme definici makra pro usnadnění života.

```
#define endmac "%␣␣␣␣\\end{macrocode}"
⟨ Hledej end macrocode 18 ⟩ ≡
  if (¬strcmp(Line, "%---")) strcpy(Line, endmac);
  outf ≪ Line ≪ endl;
  if (¬strcmp(Line, endmac)) macro ← 0;
```

Tento kód je použit v sekci 16.

**19.** Nyní vyzkoušíme, zda řádek obsahuje některý ze zbývajících příkazů. Dočasně použijeme proměnnou *macro* pro jiný účel. Hodnota $+1$ bude znamenat zahájení příslušného typu zpracování, hodnota $-1$ znamená jeho konec. Současně se nastaví proměnné *one* a *two*. Přepínač *macro* bude vynulován v následující sekci.

```
⟨ Zjisti typ příkazu 19 ⟩ ≡
  if (¬strcmp(Line, "%+")) {
    macro ← 1; one ← 1; two ← 0;
  }
  if (¬strcmp(Line, "%-")) {
    macro ← −1; one ← two ← 0;
  }
  if (¬strcmp(Line, "%%+")) {
    macro ← 1; one ← 0; two ← 1;
  }
  if (¬strcmp(Line, "%%-")) {
    macro ← −1; one ← two ← 0;
  }
```

Tento kód je použit v sekci 16.

**20.** Je-li nastaven přepínač *macro*, je na současném řádku formátovací příkaz a celý řádek se ignoruje. V opačném případě zapíšeme úvodní procenta podle nastavení proměnných *one* a *two* a samozřejmě celý řádek.

```
⟨ Zapiš upravený řádek 20 ⟩ ≡
  if (macro) macro ← 0;
  else {
    if (one) outf ≪ "%␣";
    else if (two) outf ≪ "%%␣";
    outf ≪ Line ≪ endl;
  }
```

Tento kód je použit v sekci 16.

**21.** Zpětná konverze je poněkud složitější. Naprogramujeme ji tedy jako stavový automat s následujícími deseti stavy:

**0:** normální výstup.
**1:** nalezen jeden řádek začínající znakem %.
**2:** nalezeny dva řádky začínající znakem %.
**3:** nalezen jeden řádek začínající znaky %%.
**4:** nalezeny dva řádky začínající znaky %%.
**5:** přerušení výstupu řádků typu %.
**6:** přerušení výstupu řádků typu %%.
**7:** prostředí macrocode uvnitř normálního výstupu.
**8:** prostředí macrocode během výstupu řádků typu %.
**9:** prostředí macrocode během výstupu řádků typu %%.

Je zřejmé, že stavy 1–4 brání vytváření hloupých sekvencí

```
%+
Jednořádkový komentář
%−
```

Nadefinujeme si tedy odpovídající prahovou hodnotu.

```
#define threshold 3
```

**22.** Stavový automat musí rozeznat následující typy řádků:

**0:** prázdný řádek.
**1:** řádek začínající znakem %.
**2:** řádek začínající znaky %%.
**3:** `%␣␣␣␣\begin{macrocode}`.
**4:** `%␣␣␣␣\end{macrocode}`.
**5:** jiný.

