1. července 2008 v 02:00

1. PROGRAM VLNA. Program čte vstupní textový soubor a nahrazuje za specifikovanými jednopísmennými slovy (např. v, k, u) mezery symbolem „~“. To zabrání při následném zpracování TEXem zlomit řádek na nevhodných místech, která jsou v rozporu s typografickou normou.

Program sestává z těchto hlavních celků:

⟨ Hlavičkové soubory k načtení 3 ⟩
⟨ Globální deklarace 4 ⟩
⟨ Pomocné funkce 6 ⟩
⟨ Vlnkovací funkce *tie* 26 ⟩
⟨ Hlavní program 5 ⟩

2. Definujeme `BANNER`, což je text, který se objevi při startu programu a obsahuje číslo verze programu. Zde je názorně vidět, že míchání dvou jazyků se nevyhneme. Při tisku textů na terminál nesmíme předpokládat, že tam budou české fonty. V této dokumentaci se setkáme se třemi jazyky: angličtinou (většinou v kódu programu, cestinou v /* komentářích */ a češtinou jinde. Tu cestinu si vynutil fakt, že DOS-ovská varianta `tangle` a `weave` se nesnáší s akcentovanými písmeny v /* komentářích */. A nyní už slíbený (vícejazyčný) `BANNER`.

```
#define BANNER "This␣is␣program␣vlna,␣version␣1.2,␣(c)␣1995,␣2002␣Petr␣Olsak\n"
```

3. V programu jsou použity knihovní funkce, jejíchž prototypy jsou definovány ve třech standardních hlavičkových souborech.

⟨ Hlavičkové soubory k načtení 3 ⟩ ≡

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

Tento kód je použit v sekci 1.

4. Definujeme konstanty pro návratový kód. `OK` pro úspěšný běh, `WARNING` při výskytu aspoň jedné varovné zprávy, `IO_ERR` pro chybu v přístupu ke vtupním nebo výstupním souborům, `BAD_OPTIONS` pro syntaktickou chybu na příkazové řádce a `BAD_PROGRAM` pro případ havárie programu. Ta by neměla nikdy nastat. Proměnná *status* bude obsahovat návratový kód a proměnná *prog_name* bude ukazovat na text nultého parametru příkazové řádky.

```
#define OK 0
#define WARNING 1
#define IO_ERR 2
#define BAD_OPTIONS 3
#define BAD_PROGRAM 4
```

⟨ Globální deklarace 4 ⟩ ≡

```
char *prog_name;
int status;
```

Viz také sekce 7, 15, 16, 20, 25 a 33.

Tento kód je použit v sekci 1.

5. Základní rozvržení funkce *main*.

```
⟨ Hlavní program 5 ⟩ ≡
  int main(argc, argv)
      int argc;
      char **argv;
  {
    ⟨ Lokální proměnné funkce main 9 ⟩;
    prog_name = argv[0];
    status = OK;
    ⟨ Načtení parametrů příkazového řádku 8 ⟩;
    if (¬silent) fprintf(stderr, BANNER);
    ⟨ Inicializace datových struktur 21 ⟩;
    ⟨ Zpracování souborů 11 ⟩;
    return status;
  }
```

Tento kód je použit v sekci 1.

6. Parametry příkazového řádku. Program čte z příkazového řádku postupně (nepovinné) parametry, které začínají znakem „-“. Pak následují jména vstupních a výstupních souborů.

- `-f` ... program pracuje jako filtr (viz sekce ⟨ Zpracování souborů 11 ⟩). Není-li tento parametr použit, program pracuje v tzv. standardním režimu, kdy jednotlivé soubory jsou vstupní i výstupní.
- `-s` ... program nevypíše `BANNER`, ani sumarizaci, ani varování, při nichž není program předčasně ukončen. Všechny tyto výpisy směřují do *stderr*, takže pokud program pracuje v režimu „filtr“, není nutné tento parametr použít.
- `-r` ... program maže pracovní soubor (soubory), které vytváří ve standardním režimu (tj. není použit `-f`). V reřimu filter nemá tento parametr vliv.
- `-v` ... parametr definuje skupinu písmen, které budou interpretovány jako neslabičné předložky. Např. `-v KkSsVvZzOoUuAI`. Pokud není parametr uveden, je použita skupina uvedená v tomto příkladě.
- `-m` ... program neprovádí kontrolu math/text módů, tj. vlnkuje i uvnitř matematického módu TEXu. (Implicite tam nevlnkuje).
- `-n` ... prorgram neprovádí kontrolu verbatim módu, tj. vlnkuje i uvnitř verbatim módu definovaném běžnými prostředími. Imlicite ve verbatim prostředí nevlnkuje.
- `-l` ... LaTEX režim. Při kontrole text-math-verbatim módů jsou brány v úvahu další sekvence, obvyklé v LaTEXových dokumentech.
- `-w` ... WEB režim. Ohraničení verbatim módu je doplněno znaky používanými v dokumentech WEB (např. tento dokument). Důsledek: program vlnkuje dokumentační část každé sekce, ale nikoli kód.

Definujeme funkci *printusage*, které tiskne (při chybě) stručný přehled možných parametrů. Nepodařilo se mi zjistit, jak se ve WEBu napíše kulturně dlouhý string obsahující `\n` s formátovacími požadavky. Byl jsem nucen to takto nehezky zapsat.

⟨ Pomocné funkce 6 ⟩ ≡

```
void printusage( )
{
  fprintf(stderr, "usage: vlna [opt] [filenames]\n""  opt -f :  filter \
      mode: file1 file2 ... file1->file2\n""                  \
      file1        ... file1->stdout\n""                                  ..\
      . stdin->stdout\n""              nofilter: file1 [file2 file3 ...] all are in/ou\
      t\n""       -s :  silent: no messages to stderr\n""       -r :  rmbacku\
      p: if nofilter, removes temporary files\n""       -v charset :  set of lettres to a\
      dd tie, default: KkSsVvZzOoUuAI\n""       -m :  nomath: ignores math modes\\
      n""       -n :  noverb: ignores verbatim modes\n""       -l :  LaTeX m\
      ode\n""       -w :  web mode\n");
}
```

Viz také sekce 10, 17, 18, 19, 22, 23, 36, 38, 41, 42, 44, 45, 47, 49, 51, 52, 53 a 55.

