Zpravodaj Československého sdružení uživatelů TeXu

Jan Šustek

Generování dokumentovaného zdrojového souboru po blocích v TeXu

Zpravodaj Československého sdružení uživatelů TeXu, Vol. 33 (2023), No. 3-4, 66–101

Persistent URL: http://dml.cz/dmlcz/151992

Generování dokumentovaného zdrojového souboru po blocích v TEXu

Jan Šustek

Článek popisuje problematiku psaní programů a dokumentace k nim. Ukazuje autorův balíček gensrc nad OPmacem, který umožňuje psát program i dokumentaci k němu v jediném TEXovém souboru. Jsou ukázány další možnosti a aplikace tohoto balíčku.

Klíčová slova: dokumentace, literární programování, OPmac, gensrc

1 Úvod

Představme si, že píšeme počítačový program, který je delší než několik řádků. Potom nutně musíme k jednotlivým částem programu psát také komentáře. V opačném případě bude program špatně čitelný a špatně pochopitelný, a to jak pro druhou osobu, tak i pro nás samotné, pokud se na program podíváme po nějakém čase.

Zejména u složitějších programů používajících hlubší myšlenky pak musí být dokumentace k programu důkladnější. V následující ukázce je část zdrojového kódu programu[1] a příslušná část souboru s dokumentací.

```
Procedure PridejDoMenu;
  Begin
  inc(Podnabidek[M]);
  if Poz=Nakonec then
    begin
    Nabidka[M,Podnabidek[M]]:=Text;
    if M<>Hlavni then Proc[M,Podnabidek[M]]:=Procedura;
    end
  else
    begin
    ...
    end;
  End;
```

[1] Z důvodu jednoduššího vyjadřování budeme nadále namísto pojmu „zdrojový kód programu“ psát pouze „program“. Přitom vůbec není nutné, aby výsledkem byl nějaký program – může se jednat o libovolný text, který chceme vygenerovat.

 DOI: 10.5300/2023-3-4/66

4.3.18PridejDoMenu

Deklarace: `Procedure PridejDoMenu(M,Poz:Byte;Text:String;Procedura:TProc)`
Procedura přidá do `M`-tého menu na `Poz`-tou pozici položku s názvem `Text`, po jejímž spuštění se spustí procedura `Procedura`. Hlavnímu menu přísluší `M` rovno nule. Pro vkládání na konec daného menu může být argument `Poz` roven nule.
Přímo z hlavního menu nelze spouštět procedury, proto je přiřazení proměnné `Procedura` do proměnné `Proc` prováděno pouze pro podmenu. Výjimku tvoří ta podmenu, která obsahují právě jednu položku. Ta je vybrána a příslušná procedura spuštěna již při zvolení na hlavním menu.

Na problém narazíme v případě, že se program vyvíjí. Potom musíme udržovat dokumentaci programu kompatibilní s programem samotným. Každá změna programu se musí projevit v dokumentaci. To znamená, že změnu musíme evidovat na dvou místech – v souboru s programem a v souboru s dokumentací. Přitom je vhodné a výhodné, když se každá změna eviduje pouze na jednom místě.

V tomto článku jsou nejprve ukázána některá existující řešení uvedeného problému, konkrétně v sekci 2 je to řešení od Donalda Knutha, které použil při psaní samotného TEXu, a v sekci 3 je to řešení pro LATEX. V sekcích 4 až 6 je popsán autorův balíček gensrc pracující nad OPmacem. Implementaci balíčku popsanou v sekci 6 ocení spíše programátoři maker než běžní uživatelé TEXu. Sekce 7 pak ukazuje reálné situace, ve kterých se balíček využil.

Cílem balíčku gensrc nebylo kopírovat nebo napodobovat některá existující řešení. Nebylo ani cílem, aby byl balíček dokonalý a uměl vše. Balíček naopak dodržuje všechny tři hlavní zásady OPmacu [1]:

- V jednoduchosti je síla.
- Makra nejsou univerzální, ale jsou čitelná a srozumitelná.
- Uživatel si makra může snadno předefinovat k obrazu svému.

Původním cílem balíčku bylo plnit pracovní úkoly. A při plnění těchto úkolů se TEX a jeho makra ukázaly jako účinní a užiteční pomocníci.

Pro názornost článek obsahuje větší množství ukázek programů a dokumentace k nim. Aby ukázky byly dostatečně graficky odlišeny od textu článku, jsou vloženy do rámečků.[2] Pokud není uvedeno jinak, jsou v ukázkách použity části autorových vlastních programů.

Článek navazuje na autorovy přednášky na konferenci OSS Conf 2019 v Žilině a na konferenci TUG 2023 v Bonnu.

[2] O sazbě těchto rámečků se čtenář dočte v následujícím článku tohoto Zpravodaje, který začíná na straně 102.

2 DEK a WEB

Začátkem 80. let přišel Donald Knuth [2] s myšlenkou, že by se měly programy psát jiným způsobem, než bylo do té doby obvyklé. Namísto psaní programu jako posloupnosti instrukcí pro počítač navrhl Knuth psát program tak, jako bychom druhému člověkovi vysvětlovali, co chceme, aby počítač prováděl.

Pro psaní programů Knuth vyvinul jazyk WEB. Vstupní soubor jazyka WEB v sobě obsahuje dokumentaci proloženou částmi programu. Tento soubor se

- zpracuje programem weave, čímž se vygeneruje TEXový soubor s dokumentací, který se pak může TEXem vysázet. Dále se vstupní soubor
- zpracuje programem tangle, čímž se vygeneruje program v jazyce Pascal, který se poté může zkompilovat a spustit.

Původní varianta jazyka WEB byla vytvořena pro jazyky TEX a Pascal. Později byly vytvořeny varianty i pro jiné kombinace dokumentačního a programovacího jazyka.

Pro ukázku použití Knuthova literárního programování jsem použil dokumentovaný zdrojový kód TEXu[3] [3], konkrétně kód jeho nejdůležitější procedury main_control. Tato procedura řídí celý běh TEXu a v *TEXbooku naruby* [4] se označuje jako „hlavní procesor“. Procedura se spouští ihned po spuštění TEXu po úvodních formalitách (inicializace interních proměnných, inicializace tabulky primitivů, kontrola konzistence atd.). Po dokončení této procedury už následují pouze formality závěrečné (kontrola neuzavřených skupin při ukončení, závěrečný zápis do souboru \jobname.log, uzavření souborů atd.) a samotné ukončení TEXu.

1030. We shall concentrate first on the inner loop of *main_control*, deferring consideration of the other cases until later.

```
  define big_switch = 60   { go here to branch on the next token of input }
  define main_loop = 70   { go here to typeset a string of consecutive characters }
  define main_loop_wrapup = 80   { go here to finish a character or ligature }
  define main_loop_move = 90   { go here to advance the ligature cursor }
  define main_loop_move_lig = 95   { same, when advancing past a generated ligature }
  define main_loop_lookahead = 100   { go here to bring in another character, if any }
  define main_lig_loop = 110   { go here to check for ligatures or kerning }
  define append_normal_space = 120   { go here to append a normal space between words }
⟨ Declare action procedures for use by main_control 1043 ⟩
⟨ Declare the procedure called handle_right_brace 1068 ⟩
procedure main_control;   { governs TEX's activities }
  label big_switch, reswitch, main_loop, main_loop_wrapup, main_loop_move, main_loop_move + 1,
        main_loop_move + 2, main_loop_move_lig, main_loop_lookahead, main_loop_lookahead + 1,
        main_lig_loop, main_lig_loop + 1, main_lig_loop + 2, append_normal_space, exit;
  var t: integer;   { general-purpose temporary variable }
  begin if every_job ≠ null then begin_token_list(every_job, every_job_text);
big_switch: get_x_token;
reswitch: ⟨ Give diagnostic information, if requested 1031 ⟩;
```

[3]Autor tohoto článku předpokládá, že by se uživateli TEXu mohlo zdát zajímavé vidět vnitřnosti svého oblíbeného programu. :-)

```
case abs(mode) + cur_cmd of
hmode + letter, hmode + other_char, hmode + char_given: goto main_loop;
hmode + char_num: begin scan_char_num; cur_chr ← cur_val; goto main_loop; end;
hmode + no_boundary: begin get_x_token;
  if (cur_cmd = letter) ∨ (cur_cmd = other_char) ∨ (cur_cmd = char_given) ∨ (cur_cmd = char_num)
      then cancel_boundary ← true;
  goto reswitch;
  end;
hmode + spacer: if space_factor = 1000 then goto append_normal_space
  else app_space;
hmode + ex_space, mmode + ex_space: goto append_normal_space;
⟨ Cases of main_control that are not part of the inner loop 1045 ⟩
end; { of the big case statement }
goto big_switch;
main_loop: ⟨ Append character cur_chr and the following characters (if any) to the current hlist in the
    current font; goto reswitch when a non-character has been fetched 1034 ⟩;
append_normal_space: ⟨ Append a normal inter-word space to the current list, then goto big_switch 1041 ⟩;
exit: end;
```

Pro zpřehlednění logické struktury procedury jsou použity ⟨bloky⟩, které jsou definovány v dalších sekcích dokumentace. Procedura `main_control` je definována v sekci 1030 a v jejím těle se používají některé další procedury, které musejí být (dle syntaxe jazyka Pascal) definovány dříve než procedura `main_control`. Proto bylo nutné blok ⟨Declare ... 1043⟩ vložit před definici procedury `main_control`. Na druhou stranu: kvůli logickým návaznostem v dokumentaci mohou být tyto další procedury definovány až v sekci 1043.

Uvedený blok není definován celý najednou, ale postupně v 57 částech, jedna procedura po druhé. Jedna z těchto částí je v sekci 1075 a jak je vidět, i uvnitř bloku se mohou vyskytovat další vnořené bloky.

1075. The *box_end* procedure does the right thing with *cur_box*, if *box_context* represents the context as explained above.

```
⟨ Declare action procedures for use by main_control 1043 ⟩ +≡
procedure box_end(box_context : integer);
  var p: pointer; { ord_noad for new box in math mode }
  begin if box_context < box_flag then
    ⟨ Append box cur_box to the current list, shifted by box_context 1076 ⟩
  else if box_context < ship_out_flag then ⟨ Store cur_box in a box register 1077 ⟩
    else if cur_box ≠ null then
        if box_context > ship_out_flag then ⟨ Append a new leader node that uses cur_box 1078 ⟩
        else ship_out(cur_box);
  end;
```

Tyto vnořené bloky jsou pak definovány v dalších sekcích dokumentace.

