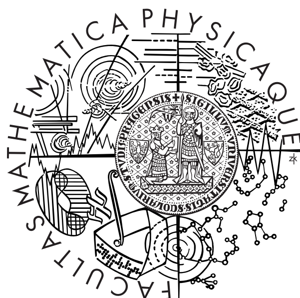
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jan Procházka

Generátor kompilátorů založený na restartovacích automatech

Kabinet software a výuky informatiky

Vedoucí diplomové práce: RNDr. František Mráz, CSc.
Studijní program: Informatika, Teoretická informatika

2010

Na tomto místě nemohu nepoděkovat svému vedoucímu, panu doktoru Mrázovi, a to hlavně za expresní připomínkování vznikajícího textu v posledních týdnech. Dále děkuji panu doktoru Plátkovi. Byly to peníze z jeho grantu, které mi umožnily odprezentovat dílčí výsledky této práce na konferenci CIAA 2009 v Sydney. V neposlední řadě pak děkuji svým nejbližším, bez jejichž podpory by tato práce asi nikdy nevznikla.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 14. dubna 2010 Jan Procházka

Obsah

1 Úvod **6**

2 Teoretický úvod **8**
2.1 Generátor kompilátoru 8
2.1.1 `yacc` – základní koncept 9
2.2 Restartovací automaty 13
2.2.1 Definice 14
2.2.2 Speciální automaty 15
2.2.3 Ekvivalentní definice 16
2.3 Restartovací automat se sémantikou 18
2.4 Notace pro regulární jazyky 20
2.4.1 SNOBOL 20
2.4.2 QED 22
2.4.3 Perl 23
2.4.4 CCRA 26

3 Popis implementace **28**
3.1 Volba jazyka 28
3.2 Formát vstupního souboru 28
3.3 Reprezentace automatů 30
3.4 Regulární výrazy 31
3.5 Nedeterminismus 32
3.6 Soubory parseru 34
3.6.1 Hlavičkový soubor 34
3.6.2 Knihovní soubor 34
3.6.3 Soubor přechodových tabulek 35
3.7 Usnadnění návrhu 35

4 Závěr **37**

Literatura **39**

A CCRA – uživatelská příručka **41**
A.1 Úvod 41
A.2 Základní koncepty 42
A.2.1 Jazyk a jeho popis 42
A.2.2 Meta-instrukce 42
A.2.3 Od meta-instrukcí ke vstupu CCRA 43
A.2.4 Sémantická hodnota 44
A.2.5 Sémantické akce 45
A.2.6 Generované soubory 45
A.2.7 Kroky při používání CCRA 46
A.2.8 Struktura zápisu gramatiky pro CCRA 46
A.3 Příklady 47
A.3.1 context2 47
A.3.2 palindrom 52
A.4 Vstupní soubor CCRA 57
A.4.1 Prolog 58
A.4.2 CCRA deklarace 59
A.4.3 Gramatická pravidla 60
A.4.4 Epilog 64
A.5 Rozhraní parseru 64
A.5.1 Vstupní a pomocné symboly 64
A.5.2 Lexikální analýza 65
A.5.3 Použití parseru 66
A.5.4 Problémy nedeterminismu 67
A.5.5 Externí tabulky 68
A.5.6 Chybová hlášení 71
A.6 Parsovací algoritmus 71
A.7 Instalace CCRA 72

B Obsah přiloženého CD **74**

Název práce: Generátor kompilátorů
založený na restartovacích automatech
Autor: Jan Procházka
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. František Mráz, CSc.
e-mail vedoucího: Frantisek.Mraz@mff.cuni.cz

Abstrakt: Restartovací automaty jsou velmi silný teoretický model, který umožňuje (ve své nejobecnější variantě) rozpoznávat mnohem širší třídu jazyků, než je třída jazyků bezkontextových. Cílem této práce bylo vytvořit nástroj, který pro daný restartovací automat zapsaný pro člověka čitelnou formou generuje program počítající význam vstupního textu. Bylo proto nutné model rozšířit o sémantiku. Výsledný nástroj je generátor kompilátorů (CCRA) a vychází z nástrojů jako je `flex`, či `bison`, ale místo bezkontextové gramatiky používá restartovací automat. Je napsaný v jazyce C++ tak, aby bylo možné ho používat jak na systémech Windows, tak Linux. Stejný jazyk používá pro svůj výstup.

Klíčová slova: restartovací automaty, regulární výrazy,
generátor kompilátorů.

Title: Compiler generator based on restarting automata
Author: Jan Procházka
Department: Department of software and computer science education
Supervisor: RNDr. František Mráz, CSc.
Supervisor's e-mail address: Frantisek.Mraz@mff.cuni.cz

Abstract: Restarting automata, in their most general form, represent a very strong theoretical model recognizing much wider class of languages than the class of context-free ones. Hence, our goal was to design a tool which for a given restarting automaton (in human-readable format) generates a program computing the meaning of an input text. In order to enable that, this thesis extends the model of restarting automata by adding semantics to its meta-instruction. The resulting program is a compiler-compiler (CCRA) inspired by the tools such as `flex` or `bison`. However, the CCRA uses restarting automaton instead of a context-free grammar. The implementation as well as the output are realized in C++ which ensures the compatibility with both Windows and Linux systems.

Keywords: restarting automata, regular expressions, compiler-compiler.

Kapitola 1

Úvod

Restartovací automat je velmi silný teoretický model stavového automatu, který umožňuje realizovat závislostní analýzu. Ta je vhodná převážně k analýze vět z přirozených jazyků s volným slovosledem, tedy například češtiny. Jednou z jeho předností je způsob, jakým pracuje s chybným vstupem. I takový vstup totiž dokáže zjednodušovat a nechat jen to, co způsobuje chybu, tzv. jádro chyby [3]. Navíc se ukazuje, že vyjadřovací síla restartovacího automatu je překvapivě veliká (dokáže rozpoznávat ostrou nadmnožinu bezkontextových jazyků). Naopak největší nevýhodou tohoto modelu je, že není definován způsob, jak pracovat se sémantikou. Tento nedostatek omezuje jeho použití na kontrolu syntaxe, což pro většinu reálných aplikací není postačující.

Z toho důvodu v této práci zavádíme model restartovacího automatu rozšířený o sémantikou, na kterém je již možné založit plnohodnotný kompilátor. Cílem této práce bylo vytvořit generátor těchto kompilátorů (jménem CCRA). Hlavní motivací byla skutečnost, že existuje celá řada jazyků (typicky na hranici mezi jazyky přirozenými a umělými), které jsou sice dostatečně jednoduché, aby je bylo možné formálně popsat, ale již příliš složité, aby to mohlo být bezkontextovou gramatikou, což je požadavek většiny generátorů kompilátorů. Dobrým příkladem takového jazyka je české názvosloví organické chemie.

V kapitole 2 je možné seznámit se se všemi teoretickými východisky. Od popisu základních vlastností generátoru kompilátorů `yacc`, přes definice nejběžnějších typů restartovacích automatů a zavedení nového typu – restartovacího automatu se sémantikou až po historický přehled zápisů regulárních jazyků, včetně modifikace regulárních výrazů pro potřeby CCRA. Následující kapitola 3 obsahuje diskusi všech důležitých implementačních rozhodnutí. V kapitole 4 jsou shrnuty dosažené výsledky a naznačeny cesty, kterými je možné ve vývoji CCRA pokračovat. Podrobná

uživatelská příručka tohoto nástroje je součástí práce v podobě přílohy A. Součástí práce je CD, s jehož obsahem je možné se seznámit v příloze B.

Kapitola 2

Teoretický úvod

Cílem této práce bylo vytvořit generátor kompilátorů používající jako parser restartovací automat [2]. V této kapitole se tedy seznámíme s generátorem kompilátorů, který byl předlohou námi vytvářenému nástroji. V navazující sekci si stručně představíme základní typy restartovacích automatů. Následuje definice restartovaícho automatu jako přepisovacího systému založeného na formalismu meta-instrukcí. Pro generování kompilátorů je třeba meta-instrukce rozšířit o sémantiku. Takovéto rozšíření je navrženo v oddíle 2.3. V meta-instrukcích se používají reglární jazyky, které se obvykle zapisují jako regulární výrazy. Ty využívá celá řada programů (např. `grep`, `sed` a mnoho dalších). Regulární výrazy používané v různých programech se liší. Oddíl 2.4 proto stručně shrnuje jejich historický vývoj a následně popisuje regulární výrazy, které používá náš generátor kompilátorů.

2.1 Generátor kompilátoru

Hlavním cílem práce bylo vytvořit nástroj, který by umožňoval provádět syntaktickou analýzu za pomoci restartovacích automatů. Nezabývali jsme se zatím ale tím, jak.

Možnosti jsou v zásadě dvě. Mohla by to být aplikace, která by simulovala práci restartovacího automatu. Takové řešení by ovšem bylo velmi složité pro zápis sémantiky, nebo by znamenalo použití velmi jednoduchého jazyka. Navíc by možnosti využití takového nástroje byly značně omezené. Druhá možnost je vytvořit nástroj, který vygeneruje zdrojový kód syntaktického analyzátoru. Toto řešení má tu obrovskou výhodu, že se dá začlenit do libovolného (zaměřením i rozsahem) projektu. Výhod je ale ještě více:

- Vygenerovaný kód lze snadno udělat platformě nezávislý.
- Algoritmy, které plánujeme použít, mají relativně vysokou časovou složitost, proto je vhodné vyhnout se veškerým interpretacím, které výpočet dále zpomalují. [1]
- Vygenerovaný program se mnohem lépe ladí, protože je k dispozici široká škála ladicích nástrojů a vývojových prostředí.

Generátor kompilátorů je nástroj, který z nějakého formálního popisu jazyka vyrobí jeho syntaktický analyzátor (parser/interpretr nebo kompilátor). Nejběžnější (a zároveň nejstarší) je generátor parserů. Ten dostane jako vstup gramatiku programovacího jazyka (obvykle popsanou v BNF) a jako výstup vydá zdrojový kód parseru.

Takovýchto generátorů existuje celá řada. Liší se od sebe jak ve volbě parsovacích algoritmů (DFA, NFA, LL(k), LALR, GLR, aj.), tak volbou programovacího jazyka (od Common-Lispu po PHP) [1].

Jedním z nejstarších (ale co je podstatnější jednoznačně nejrozšířenějších) generátorem kompilátorů je program `yacc`. Je dostupný jako standardní nástroj ve většině systémů Unix[2]. Také je součástí projektu OpenSolaris. Pro svou oblibu byl `yacc` přepsán i do řady méně rozšířených programovacích jazyků (jako např. Common-Lisp, ML či ADA).

V této podkapitole se seznámíme s jeho základními vlastnostmi, abychom tyto poznatky mohli později využít při vlastním návrhu generátoru kompilátorů.

2.1.1 `yacc` – základní koncept

Formát vstupního souboru použitý v programu `yacc` je velmi blízký tzv. BNF (Backus-Naurově formě) umožňující přehledný zápis bezkontextové gramatiky.