Tento kód je použit v sekci 1.

7. Proměnné *isfilter*, *silent*, *rmbackup*, *nomath*, *noverb*, *latex*, resp. *web* říkají, že je nastaven parametr `-f`, `-s`, `-r`, `-m`, `-n`, `-l`, resp. `-w`. Proměnná *charset* ukazuje buď na implicitní skupinu znaků *charsetdefault*, nebo (při použití parametru `-v`) na text uvedený v příkazovém řádku.

⟨ Globální deklarace 4 ⟩ +≡

```
int isfilter = 0, silent = 0, rmbackup = 0, nomath = 0, noverb = 0, web = 0, latex = 0;
char charsetdefault[ ] = "KkSsVvZzOoUuAI";
char *charset = charsetdefault;
```

8. ⟨ Načtení parametrů příkazového řádku 8 ⟩ ≡

```
while (argc > 1 ∧ argv[1][0] ≡ '-') {
  if (argv[1][2] ≠ 0) printusage(), exit(BAD_OPTIONS);
  switch (argv[1][1]) {
  case 'f': isfilter = 1;
    break;
  case 's': silent = 1;
    break;
  case 'r': rmbackup = 1;
    break;
  case 'v':
    if (argc < 2) printusage(), exit(BAD_OPTIONS);
    argv++;
    argc--;
    charset = argv[1];
    break;
  case 'm': nomath = 1;
    break;
  case 'n': noverb = 1;
    break;
  case 'l': latex = 1;
    break;
  case 'w': web = 1;
    break;
  default: printusage(), exit(BAD_OPTIONS);      /* neznámý parametr */
  }
  argc--;
  argv++;
}
```

Tento kód je použit v sekci 5.

9. Zpracování souborů. Parametr `MAXLEN` definuje maximální možnou délku jména souboru, který vytvoříme jako přechodný, nebo zálohový. Dále deklarujeme proměnné typu „stream“.

```
#define MAXLEN 120
```

```
⟨ Lokální proměnné funkce main 9 ⟩ ≡
  FILE *infile, *outfile;
  char backup[MAXLEN];
  int j;
```

Tento kód je použit v sekci 5.

10. Definujeme funkci pro výpis chybového hlášení při neúspěšném otevření souboru.

```
⟨ Pomocné funkce 6 ⟩ +≡
  void ioerr(f)
       char *f;
  {
    fprintf(stderr, "%s: cannot open file %s\n", prog_name, f);
  }
```

11. Způsob zpracování souborů rozlišíme podle režimu daným přepínačem `-f`.

```
⟨ Zpracování souborů 11 ⟩ ≡
  if (isfilter) ⟨ Zpracování v režimu filter 12 ⟩
  else ⟨ Zpracování všech souborů příkazové řádky 13 ⟩
```

Tento kód je citován v sekci 6.

Tento kód je použit v sekci 5.

12. V režimu *isfilter* ≡ 1 je další zpracování závislé na počtu souborů v příkazové řádce:

- nula souborů – vstup je *stdin* a výstup je *stdout*,
- jeden soubor – je vstupní, výstup je *stdout*,
- dva soubory – první je vstupní, druhý výstupní,
- více souborů – program skončí s chybou.

```
⟨ Zpracování v režimu filter 12 ⟩ ≡
  {
    if (argc > 3) printusage( ), exit(BAD_OPTIONS);
    infile = stdin;
    outfile = stdout;
    if (argc ≥ 2) infile = fopen(argv[1], "r");
    if (infile ≡ Λ) ioerr(argv[1]), exit(IO_ERR);
    if (argc ≡ 3) outfile = fopen(argv[2], "w");
    if (outfile ≡ Λ) ioerr(argv[2]), exit(IO_ERR);
    if (argc ≥ 2) filename = argv[1];
    else filename = Λ;
    tie(infile, outfile);
    if (outfile ≠ stdout) fclose(outfile);
    if (infile ≠ stdin) fclose(infile);
  }
```

Tento kód je použit v sekci 11.

13. V režimu *isfilter* ≡ 0 jsou jednotlivé soubory v příkazovém řádku interpretovány jako vstupní i výstupní. Více souborů v příkazovém řádku má stejný efekt, jako opakované volání programu na jednotlivé soubory. V UNIXu lze tedy např. napsat `vlna␣*.tex` a program doplní vlnky do všech souborů s příponou `tex`. Toto neplatí v DOSu, protože interpretace masky je v UNIXu starostí shellu a nikoli programu samotného. Náš program masku nebude interpretovat. Je-li v tomto režimu nulový počet souborů, program se ukončí s chybou.

```
⟨ Zpracování všech souborů příkazové řádky 13 ⟩ ≡
  {
    if (argc ≡ 1) printusage( ), exit(BAD_OPTIONS);
    while (argc > 1) {
      argc −−;
      argv ++;
      ⟨ Přejmenuj vstup argv[0] na backup a otevři jej jako infile 14 ⟩;
      if (infile ≡ Λ) {
        ioerr(argv[0]);
        continue;
      }
      outfile = fopen(argv[0], "w");
      if (outfile ≡ Λ) {
        ioerr(argv[0]);
        rename(backup, argv[0]);
        status = WARNING;
        continue;
      }
      filename = argv[0];
      tie(infile, outfile);
      fclose(outfile), fclose(infile);
      if (rmbackup) remove(backup);
    }
  }
```

Tento kód je použit v sekci 11.

14. Při *isfilter* ≡ 0 program přejmenuje každý zpracovávaný soubor tak, že změní poslední písmeno názvu souboru na znak `~`. Tento přejmenovaný soubor bude otevřen jako vstupní a výstupem bude původní soubor. Vstupní soubor při *rmbackup* ≡ 0 zůstane zachován jako záloha.

Proč vlnku nepřidáváme na konec názvu souboru, ale měníme ji za poslední znak souboru? Protože chceme, aby program fungoval i v tak nemožných systémech, jako je DOS.

```
⟨ Přejmenuj vstup argv[0] na backup a otevři jej jako infile 14 ⟩ ≡
  infile = Λ;
  j = strlen(argv[0]) − 1;
  if (j ≥ MAXLEN ∨ argv[0][j] ≡ '~') {
    if (¬silent) fprintf(stderr, "%s:␣the␣conflict␣of␣file␣name␣%s\n", prog_name, argv[0]);
  }
  else {
    strcpy(backup, argv[0]);
    backup[j] = '~';
    remove(backup);
    j = rename(argv[0], backup);
    if (j ≡ 0) infile = fopen(backup, "r");
  }
```

Tento kód je použit v sekci 13.