Použitý mechanismus má navíc výhodu, že bloky mohou být použity opakovaně na více místech. Například blok ⟨Get ... 404⟩ je použit na 8 místech programu.

1077. ⟨ Store *cur_box* in a box register 1077 ⟩ ≡

```
if box_context < box_flag + 256 then eq_define(box_base − box_flag + box_context, box_ref, cur_box)
else geq_define(box_base − box_flag − 256 + box_context, box_ref, cur_box)
```

This code is used in section 1075.

```
1078.  ⟨ Append a new leader node that uses cur_box 1078 ⟩ ≡
  begin ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
  if ((cur_cmd = hskip) ∧ (abs(mode) ≠ vmode)) ∨ ((cur_cmd = vskip) ∧ (abs(mode) = vmode)) then
    begin append_glue; subtype(tail) ← box_context − (leader_flag − a_leaders);
    leader_ptr(tail) ← cur_box;
    end
  else begin print_err("Leaders␣not␣followed␣by␣proper␣glue");
    help3("You␣should␣say␣`\leaders␣<box␣or␣rule><hskip␣or␣vskip>´.")
    ("I␣found␣the␣<box␣or␣rule>,␣but␣there´s␣no␣suitable")
    ("<hskip␣or␣vskip>,␣so␣I´m␣ignoring␣these␣leaders."); back_error; flush_node_list(cur_box);
    end;
  end
This code is used in section 1075.
```

V ukázce sekce 1030 můžeme vidět také definice maker jazyka WEB. Knuth do jazyka WEB zavedl mechanismus maker kvůli zpřehlednění a zjednodušení dokumentace, ale také kvůli odstínění některých nedokonalostí tehdejších kompilátorů jazyka Pascal. Makra jazyka WEB mohou mít nula, nebo jeden parametr.

Soubor, z něhož byla programem weave vygenerována výše uvedená dokumentace, má název tex.web. V následující ukázce je zdrojový kód sekce 1075. Jsou zvýrazněny příkazy pro práci s bloky a sekcemi.

```
@ The |box_end| procedure does the right thing with |cur_box|, if
|box_context| represents the context as explained above.

@<Declare act...@>=
procedure box_end(@!box_context:integer);
var p:pointer; {|ord_noad| for new box in math mode}
begin if box_context<box_flag then @<Append box |cur_box| to the current list,
    shifted by |box_context|@>
else if box_context<ship_out_flag then @<Store \(c)|cur_box| in a box register@>
else if cur_box<>null then
  if box_context>ship_out_flag then @<Append a new leader node that
      uses |cur_box|@>
  else ship_out(cur_box);
end;
```

Zpracováním souboru tex.web programem tangle se vygeneruje program tex.pas. V ukázce je zvýrazněna ta část programu, která odpovídá sekci 1075.

```
{:1070}{1075:}procedure boxend(boxcontext:integer);var p:halfword;
begin if boxcontext<1073741824 then{1076:}begin if curbox<>0 then begin
mem[curbox+4].int:=boxcontext;
if abs(curlist.modefield)=1 then begin appendtovlist(curbox);
if adjusttail<>0 then begin if 29995<>adjusttail then begin mem[curlist.
tailfield].hh.rh:=mem[29995].hh.rh;curlist.tailfield:=adjusttail;end;
adjusttail:=0;end;if curlist.modefield>0 then buildpage;
end else begin if abs(curlist.modefield)=102 then curlist.auxfield.hh.lh
:=1000 else begin p:=newnoad;mem[p+1].hh.rh:=2;mem[p+1].hh.lh:=curbox;
```

```
curbox:=p;end;mem[curlist.tailfield].hh.rh:=curbox;
curlist.tailfield:=curbox;end;end;
end{:1076}else if boxcontext<1073742336 then{1077:}if boxcontext<
1073742080 then eqdefine(-1073738146+boxcontext,119,curbox)else
geqdefine(-1073738402+boxcontext,119,curbox){:1077}else if curbox<>0
then if boxcontext>1073742336 then{1078:}begin{404:}repeat getxtoken;
until(curcmd<>10)and(curcmd<>0){:404};
if((curcmd=26)and(abs(curlist.modefield)<>1))or((curcmd=27)and(abs(
curlist.modefield)=1))then begin appendglue;
mem[curlist.tailfield].hh.b1:=boxcontext-(1073742237);
mem[curlist.tailfield+1].hh.rh:=curbox;
end else begin begin if interaction=3 then;printnl(262);print(1065);end;
begin helpptr:=3;helpline[2]:=1066;helpline[1]:=1067;helpline[0]:=1068;
end;backerror;flushnodelist(curbox);end;end{:1078}else shipout(curbox);
end;{:1075}{1079:}procedure beginbox(boxcontext:integer);label 10,30;
```

3 Program docstrip

Pro LaTeX je k dispozici program docstrip [5], který umožňuje do jednoho zdrojového souboru psát program i dokumentaci k němu. Podobně jako v případě jazyka WEB i zde můžeme jedním zpracováním zdrojového souboru vygenerovat program a druhým zpracováním vysázet dokumentaci. Balíček má i další dovednosti.

3.1 Základní použití

Program a dokumentace se zapisují do textového souboru, který mívá příponu dtx. V tomto souboru jsou řádky programu vloženy do prostředí macrocode, které musí začínat a končit na řádcích, které na začátku mají znak procenta a přesně čtyři mezery. Řádky mimo toto prostředí tvoří dokumentaci a musí začínat znakem procenta. Pokud si odmyslíme tato procenta, píše se dokumentace stejně jako libovolný jiný LaTeXový dokument.[4]

```
% Makro |\radeknabody| zpracuje řádek a jednotlivé položky uloží
% do příslušných maker a čítačů.
%    \begin{macrocode}
\def\radeknabody#1;#2;#3;#4;#5;{%
  \def\no{#1}\def\bodyA{#2}\def\bodyB{#3}\def\bodyC{#4}\def\bodyD{#5}%
  \nacislo#2 \bodycA \nacislo#3 \bodycB
  \nacislo#4 \bodycC \nacislo#5 \bodycD}
%    \end{macrocode}
```

[4] V ukázkách je použita část programu používaného při Mezinárodní matematické soutěži Vojtěcha Jarníka. Soutěž je pořádána Ostravskou univerzitou a bližší informace k ní lze nalézt na stránkách https://vjimc.osu.cz/.

```
% Pokud je některý soutěžící diskvalifikován, do tabulky s~počtem bodů
% se mu zapíše z~prvního příkladu 666~bodů. Jeho součet bodů se nastaví
% na nulu a v~souboru \soub{results.srt?} bude uveden až na konci.
%    \begin{macrocode}
\def\nvzpracujradek{%
  \ea\radeknabody\radek
...
%    \end{macrocode}
```

Pro vygenerování souboru pdf s dokumentací vytvoříme nový LaTeXový soubor. V něm mimo jiných použijeme balíček doc. Soubor dtx pak načteme makrem `\DocInput`. LaTeXový soubor zpracujeme běžným způsobem LaTeXem.

`\nacislo` Makro `\nacislo` nastaví čítač `#2` na hodnotu `#1` nebo na hodnotu `#1` − 100, pokud `#1` ∈ [100; 200). Pokud `#1` = -, nastaví `#2` na nulu.

```
\def\nacislo#1 #2{\ifx-#1#20\else
    #2#1 \ifnum#2>99 \ifnum#2<200 \advance#2-100\fi\fi\fi}
```

`\radeknabody` Makro `\radeknabody` zpracuje řádek a jednotlivé položky uloží do příslušných maker a čítačů.

```
\def\radeknabody#1;#2;#3;#4;#5;{%
  \def\no{#1}\def\bodyA{#2}\def\bodyB{#3}\def\bodyC{#4}\def\bodyD{#5}%
  \nacislo#2 \bodycA \nacislo#3 \bodycB
  \nacislo#4 \bodycC \nacislo#5 \bodycD}
```

`\nvzpracujradek` Pokud je některý soutěžící diskvalifikován, do tabulky s počtem bodů se mu zapíše z prvního příkladu 666 bodů. Jeho součet bodů se nastaví na nulu a v souboru results.srt? bude uveden až na konci.

```
\def\nvzpracujradek{%
  \ea\radeknabody\radek
  \sumac\bodycA \advance\sumac\bodycB
  \advance\sumac\bodycC \advance\sumac\bodycD
  \ifnum\bodycA=\bodyDQ
    \counta99 \sumac0
```

Pro vygenerování programu vytvoříme další LaTeXový soubor. V něm mimo jiné použijeme balíček docstrip. Dále použijeme makro `\generateFile`, jímž načteme vstupní soubor dtx a vygenerujeme program s daným názvem. LaTeXový soubor stačí jednou zpracovat LaTeXem.

```
\def\nacislo#1 #2{\ifx-#1#20\else
    #2#1 \ifnum#2>99 \ifnum#2<200 \advance#2-100\fi\fi\fi}
\def\radeknabody#1;#2;#3;#4;#5;{%
  \def\no{#1}\def\bodyA{#2}\def\bodyB{#3}\def\bodyC{#4}\def\bodyD{#5}%
  \nacislo#2 \bodycA \nacislo#3 \bodycB
  \nacislo#4 \bodycC \nacislo#5 \bodycD}
```

```
\def\nvzpracujradek{%
  \ea\radeknabody\radek
  \sumac\bodycA \advance\sumac\bodycB
  \advance\sumac\bodycC \advance\sumac\bodycD
  \ifnum\bodycA=\bodyDQ
    \counta99 \sumac0
```

Detaily jednotlivých LaTeXových souborů a maker se čtenář dozví v dokumentaci [6; 7].