Následující příklad popisuje dva možné zápisy datumu. A sice ve formátu 1. Jan 1993 a 1. 1. 1993.

---

[1] Rychlost je ve skutečnosti v této práci upozaděna. Hlavním cílem je otestovat úplně nový koncept generátoru kompilátorů. Bylo by ovšem neprozíravé znemožnit dodatečnou optimalizaci tak podstatného parametru, jakým rychlost analýzy jazyka bez pochyby je.

[2] Ve skutečnosti se ve většině případů jedná o GNU verzi s několika vylepšeními s názvem `Bison`.

```
datum : den '.' mesic '.' rok
      | den '.' nazevMesice rok
      ;
den : NUM;
rok : NUM;
mesic : NUM;
nazevMesice : 'J' 'a' 'n'
            | 'F' 'e' 'b'
            | 'M' 'a' 'r'
            | 'A' 'p' 'r'
            | 'M' 'a' 'y'
            | 'J' 'u' 'n'
            | 'J' 'u' 'l'
            | 'A' 'u' 'g'
            | 'S' 'e' 'p'
            | 'O' 'c' 't'
            | 'N' 'o' 'v'
            | 'D' 'e' 'c'
            ;
```

Jedná se o sadu (odvozovacích) pravidel, z nichž každé je uvedeno na samostatném řádku. První pravidlo říká, že token datum sestává z tokenů `den`, `mesic` a `rok` oddělených tečkami. Jak je možné zapsat tyto části, říkají další pravidla. Pokud existuje více možných způsobů zápisu, oddělují se tyto od sebe znakem “`|`”.

Je zřejmé, že není možné se takto zanořovat do nekonečna a je tedy nutné některé tokeny popsat přímo. Možnosti jsou dvě. Pokud existuje pouze malé množství variant, je možné všechny tyto tzv. terminály vyjmenovat tak, jak je to provedeno v předcházejícím příkladu u názvu měsíce. Může se ovšem jednat o doménu, kterou není snadné (mnohdy dokonce ani možné) popsat výčtem (v našem příkladu se jedná o terminál `NUM`). V takovém případě je nutné předřadit generovanému parseru lexikální analyzátor, který zajistí to, že objeví-li se na vstupu sekvence číslic, bude předán terminál `NUM` s poznámkou, jaká je jeho hodnota. Propojení s lexikálním analyzátorem se provede velmi snadno. Stačí v prologu definovat makro `YY_DECL`, jehož hodnota je hlavička hlavní funkce lexikálního analyzátoru, která pracuje tak, že při každém zavolání vrátí další token ze vstupu.

Tím se dostáváme k širšímu rámci vstupního souboru, do kterého jsou pravidla zasazena. Soubor má následující strukturu:

```
%
Prolog
%
Deklarace

%%
Gramatická pravidla
%%
Epilog
```

Do *Prolog*u se píše kód v jazyce C, který má být vložen před vlastní kód parseru. Typicky se tedy jedná o makra, deklarace globálních proměnných a dopředné deklarace funkcí (jejich implementace jsou pak obvykle umístěny v části *Epilog*).

Aby bylo možné startovní (`datum`) a pomocné (`den`, `mesic`, `rok` a `nazevMesice`) neterminály odlišit od terminálů, je nutné je deklarovat pomocí příkazů `%start` a `%token`. Ty se uvádějí v části *Deklarace*. `%start datum` deklaruje token `datum` jako počáteční neterminál gramatiky, `%token NUM` zas `NUM` jako terminál.

V této podobě je již možné psát plně funkční syntaktické analyzátory. Chceme-li však vytvořit překladač (v našem příkladě převádějící mezi sebou různé zápisy datumu), je nutné přidat k jednotlivým pravidlům tzv. sémantické akce. Jedná se o kód příslušící nějakému pravidlu, který je vykonán v okamžiku, kdy je toto pravidlo použito. Účelem tohoto kódu je správně nastavit sémantickou hodnotu právě vznikajícího neterminálu na levé straně pravidla na základě sémantických hodnot tokenů na pravé straně pravidla (těch, ze kterých vzniká). Toto se nazývá atributový překlad. Na sémantickou hodnotu nového tokenu se odkazuje pomocí značky $$, pro použití sémantických hodnot tokenů původních se používá značka `$`n, kde n je pořadové číslo tokenu v daném pravidle. Datový typ použitý pro ukládání sémantické hodnoty je možné nastavit jako hodnotu makra `YYSTYPE`.

Nyní již máme vše potřebné, abychom si ukázali plnou verzi našeho konvertoru zápisu datumů.

Datum.y

```
%{
  #include <stdio.h>

  #define YY_DECL int yylex()
```

```
  YY_DECL;
  void yyerror(char const *s);

  #define YYSTYPE int
%}

%start datum
%token NUM

%%
datum : den '.' mesic '.' rok
        {printf("%02d-%02d-%02d\n",$5%100,$3,$1);}
      | den '.' nazevMesice rok
        {printf("%02d-%02d-%02d\n",$4%100,$3,$1);}
      ;
den : NUM {$$=$1;};
rok : NUM {$$=$1;};
mesic : NUM {$$=$1;};
nazevMesice : 'J' 'a' 'n'    {$$=1;}
            | 'F' 'e' 'b'    {$$=2;}
            | 'M' 'a' 'r'    {$$=3;}
            | 'A' 'p' 'r'    {$$=4;}
            | 'M' 'a' 'y'    {$$=5;}
            | 'J' 'u' 'n'    {$$=6;}
            | 'J' 'u' 'l'    {$$=7;}
            | 'A' 'u' 'g'    {$$=8;}
            | 'S' 'e' 'p'    {$$=9;}
            | 'O' 'c' 't'    {$$=10;}
            | 'N' 'o' 'v'    {$$=11;}
            | 'D' 'e' 'c'    {$$=12;}
            ;
%%

int yylex(){
    static int last;

    if(!last) last=getchar();
    switch(last){
        case ' ':
        case '\t':
            last=0;
            return yylex();
        case '\n':
            getchar();
            return last=0;
        default:
            if(('0'<=last) && (last<='9')){    //cislo
```

```
                yylval=0;
                while(('0'<=last) && (last<='9')){
                    yylval=10*yylval+(last-'0');
                    last=getchar();
                }
                return NUM;
            }else{                          //pismeno
                int ret=last; last=0;
                return ret;
            }
    }
}

void yyerror(char const *s){ fprintf(stderr, "%s\n", s); }
int main(void){ return yyparse(); }
```

Za povšimnutí stojí hlavně to, že uživateli stačí zapsat gramatická pravidla a jejich význam. Dalšího kódu je již potřeba opravdové minimum.

2.2 Restartovací automaty

Restartovací automat [2] je model nedeterministického stroje, který zpracovává řetězec uložený na pružné pásce realizované spojovým seznamem. Má čtecí a zapisovací hlavu s konečným výhledem a konečnou řídící jednotku umožňující několik operací:

- přechod s posunem hlavy doprava,
- přepsání symbolů ve výhledu (na něco kratšího),
- restart (tj. posun hlavy na začátek a změna stavu řídící jednotky na počáteční – neexistuje tedy žádný způsob, jak by si automat pamatoval nějakou akci, kterou vykonal před restartem) a
- přijetí slova (automat dojde až na konec slova na pásce a je v přijímacím stavu).

Tento model je motivován redukční analýzou, která je blízká mnohým přirozeným jazykům. Myšlenka je taková, že věta (vstupní slovo) je zjednodušována tak, aby její správnost (či chybnost) byla zachována. Tím je docíleno toho, že je věta buď zjednodušena natolik, že rozhodnutí

o její správnosti je snadné, nebo (je-li věta chybná) se redukce zastaví v okamžiku, kdy z původní věty zbylo jen tzv. “jádro” chyby, tzn. věta je redukována na ty části, které se chyby bezprostředně týkají.

V této kapitole si ukážeme formální podobu definice, a pak některé významnější obměny (omezení, rozšíření, ale i ekvivalenty).

2.2.1 Definice

Definice 2.1 *Restartovací automat (neboli* RRWW*-automat) je osmice* $M = (Q, \Sigma, \Gamma, ^\wedge, \$, q_0, k, \delta)$*, kde* Q *je konečná množina stavů,* Σ *je konečná vstupní abeceda,* Γ *je konečná pásková abeceda obsahující* Σ *a neobsahující* $^\wedge$[3] *a* \$ *(tj. levou a pravou zarážku),* $q_0 \in Q$ *je počáteční stav,* $k \geq 1$ *je šířka výhledového okénka a*

$$\delta : Q \times \mathcal{PC}^{(\leq k)} \rightarrow P\left(\left(Q \times \left(\{\mathsf{MVR}\} \cup \mathcal{PC}^{(\leq k-1)}\right)\right) \cup \{\mathsf{Restart}, \mathsf{Accept}\}\right)$$

je přechodová relace. Zde $P(M)$ *značí potenční množinu množiny* M *a*

$$\mathcal{PC}^{(\leq k)} := \left(^\wedge \cdot \Gamma^{k-1}\right) \cup \Gamma^k \cup \left(\Gamma^{\leq k-1} \cdot \$\right) \cup \left(^\wedge \cdot \Gamma^{\leq k-2} \cdot \$\right)$$

je množina “možných obsahů” výhledového okénka M*, kde* $\Gamma^{\leq n} := \cup_{i=0}^{n} \Gamma^i$ *a* $\mathcal{PC}^{(\leq k-1)}$ *je definováno analogicky.*

Přechodová relace popisuje čtyři různé typy přechodu:

1. *Posun doprava je ve tvaru* $(q', \mathsf{MVR}) \in \delta(q, u)$*, kde* $q, q' \in Q$ *a* $u \in \mathcal{PC}^{(\leq k)}$*,* $u \neq \$$*. Pokud je* M *ve stavu* q *a pod hlavou je výhled* u*, pak automat posune hlavu o jeden znak doprava a změní stav na* q'*.*

2. *Přepis je ve tvaru* $(q', v) \in \delta(q, u)$*, kde* $q, q' \in Q$*,* $u \in \mathcal{PC}^{(\leq k)}$*,* $u \neq \$$ *a* $v \in \mathcal{PC}^{(\leq k-1)} : |v| < |u|$*. Pokud je* M *ve stavu* q*, je možné slovo* u *ve výhledovém okénku nahradit slovem* v*, přejít do stavu* q' *a posunout hlavu bezprostředně za slovo* v*. Přitom je třeba zajistit, aby ze vstupu nezmizel symbol zarážky (*$^\wedge$ *ani* \$*). Jestliže* v *končí pravou zarážkou* \$*, tak se hlava posune pouze na pravou zarážku.*

3. *Restart je ve tvaru* $\mathsf{Restart} \in \delta(q, u)$*, kde* $q \in Q$ *a* $u \in \mathcal{PC}^{(\leq k)}$*. V takovém případě je hlava přenesena na začátek (tzn. tak, aby první symbol ve výhledu byl* $^\wedge$*) a stav je změněn na* q_0*.*

[3] V publikacích se obvykle pro levou zarážku používá znak ¢ . Zde je použit znak $^\wedge$, pro sjednocení symboliky s tou používanou v regulárních výrazech, což je, s ohledem na následující text, vhodné.