15. Patterny. Abychom mohli účelně definovat chování programu v různých situacích, zavedeme datovou strukturu `PATTERN`. Zhruba řečeno, budeme sledovat vstup znak po znaku a pokud bude část vstupu souhlasit s definovaným patternem, provedeme námi požadovanou akci. Například nejčastější aktivitu, přidání vlnky uvnitř řádku, spustíme v okamžiku, kdy vstupní text odpovídá patternu „␣(v␣p“, kde „␣“ znamená jedna nebo více mezer a tabelátorů, „(“ je nula nebo více otevíracích závorek všeho druhu, „v“ znamená jedno písmeno z množiny předložek (viz *charset*) a „p“ zde znamená libovolné písmeno. Příklad zde není zcela přesný. Přesně jsou všechny patterny pro náš program definovány v závěrečných sekcích tohoto povídání.

Pattern bude znamenat konečnou sekvenci tzv. pozic patternu (`PATITEM`). Cykly uvnitř pozic pro jednoduchost nepřipustíme. Každá pozice obsahuje řetězec znaků, uvažovaný pro danou pozici (v příkladu pozice „␣“ by obsahovala mezeru a tabelátor, zatímco pozice v odpovídá *charset*). Každá pozice má svůj přepínač (*flag*), který obsahuje informaci o tom, zda shodu testovaného znaku s některým prvkem v množině znaků budeme považovat za úspěch či neúspěch a zda pozice se ve zkoumaném řetězci může vyskytovat právě jednou nebo opakovaně. Jako druhý případ stačí implementovat „nula nebo více“ protože „jedna nebo více“ lze popsat pomocí dvou pozic, první „právě jednou“ a následující „nula nebo více“. Jednotlivé pozice jsou zřetězeny ukazatelem *next*, poslední pozice má $next \equiv \Lambda$. Stejně tak jednotlivé patterny budeme sestavovat do seznamů a budou rovněž zřetězeny ukazatelem *next*.

Pattern kromě řetězu pozic obsahuje ukazatel na funkci (proceduru) *proc*, která se má vykonat v případě, že testovaný řetězec vyhovuje patternu.

```
#define ONE 1    /* flag: prave jeden vyskyt */
#define ANY 2    /* flag: nula nebo vice */
#define ONE_NOT -1    /* flag: prave jednou, znak nesmi byt v mnozine */
#define ANY_NOT -2    /* flag: nula nebo vice, znak nesmi byt v mnozine */
```

⟨ Globální deklarace 4 ⟩ +≡

```
typedef struct PATITEM {    /* jedna pozice patternu */
  char *str;    /* seznam znaku na teto pozici */
  int flag;    /* vyznam seznamu znaku */
  struct PATITEM *next;    /* nasledujici pozice patternu */
} PATITEM;
typedef struct PATTERN {    /* jeden pattern */
  PATITEM *patt;    /* ukazatel na prvni pozici */
  void(*proc)( );    /* procedura spustena pri souhlasu patternu */
  struct PATTERN *next;    /* nasledujici v seznamu vsech patternu */
} PATTERN;
```

16. Deklarujeme některé globální proměnné pro práci s patterny. *lapi* je pole obsahující ukazatele na aktuální pozice v otevřených patternech. Říkáme, že „pattern je otevřen“, pokud zkoumaný řetězec s ním *začíná* souhlasit. Pattern se uzavře, pokud nastane jedna ze dvou možností: zkoumaný řetězec s mím souhlasí až do konce (v takovém případě se provede procedura *proc*), nebo při vyšetřování dalších znaků ze zkoumaného řetězce přestane řetězec s patternem souhlasit.

V dané chvíli může být pattern otevřen několikrát. Např. pattern `abac` je při stringu `aba` při výskytu druhého `a` otevřen podruhé. Proto pole obsahuje ukazatele na právě aktuální pozici patternu a nikoli na pattern jako takový.

V poli *lapi* budou na počátku samá Λ (to se při překladu inicializuje samo) a přemazání ukazatele na pozici konstantou Λ budeme považovat za zavření patternu. Vedle pole *lapi* souměrně udržujeme pole *lapt*, do něhož budeme ukládat ukazatele na odpovídající otevřený pattern. Tuto informaci použijeme v případě, že potřebujeme např, znát *proc* patternu.

listpatt bude ukazovat na začátek aktuálního seznamu patternů. Seznamy budeme mít dva. Jeden se použije, nacházíme-li se mimo komentář a druhý v případě, že se nacházíme v prostoru TEXovského komentáře (tj. za procentem). Starty těchto seznamů patternů jsou *normallist* a *commentlist* a aktivní *listpatt* má vždy jednu z těchto dvou hodnot.

Proměnné *lastpt* a *lastpi* použijeme pro budování řetězové struktury patternů.

Proměnná *c* obsahuje právě testovaný znak ze vstupu (který se rovněž přepíše do bufferu *buff*). Z bufferu občas ukládáme data do výstupního proudu. Děláme to ale vždy jen v okamžiku, kdy není otevřen žádný pattern. Tehdy totiž „nehrozí“ situace, že by nějaká procedura vyvolaná souhlasem patternu požadovala v tomto bufferu nějaké změny se zpětnou platností. O vyprázdnění bufferu se začneme zajímat až v okamžiku, kdy je zaplněn aspoň na hodnotu `BUFI`, abychom proceduru přepisu bufferu do výstupního proudu neaktivovali zbytečně často.

```
#define MAXPATT 200    /* maximalni pocet patternu */
#define MAXBUFF 500    /* velikost bufferu pro operace */
#define BUFI 300    /* velikost stredniho zaplneni */
```

⟨ Globální deklarace 4 ⟩ +≡

```
PATITEM *lapi[MAXPATT];    /* pole ukazatelu na aktualni pozice */
PATTERN *lapt[MAXPATT];    /* pole odpovidajicich ukazatelu na patterny */
PATTERN *listpatt, *normallist, *commentlist, *pt, *lastpt = Λ;
PATITEM *pi, *lastpi = Λ;
char c;    /* zrovna nacetny znak */
char buff[MAXBUFF];    /* prechodny buffer */
int ind;    /* aktualni pozice prechodneho bufferu */
```

17. Nyní definujeme pomocné funkce *setpattern*, *setpi* a *normalpattern*. Tyto funkce alokují paměť pomocí standardní funkce *malloc*. Abychom mohli ohlídat případnou chybu při alokaci, budeme allokovat paměť zprostředkovaně pomocí funkce *myalloc*.