3.2 Další funkce

V souboru dtx mimo prostředí `macrocode` lze využít další funkce balíčku docstrip. Prostředí `macro` označuje, že část programu a dokumentace se týká definice konkrétního příkazu.[5] Makro `\changes` označuje, že na daném místě došlo v dané verzi programu k určité změně.

```
% \begin{macro}{\nvzpracujradek}
% Pokud je některý soutěžící diskvalifikován, do tabulky s~počtem bodů
% se mu zapíše z~prvního příkladu 666~bodů. Jeho součet bodů se nastaví
% na nulu a v~souboru \soub{results.srt?} bude uveden až na konci.
% \changes{100908}{100908}{Pročištění}
% \changes{140314}{140314}{Logika diskvalifikací}
% \changes{140314}{140314}{Přidáno top competitors}
%    \begin{macrocode}
\def\nvzpracujradek{%
  \ea\radeknabody\radek
  \sumac\bodycA \advance\sumac\bodycB
  \advance\sumac\bodycC \advance\sumac\bodycD
  \ifnum\bodycA=\bodyDQ
...
%    \end{macrocode}
% \end{macro}
```

Při použití výše uvedených maker lze v dokumentaci makrem `\PrintChanges` vygenerovat seznam všech změn v jednotlivých verzích programu. Jestliže ke změně došlo uvnitř prostředí `macro`, přiřadí se změna přímo ke konkrétnímu příkazu. Pro zaznamenávání změn je třeba v preambuli souboru použít makro `\RecordChanges`.

[5] Balíček docstrip je uzpůsoben ke generování LaTeXových balíčků a zřejmě proto jsou pro účely tohoto balíčku části programu označeny jako „macro“. Zde, opět z důvodu srozumitelnosti, budeme používat pojem „příkaz“.

Podobně lze v dokumentaci makrem `\PrintIndex` vytvořit rejstřík všech příkazů[6] použitých v programu. V rejstříku je u konkrétního příkazu možné zobrazit číslo stránky nebo řádku,[7] na nichž je příkaz použit. Podtržení čísla znamená, že na daném místě je příkaz definován. Pro umožnění tvorby rejstříku je třeba v preambuli souboru použít makro `\PageIndex` nebo `\CodelineIndex`.

[6] Jako příkazy se v programu vyhledají všechny názvy začínající znakem \, případně jiným nastaveným znakem.

[7] Čísla řádků v rejstříku odpovídají číslům řádků v dokumentaci. Tato čísla se však mohou lišit od čísel řádků ve vygenerovaném programu.

4 Jednoduché řešení v OPmacu

Autor článku namísto LaTeXu raději pracuje v Plainu s balíkem OPmac [1]. Cílem této sekce je vytvořit nad OPmacem soubor gensrc.tex[8] a v něm nadefinovat makra

- `\SRCFILENAME` udávající název vygenerovaného programu,
- `\BEGSRC` a `\ENDSRC` ohraničující řádky programu.

S balíčkem gensrc.tex se pak v jediném průchodu TeXem vytvoří jak program, tak dokumentace. Jak bude tento program a dokumentace vypadat, to si zajisté čtenář snadno domyslí.[9]

```
\input opmac
\input gensrc
\SRCFILENAME style.tex
\activettchar"
Format of the page is~A4 with~2cm~margins.
The basic font size is set to 12\,pt.
\BEGSRC
\margins/1 a4 (2,2,2,2)cm
\typosize[12/14]
\ENDSRC
Macro "\safedef" defines a control sequence which is not yet defined.
If it is already defined then its new definition is ignored.
\BEGSRC
\def\safedef#1{\ifdefined#1\begingroup\afterassignment\endgroup\fi\def#1}
\ENDSRC
Sections are defined in the same way as in~\OPmac.
\BEGSRC
\safedef\section{\sec}
\ENDSRC
\bye
```

4.1 Základní použití

Součástí dokumentace jsou i části programu. Proto není divu, že základem souboru gensrc.tex bude makro pro sazbu verbatim textu. K tomu je v OPmacu následujícím způsobem definováno makro `\begtt`. Detailní popis jednotlivých v něm použitých triků čtenář najde v *TeXbooku naruby* [4, sekce 1.3].

```
\def\begtt{\par\ttskip\bgroup \wipeepar
  \setverb\adef{ }{ }
```

[8] Soubor gensrc.tex tvoří řádky 17 až 45 na následujících stranách, přičemž je třeba vymazat šedý text. Navíc je možné při použití řádků 39 až 45 dále vymazat řádky 29 až 32.

[9] Zvídavému TeXistovi doporučuji si prozkoumat, jak funguje definice makra `\safedef` uvedená v ukázce.

```
  \ifx\savedttchar\undefined \else \catcode\savedttchar=12 \fi
  \parindent=\ttindent \vskip\parskip \parskip=0pt
  \tthook\relax
  \ifnum\ttline<0 \else
    \tenrm \thefontscale[700]\let\sevenrm=\thefont
    \everypar={\global\advance\ttline by1
      \llap{\sevenrm\the\ttline\kern.9em}}\fi
  \def\par##1{\endgraf\ifx##1\egroup\else
    \penalty\ttpenalty\leavevmode\fi ##1}
  \obeylines \startverb}
\def\setverb{\frenchspacing\def\do##1{\catcode`##1=12}%
  \dospecials \catcode`\*=12 }
{\catcode`\|=0 \catcode`\\=12
|gdef|startverb#1\endtt{|tt#1|egroup|par|ttskip|testparA}}
```

Jednoduchými úpravami výše uvedených maker lze vytvořit makro `\BEGSRC`. Konkrétně se jedná o následující kosmetické úpravy, přičemž pouze první dva body jsou opravdu potřebné.

- Řídicí sekvence `\begtt`, `\setverb`, `\tthook`, `\ttline`, `\startverb`, `\endtt` jsou přejmenovány.
- Registr `\everypar` se nastaví nezávisle na podmínce `\SRCline<0`. Toto nastavení je lokální uvnitř prostředí `\BEGSRC...\ENDSRC`.[10]
- Makro `\wipeepar` globálně promazává registr `\everypar`. Toto není potřeba, a proto zde makro `\wipeepar` není nutné.
- Je použito `\csname` pro případ, že by háček `\SRChook` nebyl definován.
- Namísto makra `\leavevmode` je použit primitiv `\quitvmode`, který v daném místě neexpanduje registr `\everypar`. Navíc příkaz `\expandafter` zajistí, že TEX na token `\fi` narazí ještě ve vertikálním módu.

Další úpravou je přidání jedné definice.

- Makro `\SRCFILENAME` nemá analogii v případě makra `\begtt`. Makro otevře pro zápis soubor `\SRCfile`. V případě, že je makro použito podruhé (tj. z jednoho zdrojového souboru začínáme generovat další program), je aktuální soubor nejprve uzavřen. Tento krok v případě prvního použití makra nenahlásí chybu.

A samozřejmě je nutná i jedna funkční úprava.

- Na začátku každého řádku programu je zavoláno makro `\SRCgetline`. Makro načte argument až po znak konce řádku[11] (tj. načte jeden řádek), tento argument vysází a zapíše jej do souboru.

Více úprav není třeba. Všechny výše uvedené úpravy jsou barevně vyznačeny.

[10] Pro přehlednost budeme dále používat text „prostředí `\BEGSRC`“.

[11] V makru `\BEGSRC` je použito makro `\obeylines`, které nastaví znak konce řádku jako aktivní. Proto je nutný onen trik s vlnovkou na řádcích 33 a 34.

```
\def\BEGSRC{\par\ttskip\bgroup \wipeepar
  \setSRC\adef{ }{ }
  \ifx\savedttchar\undefined \else \catcode\savedttchar=12 \fi
  \parindent=\ttindent \vskip\parskip \parskip=0pt
  \csname SRChook\endcsname\relax
  \ifnum\ttline<0 \else
    \tenrm \thefontscale[700]\let\sevenrm=\thefont
    \everypar={\global\advance\SRCline by1
      \llap{\sevenrm\the\SRCline\kern.9em}\SRCgetline}\fi
  \def\par##1{\endgraf\ifx##1\egroup\else
    \penalty\ttpenalty\expandafter\quitvmode\fi ##1}
  \obeylines \startSRC}
\def\setSRC{\frenchspacing\def\do##1{\catcode`##1=12}%
  \dospecials \catcode`\*=12 }
{\catcode`\|=0 \catcode`\\=12
|gdef|startSRC#1\ENDSRC{|tt#1|egroup|par|ttskip|testparA}}
\begingroup\lccode`\~13
  \lowercase{\endgroup\def\SRCgetline#1~}%
    {#1\immediate\write\SRCfile{#1}\par}
\def\SRCFILENAME{\immediate\closeout\SRCfile
  \immediate\openout\SRCfile=}
\newcount\SRCline \newwrite\SRCfile
```

4.2 Snadná vylepšení

Změnou definice makra `\startSRC` umožníme uživateli použít háček `\endSRChook`, který se expanduje v místě `\ENDSRC`. Pomocí `\csname` opět umožníme bezchybnou expanzi v případě, že háček není definován.

```
{\catcode`\|=0 \catcode`\\=12
  |gdef|startSRC#1\ENDSRC{|tt#1|egroup|par|ttskip
    |csname endSRChook|endcsname|testparA}}
```

Některé textové editory odsazují řádky pomocí tabulátoru, jiné pomocí mezer. To například v jazyce Python může činit problém. Definicí aktivního tabulátoru expandujícího se na posloupnost několika mezer tento problém vyřešíme. Při definování musíme opatrně měnit kategorie znaků tabulátoru a mezery.

```
{\catcode9=12
  \gdef\setSRC{\def\do##1{\catcode`##1=12}\dospecials
    \catcode`\*=12 \adef{^^I}{\SRCtab}}}
{\catcode32=13 \gdef\SRCtab{    }}
```

5 Generování po blocích

V této sekci si naplánujeme vylepšení souboru gensrc.tex po vzoru sekce 2. Navíc součástí tohoto vylepšení bude práce s lokálně aditivním odsazením řádků, což velmi pomůže při generování programů například v jazyce Python. Oproti sekci 2 se omezíme na situaci, kdy na řádku s vkládaným blokem není kromě bloku a odsazení žádný další text.

V této sekci pouze popíšeme, jaká makra budeme používat a co tato makra budou dělat. Implementace těchto maker bude následovat v sekci 6. Doporučuji čtenáři, aby si po přečtení sekce 6 znovu přečetl podsekce 5.3 a 5.4.

Nový soubor gensrc.tex tvoří řádky 46 až 207 na stranách 83 až 92. Soubor je ke stažení online [8].