4. *Accept je ve tvaru* Accept $\in \delta(q, u)$, *kde* $q \in Q$ *a* $u \in \mathcal{PC}^{(\leq k)}$. *V ten okamžik* M *končí výpočet a vstup je přijat.*

Nejprve je stroj v počáteční konfiguraci (tj. ve stavu q_0 *a na pásce se nachází vstup). Následuje sekvence posunů hlavy doprava a (nedeterministické) hledání vhodného místa pro přepis. Po přepisu může hlava ještě zkontrolovat zbytek pásky, ale aby mohl výpočet pokračovat, musí se následně provést restart. Po něm automat začíná opět hledat místo pro přepis a postup se opakuje. Celý výpočet probíhá v takovýchto cyklech (od restartu do restartu). Aby bylo slovo přijato, musí existovat výpočet začínající v počáteční konfiguraci a končící přechodem* Accept. *Lze snadno nahlédnout, že výpočet skončí vždy po lineárním počtu cyklů (od délky slova), protože v každém cyklu je slovo zkráceno o 1 až* k *symbolů. Pokud se automat nachází ve stavu* q *a obsah okénka je* u *a* $\delta(q, u) = \emptyset$*, vstup není přijat.*

PŘÍKLAD:

Mějme jazyk $L = \{a^m b^n, a^n : m \geq n \geq 0\}$ nad abecedou $\Sigma = \{a, b\}$. Automat M_L rozpoznávající L můžeme definovat následovně:

$$M_L = (\{q_0, q_R\}, \Sigma, \Sigma, ^\wedge, \$, q_0, 4, \delta).$$

$$\begin{array}{lcllcl}
\delta(q_0, ^\wedge \$) & = & \{\mathsf{Accept}\}, & \delta(q_0, aaaa) & = & \{(q_0, \mathsf{MVR})\}, \\
\delta(q_0, ^\wedge a\$) & = & \{(q_R, ^\wedge \$)\}, & \delta(q_0, aaab) & = & \{(q_0, \mathsf{MVR})\}, \\
\delta(q_0, ^\wedge ab\$) & = & \{(q_R, ^\wedge \$)\}, & \delta(q_0, aabb) & = & \{(q_R, ab)\}, \\
\delta(q_0, ^\wedge aa\$) & = & \{(q_R, ^\wedge a\$)\}, & \delta(q_0, aaa\$) & = & \{(q_R, aa\$)\}, \\
\delta(q_0, ^\wedge aaa) & = & \{(q_0, \mathsf{MVR})\}, & \delta(q_R, \$) & = & \{\mathsf{Restart}\}, \\
\delta(q_0, ^\wedge aab) & = & \{(q_0, \mathsf{MVR})\}, & \delta(q_R, u) & = & \{(q_R, \mathsf{MVR})\}.
\end{array}$$

Symbol u použitý jako obsah pásky v poslední rovnosti znamená: “Všechny možné obsahy, které nejsou pokryty předchozími případy.”

2.2.2 Speciální automaty

Kromě základní, v předchozím oddíle definované, verze restartovacích automatů je možné se v literatuře setkat s celou řadou modifikací. Hlavní motivací pro zavádění a zkoumání důsledků těchto omezení je silná nedeterminističnost a tedy i vysoká časová složitost (NTIME(n^2)) základní verze. Existují však i modifikace motivované snahou o dosažení maximální možné výrazové síly. Jejich časová složitost je logicky mnohy ještě vyšší.

Definice 2.2 *Restartovací automat je deterministický (označuje se prefixem* det-*), pokud je jeho přechodová relace* δ *funkce.*

Definice 2.3 *Řada omezení* RRWW-*automatu se vyjadřuje kombinací následujících dvou typů:*

- *Omezení pohybu hlavy (vyjádřeno první částí jména třídy):* RR- *znamená bez omezení a* R- *znamená, že každý přepis je bezprostředně následován restartem.*
- *Omezení instrukcí přepisu (vyjádřeno druhou částí jména třídy):* -WW *znamená opět bez omezení,* -W *znamená, že nejsou dovoleny žádné pomocné symboly ($\Gamma = \Sigma$) a absence druhé části jména třídy označuje, že každý krok přepisu je pouhým vymazáním některých (ne nutně po sobě jdoucích) symbolů z aktuálního obsahu výhledového okénka.*

Definice 2.4 RRWW-*automat se nazývá zmenšující (shrinking), pokud existuje váhová funkce $w : \Gamma \to \mathbb{R}^+$ taková, že pro každý přepis ve tvaru $(q', v) \in \delta(q, u)$ platí, namísto původní podmínky $|v| < |u|$ její zobecnění $\sum_{c \in v} w(c) < \sum_{c \in u} w(c)$.*

2.2.3 Ekvivalentní definice

Následující lemma je sice relativně slabé, ale pro naše potřeby velmi důležité. Uvádíme ho zde bez důkazu. Ten je však možné provést standardními prostředky teorie automatů [2].

Lemma 2.5 *[2] Pro každý* RRWW-*automat M existuje takový s ním ekvivalentní* RRWW-*automat M', že*

$$\mathsf{Accept}, \mathsf{Restart} \in \delta_{M'}(q, u) \Rightarrow u = u'\$, \text{ kde } u' \in \Gamma^*.$$

To znamená, že každý RRWW-automat je možné upravit tak, aby se Restart a Accept prováděli pouze v tom případě, že už automat viděl celé slovo na pásce. Na základě tohoto lemmatu lze každý cyklus M' rozložit na tři fáze:

1. sekvence posunů doprava,
2. jedna instrukce přepisu,
3. sekvence posunů doprava (až k pravé zarážce) zakončená restartem.

První a třetí fáze v podstatě představuje kontrolu regulárního jazyka. To umožňuje zavést zápis pomocí tzv. meta-instrukcí. Díky tomuto formalismu, můžeme restartovací automat zadefinovat ekvivalentně jako přepisovací systém a to následujícím způsobem:

Definice 2.6 *Restartovací automat je systém $M = (\Sigma, \Gamma, I)$, kde Σ je vstupní abeceda, Γ je pracovní abeceda obsahující Σ a I je konečná množina meta-instrukcí následujících dvou typů:*

1. *Meta-instrukce přepisu je ve tvaru $(R_1, x \to y, R_2)$, kde $x, y \in \Gamma^*$ takové, že $|x| > |y|$ a $R_1, R_2 \subset \Gamma^*$ jsou regulární jazyky nazývané levý a pravý kontext.*

2. *Přijímací meta-instrukce má tvar (R, Accept), kde $R \subset \Gamma^*$ je regulární jazyk.*

Vstupní slovo w může být přepsáno na slovo w', pokud existuje meta-instrukce $(R_1, x \to y, R_2)$ taková, že $w = uxv$, kde $u \in R_1$, $v \in R_2$ a $w' = uyv$ – píšeme $uxv \vdash uyv$.

Označme $\vdash^*$ reflexivní a tranzitivní uzávěr $\vdash$. Pak vstupní slovo w je přijímáno automatem pokud existuje slovo z takové, že $w \vdash^* z$ a $z \in R$ pro nějakou přijímací meta-instrukci (R, Accept).

V této definici vystupují regulární jazyky R_1, R_2 a R, které je potřeba zapsat. Obvykle se k těmto účelům používají regulární výrazy, které jsou podrobně rozebrány v oddíle 2.4).

Příklad:

Ukažme si tento zápis na stejném jazyce, jaký byl použit v předchozím příkladě ($L = \{a^m b^n, a^n : m \geq n \geq 0\}$), abychom je mohli snadno porovnat.

$$M'_L = (\{a, b\}, \{a, b\}, I),$$

$$I = \{ \quad (a^*, \mathsf{Accept}), \quad (ab, \mathsf{Accept}), \quad (a^*, aabb \to ab, b^*) \quad \}.$$

Na rozdíl od automatu M_L, automat M'_L přijímá vstupní slovo ab bez toho, aby ho přepisoval na prázdné slovo. A vstupní slova a^n dokáže přijímat dokonce bez jediného přepisu (a tedy i restartu).

Je evidentní, že zápis pomocí meta-instrukcí mnohem čitelnější. Přináší ovšem i jednu nevýhodu. Automat zapsaný pomocí meta-instrukcí je vždy nedeterministický, což znamená, že jeho výpočet může trvat výrazně déle, než je nutné. Výhoda čitelnějšího zápisu je však (obzvláště při návrhu) dominující.

2.3 Restartovací automat se sémantikou

Jak již bylo řečeno, restartovací automaty jsou lingvisticky motivovaný nástroj na redukční analýzu. Následující příklad: "Petr má modré oči." lze redukovat na "Petr má oči." Co očekáváme od sémantiky, je to, že si udělá někam poznámku, že ke slovu (tokenu) "oči" patří atribut s hodnotou "modrá". Nejpřirozenější a nejpřehlednější způsob, jak to udělat, je mít u každého tokenu množinu příznaků, jejichž hodnoty mohou být během výpočtu měněny. Uvědomme si však, že meta-instrukce má za úkol najít místo, kterému rozumí a to zjednodušit. Ve skutečnosti i kdybychom při realizaci použili původní definici se stavovým automatem, nemělo by smysl dělat sémantickou analýzu nějakého úseku vstupu dříve než syntaktickou. Tedy jediný okamžik, kdy má smysl měnit hodnoty příznaků u jednotlivých tokenů, je během přepisu.

Jak lze snadno nahlédnout, aby měl tento model nějakou vyjadřovací sílu, je potřeba umět nastavovat tokenu příznaky na základě hodnot příznaků u ostatních tokenů. To ovšem může přinést řadu problémů, které však mají společnou příčinu. Máme-li meta-instrukci $(R_1, u \rightarrow v, R_2)$, tak je z celého obsahu pásky znám jen úsek odpovídající podslovu u. O jeho předponě a příponě víme pouze tolik, že patří do regulárních jazyků R_1 a R_2. Pokud bychom tyto reprezentovali pomocí regulárních výrazů, tak jak je známe z jazyka `Perl` (či utility `sed`), mohli bychom využít již vybudovaný koncept zpětných referencí. (Každému výrazu uzavřenému do složených závorek je přiřazen manipulátor – pořadové číslo příslušné levé závorky. Pomocí tohoto manipulátoru je možné se na obsah dané závorky odkazovat). To ovšem neřeší problém zcela, protože v okamžiku, kdy se pokusíme použít takovýmto způsobem odkaz na token v části regulárního výrazu, která se může opakovat, není jasné, kolikátý výskyt se má použít a různé výskyty téhož tokenu mohou mít naprosto rozdílný význam. Takovýto problém ale může nastat už na syntaktické úrovni (např. u regulárního výrazu `([ab]c)*` není pro vstup `acbc` jednoznačné, jestli na dotaz na obsah závorky má být odpověď `ac`, nebo `bc`. To některé implementace řeší tak, že vrací seznam všech výskytů. Takové řešení už je funkční, navíc složité je jen pro složité regulární výrazy, pokud se nejedná o repetici alternativ (např. `(a|[bc])*`, ale i `.*`), tak není potřeba ověřovat typ tokenu, pokud požadovaný token není ani v repetici, pak není třeba ani testovat kolikrát byl rozpoznán.