**23.** V každém stavu se provede jedna z následujících procedur:

| | |
|---|---|
| ⟨ **_PA_** ⟩ | zapiš vše, vynuluj *idx*. |
| ⟨ **_KA_** ⟩ | inkrementuj *idx*. |
| ⟨ **_C1_** ⟩ | zapiš všechny řádky s odstraněným znakem %, vynuluj *idx*. |
| ⟨ **_MC1_** ⟩ | zapiš příkaz %+ a proveď ⟨ _C1_ ⟩. |
| ⟨ **_C2_** ⟩ | zapiš všechny řádky s odstraněnými znaky %% a vynuluj *idx*. |
| ⟨ **_MC2_** ⟩ | zapiš příkaz %%+ a proveď ⟨ _C2_ ⟩. |
| ⟨ **_KP_** ⟩ | zapiš všechny řádky s výjimkou posledního, zkopíruj řádek[*idx*] do řádku[0] a nastav *idx*=1. |
| ⟨ **_K1_** ⟩ | zapiš příkaz %− a proveď ⟨ _KP_ ⟩. |
| ⟨ **_K2_** ⟩ | zapiš příkaz %%− a proveď ⟨ _KP_ ⟩. |
| ⟨ **_S1_** ⟩ | zapiš příkaz %− a proveď ⟨ _PA_ ⟩. |
| ⟨ **_S2_** ⟩ | zapiš příkaz %%− a proveď ⟨ _PA_ ⟩. |
| ⟨ **_ERR_** ⟩ | zobraz chybovou zprávu a proveď ⟨ _PA_ ⟩. |

Definici enumerací musíme zařadit mezi globální proměnné, protože je budeme potřebovat později v definici stavového automatu.

```
⟨ Globální proměnné 10 ⟩ +≡
  enum ProcSteps {
    _PA_, _KA_, _C1_, _MC1_, _C2_, _MC2_, _KP_, _K1_, _K2_, _S1_, _S2_,
        _ERR_
  };
```

**24.** Stavový automat bude definován jako pole struktur. Struktura musí mít dvě položky: číslo nového stavu a označení procedury, která se má provést. Deklaraci odpovídající struktury přidáme k definici globálních proměnných, přestože zde pouze definujeme typ bez alokace paměti.

```
⟨ Globální proměnné 10 ⟩ +≡
  struct StateMachine {
    int NewState, ProcStep;
  };
```

**25.** Stavový automat je definován následující tabulkou. Ke každému stavu a typu vstupního řádku přísluší číslo nového stavu a označení požadované procedury (viz sekce 23).

| Stav | prázdný řádek | % | %% | %+ + + | %− − − | jiný |
|---|---|---|---|---|---|---|
| 0 | 0 ⟨ _PA_ ⟩ | 1 ⟨ _KA_ ⟩ | 3 ⟨ _KA_ ⟩ | 7 ⟨ _PA_ ⟩ | 0 ⟨ _ERR_ ⟩ | 0 ⟨ _PA_ ⟩ |
| 1 | 0 ⟨ _PA_ ⟩ | 1 ⟨ _KA_ ⟩ | 3 ⟨ _KP_ ⟩ | 7 ⟨ _PA_ ⟩ | 0 ⟨ _ERR_ ⟩ | 0 ⟨ _PA_ ⟩ |
| 2 | 0 ⟨ _PA_ ⟩ | 5 ⟨ _MC1_ ⟩ | 3 ⟨ _KP_ ⟩ | 7 ⟨ _PA_ ⟩ | 0 ⟨ _ERR_ ⟩ | 0 ⟨ _PA_ ⟩ |
| 3 | 0 ⟨ _PA_ ⟩ | 1 ⟨ _KP_ ⟩ | 4 ⟨ _KA_ ⟩ | 7 ⟨ _PA_ ⟩ | 0 ⟨ _ERR_ ⟩ | 0 ⟨ _PA_ ⟩ |
| 4 | 0 ⟨ _PA_ ⟩ | 1 ⟨ _KP_ ⟩ | 6 ⟨ _MC2_ ⟩ | 7 ⟨ _PA_ ⟩ | 0 ⟨ _ERR_ ⟩ | 0 ⟨ _PA_ ⟩ |
| 5 | 5 ⟨ _PA_ ⟩ | 5 ⟨ _C1_ ⟩ | 3 ⟨ _K1_ ⟩ | 8 ⟨ _PA_ ⟩ | 0 ⟨ _ERR_ ⟩ | 0 ⟨ _S1_ ⟩ |
| 6 | 5 ⟨ _PA_ ⟩ | 1 ⟨ _K2_ ⟩ | 6 ⟨ _C2_ ⟩ | 9 ⟨ _PA_ ⟩ | 0 ⟨ _ERR_ ⟩ | 0 ⟨ _S2_ ⟩ |
| 7 | 7 ⟨ _PA_ ⟩ | 7 ⟨ _PA_ ⟩ | 7 ⟨ _PA_ ⟩ | 7 ⟨ _ERR_ ⟩ | 0 ⟨ _PA_ ⟩ | 7 ⟨ _PA_ ⟩ |
| 8 | 8 ⟨ _PA_ ⟩ | 8 ⟨ _PA_ ⟩ | 8 ⟨ _PA_ ⟩ | 8 ⟨ _ERR_ ⟩ | 0 ⟨ _PA_ ⟩ | 8 ⟨ _PA_ ⟩ |
| 9 | 9 ⟨ _PA_ ⟩ | 9 ⟨ _PA_ ⟩ | 9 ⟨ _PA_ ⟩ | 9 ⟨ _ERR_ ⟩ | 0 ⟨ _PA_ ⟩ | 9 ⟨ _PA_ ⟩ |