5.1 Syntaxe

Makro `\BEGSRC` je možné volat následujícími způsoby.[12]

1. `\BEGSRC<InterniNazev>{Zobrazený název}`
2. `\BEGSRC<InterniNazev>`
3. `\BEGSRC`

Přesná syntaxe je patrná z následující ukázky. Každý blok má svůj (jednoduchý) interní název a dále název, který se zobrazí v dokumentaci. V této ukázce je navíc použit háček `\SRChook`, který způsobí, že řádky programu budou v dokumentaci vysázeny červeně.

```
\input gensrc
\def\SRChook{\longlocalcolor\Red}
\SRCFILENAME program.txt
Here we define a block which will be inserted to another place.
\BEGSRC<InternalLabel>{Displayed label}
second line
  |<InnerBlock>
\ENDSRC
A block can be defined by parts.
\BEGSRC<InternalLabel>
fourth line
\ENDSRC
And here we insert the block.
\BEGSRC
first line
    |<InternalLabel>
\ENDSRC
```

[12] V dalším textu se často budeme na uvedené číslování odkazovat.

```
Finally we define the inner block.
\BEGSRC<InnerBlock>{Inner block}
third line
\ENDSRC
\bye
```

Pro správné vysázení dokumentace jsou nutné dva průchody TEXem, protože v prvním průchodu ještě nejsou známy zobrazené názvy bloků.

Here we define a block which will be inserted to another place.

⟨*Displayed label*⟩ ≡
```
1 second line
2   ⟨Inner block⟩
```

A block can be defined by parts.

⟨*Displayed label*⟩ +≡
```
3 fourth line
```

And here we insert the block.

```
4 first line
5     ⟨Displayed label⟩
```

Finally we define the inner block.

⟨*Inner block*⟩ ≡
```
6 third line
```

Z podobného důvodu jsou nutné dva průchody TEXem pro správné vygenerování programu. Tyto dva průchody však již máme za sebou, když jsme vysázeli dokumentaci. Výsledek je dle očekávání následující:

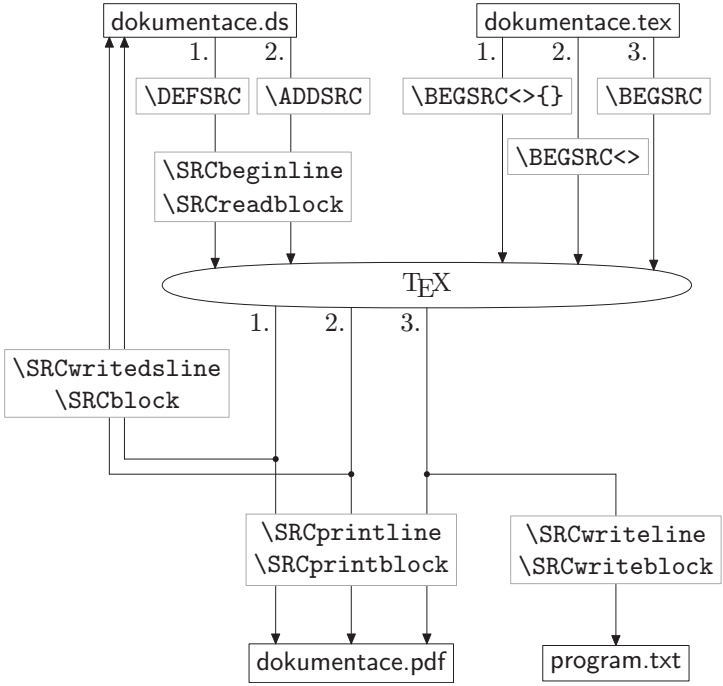
```
first line
    second line
      third line
    fourth line
```

5.2 Přenos informací

Dále v textu budeme předpokládat, že máme vstupní soubor dokumentace.tex, ze kterého chceme vytvořit dokumentaci v souboru dokumentace.pdf a vygenerovat program v souboru program.txt. Makra budou využívat soubor `\jobname`.ds,[13] v našem případě tedy soubor dokumentace.ds.[14] Graf na straně 80 ukazuje, jak se u jednotlivých způsobů volání makra `\BEGSRC` přenáší informace mezi soubory. V rámečcích u šipek jsou uvedena makra, která příslušnou informaci přenáší.

[13] Písmena „ds“ vznikla ze slov „dočasný soubor“.

[14] Pro zpřehlednění textu budeme dále jednotlivé soubory označovat pouze jako „tex“, „pdf“, „ds“, „txt“, přičemž budeme vynechávat i slovo „soubor“.



5.3 První průchod

V ukázce v podsekci 5.1 je čtyřikrát použito prostředí `\BEGSRC`. Vnitřek jednotlivých prostředí se ukládá do makra `\SRCcontent`. Toto makro má v místech `\ENDSRC` postupně následující obsah:

```
\SRConeline{second line}
\SRCblock{  }{InnerBlock}
```

```
\SRConeline{fourth line}
```

```
\SRConeline{first line}
\SRCblock{    }{InternalLabel}
```

```
\SRConeline{third line}
```

Makro `\doSRC` definuje příslušná makra a do `ds` postupně zapíše následující text:

```
\DEFSRC {InternalLabel}{Displayed label}
\SRCbeginline :second line\SRCendline
\SRCreadblock {  }{InnerBlock}
\ENDDEFSRC
\ADDSRC {InternalLabel}
\SRCbeginline :fourth line\SRCendline
\ENDDEFSRC
\DEFSRC {InnerBlock}{Inner block}
\SRCbeginline :third line\SRCendline
\ENDDEFSRC
```

Makra `\csname SRCtit:\SRCid\endcsname` obsahují zobrazené názvy bloků a ty v prvním průchodu ještě nejsou známy, proto tato makra expand procesor interpretuje jako `\relax` a v dokumentaci se namísto názvů bloků zobrazí prázdné závorky ⟨⟩:

Here we define a block which will be inserted to another place.

⟨*Displayed label*⟩ ≡
```
1 second line
2   ⟨⟩
```

A block can be defined by parts.

⟨⟩ +≡
```
3 fourth line
```

And here we insert the block.

```
4 first line
5     ⟨⟩
```

Finally we define the inner block.

⟨*Inner block*⟩ ≡
```
6 third line
```

Podobně makra `\csname SRCcon:\SRCid\endcsname` obsahují text jednotlivých bloků, který ale také v prvním průchodu není znám. Onen `\relax` způsobí, že se do `txt` příslušné řádky nezapíšou a že v `txt` bude pouze následující obsah:

```
first line
```

5.4 Druhý průchod

Na začátku druhého průchodu se načte `ds`. Makra z podsekce 6.7 způsobí, že se interně provedou následující definice:

```
\expandafter\def\csname SRCtit:InternalLabel\endcsname{%
  Displayed label}
```

```
\expandafter\def\csname SRCcon:InternalLabel\endcsname{%
  \SRConeline{second line}%
  \SRCblock{  }{InnerBlock}%
  \SRConeline{fourth line}}
\expandafter\def\csname SRCtit:InnerBlock\endcsname{%
  Inner block}
\expandafter\def\csname SRCcon:InnerBlock\endcsname{%
  \SRConeline{third line}}
```

Druhý průchod poté pokračuje stejně jako první průchod, až na dva rozdíly. Prvním rozdílem je, že po těchto definicích již TEX zná vše potřebné a dokumentaci vysází správně, jak je na straně 79. Druhým rozdílem je, že se správně vygeneruje i txt. Podívejme se, jak probíhá ono generování.

Pouze ve třetím případě je makro `\BEGSRC` voláno třetím způsobem, tj. způsobem, kdy se zapisuje do txt. Makro `\SRCcontent` se postupně expanduje takto:

```
\SRCcontent
```

```
\SRConeline{first line}
\SRCblock{    }{InternalLabel}
```

```
\SRCwriteline{first line}
\SRCwriteblock{    }{InternalLabel}
```

```
\csname SRCwritehook\endcsname
\immediate\write\SRCfile{\SRCodsazeni first line}
\begingroup
\addto\SRCodsazeni{    }
\csname SRCcon:InternalLabel\endcsname
\endgroup
```

```
\csname SRCwritehook\endcsname
\immediate\write\SRCfile{\SRCodsazeni first line}
\begingroup
\addto\SRCodsazeni{    }
\SRConeline{second line}
\SRCblock{  }{InnerBlock}
\SRConeline{fourth line}
\endgroup
```

Po několika dalších expanzích se dostaneme na následující příkazy:

```
\csname SRCwritehook\endcsname
\immediate\write\SRCfile{\SRCodsazeni first line}
\begingroup
\addto\SRCodsazeni{    }
\csname SRCwritehook\endcsname
\immediate\write\SRCfile{\SRCodsazeni second line}
\begingroup
\addto\SRCodsazeni{  }
\csname SRCwritehook\endcsname
\immediate\write\SRCfile{\SRCodsazeni third line}
\endgroup
\csname SRCwritehook\endcsname
\immediate\write\SRCfile{\SRCodsazeni fourth line}
\endgroup
```

Tyto příkazy způsobí, že se do txt zapíše správný text, který je uveden na straně 79.

6 Implementace

6.1 Začátek

Protože se v souboru gensrc.tex opakovaně používají makra OPmacu, je OPmac načten přímo v tomto souboru.[15]

```
\input opmac
```

Pro označení bloků kódu v pdf je použito makro `\SRCangle` a uvnitř něj je použit přepínač fontu `\sl`. Přitom je nutné font `\tensl` registrovat, aby se správně škálovala jeho velikost.

```
\regfont\tensl
\def\SRCangle#1{{$\langle$\sl#1\/$\rangle$}}
```

Stejně jako dříve se název generovaného souboru zadá makrem `\SRCFILENAME`. Interně bude tento soubor reprezentován řídicí sekvencí `\SRCfile`. Při opakovaném použití `\SRCFILENAME` se soubor nejdříve zavře a poté se začne začne generovat nový.

[15] Na konferenci TUG 2023 zazněl námět, aby makra fungovala také v LaTeXu. Jak je vidět v následujícím článku tohoto Zpravodaje na straně 106, je možné vytvořit balíček, který by fungoval jak v Plainu, tak v LaTeXu, pokud byl balíček původně naprogramován v Plainu. V tom případě by se v LaTeXové větvi samozřejmě OPmac nenačítal, musela by se ale nadefinovat všechna používaná OPmacová makra a na všech místech, kde LaTeX používá jiné mechanismy (například práce s fonty), by se musela použít makra, jejichž expanze by závisela na použitém formátu TeXu.

```
\newwrite\SRCfile
\def\SRCFILENAME{\immediate\closeout\SRCfile
  \immediate\openout\SRCfile=}
```

Prostředí `\BEGSRC` interně používá přepínače `\ifsaveSRC`, `\ifaddingSRC` a `\ifprintSRC`, aby se v různých situacích chovalo správně.