Jedná se o velmi silný koncept. Vrátíme-li se však opět k základním vlastnostem meta-instrukcí, tak jedna z nich bude určitě lokálnost. Jazyky R_1 a R_2 slouží pouze pro usazení pravidla do kontextu, ale vlastní zjednodušování vstupu – přepis – je prováděno pouze pomocí toho, co

je obsažené v podslově u. A tak nejlepším řešením je zakázat manipulaci s příznaky tokenů mimo u (a v). (Navíc tokeny v u jsou následně odstraněny a tokeny ve v jsou nově vytvořené, takže můžeme říci, že ty v u jsou určeny pouze ke čtení a ty ve v pouze pro zápis.) Důvod volby tohoto přístupu je ještě jeden – velmi dobře se totiž shoduje s přístupem používaným v nástroji `yacc`.

Co ovšem dělat, pokud máme meta-instrukci, která provádí čisté mazání? (Tj. v je prázdné slovo.) V takovém případě nemáme kam případnou sémantickou informaci uložit. Možnosti jsou v zásadě dvě. Buď potřebné informace můžeme ukládat do nějaké globální proměnné, nebo se takovým meta-instrukcím vyhneme. Při použití globální proměnné je nutná velká obezřetnost s ohledem na potenciální back-tracking. Tento problém je ale ve skutečnosti lehce umělý. Pokud se totiž nejedná o zjednodušování, ale opravdu o odstraňování, nemělo by být potřeba žádnou sémantickou hodnotu ukládat. Chceme-li například rozpoznávat jazyk $\{a^n b^n, n \in N\}$, je výhodné použít meta-instrukci $(a^+, ab \rightarrow, b^+)$. Považujeme-li za sémantickou hodnotu daného slova hodnotu n, potře-bujeme ji někde mít uloženu. V takovém případě je vhodné upravit uvedenou meta-instrukci do podoby $(a^+, abb \rightarrow b, b^*)$. Nyní již není problém n počítat jako sémantickou hodnotu symbolu b po přepisu.

Řešení je to sice nejčistší, ale vytváří další komplikaci. Dosud jsme se zabývali pouze meta-instrukcemi pro přepis, co ale ty pro přijetí – (R, Accept)? Zde také vystupuje regulární jazyk a co hůř, zde je naopak nutné s ním pracovat i na sémantické úrovni. Variant je opět několik. První z nich je samozřejmě ta před chvílí zavržená. K té se ale vracet nechceme.

Zamysleme se nad tím, co by vlastně mělo být výsledkem sémantické analýzy? Mohl by to být – tak jak jsme zvyklí – jeden token ("počáteční neterminál") a jeho sémantika by byla sémantika celého vstupu. My zde ale nemáme žádný "počáteční neterminál", syntaktická analýza skončí v okamžiku, kdy obsah pásky patří do nějakého regulárního jazyka. Takový vstup může být ale potenciálně libovolně dlouhý. Představa, že by bylo možné něco takového v jednom kroku sémanticky poskládat do jednoho tokenu, je dosti nereálná.

Zkusme tedy najít kompromis. Zaveďme skutečně nový pomocný symbol, který bude fungovat tak, že v okamžiku, kdy se objeví na pásce, výpočet končí a vstup je přijat. Zásadní rozdíl je ale v tom, že nebudeme vyžadovat, aby to byl po skončení výpočtu jediný symbol na pásce. V ten okamžik můžeme zrušit původní přijímající meta-instrukce, protože máme ekvivalentní nástroj (což lze snadno nahlédnout). Pokud tedy byla

analýza vstupu úspěšná, je jejím výsledkem posloupnost tokenů, v níž každý může mít svou sémantickou hodnotu.

2.4 Notace pro regulární jazyky

V oddíle 2.2.3 jsme si zadefinovali restartovací automat pomocí metainstrukcí (definice 2.6). Ty obsahují zápis regulárních jazyků. K tomu účelu bylo v historii zavedeno mnoho způsobů, jejichž stručný přehled je obsahem této podkapitoly. Za tímto přehledem je uveden popis zápisu použitý v CCRA.

2.4.1 SNOBOL [6]

Za prvním pokus o práci s regulárními jazyky lze označit rozpoznávání vzorů (*pattern matching*) v jazyce SNOBOL, který využíval notaci označovanou jako regulární množiny (*regular sets*).

SNOBOL (StriNg Oriented symBOlic Language) je programovací jazyk vytvořený v 60. letech pro manipulaci s textovými řetězci. Hlavní motivací jeho vzniku bylo uvědomění si faktu, že na různé problémy jsou vhodné různé nástroje a že se tato skutečnost týká i programovacích jazyků.

Základní operace SNOBOL-u jsou tvoření řetězců, rozpoznávání vzorů a nahrazování. Umožňuje i základní aritmetické operace, ty jsou ale prováděny také nad textovými řetězci.

Vytvoření řetězce

Proměnná obsahující textový řetězec se vytvoří obvyklým způsobem, tedy příkaz `LINE.1 = "AROUND THE SUN"` dosadí do proměnné `LINE.1` text `AROUND THE SUN`. Víceřádkový text lze pak vytvořit pomocí lomítka (`"/"`) jakožto oddělovače řádků, tedy například:

```
TEXT = LINE.1 "/" LINE.2
```

Zde je i vidět, jakým způsobem je realizováno zřetězení, totiž oddělením alespoň jednou mezerou.

Rozpoznávání vzorů

Rozpoznávání vzorů je základním rysem jazyka SNOBOL. V nejjednodušší podobě můžeme psát např. `LINE.1 "ROUND"` pro test, zda proměnná `LINE.1` obsahuje podřetězec `"ROUND"`. Takovýto test má však

jasné limity. Může nás zajímat (a v praxi často také zajímá) nejen to, zda je obsažen podřetězec, ale navíc zda je (ne nutně bezprostředně) následován jiným podřetězcem. I takovýto test je možné realizovat, a to v následující podobě. Kód `LINE.1 "ROUND" *VAR* "SUN"` nejenže otestuje, zda je v proměnné `LINE.1` obsaženo podslovo `"ROUND"` následované podslovem `"SUN"`, ale navíc je případný podřetězec vyskytující se mezi nimi uložen do nové proměnné `VAR`.

Další vyjadřovací síla (jdoucí nad rámec regulárních jazyků) je ukryta ve zpětné referenci. Výše popsaným způsobem nově vytvořenou proměnnou `VAR` je totiž možné využít později v témže výrazu. Např. pravidlo `TEXT "(" *X* ")" *Y* "(" X ")"` bude pro hodnotu proměnné `TEXT = "(C,D)(A,B)(D,C)(A,B)"` aplikováno úspěšně a do proměnné `X` bude dosazen řetězec `"A,B"`.

Jazyk dále obsahuje podporu pro kontrolu uzávorkování, která je často potřeba u algebraických struktur. Chceme-li, aby proměnná `A` obsahovala správně uzávorkovaný výraz (zdůrazněme zde, že uložený v řetězci), uzavře se nejen mezi hvězdičky, ale předtím do závorek (tedy `*(A)*`).

Poslední podmínka, kterou je možné pomocí vzorů popsat, je fixní délka řetězce, která se vyjádří hodnotou uvedenou za jménem proměnné odděleným lomítkem. Tedy má-li mít proměnná `PAD` délku 3 znaky, zapíše se do vzoru jako `*PAD/"3"*`.

Rozpoznávací (*matching*) algoritmus

V některých případech může být přinejmenším sporné, jak by se měl testovací algoritmus zachovat. Pro takový případ existují následující pravidla:

1. Hledání probíhá zleva doprava.

2. Pro každý prvek se hledá nejkratší možná alternativa splňující podmínku.

3. Pokud pro nějaký prvek není možné stanovené podmínky splnit, hledá se nové řešení pro prvek předchozí.

4. Pokud je poslední prvek vzoru nová proměnná (neomezená ani na délku ani na závorky), je rozšířena až do konce řetězce.

5. Test je úspěšný, pokud se podaří splnit omezení posledního prvku, a neúspěšný, pokud se nepodaří splnit omezení na prvek první.

2.4.2 QED [7]

QED je řádkově orientovaný textový editor, jehož vznik se datuje k roku 1965. Jeho autoři si všimli časté potřeby uživatelů upravit text dokumentu na mnoha (v jistém smyslu podobných) místech stejným způsobem. Začlenili tedy do svého editoru nástroj inspirovaný regulárními množinami. Poprvé se tak objevují regulární výrazy.

Regulární výrazy

Regulární výrazy jsou definovány rekurzivně následujícím způsobem. (Označení výraz zde znamená regulární výraz.)

- Běžný znak je regulární výraz odpovídající tomuto znaku.
- `^` – znak `null` na začátku řádku.
- `$` – znak `null` před znakem `<NL>` (*newline* – znak konce řádku).
- `.` – libovolný znak kromě `<NL>`.
- `[`*string*`]` – libovolný znak v *string* a žádný jiný.
- `[^`*string*`]` – libovolný znak kromě `<NL>` a těch obsažených v *string*.
- *RegExp*`*` – libovolný počet (včetně 0) opakování výrazu *RegExp*.
- *RegExp1 RegExp2* – regulární výraz odpovídající sousednímu výskytu výrazů *RegExp1* a *RegExp2* v textu.
- *RegExp1* `|` *RegExp2* – regulární výraz pro text odpovídající alespoň jednomu z výrazů *RegExp1* a *RegExp2*.
- `(`*RegExp*`)` – zápis slouží pouze k úpravě priority při zpracovávání.
- {*RegExp*}*x* – regulární výraz říkající, že pokud byl rozpoznán text výrazem *RegExp*, má se tento text uložit do bufferu *x* (kde *x* je libovolný znak).
- \`E`*rexname*`"` – regulární výraz odpovídající výrazu dříve uloženému pod názvem *rexname*.
- Prázdný regulární výraz odpovídá výrazu naposledy použitému.
- Nic jiného není regulární výraz.

- Žádný regulární výraz nemůže odpovídat textu rozdělenému do více řádek.

2.4.3 Perl [8]

Od vzniku editoru QED bylo implementováno nepřeberné množství nejrůznějších rozšíření původní koncepce regulárních výrazů. V současné době je však vývoj velmi pozvolný (není totiž snadné změnit něco, co používá veliké množství lidí) a pomyslným etalonem se stala implementace začleněná v programovacím jazyku Perl. Odtud ji převzala řada dalších programovacích jazyků (Java, Python, PHP, Ruby atd.).

Základní syntax původních regulárních výrazů je v Perlu zachována (konkrétně význam konstrukcí ^, `$`, `.`, `[`*str*`]`, `[^`*str*`]`, `*`, `|` a `(`*RegExp*`)`). Přidává však různé další konstrukce zjednodušující zápis.