⟨ Pomocné funkce 6 ⟩ +≡

```
void *myalloc(size)
    int size;
{
  void *p;
  p = malloc(size);
  if (p ≡ Λ) {
    fprintf(stderr, "%s, no memory, malloc failed\n", prog_name);
    exit(BAD_PROGRAM);
  }
  return p;
}
```

18. Funkce *setpattern* alokuje paměťové místo struktury **PATTERN** a napojí ji pomocí proměnné *lastpt* na už alokovaný řetěz patternů. Vrátí ukazatel na nově alokované místo. Jednotlivé pozice patternu se musí následovně alokovat pomocí *setpi*.

```
⟨ Pomocné funkce 6 ⟩ +≡
  PATTERN *setpattern(proc)
  void(*proc)( );
  {
    PATTERN *pp;
    pp = myalloc(sizeof(PATTERN));
    pp→proc = proc;
    pp→next = Λ;
    pp→patt = Λ;
    if (lastpt ≠ Λ) lastpt→next = pp;
    lastpt = pp;
    lastpi = Λ;
    return pp;
  }
```

19. Funkce *setpi* alokuje paměťové místo pro jednu pozici patternu. Provede zřetězení tak, aby první pozice řetězu pozic byla zaznamenána v položce *patt* ve struktuře **PATTERN** a další byly provázány položkou *next* ve struktuře **PATITEM**. Poslední pozice má *next* ≡ Λ.

```
⟨ Pomocné funkce 6 ⟩ +≡
  void setpi(str, flag)
      char *str;
      int flag;
  {
    PATITEM *p;
    p = myalloc(sizeof(PATITEM));
    p→str = str;
    p→flag = flag;
    p→next = Λ;
    if (lastpi ≡ Λ) lastpt→patt = p;
    else lastpi→next = p;
    lastpi = p;
  }
```

20. Připravme si půdu pro funkci *normalpattern*. Tato funkce alokuje strukturu pro jeden pattern včetně pozic patternu na základě vstupního stringu. Každá pozice patternu obsahuje v množině znaků jediný znak a má *flag* = `ONE`. Znaky ve vstupním stringu odpovídají po řadě jednotlivým pozicím. Vytvoří se vlastně jakýsi absolutní pattern, tj. testovaný řetězec se musí přesně shodovat s uvedeným stringem. Výjimku tvoří znak `"."`, který se interpretuje jako nula nebo více mezer. Chceme-li tečku vnutit do patternu, napíšeme dvě tečky za sebou.

Nejdříve deklarujeme pole všech možných jednopísmenných stringů.

```
⟨ Globální deklarace 4 ⟩ +≡
  char strings[512];
  int i;
```

21. Inicializujeme toto pole (znak, nula, znak, nula, atd...).

```
⟨ Inicializace datových struktur 21 ⟩ ≡
  for (i = 0; i < 256; i++) {
    strings[2 * i] = (char) i;
    strings[2 * i + 1] = 0;
  }
```

Viz také sekce 34, 37, 39, 40, 43, 46, 48, 50 a 54.

Tento kód je použit v sekci 5.

22. Definujme funkci *normalpattern*.

```
⟨ Pomocné funkce 6 ⟩ +≡
  PATTERN *normalpattern(proc, str)
  void(*proc)( );
  char *str;
  {
    PATTERN *pp;
    int j = 0;
    pp = setpattern(proc);
    while (str[j]) {
      if (str[j] ≡ '.') {
        j++;
        if (str[j] ≠ '.') {
          setpi(blankscr, ANY);
          continue;
        }
      }
      setpi(&strings[(unsigned char) str[j] * 2], ONE);
      j++;
    }
    return pp;
  }
```

23. Funkce *match*. Definujeme funkci, která na základě hodnoty znaku c (proměnná c je definována jako globální), a pozice patternu p (parametr funkce) vrátí informaci o tom, zda znak souhlasí s patternem. Záporná čísla `FOUND`, resp. `NOFOUND` znamenají, že je třeba uzavřít pattern s tím, že vzor odpovídá, resp. neodpovídá patternu. Nezáporné číslo vrátí v případě, že zkoumaný vstup stále souhlasí s patternem, ale není ještě rozhodnuto. Velikost návratové hodnoty v takovém případě udává, o kolik pozic je třeba se posunout v patternu, abychom měli ukazatel na pozici patternu v souhlase s novou situací, způsobenou znakem c.

Pokud je c v množině znaků pro danou pozici $p\rightarrow str$, bude $m \equiv 1$, jinak je $m \equiv -1$. Pokud tímto číslem pronásobíme hodnotu $p\rightarrow flag$, nemusíme větvení podle $p\rightarrow flag$ programovat dvakrát. Hodnoty *flag* jsou totiž symetrické podle nuly, např. `ANY` $\equiv -$`ANY_NOT`.

```
#define FOUND -1
#define NOFOUND -2
⟨ Pomocné funkce 6 ⟩ +≡
  int match(p)
      PATITEM *p;
  {
    int m;
    if (strchr(p→str, c) ≠ Λ) m = 1;      /* Znak nalezen */
    else m = -1;      /* Znak nenalezen */
    switch (m * p→flag) {
    case ANY: return 0;      /* Souhas, neni nutny posun */
    case ONE:
      if (p→next ≡ Λ) return FOUND;
      return 1;      /* Souhas, nutny posun o 1 */
    case ONE_NOT: return NOFOUND;      /* Nesouhlas */
    case ANY_NOT: ⟨ Vrať hodnotu podle následující pozice patternu 24 ⟩;
    }
    return 0;      /* Tady bychom nikdy nemeli byt, return pro potlaceni varovani */
  }
```

24. O kolik pozic je třeba se posunout a s jakým výsledkem zjistíme rekurzivním voláním funkce *match*.

```
⟨ Vrať hodnotu podle následující pozice patternu 24 ⟩ ≡
  switch (m = match(p→next)) {
  case NOFOUND: return NOFOUND;
  case FOUND: return FOUND;
  default: return 1 + m;
  }
```

Tento kód je použit v sekci 23.