Pokud je nastaveno `\saveSRCtrue`, znamená to, že se definuje blok a ten se zapíše do `ds`. Pokud je nastaveno `\saveSRCfalse`, pak se zapisuje přímo do `txt`.

Pokud je nastaveno `\addingSRCtrue`, znamená to, že se přidávají řádky k již definovanému bloku. Pokud je nastaveno `\addingSRCfalse`, je příslušný blok vynulován a řádky jsou uloženy přidáním k takto vynulovanému bloku.

Pokud je nastaveno `\printSRCtrue`, bude se aktuální prostředí `\BEGSRC` sázet do `pdf`. V případě, že je nastaveno `\printSRCfalse`, bude se sázet do boxu `\nonprintbox`, který se nevytiskne. Implicitní hodnota je `\printSRCtrue`.

```
\newif\ifsaveSRC
\newif\ifaddingSRC
\newif\ifprintSRC \printSRCtrue
\newbox\nonprintbox
```

Makro `\SRCensurehmode\token` zajistí, že se ve vertikálním módu bude `\token` chovat tak, že bezpečně přejde do módu horizontálního a tam zůstane neexpandovaný.

```
\def\SRCensurehmode#1{\def#1{\quitvmode\noexpand#1}}
```

Uvnitř prostředí `\BEGSRC` má znak `|` kategorii 0. Pro jeho sazbu a export se použije řídicí sekvence `||`.

```
\def\|{|}
```

Může se stát, že prostředí `\BEGSRC` zavoláme uvnitř prostředí `\begitems`, v němž je hvězdička aktivním znakem. Proto ji musíme na všech potřebných místech deaktivovat. To uděláme nejjednodušeji tak, že ji zařadíme mezi znaky, které OPmac interně považuje za speciální.

```
\addto\dospecials{\do\*}
```

6.2 Makro `\BEGSRC`

Makro `\BEGSRC` nejdříve přes `\futurelet` zjistí, jak je voláno, a nastaví příslušné přepínače. Pak se uloží (vizte podsekci 6.4) jednotlivé řádky prostředí `\BEGSRC` dovnitř maker `\SRConeline` a `\SRCblock` a všechny řádky se takto postupně vloží do makra `\SRCcontent`. V místě `\ENDSRC` (vizte podsekci 6.5) se pak v závislosti na jednotlivých přepínačích nadefinují makra `\SRConeline` a `\SRCblock` a expanduje se makro `\SRCcontent`.

Již v makru `\BEGSRC` a jeho větvích jsou opatrně měněny kategorie znaků, aby se případný první token prostředí načetl do `\futurelet` s kategorií platnou uvnitř prostředí, ale aby uvnitř argumentů maker `\BEGSRCb` a `\BEGSRCd` byly kategorie ještě původní.

```
\def\BEGSRC{\ifprintSRC\par\ttskip\fi
  \bgroup
  \catcode`\|=0 \catcode`\\=12
  \futurelet\BEGSRCt\BEGSRCa}
```

Makro `\BEGSRCa` zjistí, kterým způsobem je makro `\BEGSRC` voláno, a podle toho buď zavolá další makro `\BEGSRCb` (první dva způsoby), nebo zavolá přímo makro `\BEGSRCz` (třetí způsob). Zároveň nastaví přepínač `\ifsaveSRC`.

```
\def\BEGSRCa{\ifx<\BEGSRCt
    \saveSRCtrue
    \catcode`\|=12 \catcode`\\=0
    \expandafter\BEGSRCb
  \else
    \saveSRCfalse
    \expandafter\BEGSRCz
  \fi}
```

Makro `\BEGSRCb` uloží `InterniNazev` bloku pro další použití do makra `\SRCid` a zavolá rozhodovací makro `\BEGSRCc`.

```
\def\BEGSRCb<#1>{\def\SRCid{#1}%
  \catcode`\|=0 \catcode`\\=12
  \futurelet\BEGSRCt\BEGSRCc}
```

Makro `\BEGSRCc` zjistí, kterým způsobem je makro `\BEGSRC` voláno, a podle toho buď zavolá další makro `\BEGSRCd` (první způsob), nebo zavolá přímo makro `\BEGSRCz` (druhý způsob). Zároveň nastaví přepínač `\ifaddingSRC`.

```
\def\BEGSRCc{\ifx\bgroup\BEGSRCt
    \addingSRCfalse
    \catcode`\|=12 \catcode`\\=0
    \expandafter\BEGSRCd
  \else
    \addingSRCtrue
    \expandafter\BEGSRCz
  \fi}
```

Makro `\BEGSRCd` uloží `Zobrazený název` bloku pro další použití do makra `\csname SRCtit:\SRCid\endcsname` a zavolá makro `\BEGSRCz`.

```
82 \def\BEGSRCd#1{\sdef{SRCtit:\SRCid}{#1}\BEGSRCz}
```

Makro `\BEGSRCz` je analogií makra `\BEGSRC` z řádků 17 až 28, přičemž již respektuje nastavené přepínače. Na začátku prostředí `\BEGSRC` je možné použít háček `\SRChook`, na konci prostředí háček `\endSRChook`. Na začátku každého řádku je možné použít háček `\SRClinehook`. Každý řádek je načten a zpracován makrem `\SRCgetline`. Makro `\<` mimo `\SRCgetline` nedělá nic a v horizontálním módu na něj expand procesor nenarazí. Je však třeba makro `\<` nadefinovat, ať ve vertikálním módu bezpečně přejde do módu horizontálního a v něm ať zůstane neexpandované.

```
83 \def\BEGSRCz{\ifprintSRC\else\setbox\nonprintbox\vbox\bgroup\fi
84   \ifsaveSRC
85     \noindent\SRCangle{\csname SRCtit:\SRCid\endcsname}%
86       ${}\ifaddingSRC\mathrel{{+}{\equiv}}\else\equiv\fi$%
87     \par\nobreak
88   \fi
89   \def\SRCcontent{}%
90   \setSRC
91   \SRCensurehmode\<%
92   \SRCensurehmode\label
93   \parindent\ttindent
94   \csname SRChook\endcsname
95   \tenrm\thefontscale[700]\let\sevenrm\thefont
96   \everypar{\SRCgetline}%
97   \def\par##1{\endgraf\ifx##1\egroup\else\penalty\ttpenalty
98     \fi##1}%
99   \obeylines\startSRC}
```

6.3 Makro `\setSRC`

Se sazbou znaků uvnitř verbatim prostředí, které je vybaveno nějakou přidanou hodnotou, nutně souvisejí následující problémy.

- Co nejvíce znaků musí být přímo tisknutelných, tj. musejí mít kategorii 11 nebo 12. To zajistí makro `\dospecials` a definice makra `\do`.
- Kdyby měla mezera kategorii 10 (jako obvykle), pak by více mezer za sebou splynulo do jedné. Výhodné je nastavit mezeru jako aktivní znak a ten pak expandovat na normální mezeru.
- Některý znak musí mít kategorii 0 nebo 13, aby uvnitř prostředí `\BEGSRC` bylo možné zavolat makro, které zařídí onu přidanou hodnotu. V našem případě to je znak `|`, který bude uvozovat řídicí sekvence.
- Znak uvedený v předchozím bodě musí být možné také vytisknout. To jsme již zajistili na řádku 57.

Uvnitř prostředí `\BEGSRC` se navíc tabulátor expanduje na posloupnost mezer. A jestliže uživatel makrem `\activettchar` nastavil znak pro přechod do odstavcového verbatim módu, nastaví se tomuto znaku kategorie 12.

```
{\catcode9=12
  \gdef\setSRC{\def\do##1{\catcode`##1=12}\dospecials
    \ifx\savedttchar\undefined\else\catcode\savedttchar=12 \fi
    \adef{ }{\ }%
    \catcode`\|=0 \adef{^^I}{\SRCtab}}}
{\catcode32=13 \gdef\SRCtab{    }}
```

6.4 Makro `\SRCgetline`

Makro `\SRCscan` zjistí, zda argument makra `\SRCgetline` obsahuje `\<popisek>` a jaké je před ním odsazení.[16] Oboje uloží do příslušných maker. V argumentu mají mezery kategorii 13 a vlnovky kategorii 12. V níže použitém testu mají vlnovky kategorii 13, proto nemůže dojít k chybnému testu. Jestliže je odsazení nulové, platí `#1=\noexpand` dle definice makra `\SRCensurehmode`. V tom případě se `\SRCods` opraví na nic.

```
\def\SRCnoex{\noexpand}
\def\SRCscan#1\<#2>#3\SRCend{%
  \ifx~#3~%
    \let\SRCnaz\relax
  \else
    \def\SRCods{#1}%
    \ifx\SRCods\SRCnoex
      \def\SRCods{}%
    \fi
    \def\SRCnaz{#2}%
  \fi}
```

Makro `\SRCgetline` načte a vytiskne jeden řádek.

- Pokud řádek obsahuje `\<popisek>`, uloží se `popisek` do makra `\SRCnaz` a odsazení před ním do makra `\SRCods`. Případný další text za `\<...>` se ignoruje. Do makra `\SRCcontent` se přidá `\SRCblock{odsazení}{popisek}`.
- V opačném případě se do makra `\SRCcontent` přidá `\SRConeline{#1}`.