Opakování

Počet opakování lze určovat nejen symbolem `*` za daným výrazem, ale těmito způsoby:

- `*` – libovolný počet opakování (včetně nuly).
- `+` – alespoň jeden výskyt.
- `?` – žádný, nebo jeden výskyt.
- {*n*} – přesně *n* výskytů.
- {*n*`,`} – alespoň *n* výskytů.
- {`,`*m*} – žádný až *m* výskytů.
- {*n*`,`*m*} – alespoň *n* výskytů, ale nejvýše *m*.

Kvantifikátory pracují za normálních okolností hladově, tzn. že při rozpoznávání vstupu “aaa” výrazem `a*a*` budou použita 3 “a” a následně 0. V některých složitějších konstrukcích však může být výhodné opačné chování, totiž rozpoznávat nejmenší možný počet (tedy v předchozím příkladě nejprve 0, a poté 3). Toho lze dosáhnout tak, že se za příslušný kvantifikátor napíše otazník (tedy např. `a*?a*?`).

Za normálních okolností, pokud se Perlu nepodaří rozpoznat zbytek výrazu, začne prohledávat do hloubky. To může být velmi nepříjemné. Lze tak dosáhnout až exponenciální časové složitosti – např. pro výraz `((a{0,5}){0,5})*[c]` se při vstupu “aaaaaaaaaaa” otestují všechny

kombinace délek než je vstup odmítnut. Navíc může být v mnoha případech takové hledání evidentně zbytečné, a tak je umožněno toto prohledávání zakázat připsáním `+` za příslušný kvantifikátor.

Třídy znaků

Aby nebylo nutné neustálým vypisováním velmi často používaných skupin znaků regulární výraz znepřehledňovat (např. výraz pro malé písmeno české abecedy – `[a-zěščřžýáíéťďóňúů]`), definuje Perl (mimo jiné) následující zkratky:

- `\w` – znak identifikátoru (číslice, písmena a podtržítko),
- `\s` – bílý znak (mezera, tabulátor),
- `\d` – číslice.

Pokud je použito velké písmeno namísto malého, znamená to doplněk původní množiny.

Je také možné použít POSIXový zápis `[:`*class*`:]`, kde *class* může nabývat následujících hodnot:

- **alpha** – písmeno,
- **alnum** – alfanumerický znak,
- **ascii** – znak s ordinálním číslem od 0 do 127,
- **blank** – mezera nebo tabulátor,
- **cntrl** – řídící znak (např. `<NL>`, `<BS>`, `<DEL>`, apod.),
- **digit** – číslice,
- **graph** – alfanumerický nebo interpunkční znak,
- **lower** – malé písmeno,
- **print** – alfanumerický, interpunkční nebo bílý znak,
- **punct** – interpunkční znak,
- **space** – bílý znak,
- **upper** – velké písmeno,
- **word** – znak identifikátoru,
- **xdigit** – hexadecimální číslice (tedy `[0-9A-Fa-f]`).

Seskupování

Seskupování výrazů se provádí pomocí uzavření do kulatých závorek. Obvykle je použito za účelem úpravy priority operátorů. Má však ještě jedno využití. Na část vstupu, která je rozpoznána takto uzávorkovaným podvýrazem, je možné se odkazovat pomocí výrazu \n, kde n je pořadové číslo levé závorky v rámci daného regulárního výrazu.

V případě, že je výraz složitý a závorek je mnoho, je možné jejich číslování zpřehlednit tak, že se u skupiny, jejichž očíslování není žádoucí neuvede levá závorka a namísto ní se použije `?:`.

Kotvy

Kotvy jsou nástroj, který umožňuje ukazovat na místa v textovém řetězci, ale bez toho, aby byla zahrnuta. To je výhodné ve dvou případech – pokud se nejedná o žádný konkrétní znak (jako například konec řádky) a pokud by jeho zahrnutí způsobovalo problémy (např. kvůli opakováním či zpětným referencím).

Základními typy kotev jsou znaky ^ a `$` pro začátek a konec řádky. Perl ovšem podporuje ještě další kotvy a sice:

- `\<` – pro začátek slova,
- `\>` – pro konec slova,
- `\b` – pro hranici (začátek nebo konec) slova,
- `\B` – pro cokoliv kromě hranice slova,
- `\A` – pro začátek vstupu,
- `\Z` – pro konec vstupu nebo místo před znakem `<NL>` na konci vstupu a
- `\z` – pro konec vstupu.

Modifikátory

Modifikátory jsou nástroj umožňující změnit chování jednotlivých operátorů v regulárním výrazu. Většinou se používají mimo regulární výrazy, ale někdy je možné je do nich přímo vepsat (buď `(?`*modifikátor*`)` pro zapnutí nebo `(?-`*modifikátor*`)` pro vypnutí).

Nejdůležitější modifikátory jsou tyto:

- **m** – Umožňuje pracovat s řetězci přes hranici řádky (kotvy ^ a `$` pak mohou být použity i uvnitř výrazu).
- **s** – Označí vstup jako jednořádkový (`.` pak může odpovídat i znaku `<NL>`).
- **i** – Vypíná rozlišování malých a velkých písmen.

2.4.4 CCRA

Dá se říci, že pokud dnes aplikace pracuje s regulárními výrazy, velmi pravděpodobně je použitá syntax podmnožinu syntaxe regulárních výrazů použité v jazyce Perl. Bohužel v námi vytvářeném generátoru kompilátorů toto není možné. Hlavní důvod je ten, že CCRA, stejně jako jiné generátory parserů, umožňuje předřazení lexikálního analyzátoru – scanneru. To ale znamená, že regulární výrazy nepracují pouze se znaky, ale mohou pracovat i s názvy symbolů, čemuž je nutné uzpůsobit jejich zápis.

Bylo by sice možné vyjít ze zápisu regulárních množin v jazyce SNOBOL. Ten ale v dnešní době není vůbec rozšířený. Navíc je výrazová síla regulárních množin velmi omezená. Chceme-li však při návrhu vyjít ze syntaxe regulárních výrazů jazyka Perl, je vhodné odstranit všechny výrazové prostředky, jdoucí nad rámec regulárních jazyků. Tedy ku příkladu zpětnou referenci, ta totiž umožňuje rozpoznávat mimo jiné neregulární jazyk $\{u.u : u \in \Sigma^*\}$).

Podporovány jsou následující konstrukce regulárních výrazů:

- *id* – regulární výraz odpovídající symbolu s názvem *id*,
- `"`*str*`"` – regulární výraz odpovídající posloupnosti znaků *str*,
- ^ – levá zarážka, tedy začátek vstupu,
- `$` – pravá zarážka, tedy konec vstupu,
- `.` – libovolný znak,
- `[`*str*`]` – libovolný znak obsažených v *str*,
- `[^`*str*`]` – libovolný znak neobsažený v *str*,
- *RegExp1 RegExp2* – regulární výraz odpovídající sousednímu výskytu výrazů *RegExp1* a *RegExp2*,

- *RegExp1* `|` *RegExp2* – regulární výraz pro vstup odpovídající alespoň jednomu z výrazů *RegExp1* a *RegExp2*,
- `(`*RegExp*`)` – regulární jazyk *RegExp* (pouze k úpravě priority),
- *RegExp*{m`,`n} – regulární výraz odpovídající m až n opakování výrazu *RegExp*; neuvedení m znamená jeho hodnotu 0, neuvedení n znamená jeho hodnotu nekonečno,
- *RegExp*{n} – ekvivalent zápisu *RegExp*{n`,`n},
- *RegExp*`*` – ekvivalent zápisu *RegExp*{*0*`,`},
- *RegExp*`+` – ekvivalent zápisu *RegExp*{*1*`,`},
- *RegExp*`?` – ekvivalent zápisu *RegExp*{`0,1`}.

Kapitola 3

Popis implementace

V předchozí kapitole byla popsána teoretická východiska této práce. Při implementaci sebelépe popsaného teoretického modelu je nutné činit velké množství implementačních rozhodnutí. Právě těmto rozhodnutím se věnuje tato kapitola.

3.1 Volba jazyka

První rozhodnutí, které bylo nutné učinit, byla volba vhodného programovacího jazyka. A to navíc hned dvakrát. Jazyk, ve kterém bude napsán nástroj CCRA, a jazyk použitý generovanými parsery. S ohledem na maximální možnou přenositelnost a rychlost výsledného programu padla volba na jazyk C++. Jako jazyk vygenerovaného kódu byl sice zvažován i jazyk C. Ten však byl nakonec zavrhnut, protože výhoda možnosti zapouzdřovat kód v C++ byla shledána jako mnohem užitečnější, než výhoda použít vygenerovaný parser i na jednočipu (což byla jediná uvažovaná výhoda jazyka C). Mimo jiné i z toho důvodu, že časová složitost použitých algoritmů takové použití minimálně velmi komplikuje, pokud přímo neznemožňuje.

3.2 Formát vstupního souboru

Jak již bylo řečeno, výsledkem této práce má být generátor kompilátorů – CCRA. V předchozí kapitole byly představeny základní vlastnosti programu `yacc`, které by bylo dobré zahrnout do vyvíjeného generátoru kompilátorů, protože používání tohoto nástroje je velmi rozšířené a je žádoucí, aby bylo učení se nového nástroje pro uživatele co nejjednodušší. Mimo jiné to také umožňuje snadno (např. skriptem) převádět vstupní

soubor pro yacc na vstupní soubor pro CCRA. Další výhodou tohoto přístupu je, že implementace stejného rozhraní jaké nabízí yacc zajišťuje podporu pro další nástroje (např. program flex – generátor lexikálních analyzátorů). Některé věci bylo však samozřejmě nutné upravit.

Kromě takových drobností, jako že bude použita jiná množina přepínačů, bylo samozřejmě nutné změnit hlavně formu zápisu gramatických pravidel. Zápis pravidel v nástroji yacc totiž vychází z BNF (Backus-Naurovy formy), která je ovšem pro zápis meta-instrukcí zcela nevhodná. I zde však byla vyvinuta snaha o dosažení maximální podobnosti.

Příklad gramatického pravidla zapsaný ve formátu pro nástroj yacc vypadá následovně:

```
vyraz: vyraz "+" vyraz { $$=$1+$2; };
```

Příklad zápisu gramatického pravidla pro nástroj CCRA odpovídající meta-instrukci $([ab]^*, a\,S\,a \rightarrow S, [ab]^*)$ vypadá takto:

```
^[ab]* / "a" S "a" -> S / [ab]*$ @{ $$1=$2+"a"; };
```

Zápis regulárních výrazů byl navržen tak, aby bylo možné ho používat ve vstupním souboru v nezměněné podobě. Nahrazení šipky dvojicí symbolů “->” je relativně intuitivní. Změna oddělovače levého a pravého kontextu od popisu přepisu z čárky na lomítko sice nebyla nutná, byla ale učiněna s ohledem na vyšší přehlednost. Čárka by se totiž mnohdy díky složitosti zápisu regulárního výrazu velmi obtížně hledala. Z tohoto důvodu bylo také umožněno od sebe jednotlivé části oddělovat libovolnou sekvencí bílých znaků (tedy i odřádkováním) a komentářů (ve formátu C++). Poslední úprava zápisu syntaxe meta-instrukcí se týká práce se zarážkami. V definici zápisu kontextů se totiž uvažuje to, že celá část vstupu před místem přepisu má ležet v popsaném jazyce. V praxi ovšem velmi často záleží pouze, pokud vůbec, jen na malém okolí místa přepisu, v důsledku čehož všechny meta-instrukce začínají i končí výrazem “.*”. Proto byla zavedena konvence, že je možné tento výraz vynechat, a pokud jeho přidání není žádoucí, napíše se před/za výraz značka levé/pravé zarážky (^ / $).