25. Vlnkovací funkce. Nejprve připravíme globální deklarace pro „vlnkovací“ funkci *tie*. Funkce *tie* „ovlnkuje“ vstupní soubor *infile* a vytvoří soubor *outfile*. Při *silent* = 0 tiskne závěrečnou zprávu o zpracování. V této zprávě se objeví jméno souboru, které se funkce „dozví“ prostřednictvím globální proměnné *filename*. Proměnná *numline* počítá řádky, proměnná *numchanges* sčítá změny, tj. počet doplněných vlnek. Proměnná *mode* nabývý některé z hodnot `TEXTMODE`, `MATHMODE`, `DISPLAYMODE` a `VERBMODE` podle stavu ve čteném textu.

```
#define TEXTMODE 0
#define MATHMODE 1
#define DISPLAYMODE 2
#define VERBMODE 3
```

```
⟨ Globální deklarace 4 ⟩ +≡
  char *filename;      /* jmeno zpracovavaneho souboru */
  long int numline, numchanges;      /* pro zaverecnou statistiku */
  int mode;
```

26. Nyní definujeme vlnkovací funkci *tie*. Veškerá činnost se opírá o strukturu patternů. Výhodné je (z důvodu rychlosti) „natvrdo“ zde implementovat jen přepínání mezi stavem čtení z oblasti komentáře (*listpatt* ≡ *commentlist*) a mimo komentář (*listpatt* ≡ *normallist*);

```
⟨ Vlnkovací funkce tie 26 ⟩ ≡
  void tie(input, output)
      FILE *input, *output;
  {
    int ap;      /* ap je pocet otevrenych patternu */
    register int k, m, n;
    int ic;
    PATTERN *pp;
    PATITEM *pi;
    ⟨ Inicializace proměnných při startu funkce tie 27 ⟩;
    while (¬feof(input)) {
      ⟨ Otevři nové patterny 30 ⟩;
      if (ap ≡ 0 ∧ ind > BUFI ∧ c ≠ '\\') ⟨ Vyprázdni buffer 28 ⟩;
      if (ind ≥ MAXBUFF) {
        fprintf(stderr, "Operating␣buffer␣overflow,␣is␣anything␣wrong?\n");
        exit(BAD_PROGRAM);
      }
      if ((ic = getc(input)) ≡ EOF)      /* opravil Cejka Rudolf */
        break;
      buff[ind++] = c = ic;
      if (c ≡ '\n') numline++, listpatt = normallist;
      if (c ≡ '%' ∧ mode ≠ VERBMODE ∧ buff[ind − 2] ≠ '\\') listpatt = commentlist;
      ⟨ Projdi otevřené patterny 29 ⟩;
    }
    ⟨ Vyprázdni buffer 28 ⟩;
    if (¬web) checkmode();      /* zaverecna kontrola modu */
    if (¬silent) ⟨ Tiskni závěrečnou zprávu 32 ⟩;
  }
```

Tento kód je použit v sekci 1.

27. ⟨ Inicializace proměnných při startu funkce *tie* 27 ⟩ ≡

```
for (k = 0; k < MAXPATT; k++) lapi[k] = Λ;
c = '\n';
buff[0] = mode = ap = 0;
ind = 1;
numline = 1;
numchanges = 0;
mode = TEXTMODE;
```

Viz také sekci 35.

Tento kód je použit v sekci 26.

28. Při manipulaci s bufferem byl použit jeden trik. Veškeré načtené znaky začínají až od *buff*[1], zatímco *buff*[0] je rovno nule. Je to proto, že některé algoritmy se vrací o jeden znak zpět za svůj pattern, aby zjistily, zda tam není symbol „\“ (například na výskyt sekvence \% je třeba reagovat jinak, než na výskyt obyčejného procenta). Kdybychm zazačali od *buff*[0], v některých situacích bychom se ptali, zda *buff*[−1] ≡ '\\', tj. sahali bychom na neošetřené místo v paměti.

⟨ Vyprázdni buffer 28 ⟩ ≡

```
{
  buff[ind] = 0;
  fputs(&buff[1], output);
  ind = 1;
}
```

Tento kód je použit v sekci 26.

29. Při procházení otevřenými patterny posunujeme v poli *lapi* pozice jednotlivých patternů podle pokynů funkce *match*, případně pattern zavřeme a případně vyvoláme proceduru patternu.

Některé patterny v poli *lapi* už mohou být zavřeny, takže je nutno s tímto polem pracovat jako s jakýmsi děravým sýrem.

⟨ Projdi otevřené patterny 29 ⟩ ≡

```
n = ap;
k = 0;
while (n) {
  while (lapi[k] ≡ Λ) k++;      /* zastav se na prvnim ukazateli na pattern */
  switch (m = match(lapi[k])) {
  case FOUND: (*lapt[k]→proc)( );      /* Pattern nalezen, spustit proceduru */
  case NOFOUND: lapi[k] = Λ;      /* Deaktivace patternu */
    ap−−;
    break;
  default:
    while (m−−) lapi[k] = lapi[k]→next;      /* dalsi pozice patternu */
  }
  k++;
  n−−;
}
```

Tento kód je použit v sekci 26.

30. Při otevírání nových patternů, které nejsou v tuto chvíli zablokovány, se hned vypořádáme s takovými patterny, které nám dávají rovnou odpověď typu `FOUND` nebo `NOFOUND`. V takových případech ani nezanášíme ukazatel na pozici do pole *lapi*.

```
⟨ Otevři nové patterny 30 ⟩ ≡
  pp = listpatt;
  while (pp ≠ Λ) {
    switch (m = match(pp→patt)) {
    case FOUND: (*pp→proc)( );      /* spustit proceduru */
    case NOFOUND: break;
    default: ⟨ Vytvoř ukazatel na nový pattern a break 31 ⟩;
    }
    pp = pp→next;
  }
```

Tento kód je použit v sekci 26.