Pro lepší hospodaření se zásobníkem budeme automat definovat globálně.

```
⟨ Globální proměnné 10 ⟩ +≡
  StateMachine RM[][6] ← {
      {{0, _PA_}, {1, _KA_}, {3, _KA_}, {7, _PA_}, {0, _ERR_}, {0, _PA_}},
    /* stav=0 */
  {{0, _PA_}, {2, _KA_}, {3, _KP_}, {7, _PA_}, {0, _ERR_}, {0, _PA_}},
    /* stav=1 */
  {{0, _PA_}, {5, _MC1_}, {3, _KP_}, {7, _PA_}, {0, _ERR_}, {0, _PA_}},
    /* stav=2 */
  {{0, _PA_}, {1, _KP_}, {4, _KA_}, {7, _PA_}, {0, _ERR_}, {0, _PA_}},
    /* stav=3 */
  {{0, _PA_}, {1, _KP_}, {6, _MC2_}, {7, _PA_}, {0, _ERR_}, {0, _PA_}},
    /* stav=4 */
  {{5, _PA_}, {5, _C1_}, {3, _K1_}, {8, _PA_}, {0, _ERR_}, {0, _S1_}},
    /* stav=5 */
  {{6, _PA_}, {1, _K2_}, {6, _C2_}, {9, _PA_}, {0, _ERR_}, {0, _S2_}},
    /* stav=6 */
  {{7, _PA_}, {7, _PA_}, {7, _PA_}, {7, _ERR_}, {0, _PA_}, {7, _PA_}},
    /* stav=7 */
```

```
  {{8, _PA_}, {8, _PA_}, {8, _PA_}, {8, _ERR_}, {5, _PA_}, {8, _PA_}},
    /* stav=8 */
  {{9, _PA_}, {9, _PA_}, {9, _PA_}, {9, _ERR_}, {6, _PA_}, {9, _PA_}}
    /* stav=9 */
  };
```

**26.** Podobně jako při opačném procesu, i nyní nejprve otevřeme soubory, provedeme inicializaci proměnných, zpracujeme soubor a nakonec všechny soubory zavřeme.

```
⟨ Odstraň procento 26 ⟩ ≡
  void RemovePerCent(const char *fn)
  {
    ⟨ Otevři soubory 9 ⟩
    ⟨ Rem: inicializuj proměnné 27 ⟩
    ⟨ Rem: zpracuj soubor 28 ⟩
    ⟨ Zavři soubory 11 ⟩
  }
```

Tento kód je použit v sekci 6.

**27.** Do proměnné *Line* budeme načítat vstupní řádky a *idx* bude obsahovat počet načtených řádků. Další proměnné budou obsahovat stav automatu a typ procedury, která se má provést. Jejich přesný význam bude jasnější později, nyní pouze některé z nich vynulujeme.

```
⟨ Rem: inicializuj proměnné 27 ⟩ ≡
  char Line[threshold][256];
  int idx, state, empty, linetype, newstate, proctype, maxstate;
  maxstate ← sizeof (RM)/sizeof (RM[1]);
  idx ← state ← empty ← 0;
```

Tento kód je použit v sekci 26.