Co se ostatního formátování týče, bylo přejato možné maximum z formátu programu yacc. Tedy jednotlivá pravidla jsou ukončena středníkem a sémantická akce, pokud je definována, je uzavřena do složených závorek. Bohužel je složená závorka součástí syntaxe regulárních výrazů, což LA-LR(1) parseru použitému ke zpracování souboru s definicí automatu znemožňuje rozhodnout, jedná-li se o začátek sémantického bloku nebo o součást pravého kontextu. Z toho důvodu byl formát upraven tak, že

pokud má meta-instrukce definovanou sémantickou akci, je tato, kromě uzavření do složených závorek, oddělena od pravého kontextu znakem ‘@’.

3.3 Reprezentace automatů

Formální popis restartovacího automatu je jedna věc, ale jeho implementace může vypadat úplně jinak. Základní definice RRWW-atomatu (definice 2.1, str. 14) nabízí reprezentaci pomocí jednoho nedeterministického konečného automatu. Toto řešení má tu ohromnou výhodu, že zpracování vstupu probíhá velmi rychle. Seznam všech možných přepisů daného vstupu lze totiž získat jediným průchodem automatu přes něj. V CCRA je ovšem automat popsán pomocí meta-instrukcí. Každou meta-instrukci $(R_1, u \to v, R_2)$ je možné reprezentovat dvěma nedeterministickými konečnými automaty popisujícími jazyky R_1 a R_2 a dvěma sekvencemi symbolů u a v.

Pokud bychom chtěli reprezentovat automat popsaný pomocí meta-instrukcí jedním nedeterministickým automatem, vypadal by převod tak, že nejprve by se na takový automat převedla každá meta-instrukce zvlášť (až na drobné komplikace s místem přepisu pouhým zřetězením jazyků R_1, $\{u\}$ a R_2). Všechny takto vytvořené automaty dohromady tvoří hledaný nedeterministický automat. Jeho práci pak můžeme simulovat tak, že si udržujeme množinu stavů, ve kterých se automat může v daném okamžiku nacházet. Této množině budeme dále říkat aktivní stavy.

Takovýmto převodem jsme však moc nezískali, pouštíme sice pouze jeden automat, ten má ale během výpočtu velmi vysoký počet aktivních stavů. Tento počet je však možné snižovat – automaty odpovídající jednotlivým meta-instrukcím je možné převést na deterministické. Tím dosáhneme toho, že počet aktivních stavů je maximálně roven počtu meta-instrukcí. Totéž je dokonce možné (s drobnými komplikacemi) udělat s výsledným automatem. Tím bychom sice snížili počet aktivních stavů na jeden, ale za cenu potenciálně exponenciálního nárůstu počtu stavů automatu s počtem meta-instrukcí, což může být neúnosné. Pro složitější jazyky totiž může být meta-instrukcí velmi mnoho.

Je tedy nutné použít realizaci s nižšími paměťovými nároky, a to i za cenu nižší rychlosti. V podstatě nejjednodušší varianta by byla nic neupravovat a pamatovat si jednotlivé meta-instrukce tak, jak bylo popsáno, jako dvojici automatů pro jazyky R_1 a R_2 a dvojici seznamů u a v. Při takové reprezentaci by však bylo velmi obtížné hledat místa, kde je možné danou meta-instrukci aplikovat. Bylo tedy zvoleno následující řešení.

Automat M_1 popisující jazyk R_1 je ponechán beze změny. Automat M_2 popisující R_2 je modifikován tak, že popisuje jazyk $R_2^R.\{u\}^R$ (tj. zrcadlový obraz jazyka R_2 zřetězený se zrcadlovým obrazem jazyka $\{u\}$). To umožňuje hledat pozice možné aplikovat meta-instrukce tak, že je celý vstup zpracován automatem M_1, s tím, že se označí každá pozice, na které se automat nacházel v přijímacím stavu. Následně se obdobným způsobem pustí automat M_2, ale na vstupu zprava doleva. Místa, kde je možné danou meta-instrukci aplikovat, jsou pak ta, která byla označena v obou bězích. Tento postup je zopakován s každou meta-instrukcí. Hledaný seznam možných aplikací meta-instrukcí je pak vytvořen sjednocením takto nalezených seznamů.

3.4 Regulární výrazy

Proč je použit vlastní formát zápisu regulárních výrazů, je vysvětleno v oddíle 2.4.4 (str. 26). Navíc je nutné mít možnost exportu výrazu ve vlastním formátu, proto nebylo možné použít žádnou již existující knihovnu na regulární jazyky a bylo tedy nutné napsat svou vlastní.

Protože je potřeba zpracovaný regulární výraz exportovat jako konečný nedeterministický automat, je vhodné ho v této podobě rovnou vytvářet.

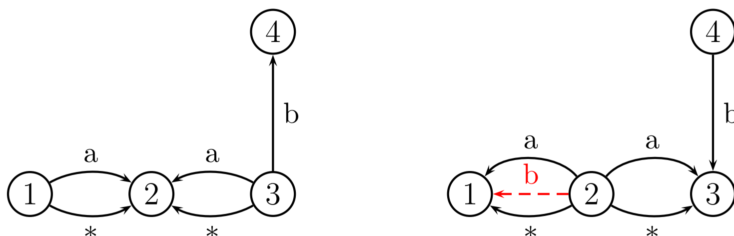
Během implementace se však objevila jedna nepříjemná vlastnost obvyklé reprezentace (kde každý stav popisuje přechody z něj v podobě seznamu dvojic symbol-cílový stav) projevující se obzvlášť u rozsáhlejších abeced. Automat pro regulární výraz “.” vypadá tak, že má jeden vstupní a jeden výstupní stav a mezi nimi vede přechod přes všechny existující symboly.

Z tohoto důvodu byla reprezentace automatu upravena tak, že je možné kromě běžných přechodů přes symbol definovat také přechod přes všechny ostatní symboly. Tato úprava výrazně snižuje paměťové nároky automatu. Navíc díky tomu není nutné, aby uživatel deklaroval na začátku vstupního souboru všechny použité symboly. Je však nutné upravit algoritmy, které automaty upravují (jako zřetězení automatů, otočení automatu, redukce automatu a pod.)

Jako příklad použijme automat na obrázku 3.1 vlevo. Symbol ‘*’ zde značí zmiňovaný přechod přes všechny symboly neuvedené v daném stavu (tzv. *-přechod). Máme-li tedy abecedu $\{a, b, c\}$, tak *-přechod ze stavu 1 do stavu 2 lze použít pro symboly b a c, kdežto *-přechod ze stavu 3 do stavu 2 pouze pro symbol c. Tuto vlastnost *-přechodů chceme zachovat i po změně jejich orientace. V otočeném automatu však ze stavu 2 vedou

kromě *-přechodů už jen přechody přes symbol a. To ale znamená, že oba tyto *-přechody reprezentují přechod přes symboly b i c, což je u přechodu ze stavu 2 do stavu 3 chyba. Je však možné ji snadno opravit přidáním přechodu ze stavu 2 do stavu 1 přes symbol b.

Algoritmus, který tento problém řeší spočívá v tom, že se nejprve určí pro každý takovýto přechod seznam všech vyloučených symbolů. Pokud se při otočení automatu sejde v jednom stavu více *-přechodů, je použit *-přechod pouze pro symboly z průniku těchto seznamů a pro všechny ostatní symboly (tj. symboly, které se nevyskytují ve všech seznamech) jsou definovány další přechody na příslušných místech. Tento postup používající seznamy vyloučených symbolů je použit na všech místech, na kterých je to potřeba.



Obrázek 3.1: Demonstrace problému při otáčení automatu. Původní automat (vlevo) a jeho otočení (vpravo).

Další optimalizace v reprezentaci automatu byla učiněna vzhledem k tomu, že použité algoritmy pro stavbu automatu odpovídajícího regulárnímu výrazu způsobují to, že velmi často nastává situace, kdy existuje více přechodů přes jeden symbol, ale těchto symbolů je málo. Z toho důvodu byla zvolena taková reprezentace, kdy jsou symboly, pro které je definována přechodová funkce, udržovány v kontejneru `map` (typicky realizovaném jako červeno-černý strom) a ke každému takovému symbolu je udržován seznam stavů, do kterých je možné přes něj přejít. Aby bylo možné snadno a rychle zjišťovat, zda se již v seznamu daný cíl nachází, je tento seznam realizován pomocí kontejneru `set` (taktéž typicky realizovaném jako červeno-černý strom).

3.5 Nedeterminismus

Protože restartovací automat je nedeterministický model, je nutné prohledat všechny možné výpočty automatu nad daným vstupem. Na to jak toto prohledávání realizovat existuje celý obor – programování s omezujícími podmínkami. CCRA však používá nejjednodušší možný přístup

– prohledávání do hloubky. Tato volba má však ještě jeden důvod. Často je nutné při určování sémantické hodnoty výrazu pracovat s globálními proměnnými. Pokud se však výpočet nachází ve slepé větvi a začíná backtrackovat, je vhodné a někdy i nutné, aby ony globální proměnné opravil do původního stavu. To se právě při použití prohledávání do hloubky dělá relativně snadno pomocí implementace destruktorů.

Nevýhodou prohledávání do hloubky je to, že může trvat velmi dlouho (v nejhorším případě je nutné projít celý rozhodovací strom jehož výška je u zmenšujícího restartovacího automatu rovna součtu vah symbolů na vstupu). Toto prohledávání je mnohdy možné urychlit tím, že jsou dříve prohledány ty větve výpočtu, které mají větší šanci dosáhnout přijímacího stavu. Proto použitá implementace prohledávání umožňuje uživateli v každém okamžiku určit pořadí možných větví výpočtu, v jakém mají být prozkoumány, případně rovnou některé nabízené větve zamítnout. Tím je umožněno využít specifika dané úlohy ke zrychlení nalezení přijímací větve výpočtu. Pokud uživatel toto rozhodování nechá bez zásahu, je větev výpočtu vybírána tak, že dříve se zkoumá levější místo přepisu (měřeno podle pozice prvního přepisovaného symbolu). Pokud je možné na tomto místě aplikovat více meta-instrukcí, je použita dříve ta, která je ve vstupním souboru definována dříve. Toto chování se nejen dobře představuje, což je dobré při návrhu automatu, ale navíc se velmi blíží chování LALR parseru. Největší výhodou je to, že jsou do maximální možné míry využity již zjednodušené části vstupu, tedy že aplikace meta-instrukcí na sebe skutečně navazují, a tedy je šance rychlého nalezení přijímací posloupnosti aplikací meta-instrukcí.