31. Není-li hned známa odpověď, zda pattern vyhovuje či nikoli, překontrolujeme nejdříve, zda už není pattern ve stejné pozici otevřený. Pak najdeme první „díru“ v tabulce *lapi* a tam uhnízdíme nový ukazatel na pozici v patternu.

```
⟨ Vytvoř ukazatel na nový pattern a break 31 ⟩ ≡
  pi = pp→patt;
  while (m−−) pi = pi→next;
  n = ap;
  k = 0;
  while (n) {
    if (lapi[k] ≡ pi) break;
    if (lapi[k++] ≠ Λ) n−−;
  }
  if (¬n) {
    k = 0;
    while (lapi[k] ≠ Λ) k++;
    if (k ≥ MAXPATT) {
      fprintf(stderr, "I␣cannot␣allocate␣pp,␣is␣anything␣wrong?\n");
      exit(BAD_PROGRAM);
    }
    lapt[k] = pp;
    lapi[k] = pi;
    ap++;
  }
```

Tento kód je použit v sekci 30.

32. Poslední věcí ve funci *tie* je tisk závěrečné statistiky zpracování.

```
⟨ Tiskni závěrečnou zprávu 32 ⟩ ≡
  fprintf(stderr, "~~~␣file:␣%s\t␣␣lines:␣%ld,␣changes:␣%ld\n", filename, numline, numchanges);
```

Tento kód je použit v sekci 26.

33. Inicializace patternů. Po vytvoření předchozího kódu opírajícího se o patterny máme nyní v ruce poměrně silný nástroj na definování různých činností programu prostým vytvořením patternu a příslušné jeho procedury. Pokud budeme chtít v budoucnu nějaký rys programu přidat, pravděpodobně to bude snadné.

Nejprve deklarujeme některé často používané skupiny znaků v patternech.

```
⟨ Globální deklarace 4 ⟩ +≡
  char tblanks[] = "␣~\t";
  char blanks[] = "␣\t";
  char blankscr[] = "␣\t\n";
  char tblankscr[] = "␣~\t\n";
  char nochar[] = "%~\n";
  char cr[] = "\n";
  char prefixes[] = "[({~";
  char dolar[] = "$";
  char backslash[] = "\\";
  char openbrace[] = "{";
  char letters[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
  PATTERN *vlnkalist, *mathlist, *parcheck, *verblist;
```

34. Začneme definicí nejčastěji používaného patternu na vlnkování uvnitř řádku. Připomeňme, že opakované volání funkce *setpattern* vytváří interně seznam patternů, přičemž o jejich propojení se nemusíme starat. Vyzvedneme si z návratového kódu funkce pouze ukazatel na první položku seznamu *normallist*. Stejně tak opakované volání funkce *setpi* vytváří seznam pozic pro naposledy deklarovaný pattern.

```
⟨ Inicializace datových struktur 21 ⟩ +≡
  vlnkalist = setpattern(vlnkain);
  setpi(tblankscr, ONE);
  setpi(tblanks, ANY);
  setpi(prefixes, ANY);
  setpi(charset, ONE);
  setpi(blanks, ONE);
  setpi(blanks, ANY);
  setpi(nochar, ONE_NOT);
```

35. ⟨ Inicializace proměnných při startu funkce *tie* 27 ⟩ +≡

```
  listpatt = normallist = vlnkalist;
```

36. ⟨ Pomocné funkce 6 ⟩ +≡

```
  void vlnkain()
  {
    char p;
    ind--;
    p = buff[ind--];
    while (strchr(blanks, buff[ind]) ≠ Λ) ind--;
    ind++;
    buff[ind++] = '~';
    buff[ind++] = p;
    numchanges++;
  }
```

37. Podobně pro tvorbu vlnky „přes řádek“ vytvoříme pattern a kód procedury.

```
⟨ Inicializace datových struktur 21 ⟩ +≡
  setpattern(vlnkacr);
  setpi(tblankscr, ONE);
  setpi(tblanks, ANY);
  setpi(prefixes, ANY);
  setpi(charset, ONE);
  setpi(blanks, ANY);
  setpi(cr, ONE);
  setpi(blanks, ANY);
  setpi(nochar, ONE_NOT);
```

38. V proceduře k tomuto patternu musíme ošetřit případ typu „`a~v\np`“, kdy nelze prostě přehodit „`\n`“ za „`v`“, protože bychom roztrhli mezeru svázanou vlnkou už dříve. Proto musíme vyhledat vhodné místo pro roztržení řádku, které bude až *před* znakem „`a`“. Při důsledném ošetření tohoto fenoménu můžeme dokonce narazit na situaci „`\n␣v~v~v\np`“, kde nemůžeme vložit „`\n`“ před první výskyt „`v`“, protože bychom dostali „`\n\n`“, tedy prázdný řádek. Ten je v TEXu interperetován odlišně. V této výjimečné situaci pouze zrušíme stávající (v pořadí druhé) „`\n`“ a nebudeme vytvářet nové. Na výstupu bude soubor o jeden řádek kratší.

```
⟨ Pomocné funkce 6 ⟩ +≡
  void vlnkacr( )
  {
    char p;
    int i, j;
    ind−−;
    p = buff[ind−−];
    while (strchr(blankscr, buff[ind]) ≠ Λ) ind−−;
    i = ind;      /* misto predlozky, kterou chceme vazat */
    while (i ≥ 0 ∧ (strchr(blankscr, buff[i]) ≡ Λ)) i−−;
    j = i;
    while (i ≥ 0 ∧ (strchr(blanks, buff[i]) ≠ Λ)) i−−;
    if (i ≥ 0 ∧ buff[i] ≡ '\n') j = −1;
    if (j ≥ 0) buff[j] = '\n';
    else numline−−;
    ind++;
    buff[ind++] = '~';
    buff[ind++] = p;
    numchanges++;
  }
```