Oddělovačem makra `\SRCgetline` je token $\boxed{\texttt{\^{}\^{}M}}_{13}$, který se vytvoří díky makru `\obeylines` použitému na řádku 99.

```
\begingroup\lccode`\~13
    \lowercase{\endgroup\def\SRCgetline#1~}{%
```

[16] Jako odsazení se chápe text (zpravidla posloupnost mezer) před prvním výskytem `\<...>`.

```
\SRCscan#1\<>\SRCend
\ifx\SRCnaz\relax
  \addto\SRCcontent{\SRConeline{#1}}%
  \SRCprintline{#1}%
\else
  \expandafter\expandafter\expandafter\addto
    \expandafter\expandafter\expandafter\SRCcontent
    \expandafter\expandafter\expandafter{%
      \expandafter\expandafter\expandafter\SRCblock
      \expandafter\expandafter\expandafter{%
      \expandafter\SRCods\expandafter}\expandafter{\SRCnaz}}%
  \SRCprintblock{\SRCods}{\SRCnaz}%
\fi\par}
```

Makro `\SRCprintline` vysází jeden řádek programu. Čísla vysázených řádků jsou uložena v registru `\SRClinenum`. Pokud je `\SRClinenum<0`, pak se číslo nevysází. Na začátku každého řádku je možné použít `\SRClinehook`.

```
\newcount\SRClinenum
\def\SRCprintline#1{\SRCprintlinenum#1\par}
\def\SRCprintlinenum{\ifprintSRC
    \ifnum\SRClinenum<0 \else
      \global\advance\SRClinenum1
    \fi
  \fi
  \csname SRClinehook\endcsname
  \quitvmode
  \ifnum\SRClinenum<0 \else
    \llap{\sevenrm\the\SRClinenum\kern.9em}%
  \fi}
```

Makro `\SRCprintblock` vysází odkaz na blok. Mezery v odsazení mají kategorii 13 a chovají se jako ostatní mezery uvnitř prostředí `\BEGSRC`.

```
\def\SRCprintblock#1#2{\SRCprintlinenum
  #1\SRCangle{\csname SRCtit:#2\endcsname}\par}
```

6.5 „Makro“ `\ENDSRC`

Makra definovaná v této podsekci se budou provádět v místě použití `\ENDSRC`. Proto má podsekce tento název. Ve skutečnosti ale žádná sekvence `\ENDSRC` není definována. Načtení řádků až po tokeny $\boxed{\backslash}_{12}\,\boxed{\mathtt{E}}_{11}\,\boxed{\mathtt{N}}_{11}\,\boxed{\mathtt{D}}_{11}\,\boxed{\mathtt{S}}_{11}\,\boxed{\mathtt{R}}_{11}\,\boxed{\mathtt{C}}_{11}$ řeší makro `\startSRC`. Pokud je nastaveno `\printSRCfalse`, pak je teoreticky možné uvolnit paměť nastavením `\nonprintbox` na prázdný vbox. Nicméně toto

realizováno není, což umožňuje pomocí háčku `\endSRChook` obsah `\nonprintbox` dále zpracovat.

Samotné vysázení jednotlivých řádků je vyřešeno expanzí `|tt#1` na řádku 147, kdy se prostřednictvím `\everypar` (vizte řádek 96) volá makro `\SRCgetline` a to se expanduje na `\SRCprintline` nebo na `\SRCprintblock`.

```
{\catcode`\|=0 \catcode`\\=12
  |gdef|startSRC#1\ENDSRC{|tt#1|doSRC
    |ifprintSRC
      |egroup|par|ttskip
    |else
      |egroup|egroup
    |fi
    |csname endSRChook|endcsname|testparA}}
```

Makro `\doSRC` zajistí, že řádky a bloky budou dále zpracovány v závislosti na způsobech volání makra `\BEGSRC`.

- U prvního způsobu se do `ds` zapíše řádek
  `\DEFSRC {InterniPopisek}{Formátovaný popisek}`
- U druhého způsobu se do `ds` zapíše řádek
  `\ADDSRC {InterniPopisek}`
- V obou případech se pak zavolá háček `\SRCdshook`, v němž se například mohou lokálně nadefinovat další makra. Dále se jednotlivé řádky obsažené v `\SRCcontent` zapíšou do `ds` makrem `\SRCwritedsline` a bloky makrem `\SRCreadblock`.[17] Po ukončení bloku se do `ds` zapíše token `\ENDDEFSRC`.
- U třetího způsobu se pak pomocí maker `\SRCwriteline` a `\SRCwriteblock` zapíše obsah `\SRCcontent` přímo do `txt`.

```
\def\doSRC{\everypar{}%
  \def\ { }%
  \ifsaveSRC
    \ifaddingSRC
      \immediate\write\SRCdsfile{\noexpand\ADDSRC{\SRCid}}%
    \else
      \immediate\write\SRCdsfile{\noexpand\DEFSRC
        {\SRCid}{\csname SRCtit:\SRCid\endcsname}}%
    \fi
    \let\SRConeline\SRCwritedsline
    \let\SRCbeginline\relax
    \let\SRCendline\relax
```

[17] Trik na řádcích 166 a 167 způsobí, že se token `\SRCblock` zapíše do `ds` jako neexpandovaný token `\SRCreadblock`, za nímž budou následovat původní „argumenty“ makra `\SRCblock` vložené na řádku 129.

```
    \def\SRCblock{\SRCreadblock}%
    \let\SRCreadblock\relax
    \csname SRCdshook\endcsname
    \immediate\write\SRCdsfile{\SRCcontent}%
    \immediate\write\SRCdsfile{\noexpand\ENDDEFSRC}%
  \else
    \let\SRConeline\SRCwriteline
    \let\SRCblock\SRCwriteblock
    \SRCcontent
  \fi}
```

6.6 Cesta z tex do ds

Makro `\SRCwritedsline` zapíše do ds jeden řádek ve tvaru

```
\SRCbeginline : ... \SRCendline
```

Uvědomme si, že při zápisu do souboru se za název řídicí sekvence skládající se z písmen automaticky vkládá mezera. Tato mezera a všechny bezprostředně navazující mezery jsou poté při načítání souboru pohlceny token procesorem. Znak dvojtečka způsobí, že případné mezery na začátku řádku programu (tj. za touto dvojtečkou) nebudou při načítání souboru pohlceny.

```
\def\SRCwritedsline#1{\SRCbeginline:#1\SRCendline}
```

Jak již bylo řečeno v poznámce 17, blok programu se do ds zapíše ve tvaru

```
\SRCreadblock {   }{...}
```

6.7 Cesta z ds do expand procesoru

Již víme, že uvnitř ds jsou uloženy jednotlivé bloky programu v následujícím tvaru.

```
\DEFSRC {InterniPopisek}{Formátovaný popisek}
  \SRCbeginline :...\SRCendline ...\SRCreadblock {   }{...}...
\ENDDEFSRC
\ADDSRC {InterniPopisek}
  \SRCbeginline :...\SRCendline ...\SRCreadblock {   }{...}...
\ENDDEFSRC
```

Makro `\DEFSRC#1#2` uloží argument `#1` do makra `\SRCid` a argument `#2` do makra `\csname SRCtit:\SRCid\endcsname`. Další obsah až po `\ENDDEFSRC` se postupně uloží do makra `\csname SRCcon:\SRCid\endcsname`.

```
\def\DEFSRC#1#2{\def\SRCid{#1}\sdef{SRCtit:#1}{#2}%
  \sdef{SRCcon:#1}{}}
```

Podobně makro `\ADDSRC#1` uloží argument `#1` do makra `\SRCid` a další obsah až po `\ENDDEFSRC` se do makra `\csname SRCcon:\SRCid\endcsname` přidá postupně.

```
179 \def\ADDSRC#1{\def\SRCid{#1}}
```

Makro `\SRCbeginline:` načte verbatim text až po tokeny $\boxed{\backslash}_{12}\ \boxed{S}_{11}\ \boxed{R}_{11}\ \boxed{C}_{11}\ \boxed{e}_{11}\ \boxed{n}_{11}\ \boxed{d}_{11}\ \boxed{l}_{11}\ \boxed{i}_{11}\ \boxed{n}_{11}\ \boxed{e}_{11}$ a tento text přidá k aktuálnímu obsahu makra `\csname SRCcon:\SRCid\endcsname` jako argument makra `\SRConeline`. Všechny mezery se načtou s kategorií 13 (díky makru `\setSRC` a jeho řádku 103) a pak se uvnitř makra `\doSRC` (díky řádku 155) postupně expandují na mezery kategorie 10. Uvnitř definice `\SRCbeginlineA` je třeba s mezerami pracovat opatrně.

```
180 \def\SRCbeginline:{\bgroup\setSRC \catcode`\|=12 \SRCbeginlineA}
181 {\catcode`\|=0 \catcode`\\=12
182   |gdef|SRCbeginlineA#1\SRCendline{|egroup
183     |expandafter|addto|csname SRCcon:|SRCid|endcsname
184       {|SRConeline{#1}}%
185     |ignorespaces}}
```

Makro `\SRCreadblock#1#2` přidá `\SRCblock{#1}{#2}` k aktuálnímu obsahu makra `\csname SRCcon:\SRCid\endcsname`. Je nutné zajistit, aby se mezery z odsazení načítaly s kategorií 13. V souboru `ds` je za tokenem `\SRCreadblock` mezera a ta je na řádku 187 při načítání čísla pohlcena. Poté se pomocí maker `\SRCreadblockA` a `\SRCreadblockB` načtou jednotlivé argumenty.

```
186 \def\SRCreadblock{\begingroup\afterassignment\SRCreadblockA
187   \catcode32=13}
188 \def\SRCreadblockA#1{\gdef\SRCods{#1}\endgroup\SRCreadblockB}
189 \def\SRCreadblockB#1{\expandafter\addto
190   \csname SRCcon:\SRCid\expandafter\endcsname\expandafter{%
191     \expandafter\SRCblock\expandafter{\SRCods}{#1}}}
```

Makro `\ENDDEFSRC` nedělá nic a je použito pouze proto, aby byl obsah souboru `\jobname.ds` čitelnější.

```
192 \let\ENDDEFSRC\relax
```

Soubor `ds` se načítá na začátku dokumentu v průběhu načítání souboru `gensrc.tex`. Pokud soubor `ds` neexistuje, nic se nestane. Pro test existence je použito makro `\softinput` z *TEXbooku naruby* [4, sekce 7.1].

```
193 \newread\testin
194 \def\softinput #1 {\let\next=\relax \openin\testin=#1
195   \ifeof\testin \message{Warning: the file #1 does not exist}%
```

```
196   \else \closein\testin \def\next{\input #1 }\fi
197   \next}
198 \softinput\jobname.ds
```

Interně bude ds reprezentován řídicí sekvencí \SRCdsfile. Soubor se po svém načtení ihned otevře pro zápis.

```
199 \newwrite\SRCdsfile
200 \immediate\openout\SRCdsfile\jobname.ds
```

6.8 Cesta z expand procesoru do txt

V makru \SRCodsazeni je uloženo aktuální odsazení celého bloku, které je uvnitř vnořených \SRCwriteblock lokálně aditivní. Na začátku je odsazení prázdné.

```
201 \def\SRCodsazeni{}
```

Makro \SRCwriteline zapíše do generovaného programu jeden řádek. Na začátku každého zapsaného řádku je možné použít \SRCwritehook.

```
202 \def\SRCwriteline#1{\csname SRCwritehook\endcsname
203   \immediate\write\SRCfile{\SRCodsazeni#1}}
```

Makro \SRCwriteblock lokálně přidá nové odsazení k aktuální hodnotě makra \SRCodsazeni. Mezery v odsazení mají kategorii 13. S tímto odsazením se pak expanduje obsah aktuálního bloku.