Dalším problémem, který sebou nedeterminismus přináší, je možná nejednoznačnost výpočtu. Na syntaktické úrovni to ještě nevadí – vstup je přijat, pokud existuje alespoň jedna posloupnost aplikací meta-instrukcí vedoucí k přijetí. Existuje-li tedy druhá taková posloupnost, nic to nemění na tom, že je slovo přijato. Na sémantické úrovni je to ale něco jiného. Může se stát (a v praxi se, bohužel, stává), že každý takový výpočet vede k jiné sémantické hodnotě. Tedy že vstup nemá jednoznačný význam. V mnohých aplikacích sice není nutné získat všechny tyto hodnoty, ale obvykle je vhodné dokázat tuto situaci alespoň detekovat.

V CCRA byl pro tyto potřeby implementován mnohem obecnější nástroj. V sémantické akci je totiž umožněno aplikaci dané meta-instrukce ještě zamítnout (jedná se o tzv. sémantické odmítnutí). To se velmi hodí například u meta-instrukcí typu

$$I_1 = (.*, \textit{delka_seznamu}\ \textit{seznam} \rightarrow \textit{prvky}, .*).$$

Pokud je totiž *seznam* vytvářen pomocí nějaké takovéto meta-instrukce

$$I_2 = (.*, seznam\ prvek \rightarrow seznam, .*),$$

tak je aplikace I_1 v okamžiku, kdy počet symbolů *prvek* v symbolu *seznam* je menší, než deklarovaný počet v symbolu *delka_seznamu*, buď předčasná, nebo se jedná o sémantickou chybu ve vstupu.

Sémantické odmítnutí je možné také využít k nalezení všech možných sémantických hodnot vstupu. Dosáhne se toho tak, že se do každé původně přijímací meta-instrukce přidá tento příkaz. Předtím se však přidá sémantická hodnota do seznamu dosud nalezených sémantických hodnot. Díky tomu je vstup sice nakonec vždy odmítnut, ale v tomto seznamu jsou uloženy sémantické hodnoty všech možných přijímacích výpočtů. K určení, zda byl vstup přijat, tedy neslouží návratová hodnota parseru, ale to, zda je tento seznam sémantických hodnot neprázdný.

3.6 Soubory parseru

Rozložení vygenerovaného parseru do souborů je důležitým koncepčním rozhodnutím. Že je pro kód nutné použít "*.cpp" soubor, je zřejmé, ale jaké další soubory a jestli vůbec nějaké je již třeba zvážit.

3.6.1 Hlavičkový soubor

Pokud bychom celý parser uložili do jediného souboru, bylo by nutné, aby byl tento soubor hlavním souborem celého projektu. Je totiž žádoucí umožnit lexikálnímu analyzátoru používat deklarované symboly, stejně jako nadřazenému kódu (pokud nějaký je) umožnit vygenerovaný parser volat. Z toho důvodu je nutné vytvořit také hlavičkový soubor ("*.h") obsahující právě zmiňované deklarace a v hlavním ("*.cpp") souboru nechat jen implementace jednotlivých metod, případně pomocných funkcí.

3.6.2 Knihovní soubor

Někdy se může stát, že je potřeba v projektu použít více parserů najednou. Je zřejmé, že velká část kódu takových parserů je stejná a může být napsána tak, že vůbec nezávisí na zpracovávaném jazyce. Jedná se hlavně o obecný popis meta-instrukcí (jejich načítání, použité regulární výrazy, hledání míst možné aplikace, apod.). Všechen takový kód, který je možné mezi parsery sdílet je nebezpečný tím, že pokud není uzavřen v každém parseru do unikátního jmenného prostoru (nebo třídy), hrozí

kolize názvů. Navíc je zbytečné, aby byl daný kód v programu skutečně přítomen ve více exemplářích.

Z toho důvodu byl všechen společný kód (u kterého to bylo možné) oddělen od kódu závislého na konkrétním generovaném parseru a byl uložen do souboru “CCRA.hxx”, který je vždy vložen (pomocí direktivy `#include`) do generovaného hlavičkového souboru. Navíc protože všechny funkce (a metody) v tomto souboru pracují s šablonami, není třeba linkovat žádný další soubor.

3.6.3 Soubor přechodových tabulek

Také bylo nutné rozhodnout, zda přechodové tabulky vložit přímo do kódu, nebo je načítat z externího souboru. V programu `bison` jsou obě možnosti, ale významný rozdíl je v tom, že v CCRA nejsou přechodové tabulky pouze rozsáhlá pole, ale složené objekty. Pokud bychom je tedy chtěli mít uložené ve statické konstantě, byl by problém s následným použitím, protože bychom nemohli použít metody, které je modifikují. Je tedy nutné použít statické proměnné (statičnost je výhodná v tom, že umožňuje používat v projektu více kopií parseru najednou a přitom mít přechodové tabulky v paměti pouze jednou). Dalším důvodem pro načítání z externího souboru je skutečnost, že u složitějších jazyků mohou být přechodové tabulky značně rozsáhlé a obzvláště u větších projektů nemusí být úplně žádoucí, aby byly v paměti po celou dobu běhu programu. Zvolené řešení je tedy takové, že v konstruktoru třídy parseru se zkontroluje, zda jsou tabulky již načteny a pokud tomu tak není, tak se načtou.

3.7 Usnadnění návrhu

Protože ladění generátoru zdrojového kódu není vůbec jednoduchá záležitost, byl vytvořen následující systém umožňující jeho maximální podporu.

Nejprve byl napsán projekt představující ukázkový výstup programu (program `Template`). Jedná se o parser velmi jednoduchého jazyka ($L = \{0^*.u.u^R.0^* : u \in \Sigma^*\}$) nad malou abecedou ($\Sigma = \{0, 1, 2\}$). Jako sémantická hodnota vstupu je brána číselná hodnota podslova u. V okamžiku, kdy se zdál být tento parser plně funkční, bylo přistoupeno k tomu, že byly pojmenovány a uzávorkovány pomocí speciální formy komentářů ty části kódu, které jsou závislé na zpracovávaném jazyce. Např. kódy sémantických akcí byly označeny takto:

```
////MACRO: USER_CODE_SEMANTIC_BLOCKS-BEGIN
        ... problémově závislý kód,
        který je třeba smazat ...
////MACRO: USER_CODE_SEMANTIC_BLOCKS-END
```

Dalším krokem bylo napsání programu (`Templater`), který zdrojové soubory takto označkované převede do šablony v takovém formátu, který bude možné za běhu CCRA snadno doplnit. Tento program tedy dělá to, že vymaže kódy mezi komentáři ve výše popsaném tvaru, ale na začátek souboru vypíše seznam jmen těchto oblastí s číslem řádky, na kterou patří vložit. Takto vytvořená šablona výstupu je již součástí instalace CCRA v podobě načítaných souborů.

Díky této automatizaci se dají velmi snadno realizovat zásahy do podoby, v jaké mají být parsery generovány. Změnu je totiž možné implementovat přímo v programu `Template` a v okamžiku, kdy je vše funkční zde, stačí k její propagaci do CCRA jedno spuštění programu `Templater`.

Kapitola 4

Závěr

Podařilo se splnit cíl práce navrhnout a implementovat generátor kompilátorů (CCRA), který je díky tomu, že vychází z modelu restartovacího automatu schopen zpracovávat rozsáhlou třídu jazyků (nadmnožinu jazyků bezkontextových). Touto vlastností se výrazně odlišuje od jiných generátorů kompilátorů, které jsou obvykle schopné generovat kompilátory pouze pro úzkou podmnožinu bezkontextových jazyků. Přes tuto odlišnost se podařilo dosáhnout velké míry podobnosti ve způsobu práce s velmi rozšířeným programem `yacc`. Díky této podobnosti, je naučení se obsluhy nového nástroje velice intuitivní. Jedna z těchto podobností spočívá i v tom, že program generuje kód v jazyce C++, který je možné začlenit i do rozsáhlých projektů. Navíc je možné podobu tímto nástrojem generovaných kompilátorů snadno modifikovat zásahem do šablonových souborů.

Ačkoliv je zvolený model restartovacího automatu velmi silný, má i několik nevýhod. Hlavní z nich je řádově nižší rychlost zpracování vstupu, což je mnohdy nepřekonatelný handicap. Ačkoliv je tedy možné pomocí CCRA realizovat tvorbu kompilátoru pro libovolný bezkontextový jazyk, je takové rozhodnutí pro celou řadu z nich absolutně nevhodné. Do této kategorie se řadí i programovací jazyky, je nemyslitelné zpracovávat restartovacím automatem vstup obsahující desetitisíce řádek kódu, což je u větších softwarových projektů běžný rozsah. Jazyky, pro jejichž zpracování je CCRA vhodný, jsou tedy jazyky, jejichž syntax je složitá, ale velikost vstupu je malá. Velmi dobrým příkladem takového jazyka je české názvosloví organické chemie.

Existuje mnoho funkcí, o které by se mohlo být zajímavé pokusit program rozšířit. Protože, jak bylo řečeno, je největším problémem nástroje rychlost vygenerovaného kompilátoru, směřuje řada z navrhovaných rozšíření k optimalizaci právě tohoto parametru.

Velmi vhodné by bylo rozšířit CCRA o sadu testů určující, jaký nejjednodušší (resp. spíš nejrychlejší) model restartovacího automatu je dostačující použít, a tuto informaci následně zužitkovat při generování kódu.

Dále by bylo možné zrychlit vytváření seznamu míst možných aplikací jednotlivých meta-instrukcí tím, že by se pro každou meta-instrukci ve tvaru $(R_1, u \rightarrow v, R_2)$ udržovala informace o tom, které ze symbolů obsažených v u jsou a které nejsou ve vstupu obsaženy. Pokud některý symbol chybí, bylo by možné meta-instrukci vyřadit ze seznamu meta-instrukcí, pro které je hledán seznam míst možné aplikace, protože žádné takové místo není. Do tohoto seznamu by se vrátila až v okamžiku, kdy jsou všechny chybějící symboly přidány pomocí aplikací jiných meta-instrukcí. Tato úprava by pravděpodobně kompilátor výrazně zrychlila. Většina jazyků totiž obsahuje mnoho pravidel, která umožňují zjednodušit jeden konkrétní vstupní symbol. Pokud se však tento symbol ve vstupu nevyskytuje, je daná meta-instrukce testována po každém přepisu úplně zbytečně.

Jistého zrychlení by také bylo možné dosáhnout zjednodušením hledání možných míst aplikace těch meta-instrukcí, které mají prázdný levý nebo pravý kontext. V ten okamžik totiž toto hledání degeneruje na hledání podřetězce, ke kterému jsou ale použity dva průchody přes vstup (přičemž jeden nic nedělá). Takových meta-instrukcí je v jazyce zpravidla většina. Dalšího výrazného zrychlení by tedy bylo možné dosáhnout využitím algoritmu Aho-Corasick [9], který umožňuje hledání míst pro aplikaci všech těchto meta-instrukcí najednou.