39. Nyní vytvoříme patterny pro případy typu `\uv{v lese}`.

```
⟨ Inicializace datových struktur 21 ⟩ +≡
  setpattern(vlnkain);      /* na radku */
  setpi(tblankscr, ONE);
  setpi(backslash, ONE);
  setpi(letters, ONE);
  setpi(letters, ANY);
  setpi(openbrace, ONE);
  setpi(prefixes, ANY);
  setpi(charset, ONE);
  setpi(blanks, ONE);
  setpi(blanks, ANY);
  setpi(nochar, ONE_NOT);
  setpattern(vlnkacr);      /* pres radek */
  setpi(tblankscr, ONE);
  setpi(backslash, ONE);
  setpi(letters, ONE);
  setpi(letters, ANY);
  setpi(openbrace, ONE);
  setpi(prefixes, ANY);
  setpi(charset, ONE);
  setpi(blanks, ANY);
  setpi(cr, ONE);
  setpi(blanks, ANY);
  setpi(nochar, ONE_NOT);
```

40. Vytvoříme patterny a proceduru pro potlatčení tvorby vlnky u písmen těsně následujících sekvence `\TeX` a `\LaTeX`. Tj. nechceme, aby např z textu „`Vlastnosti␣\TeX␣u␣jsou...`“ jsme dostali text s nesprávně vázaným písmenem „`Vlastnosti␣\TeX␣u~jsou...`“.

```
⟨ Inicializace datových struktur 21 ⟩ +≡
  normalpattern(tielock, "\\TeX");
  setpi(blankscr, ONE);
  normalpattern(tielock, "\\LaTeX");
  setpi(blankscr, ONE);
```

41. Procedura *tielock* obsahuje nečistý trik. Při provádění procedury je právě načten znak z *blankscr* a je uložen do buff. Testy na otevírání nových patternů pro tento znak teprve budou následovat a testují se na hodnotu proměnné *c*. Stačí tedy změnit hodnotu *c* a vlnkovací patterny se neotevřou.

```
⟨ Pomocné funkce 6 ⟩ +≡
  void tielock()
  {
    c = 1;
  }
```

42. Ošetříme nyní přechod do/z matematického režimu TEXu. Uvnitř math módu vlnky neděláme. Při zjištěném nesouladu v přechodech mezi math-módy spustíme následující proceduru.

```
⟨ Pomocné funkce 6 ⟩ +≡
  void printwarning()
  {
    if (¬silent)
      fprintf(stderr, "~!~␣warning:␣text/math/verb␣mode␣mismatch,␣␣file:␣%s,␣␣line:␣%ld\n",
          filename, numline − (c ≡ '\n' ? 1 : 0));
    status = WARNING;
  }
```

43. Začneme patterny pro přechod do/z matematického režimu, ohraničeného jedním dolarem, nebo v LaTeXu příslušnými sekvencemi. Sekvence LaTeXu `\(` a `\)` nejsou zahrnuty, protože bývají často předefinovány k jiným užitečnějším věcem.

```
⟨ Inicializace datových struktur 21 ⟩ +≡
  if (¬nomath) {
    mathlist = setpattern(onedollar);
    setpi(dolar, ONE);
    setpi(dolar, ONE_NOT);
    if (latex) {
      normalpattern(mathin, "\\begin.{math}");
      normalpattern(mathout, "\\end.{math}");
    }
  }
```

44.
```
⟨ Pomocné funkce 6 ⟩ +≡
  void mathin( )
  {
    if (mode ≠ TEXTMODE) printwarning( );
    mode = MATHMODE;
    normallist = listpatt = mathlist;
  }
  void mathout( )
  {
    if (mode ≠ MATHMODE) printwarning( );
    mode = TEXTMODE;
    normallist = listpatt = vlnkalist;
  }
```

45. Při programování procedury *onedollar* nesmíme zapomenout na výskyt sekvence `\$`. V tom případě akci ignorujeme. Podobně u sekvence `$$` souhlasí ten druhý dolar s naším patternem, ale to už jsme uvnitř display módu. V takovém případě také nic neděláme.

```
⟨ Pomocné funkce 6 ⟩ +≡
  void onedollar( )
  {
    if (buff[ind − 3] ≡ '\\' ∨ (buff[ind − 3] ≡ '$' ∧ buff[ind − 4] ≠ '\\')) return;
    if (mode ≡ DISPLAYMODE) printwarning( );
    else {
      if (mode ≡ TEXTMODE) mathin( );
      else mathout( );
    }
  }
```

46. Pokud najdeme prázdný řádek, překontrolujeme, zda náhodou nejsme v math-módu. Pokud ano, vypíšeme varování a přejdeme do textového módu.

```
⟨ Inicializace datových struktur 21 ⟩ +≡
  parcheck = setpattern(checkmode);
  setpi(cr, ONE);
  setpi(blanks, ANY);
  setpi(cr, ONE);
```

47. ⟨ Pomocné funkce 6 ⟩ +≡

```
void checkmode( )
{
  if (mode ≠ TEXTMODE) {
    printwarning( );
    mode = TEXTMODE;
    normallist = listpatt = vlnkalist;
  }
}
```

48. Nyní ošetříme výskyt dvou dolarů, tj. vstup do/z display módu. Rovněž mysleme na LaTEXisty a jejich prostředí pro display-mód. Protože je možná alternativa s hvězdičkou na konci názvu prostředí, raději už uzavírací závorku do patternu nezahrnujeme.

⟨ Inicializace datových struktur 21 ⟩ +≡

```
if (¬nomath) {
  normalpattern(twodollars, "$$");
  if (latex) {
    normalpattern(displayin, "\\begin.{displaymath");
    normalpattern(displayin, "\\begin.{equation");
    normalpattern(displayout, "\\end.{displaymath");
    normalpattern(displayout, "\\end.{equation");
  }
}
```

49. ⟨ Pomocné funkce 6 ⟩ +≡

```
void displayin( )
{
  if (mode ≠ TEXTMODE) printwarning( );
  mode = DISPLAYMODE;
  normallist = listpatt = parcheck;
}
void displayout( )
{
  if (mode ≠ DISPLAYMODE) printwarning( );
  mode = TEXTMODE;
  normallist = listpatt = vlnkalist;
}
void twodollars( )
{
  if (buff[ind − 3] ≡ '\\') return;
  if (mode ≡ DISPLAYMODE) displayout( );
  else displayin( );
}
```