Jelikož expand procesor provádí úplnou expanzi, jsou takto na řádku 206 expandovány i případné vnořené bloky programu.

Navíc díky použití \csname nedojde k chybě v případě, že blok s daným názvem není definován, například v prvním průchodu TEXem.

```
204 \def\SRCwriteblock#1#2{\begingroup
205   \addto\SRCodsazeni{#1}%
206   \csname SRCcon:#2\endcsname
207   \endgroup}
```

7 Aplikace

7.1 Imitování podprogramů

V balíčku gensrc jsou bloky programu interně definovány jako makra. Proto je možné stejný blok použít na více místech a do txt jej exportovat vícekrát. Takto je možné celkem přímočaře imitovat volání podprogramů v programovacích jazycích, které volání podprogramů neumožňují.

Jak se ale vyrovnat s podprogramy, které mají argumenty? Použijeme globální proměnné, které si rezervujeme pouze pro použití uvnitř daného „podprogramu“.

V našem příkladě to budou proměnné SDRin1 a SDRin2, jejichž hodnotu nastavíme před voláním „podprogramu“ a které budou sloužit jako jeho argumenty, a proměnná SDRout, do níž „podprogram“ uloží svou výstupní hodnotu.

V ukázkách je část zdrojového souboru. Jak vypadá vygenerovaný program, si lze snadno domyslet.

```
\BEGSRC<SDR>{Concat solutions}
SDRout=SDRin2.clone();
if (SDRin1.max > SDRin2.max) {SDRout.max = SDRin1.max;}
else {SDRout.max = SDRin2.max;}
SDRout.position = SDRin1.position.concat(SDRin2.position);}
\ENDSRC
```

```
\BEGSRC
SDRin1=mainsolution.clone();
SDRin1.position=mainsolution.position.clone();
SDRin2=ZPRout.clone();
SDRin2.position=ZPRout.position.clone();
|<SDR>
mainsolution=SDRout.clone();
mainsolution.position=SDRout.position.clone();
\ENDSRC
```

Tímto způsobem sice není možné imitovat rekurzivní volání podprogramů, nicméně toto v rámci řešení původního úkolu nebylo potřeba.

7.2 Koordinace více programů

Představme si situaci, kdy píšeme několik programů, třeba i v různých programovacích jazycích, a tyto programy mají vzájemně spolupracovat. Například si programy mohou předávat data, kdy výstup některé funkce jednoho programu je použit pro vstup příslušné funkce druhého programu. Při vývoji programů pak potřebujeme, aby tyto funkce byly spolu kompatibilní. Pro lepší údržbu a pochopení je výhodnější, když jsou ve zdrojovém souboru a v dokumentaci tyto funkce poblíž sebe.

Z pohledu balíčku gensrc se jedná o situaci, kdy průběžně generujeme dva soubory txt, přičemž se dokonce může stát, že některé části kódu budou společné. Soubory txt můžeme generovat každý se svým blokem a tyto bloky poté definovat průběžně.

```
\SRCFILENAME programA.txt
\BEGSRC
|<A>
\ENDSRC
```

```
\SRCFILENAME programB.txt
\BEGSRC
|<B>
\ENDSRC
\BEGSRC<A>{Program A}
|<firstpart>
in the first
|<secondpart>
\ENDSRC
\BEGSRC<B>{Program B}
|<firstpart>
in the second
|<secondpart>
\ENDSRC
\BEGSRC<firstpart>{First part}
This will be
\ENDSRC
\BEGSRC<secondpart>{Second part}
program.
\ENDSRC
```

Může být užitečné v dokumentaci odlišit barvou, ke kterému generovanému souboru příslušná část kódu patří. K tomu lze využít háček `\SRChook`.

V následující ukázce navíc háčky nastavíme tak, aby jejich změna byla lokální a aby se po ukončení prostředí `\BEGSRC` předefinovaly zpět na předchozí hodnotu. Rozbor ukázky ponechávám na čtenáři.

```
\def\switchSRC#1#2{%
  \sdef{#1BEGSRC}{%
    \let\exSRChook\SRChook
    \def\SRChook{\longlocalcolor#2%
      \global\let\SRChook\exSRChook}%
    \BEGSRC}}
\switchSRC{a}{\Red}
\switchSRC{b}{\Green}
```

```
\SRCFILENAME programA.txt
\aBEGSRC
|<A>
\ENDSRC
\SRCFILENAME programB.txt
\bBEGSRC
|<B>
\ENDSRC
```

```
\aBEGSRC<A>{Program A}
|<firstpart>
in the first
|<secondpart>
\ENDSRC
\bBEGSRC<B>{Program B}
|<firstpart>
in the second
|<secondpart>
\ENDSRC
\BEGSRC<firstpart>{First part}
This will be
\ENDSRC
\BEGSRC<secondpart>{Second part}
program.
\ENDSRC
```

Příslušná dokumentace pak vypadá následovně.

```
 1 ⟨Program A⟩

 2 ⟨Program B⟩

⟨Program A⟩ ≡
 3 ⟨First part⟩
 4 in the first
 5 ⟨Second part⟩

⟨Program B⟩ ≡
 6 ⟨First part⟩
 7 in the second
 8 ⟨Second part⟩

⟨First part⟩ ≡
 9 This will be

⟨Second part⟩ ≡
10 program.
```

7.3 Křížové odkazy

V balíčku gensrc není implementován mechanismus maker jako v jazyku WEB. Pokud si ale uvědomíme, že uvnitř prostředí \BEGSRC má znak | kategorii 0, tj. uvozuje název řídicí sekvence, pak není problém

```
v programu použít jednoduché |makro nebo dokonce
použít |Makro(s jedním)(i více) argumenty,
```

pokud si řídicí sekvence \makro a \Makro předem nadefinujeme. Pouze si musíme dát pozor na to, aby v okamžiku definování měly znaky, které oddělují argumenty, kategorii 12, případně si musíme správné načtení argumentů zajistit jiným způsobem.

V následujících ukázkách budeme používat také makra, která nebudeme definovat globálně, ale využijeme mechanismus háčků k tomu, abychom v různých okamžicích jejich definici měnili lokálně.

Když autor tohoto článku začínal s programováním, měl počítač Commodore 64 s jazykem BASIC. Po třech desetiletích objevil na internetu emulátor tohoto počítače [9] a z nostalgie si v něm trochu zaprogramoval.

Emulátor má jako počítač Commodore 64 stejně nízkou rychlost, stejně nízkou přesnost reálné aritmetiky a stejně nízké rozlišení obrazovky. Tyto nevýhody autor přetavil ve výhody a vytvořil názorný program používaný pro řešení úloh v jím vyučovaném předmětu Numerická matematika.

Pro jazyk BASIC je charakteristické, že

- program tvoří posloupnost příkazů, která nijak neodpovídá logické struktuře programu.
- Na začátku každého řádku je uvedeno číslo tohoto řádku. Bývá zvykem, že posloupnost čísel řádků má krok 10.
- Program obsahuje skoky na řádky s daným číslem.

Tyto problémy jsou dobře vidět na následující ukázce části programu.

```
1190 let ra=15: gosub 2080: input a
1200 let ra=16: gosub 2080: input b
1210 if b<a then let c=a: let a=b: let b=c
1220 let x=a: gosub 3100: let d=f
1230 let x=b: gosub 3100
1240 if sgn(f)*sgn(d)<0 then 1270
1250 let ra=17: gosub 2270
1260 goto 1390
```

S balíčkem gensrc je možné uvedené problémy vyřešit následovně.

- Logickou strukturu programu lze vytvořit a udržovat pomocí bloků.
- Číslo řádku je známo až po úplné expanzi všech bloků při zápisu do souboru txt. Na tom místě lze využít háček `\SRCwritehook`.
- Pro skoky lze použít mechanismus křížových odkazů. Přitom číslo řádku v dokumentaci je obecně jiné než číslo řádku v programu. V dokumentaci pak křížové odkazy mohou být aktivní.

V ukázce je příslušná část dokumentace.

3.2.4 Bisekce

Na začátku bisekce uživatel zadá krajní body do proměnných `A`, `B`.

⟨*Bisekce*⟩ ≡

```
370 LET RA=15: GOSUB 139: INPUT A
371 LET RA=16: GOSUB 139: INPUT B
```

Ze slušnosti je zařízeno, aby platilo `A < B`.

⟨*Bisekce*⟩ +≡
```
372 IF B<A THEN LET C=A: LET A=B: LET B=C
```

Pokud nejsou v krajních bodech různá znaménka, vypíše se chybové hlášení a metoda se ukončí.

⟨*Bisekce*⟩ +≡
```
373 LET X=A: GOSUB 173: LET D=F
374 LET X=B: GOSUB 173
375 IF SGN(F)*SGN(D)<0 THEN 378
376 LET RA=17: GOSUB 158
377 GOTO 390
```

Podívejme se, jak lze uvedený mechanismus čísel řádků a křížových odkazů implementovat.

Pro křížové odkazy budeme používat řídicí sekvence `\LL(#1)` a `\RR(#1)`, kde oddělovače argumentů mají kategorii 12. Tyto sekvence budou na grafu na straně 80 cestovat všemi šipkami. Pomocí háčků budeme na jednotlivých místech definovat, jak se mají tyto sekvence expandovat nebo neexpandovat.