**28.**

```
⟨ Rem: zpracuj soubor 28 ⟩ ≡
  while (inf ∧ outf ∧ ¬inf.eof()) {
    ⟨ Rem: načti vstupní řádek 29 ⟩
    ⟨ Zkontroluj stav 30 ⟩
    ⟨ Rem: zjisti typ řádku 31 ⟩
    ⟨ Rem: zjisti nový stav a typ procedury 32 ⟩
    ⟨ Rem: zpracování prázdného řádku 33 ⟩
    ⟨ Proveď příslušnou proceduru 35 ⟩
    ⟨ Nastav nový stav 34 ⟩
  }
```

Tento kód je použit v sekci 26.

**29.** Po načtení řádku a přeskočení všech znaků po `\n` včetně opět odstraníme případné koncové mezery.

```
⟨ Rem: načti vstupní řádek 29 ⟩ ≡
  inf.get(Line[idx], sizeof (Line[idx])); inf ≫ endl;
  CutStg(Line[idx]);
```

Tento kód je použit v sekci 28.

**30.** Nyní ověříme, zda je automat v přípustném stavu. Kontrola je vlastně zbytečná, neboť do nepřípustného stavu se automat může dostat pouze tehdy, je-li program poškozen (např. virem).

```
⟨ Zkontroluj stav 30 ⟩ ≡
  if (state < 0 ∨ state ≥ maxstate) {
    cout ≪ "Nepřípustný␣stav␣" ≪ state ≪ endl;
    cout ≪ "Program␣je␣poškozen\n";
    exit(EXIT_FAILURE);
  }
```

Tento kód je použit v sekci 28.

**31.** Dalším krokem je zjištění typu řádku tak, jak je uvedeno v sekci 22. Zjišťování se provádí jednoduchými příkazy **if**, které nepotřebují bližší komentář.

```
⟨ Rem: zjisti typ řádku 31 ⟩ ≡
  linetype ← 5;
  if (¬Line[idx][0]) linetype ← 0;
  else {
    if (¬strcmp(Line[idx], beginmac)) {
      strcpy(Line[idx], "%+++"); linetype ← 3;
    }
    else if (¬strcmp(Line[idx], endmac)) {
      strcpy(Line[idx], "%---"); linetype ← 4;
    }
    else if (Line[idx][0] ≡ '%') {
      if (Line[idx][1] ≠ '%') linetype ← 1;
      else if (Line[idx][2] ≠ '%') linetype ← 2;
    }
  }
```

Tento kód je použit v sekci 28.

**32.** Z tabulky zjistíme nový stav automatu a typ procedury, kterou máme provést.

```
⟨ Rem: zjisti nový stav a typ procedury 32 ⟩ ≡
  newstate ← RM[state][linetype].NewState;
  proctype ← RM[state][linetype].ProcStep;
```

Tento kód je použit v sekci 28.

**33.** Je-li *linetype* $\equiv 0$, znamená to, že poslední řádek je prázdný. Dva a více prázdných řádků nemají v TEXu význam, proto budeme nadbytečné prázdné řádky ignorovat. Po prázdném řádku si nastavíme switch *empty*. Je-li tento switch již nastaven, znamená to, že i předchozí řádek byl prázdný. V tom případě nastavíme nový stav automatu a skočíme rovnou na konec smyčky. Jestliže poslední řádek není prázdný, musíme vynulovat *empty*.

```
⟨ Rem: zpracování prázdného řádku 33 ⟩ ≡
  if (linetype ≡ 0) {
    if (empty) {
      ⟨ Nastav nový stav 34 ⟩
      continue;
    }
    empty ← 1;
  }
  else empty ← 0;
```

Tento kód je použit v sekci 28.