50. Následuje ošetření tzv. verbatim módu. Pro plain i LaTEX jsou nejčastější závorky pro verbatim mod tyto (variantu s `\begtt` používám s oblibou já).

```
⟨ Inicializace datových struktur 21 ⟩ +≡
  if (¬noverb) {
    verblist = normalpattern(verbinchar, "\\verb");
    setpi(blankscr, ANY);
    setpi(blankscr, ONE_NOT);
    normalpattern(verbin, "\\begtt");
    if (latex) normalpattern(verbin, "\\begin.{verbatim");
  }
  if (web) {
    normalpattern(verbin, "@<");
    normalpattern(verbin, "@d");
  }
  if (¬noverb) {
    verboutlist[0] = setpattern(verbout);
    setpi(verbchar, ONE);
    verboutlist[1] = normalpattern(verbout, "\\endtt");
    if (latex) verboutlist[2] = normalpattern(verbout, "\\end{verbatim");
  }
  if (web) {
    verboutlist[3] = normalpattern(verbout, "@␣");
    normalpattern(verbout, "@*");
    normalpattern(verbout, "@>|");
  }
```

51. Procedura *verbinchar* se od „společné“ procedury *verbin* liší v tom, že zavede do stringu *verbchar* momentální hodnotu proměnné *c*. Proto druhý výskyt této hodnoty verbatim režim ukončí.

```
⟨ Pomocné funkce 6 ⟩ +≡
  int prevmode;
  PATTERN *prevlist, *verboutlist[4];
  char verbchar[2];
  void verbinchar( )
  {
    prevmode = mode;
    verbchar[0] = c;
    c = 1;
    listpatt = normallist = verboutlist[0];
    prevlist = listpatt→next;
    listpatt→next = Λ;
    mode = VERBMODE;
  }
```

52. Při programování „obecné“ funkce *verbin* musíme dbát na to, aby zůstal aktivní pouze odpovídající „výstupní“ pattern k danému vstupnímu. Také si zapamatujeme mód, ze kterého jsme do verbatim oblasti vstoupili, abychom se k němu mohli vrátit (např. uvnitř math. módu může být `\hbox` a v něm lokálně verbatim konstrukce).

```
⟨ Pomocné funkce 6 ⟩ +≡
  void verbin( )
  {
    int i;
    i = 0;
    prevmode = mode;
    switch (c) {
    case 't': i = 1;
      break;
    case 'm': i = 2;
      break;
    case '<': ;
    case 'd': i = 3;
      if (buff[ind − 3] ≡ '@') return;      /* dvojity @ ignorovat */
      break;
    }
    listpatt = normallist = verboutlist[i];
    prevlist = listpatt→next;
    if (c ≠ '<' ∧ c ≠ 'd') listpatt→next = Λ;
    mode = VERBMODE;
  }
```

53.
```
⟨ Pomocné funkce 6 ⟩ +≡
  void verbout( )
  {
    if (mode ≠ VERBMODE) return;
    if (web ∧ buff[ind − 2] ≡ '@' ∧ buff[ind − 3] ≡ '@') return;
    mode = prevmode;
    normallist→next = prevlist;
    switch (mode) {
    case DISPLAYMODE: normallist = listpatt = parcheck;
      break;
    case MATHMODE: normallist = listpatt = mathlist;
      break;
    case TEXTMODE: normallist = listpatt = vlnkalist;
      break;
    }
  }
```

54. Nyní implementujeme vlastnost dříve používaného programu vlnka, tj. že lze jeho činnost vypnout a opět zapnout v komentářích. Vytváříme druhý nezávislý seznam patternů a proto nejprve pronulujeme *lastpt*.

```
⟨ Inicializace datových struktur 21 ⟩ +≡
  lastpt = 0;
  commentlist = normalpattern(tieoff, "%.~.-");
  normalpattern(tieon, "%.~.+");
```

55. ⟨ Pomocné funkce 6 ⟩ +≡

```
void tieoff( )
{
  normallist = Λ;
}
void tieon( )
{
  normallist = vlnkalist;
}
```

56. Další plánovaná vylepšení. Program by mohl číst definici svého chování nejen z příkazové řádky, ale v mnohem kompletnější podobě, včetně uživatelsky definovaných patternů, z komentářové oblasti ve čteném souboru. Parametry zde uvedené by mohly mít vyšší prioritu, než parametry z příkazové řádky a mohl by se třeba rozšiřovat seznam sekvencí, za nimiž písmena nemají být vázana vlnkou (zatím je implemenováno na pevno jen `\TeX` a `\LaTeX`).

57. Rejstřík.

⟨ Globální deklarace 4, 7, 15, 16, 20, 25, 33 ⟩ Použito v sekci 1.
⟨ Hlavičkové soubory k načtení 3 ⟩ Použito v sekci 1.
⟨ Hlavní program 5 ⟩ Použito v sekci 1.
⟨ Inicializace datových struktur 21, 34, 37, 39, 40, 43, 46, 48, 50, 54 ⟩ Použito v sekci 5.
⟨ Inicializace proměnných při startu funkce *tie* 27, 35 ⟩ Použito v sekci 26.
⟨ Lokální proměnné funkce *main* 9 ⟩ Použito v sekci 5.
⟨ Načtení parametrů příkazového řádku 8 ⟩ Použito v sekci 5.
⟨ Otevři nové patterny 30 ⟩ Použito v sekci 26.
⟨ Přejmenuj vstup *argv*[0] na *backup* a otevři jej jako *infile* 14 ⟩ Použito v sekci 13.
⟨ Pomocné funkce 6, 10, 17, 18, 19, 22, 23, 36, 38, 41, 42, 44, 45, 47, 49, 51, 52, 53, 55 ⟩ Použito v sekci 1.
⟨ Projdi otevřené patterny 29 ⟩ Použito v sekci 26.
⟨ Tiskni závěrečnou zprávu 32 ⟩ Použito v sekci 26.
⟨ Vlnkovací funkce *tie* 26 ⟩ Použito v sekci 1.
⟨ Vrať hodnotu podle následující pozice patternu 24 ⟩ Použito v sekci 23.
⟨ Vyprázdni buffer 28 ⟩ Použito v sekci 26.
⟨ Vytvoř ukazatel na nový pattern a **break** 31 ⟩ Použito v sekci 30.
⟨ Zpracování souborů 11 ⟩ Citováno v sekci 6. Použito v sekci 5.
⟨ Zpracování všech souborů příkazové řádky 13 ⟩ Použito v sekci 11.
⟨ Zpracování v režimu filter 12 ⟩ Použito v sekci 11.

VLNA