Při sazbě dokumentace se makro `\LL(#1)` chová stejně jako `\label[S:#1]`, přičemž se odkazuje na číslo řádku `\SRClinenum`. Makro `\RR(#1)` se chová jako `\ref[S:#1]` se změnou fontu dle řádku 142.

```
216 \def\SRChook{\def\LL(##1){%
217     \immediate\write\reffile
218       {\string\Xlabel{S:##1}{\the\SRClinenum}}%
219     \dest[ref:S:##1]}%
220   \def\RR(##1){{\sevenrm\ref[S:##1]}}}
```

Při zápisu do `ds` se makra pouze přepíšou s uvozujícím znakem `|`. Ten má při zápisu do `ds` kategorii 12 a při opětovném načtení `ds` kategorii 0. Obě makra tak budou i nadále jako makra fungovat.

```
221 \def\SRCdshook{\def\LL(##1){|LL(##1)}%
222   \def\RR(##1){|RR(##1)}}
```

Nejsložitější je zápis do `txt`, protože při zápisu do souboru není možné provádět příkazy hlavního procesoru. V našem případě potřebujeme příkazy pro

- zvýšení čísla řádku a
- zápis čísla řádku do souboru `\jobname.ref`, který se v OPmacu používá pro realizaci křížových odkazů.

Háček `\SRCwritehook` nadefinujeme tak, že namísto zápisu řádku do `txt` tento řádek načte,[18] dále zpracuje a sám zajistí zápis. Nejdříve řádek vysází do pomocného `\box0`, přičemž uvnitř tohoto boxu lokálně nadefinuje `\LL(#1)` jako

[18] V ukázce na straně 83 je vidět, kdy přesně se háček `\SRCwritehook` volá a jakým způsobem lze pomocí něj načíst argument příkazu `\write`.

`\label[B:#1]`, zatímco `\RR(#1)` ignoruje. Poté řádek zapíše do txt, přičemž na začátek řádku vloží jeho číslo, `\LL(#1)` ignoruje a `\RR(#1)` lokálně nadefinuje jako `\ref[B:#1]`.

```
\newcount\cislo
\def\SRCwritehook#1\SRCfile#2{\advance\cislo10
  {\setbox0=\hbox{%
    \def\LL(##1){\immediate\write\reffile
      {\string\Xlabel{B:##1}{\the\cislo}}}%
    \def\RR(##1){}%
    \lowercase{#2}}}%
  {\def\LL(##1){}%
    \def\RR(##1){\csname lab:B:##1\endcsname}
    \lowercase{\immediate\write\SRCfile{\the\cislo\space#2}}}}
```

Následuje ukázka příslušné části zdrojového kódu s využitím výše uvedených maker.

```
\seccc Bisekce

Na začátku bisekce uživatel zadá krajní body do proměnných "A", "B".
\BEGSRC<bisekce>{Bisekce}
|vypisretezec(15): INPUT A |LL(bisekce)
|vypisretezec(16): INPUT B
\ENDSRC
Ze slušnosti je zařízeno, aby platilo "A"${}<{}$"B".
\BEGSRC<bisekce>
IF B<A THEN LET C=A: LET A=B: LET B=C
\ENDSRC
Pokud nejsou v~krajních bodech různá znaménka, vypíše se chybové hlášení a
metoda se ukončí.
\BEGSRC<bisekce>
LET X=A: GOSUB |RR(deffce): LET D=F
LET X=B: GOSUB |RR(deffce)
IF SGN(F)*SGN(D)<0 THEN |RR(bis1)
|vypisretezecln(17)
GOTO |RR(bis0)
\ENDSRC
```

Vygenerovaný program lze programem petcat [10] převést do souboru s příponou prg, ve kterém je po bajtech uložen obsah paměti počítače Commodore 64, jaký by byl po načtení programu do počítače. Soubor prg lze spustit v emulátoru [9]. Programem prg2wav [11] lze dále vytvořit soubor s příponou wav, který odpovídá zvuku nahranému na magnetofonovou kazetu.[19] Soubor wav lze dále na

[19] Dříve narozenému čtenáři se zajisté okamžitě vybavilo ono charakteristické „chrrr píp píííp píip chrrrrr píííp“. :-)

kazetu nahrát, nicméně autorovi těchto řádků se nepodařilo jej nahrát tak, aby jej počítač Commodore 64 načetl správně.[20]

7.4 Program ve více jazycích

V této podsekci se zaměříme na situaci, kdy potřebujeme vytvořit stejný program v několika podobných programovacích jazycích. V této situaci s úspěchem využijeme TEXová makra, která se v závislosti na přepínači budou expandovat různě.

V ukázkách se bude jednat o dva dialekty jazyka SQL, přičemž aktuální dialekt bude určovat makro `\version`. Podle něj se nastaví přepínač `\ifOR` a ten poté bude rozhodovat o správné expanzi maker.

```
\def\versionOR{OR}
\unless\ifdefined\version \let\version\versionOR \fi
\newif\ifOR
\ifx\version\versionOR \ORtrue \fi
```

Vytvoříme makro `\variant#1#2#3`, které nadefinuje makro `#1` s jedním argumentem, aby se expandovalo jako `#2`, nebo jako `#3`.

```
\def\variant#1#2#3{\ifOR \sdef{#1}##1{#2}%
  \else \sdef{#1}##1{#3}\fi}
```

Dále nadefinujeme samotná makra s jejich expanzí v jednotlivých dialektech.

```
\variant{integer}{number(38)}{decimal(38)}
{\catcode`\_=12 \globaldefs=1
  \variant{rownum}{rownum}
    {(row_number() over (order by #1))}}
```

Uvnitř prostředí `\BEGSRC` můžeme makra volat s uvozujícím znakem `|`. Nicméně ukážeme si, jak je možné pro volání maker využít znak kategorie 13. Znak `@` nadefinujeme tak, aby načítal název makra až po další znak `@` a pak načítal argument makra až po třetí znak `@`. Znak musí být aktivní v okamžiku definování i uvnitř prostředí `\BEGSRC`. Případné vysázení znaku `@` se provede pomocí `@@@`.

```
{\catcode`\@=13 \gdef@#1@#2@{\csname#1\endcsname{#2}}}
\sdef{}#1{@}
\def\SRChook{\catcode`\@=13 }
```

S těmito definicemi již můžeme psát části programu.

[20] Pokud někdo ze čtenářů má s nahráváním souboru `wav` na kazetu zkušenosti, bude autor za předání informací rád.

```
\SRCFILENAME program\version.txt
\BEGSRC
select cast(NN/p as @integer@@), st+1
from FindN1,
    (select @rownum@p@ r, p from SmallFactors) a
where st=r
\ENDSRC
```

Příslušný vygenerovaný program a dokumentace při nastavení `\ORtrue` jsou následující:

```
select cast(NN/p as number(38)), st+1
from FindN1,
    (select rownum r, p from SmallFactors) a
where st=r
```

```
1 select cast(NN/p as number(38)), st+1
2 from FindN1,
3     (select rownum r, p from SmallFactors) a
4 where st=r
```

Podobně pro `\ORfalse`.

```
select cast(NN/p as decimal(38)), st+1
from FindN1,
    (select (row_number() over (order by p)) r, p from SmallFactors) a
where st=r
```

```
1 select cast(NN/p as decimal(38)), st+1
2 from FindN1,
3     (select (row_number() over (order by p)) r, p from SmallFactors) a
4 where st=r
```

Jak ale zajistit, abychom vygenerovali program najednou v obou dialektech a nemuseli přitom ručně nastavovat `\version`? Stačí si vytvořit řídicí soubor, kterým TEX spustíme dávkově.

```
pdfcsplain soubor
pdfcsplain soubor
mv soubor.pdf souborOR.pdf
pdfcsplain '\def\version{MS}\input soubor'
pdfcsplain '\def\version{MS}\input soubor'
mv soubor.pdf souborMS.pdf
```

Odkazy

1. OLŠÁK, Petr. *OPmac: Rozšiřující makra plain TEXu* [online]. [cit. 2023-11-13]. Dostupné z: https://petr.olsak.net/opmac.html.
2. KNUTH, Donald E. *Literate Programming* [online]. 1983-09. [cit. 2023-11-13]. Dostupné z: http://www.literateprogramming.com/knuthweb.pdf. Submitted to The Computer Journal.
3. KNUTH, Donald E. *TEX: The Program*. Sv. B. Reading, MA: Addison-Wesley, 1986. Computers & Typesetting. Dostupné také z: https://tug.ctan.org/systems/knuth/dist/tex/tex.web.
4. OLŠÁK, Petr. *TEXbook naruby*. 2. vyd. Konvoj, 2001. Dostupné také z: http://petr.olsak.net/ftp/olsak/tbn/tbn.pdf.
5. THE LATEX PROJECT TEAM; MITTELBACH, Frank. docstrip*: Remove comments from file* [online]. 2022-09-03. [cit. 2023-11-13]. Dostupné z: https://ctan.org/pkg/docstrip.
6. MITTELBACH, Frank. *The* doc *and* shortvrb *Packages* [online]. 2023-11-01. [cit. 2023-11-13]. Dostupné z: https://mirrors.ctan.org/macros/latex/base/doc.pdf.
7. MITTELBACH, Frank; DUCHIER, Denys; BRAAMS, Johannes; WOLIŃSKI, Marcin; WOODING, Mark. *The* DocStrip *program* [online]. 2023-11-01. [cit. 2023-11-13]. Dostupné z: http://mirrors.ctan.org/macros/latex/base/docstrip.pdf.
8. ŠUSTEK, Jan. gensrc.tex [online]. 2023-11-07. [cit. 2023-11-13]. Dostupné z: https://github.com/jsustek/gensrc/blob/main/gensrc.tex.
9. *C64 online emulator* [online]. [cit. 2023-11-13]. Dostupné z: https://virtualconsoles.com/online-emulators/c64/.
10. *VICE: The Versatile Commodore Emulator* [online]. [cit. 2023-11-13]. Dostupné z: https://vice-emu.sourceforge.io/.
11. TOM THE DRAGON. *prg2wav: C64 PRG to Turbo Tape 64* [online]. [cit. 2023-11-13]. Dostupné z: https://github.com/tomdwaggy/prg2wav.

Summary: On Generating Documented Source Code by Blocks in TEX

This paper concerns writing programs and their documentation. We show author's package gensrc running on OPmac, which allows to write both program and its documentation in one TEX file. We also show more possibilities and applications of this package.

Keywords: documentation, literate programming, OPmac, gensrc

Jan Šustek, jan.sustek@seznam.cz