**34.** Nastavení nového stavu je triviální.

```
⟨ Nastav nový stav 34 ⟩ ≡
  state ← newstate;
```

Tento kód je použit v sekcích 28 a 33.

**35.** Typ procedury máme uložen v proměnné *proctype*. Nejprve provedeme kontrolu, zda je procedura povolena. Chyba je obvykle způsobena špatným vnořením prostředí „macrocode“. Pak následuje rozhodovací blok pro jednotlivé typy procedur, které byly definovány v sekci 23.

```
⟨ Proveď příslušnou proceduru 35 ⟩ ≡
  if (proctype ≡ _ERR_) {
    cout ≪ "Chyba␣ve␣vstupním␣souboru;␣zkontroluj␣p\
        rostředí␣macrocode\n";
  }
  char *s;
  int j;
  switch (proctype) {
  case _ERR_: case _S1_: case _S2_: case _PA_: ⟨ Proc P 36 ⟩ break;
  case _K1_: case _K2_: case _KP_: ⟨ Proc K 37 ⟩ break;
  case _MC1_: case _MC2_: ⟨ Proc M 38 ⟩
  case _C1_: case _C2_: ⟨ Proc C 39 ⟩ break;
  case _KA_: idx++; break;
  default: ⟨ Default proc 40 ⟩
  }
```

Tento kód je použit v sekci 28.

**36.** Tyto procedury musí především zapsat všechny uschované řádky. Procedury ⟨ _S1_ ⟩ a ⟨ _S2_ ⟩ navíc zapíší příkaz pro ukončení příslušného typu komentáře.

```
⟨ Proc P 36 ⟩ ≡
  if (proctype ≡ _S1_) outf ≪ "%-\n";
  else if (proctype ≡ _S2_) outf ≪ "%%-\n";
  for (j ← 0; j ≤ idx; j++) outf ≪ Line[j] ≪ endl;
  idx ← 0;
```

Tento kód je použit v sekci 35.

**37.** Tyto procedury se podobají předchozím, pouze pořadí příkazů je odlišné a navíc nezapisujeme poslední řádek.

```
⟨ Proc K 37 ⟩ ≡
  for (j ← 0; j < idx; j++) outf ≪ Line[j] ≪ endl;
  if (proctype ≡ _K1_) outf ≪ "%-\n";
  else if (proctype ≡ _K2_) outf ≪ "%%-\n";
  if (idx) strcpy(Line[0], Line[idx]);
  if (idx) idx ← 1;      /* vynechat, pokud bylo idx ≡ 0 */
```

Tento kód je použit v sekci 35.

**38.** Tyto procedury zapíší příkaz pro zahájení komentáře. V příkazu **switch** nejsou ukončeny příkazem **break**, takže pokračují do procedur ⟨ _C1_ ⟩, ⟨ _C2_ ⟩.

```
⟨ Proc M 38 ⟩ ≡
  if (proctype ≡ _MC1_) outf ≪ "%+\n";
  else outf ≪ "%%+\n";
```

Tento kód je použit v sekci 35.

**39.** Zde odstraníme jeden nebo dva znaky procento (podle typu řádku) a všechny následující mezery.

```
⟨ Proc C 39 ⟩ ≡
  for (j ← 0; j ≤ idx; j++) {
    s ← Line[j] + ((proctype ≡ _C1_ ∨ proctype ≡ _MC1_) ? 1 : 2);
    while (*s ∧ isspace(*s)) s++;
    outf ≪ s ≪ endl;
  }
  idx ← 0;
```

Tento kód je použit v sekci 35.

**40.** Toto je opět situace, která za normálních okolností nemůže nastat. Pokud program dojde do tohoto stavu, znamená to, že je poškozen.

```
⟨ Default proc 40 ⟩ ≡
  cout ≪ "Program␣je␣poškozen\n";
  exit(EXIT_FAILURE);
```

Tento kód je použit v sekci 35.

---

CWEB má ještě další užitečné prostředky, které jsme zde nepoužili. Nejsou-li potlačeny, generuje weave automaticky rejstřík všech proměnných a obsah. U rozsáhlých programů to představuje nespornou výhodu.

Kultivovaně napsaný program má ještě další výhodu. Obvykle v něm po prvním napsání bývá málo chyb, takže nevyžaduje zdlouhavé ladění. Programátor totiž vše vysvětluje lidskou řečí v dokumentační části a to jej vede k podrobnější analýze algoritmu, který píše. Tím odhalí různá úskalí dříve, než se jeho počítač zacyklí v nekonečné smyčce.

Ladění se ovšem zřejmě nevyhneme u žádného většího programu. Tehdy nezbývá, než najít předpokládanou příčinu chyby, provést modifikaci zdrojového programu a zkusit znovu. Chyb ale může být více a při ladění se můžeme splést. Tak se stává, že najednou chceme cosi vrátit do původního stavu, jenže nemáme použitelnou kopii a původní stav jsme již zapomněli.

Zde se vyplatí další nástroj, který nám WEB nabízí — a to tzv. změnový soubor. Jak název napovídá, jsou v něm obsaženy změny, které hodláme provést. Jeho struktura je následující:

```
@x
Původní řádky
@y
Nové řádky
@z
```

„Původní řádky“ i „Nové řádky“ mohou obsahovat libovolný text s výjimkou příkazů @x, @y a @z. Text, který předchází @x nebo následuje za @z (až do @x) je ignorován. Můžete si zde tudíž poznamenat, proč příslušnou změnu provádíte.

Když je program konečně odladěn, je vhodné začlenit změny do zdrojového textu. K tomu lze použít program wmerge.

Změnový soubor hraje důležitou roli při implementaci TEXu, ale i jiných programů, kde je kladen důraz na přenositelnost. Představte si, že

chcete implementovat nějaký program na neobvyklém počítači s obskurním operačním systémem. Prostudujete příslušný program a uděláte si změnový soubor. Po nějakém čase dá autor programu k dispozici novou verzi. Implementace této verze bude nyní představovat jen malou (nebo dokonce žádnou!) úpravu změnového souboru.

Při podrobnějším pohledu na program PerCent zjistíte, že se vám jej nepodaří přeložit. Je to dáno tím, že nemáte soubory `ibmanip.h` a `useful.h` a navíc linker nenajde funkci *CutStg* ani operátor *endl* pro **istream**. Potřebujete tedy změnový soubor, v němž nejprve vyřadíme příkazy načítající neexistující soubory:

```
@x
#include␣<ibmanip.h>
#include␣<useful.h>
#include␣<stdlib.h>
@y
#include␣<stdlib.h>
@z
```

Dále musíme nadefinovat prototypy operátoru *endl* a funkce *CutStg*.

```
@x
@<Prototypy@>=
@y
@<Prototypy@>=
istream&␣operator␣endl;
int␣CutStg␣(char*);
@z
```

Nakonec ještě musíme příslušné funkce implementovat. Pro jednoduchost pouze přidáme odpovídající příkazy do sekce „Funkce“. Lepší by ovšem bylo, kdybychom přidali i dokumentaci.

```
@x
@<Funkce@>=
@<Přidej␣procento@>@;
@<Odstraň␣procento@>@;
@<Udělej␣záložní␣soubor@>@;
@y
@<Funkce@>=
```

```
@<Přidej␣procento@>@;
@<Odstraň␣procento@>@;
@<Udělej␣záložní␣soubor@>@;

int␣CutStg␣(char␣*s)␣{
for␣(int␣j=strlen(s)-1;␣␣j>=0␣&&␣s[j]>=0␣&&␣isspace(s[j]);␣␣j--);
s[++j]␣=␣'\0';
return␣j;
}

istream&␣endl␣(istream&␣is)␣{
char␣c;
do␣{␣is.get(c);␣}␣while␣(c␣!=␣'\n'␣&&␣!is.eof()␣&&␣is);
return␣is;
}
@z
```

Ukázali jsme si, že změnový soubor je zcela nezávislý na programovacím jazyku. Lze jej tedy použít naprosto k čemukoliv. Chcete-li např. upravit pro své potřeby balík TEXových maker, můžete si napsat změnový soubor, a protože zde nelze použít `tangle` ani `weave` z `CWEBu`, vytvoříme nový balík pomocí `wmerge`.

Rozsáhlejší balík TEXových maker se svou složitostí vyrovná programům v jiných jazycích. I zde je tedy vhodné využít koncept kultivovaného programování. Můžeme použít některý z WEBů, který není závislý na programovacím jazyce, např. `noweb` nebo `nuweb`. To má ovšem jednu nevýhodu: uživatel TEXu nemusí nutně být programátor, a nebude tudíž umět implementovat WEB na svůj počítač. Každý uživatel TEXu má však TEX. To je využito v balíku DOC.STY, který vytvořil Frank Mittelbach. Definice maker se píší obvyklým způsobem, pouze se vkládají mezi řádky s příkazy `%␣␣␣␣␣\begin{macrocode}` a `%␣␣␣␣␣\end{macrocode}`. Všimněte si čtyř mezer — ty jsou v uvedených příkazech nezbytné. Dokumentace maker se uvádí v komentářích, tedy v řádcích uvedených znaky %. Pokud získáte takový balík maker, můžete jej použít bez úprav tak, jak je. Máte-li DOC.STY, můžete si LATEXem vytisknout dokumentaci.

Tímto způsobem jsem napsal do Zpravodaje článek o vytváření rejstříků. Tak bylo zajištěno, že příklad rejstříku byl vytvořen přesně stejnými makry, která jsem popisoval.

Teď se konečně dostáváme k důvodu, proč jsem psal program PERCENT. Dokumentace pro DOC.STY se nesnadno píše, pokud musíme pro-

centa doplňovat ručně. Navíc se tím ztěžují dodatečné zásahy a TEXspell nehledá překlepy v komentářích. Ve slušném editoru sice lze vytvořit klávesová makra, která přidávají nebo ruší procenta. Jenže my nechceme doplnit procenta do celého textu, ale do mnoha kratších kousků. Navíc musíme přesně dodržet zmíněné čtyři mezery — a to vše bez trochy automatizace vede k mnoha chybám. Program PERCENT jsem skutečně vytvořil proto, abych mohl napsat článek o tvorbě rejstříků, ovšem s vědomím, že jej budu používat i v dalších projektech. Tím jsme se od WEBu vrátili k tomu, čím se tento zpravodaj zabývá především, tj. k TEXu.

wagner@csearn
wagner@earn.cvut.cz

## Recenze s přívažkem

JIŘÍ VESELÝ

Na rozdíl od knížek, které čtenáře varují na začátku první kapitoly a vybízejí ho k tomu, aby si přečetl předmluvu, já upozorňuji na odstavec začínající znamením •. Tam teprve recenze knížky

> George Grätzer: *Math into TEX. A Simple Introduction to AMS-LATEX*, Birkhäuser, Boston 1993, xxix + 294 str., 1 disk, brož., cena neznámá, ISBN 0-8176-3637-4 a také ISBN 3-7643-3637-4

opravdu začíná. Úvahy před můžete bez následků přeskočit, i když se týkají věcí kolem hlavního a jediného objektu recenzované knížky — AMS-LATEXu; snažím se upozornit na alespoň některé důležité souvislosti.

Kdybych byl stylový, psal bych tuto recenzi v AMS-LATEXu jako Grätzer svoji knížku. Konec konců každý zkušený uživatel TEXu však ví, že z kvality tištěné podoby článku nikdo nepozná, zda použitý „macro package“ je kvalitní — nakonec by to byla tedy jen výpověď o tom, že Karel Horák je schopen zvládnout přechod z tohoto poněkud exotického křížence dvou dosti odlišných „supermaker“ do stylu našeho Zpravodaje