I v oblasti zápisu meta-instrukcí do vstupního souboru je několik věcí, které by bylo možné pro větší uživatelovo pohodlí doplnit. Regulární výrazy by bylo vhodné doplnit o prostředky pro zápis tříd znaků, jak je známe z jazyka Perl. Dále se nabízí možnost zahrnout do výčtových konstrukcí (`[`*str*`]` a `[^`*str*`]`) i jiné symboly, než jsou znaky. To by bylo možné realizovat například jejich uzavřením do špičatých závorek (tedy např. `[<cislo><id>]`).

Pro větší pohodlí uživatele by bylo jistě zajímavé pokusit se lépe vyřešit problematiku chybových hlášení při odmítnutí vstupu.

Také by mohlo být zajímavé zavedení restartovacího automatu se sémantikou způsobem, který zachovává podobu přijímací meta-instrukce za tu cenu, že taková meta-instrukce již nemůže být spojena s žádnou sémantickou akcí. Sémantickou hodnotou výsledku je u takového modelu seznam všech sémantických hodnot symbolů, které na pásce zbyly.

Literatura

[1] Fraunhofer Institute For Computer Architecture And Software Technology, *The Catalog of Compiler Construction Tools*, 2006, http://catalog.compilertools.net/.

[2] F. Otto: Restarting automata and their relations to the Chomsky hierarchy. In: Z. Esik and Z. Fulop (eds.), *Developments in Language Theory*, Proc. DLT'2003, LNCS 2710, Springer, Berlin, 2003, pp. 55–74.

[3] F. Mráz, M. Plátek and M. Procházka: On Special Forms of Restarting Automata. In: A. Mateescu (ed.), *Grammars*, Proc. FCT'1999, Vol. 2, Springer, Dordrecht, 1999 , pp. 223–233.

[4] M. Kutrib, H. Messerschmidt and F. Otto: On Stateless Deterministic Restarting Automata. In: M. Nielsen, A. Kučera et al. (eds.), *Theory and Practice of Computer Science*, Proc. SOFSEM'2009, LNCS 5404, Springer, Berlin, 2009, pp. 353–364.

[5] F. Mráz, F. Otto and M. Plátek: Learning Analysis by Reduction from Positive Data. In: Y. Sakakibara et al. (eds.), *Grammatical Inference: Algorithms and Applications*, LNCS 4201, Springer, Berlin, 2006, pp. 125–136.

[6] D. J. Farber, R. E. Griswold and I. P. Polonsky: SNOBOL, A String Manipulation Language. In: R. W. Hamming (ed.), *J. ACM*, Vol. 11, Iss. 1, ACM, New York, 1964, pp. 21–30.

[7] L. P. Deutsch, B. W. Lampson: An online editor. In: G. Salton (ed.), *Comm. ACM*, Vol. 10, Iss. 12, ACM, New York, 1967, pp. 793–799.

[8] T. Stubblebine: *Regular Expression Pocket Reference*, O'Reilly, Sebastopol, 2003, pp. 3–21.

[9] A. V. Aho, M. J. Corasick: Efficient string matching: an aid to bibliographic search. In: G. Manacher, *Comm. ACM*, Vol. 18, Iss. 6, ACM, New York, 1975, pp. 333-340.

Dodatek A

CCRA – uživatelská příručka

A.1 Úvod

CCRA (Compiler-Compiler based on Restarting Automata) je široce použitelný generátor kompilátorů. Těch sice existuje nepřeberné množství, ale většinou se liší pouze tím, v jakém programovacím jazyce umí vygenerovat výstup. Takřka všechny však generují LALR(k) popř. LL(k) parsery, výjimečně pak GLR. V tomto se CCRA od ostatních generátorů výrazně odlišuje, protože generuje parser založený na restartovacích automatech, díky čemuž dokáže zpracovávat širší třídu jazyků, než jsou jazyky bezkontextové.

Při návrhu CCRA byl kladen důraz na maximální možnou kompatibilitu s nástroji `yacc`, `Bison` a jim podobnými. Hlavním důvodem je, že problémy s kontextovými rysy zpracovávaného jazyka se někdy objeví až v okamžiku, kdy už by úplná změna způsobu zápisu gramatiky mohla být velmi bolestivá. V důsledku toho je pro programátora zvyklého používat `Bison` použití CCRA velmi snadné. Jako programovací jazyk bylo zvoleno C++. Byla zvažována ještě volba jazyka C, ale s ohledem na časovou náročnost výpočtu by tato varianta přinesla pouze komplikace a žádné výhody.

Jak již bylo řečeno, CCRA generuje parser založený na restartovacím automatu. Jakým způsobem je tento model využit je vysvětleno v oddíle A.2. Zároveň se jedná o lehkou formu představení nástroje. V navazujícím oddíle jsou podrobně popsány imlementace kompilátorů dvou konkrétních jednoduchých jazyků. Oddíl A.4 obsahuje formální popis vstupního souboru CCRA. Možnostmi propojení vygenerovaného parseru s dalším kódem se zabývá oddíl A.5. Pro lepší pochopení chování vygenerovaného programu je vhodné seznámit se se způsobem jeho práce.

Tomu je věnován oddíl A.6. Poslední oddíl je stručným návodem, jak nástroj nainstalovat.

A.2 Základní koncepty

Tato kapitola seznamuje se základními koncepty, bez kterých není možné pochopit fungování CCRA.

A.2.1 Jazyk a jeho popis

Jazyk, který se má zpracovávat, je potřeba nějakým způsobem popsat. Narozdíl od ostatních generátorů kompilátorů nemůže CCRA vyjít z BNF (Backus-Naurovy formy), protože tento popis nemá dostatečnou výrazovou sílu (je takto možné popsat pouze bezkontextové gramatiky). Jediný dosud vyvinutý způsob, jak popisovat (pro člověka srozumitelným způsobem) jazyky přijímané restartovacími automaty, jsou tzv. meta-instrukce.

Restartovací automat, a tedy i parser na tomto modelu založený je silně nedeterministický. Analýza takového parseru může být tedy velmi pomalá. To je daň za velkou výrazovou sílu.

A.2.2 Meta-instrukce

Restartovací automat je lingvisticky motivovaný teoretický model automatu. Jeho hlavní myšlenka je takováto. Máme-li rozhodnout o syntaktické správnosti věty a najít v ní případnou chybu, dá se to udělat tak, že najdeme místo, které dokážeme zjednodušit (například odstraněním rozvíjejícího větného členu), a takto zjednodušenou větu považujeme za nový vstup. Takovýmto postupem dostaneme časem větu, kterou již sice zjednodušit nelze, ale o jejíž správnosti dokážeme snadno rozhodnout. Vezměme si například větu: “Petr má tmavě modré oči.” Výše popsaným postupným zjednodušováním se můžeme dostat až k holé větě, o které je už snadné rozhodnout, že je syntakticky správná: “Petr má modré oči.” $\rightsquigarrow$ “Petr má oči.” $\rightsquigarrow$ “Petr má.”

Formalismus, kterým lze – pro člověka snadno čitelným způsobem – popsat úpravy uvedené v předchozím odstavci, stejně jako ony maximálně zjednodušené věty, které patří do jazyka, se nazývá meta-instrukce. Jedná se o sadu pravidel, z nichž každé je ve tvaru $(R_1, x \to y, R_2)$ a interpretuje se takto:

Pokud vstup obsahuje podřetězec x a to, co je před ním, patří do regulárního jazyka R_1 a to, co je za ním, do regulárního jazyka R_2, pak lze meta-instrukci aplikovat, tj. přepsat podřetězec x řetězcem y. Regulární jazyky R_1 a R_2 jsou popsány regulárními výrazy a nazývají se levý a pravý kontext.

Výpočet pak vypadá jako (nedeterministická) posloupnost aplikací meta-instrukcí a končí přijetím, pokud je aplikována meta-instrukce, která obsahuje v y speciální přijímací pomocný symbol. Nelze-li vstup přijmout (neexistuje-li posloupnost aplikací meta-instrukcí končící přijetím), je vstup zamítnut.

Příklad:

Vezměme si kontextový jazyk $\{a^nb^nc^nd^n : n \geq 1\}$. Zda vstup patří do tohoto jazyka zjistíme tak, že budeme v každém cyklu mazat po jednom od každého písmene. Pokud na konci dostaneme *abcd*, vstup přijmeme. Úpravy je však možné dělat pouze lokálně, nelze tedy smazat všechna 4 písmena v jednom kroku. Musíme tedy mazat střídavě *ab* a *cd*. K tomu budeme potřebovat pomocný symbol (X).

Analýzu můžeme rozdělit do čtyř kroků, přičemž každé z nich odpovídá jedna meta-instrukce. V prvním kroku se smaže jedna dvojice symbolů *ab*, ale jako informace o tomto mazání se na daném místě napíše symbol X. V druhém kroku se provede totéž s dvojicí symbolů *cd*. Následující dva kroky spočívají v odstranění obou symbolů X. Správné pořadí kroků je zajištěno díky kontrole výskytu pomocného symbolu v kontextu. Přijímací symbol označme třeba P. Vstup je přijat, pokud se tímto střídavým způsobem podaří odstranit z něj všechny symboly.

$(a^*, ab \to X, b^*c^*d^*)$ $\qquad$ $(a^*, aXb \to ab, b^*c^*Xd^*)$

$(a^*Xb^*c^*, cd \to X, d^*)$ $\qquad$ $(a^*b^*c^*, cXd \to cd, d^*)$

$(, XX \to P,)$

Příklad zpracování vstupu pro $n = 2$, tj. *aabbccdd*:

$$aabbccdd \rightsquigarrow aXbccdd \rightsquigarrow aXbcXd \rightsquigarrow abcXd \rightsquigarrow abcd \rightsquigarrow Xcd \rightsquigarrow XX \rightsquigarrow P.$$

A.2.3 Od meta-instrukcí ke vstupu CCRA

Meta-instrukce jsou pouze formalismem, ze kterého syntax vstupu pro CCRA vychází. V některých bodech se však oba zápisy více či méně liší. Hlavním důvodem je to, že CCRA, stejně jako jiné generátory parserů, umožňuje předřazení lexikálního analyzátoru – scanneru. To ale znamená, že regulární výrazy nepracují pouze se znaky, ale mohou pracovat i s názvy symbolů. Tomu samozřejmě musí odpovídat jejich zápis, který

se proto od toho běžně užívaného lehce liší. Podrobnosti viz oddíl A.4.3, str. 61.

Meta-instrukce $(R_1, xy \rightarrow z, R_2)$ se v CCRA zapíše následovně:

```
R1 / x y -> z / R2 ;
```

Zde `R1` (resp. `R2`) je zápis jazyka R_1 (resp. R_2) pomocí regulárního výrazu a lomítka, středník a šipka jsou interpunkční znaménka používaná ve všech pravidlech.

Jako příklad si vezměme jednoduchou kalkulačku, která umí pouze sčítat a odčítat čísla zapsaná ve dvojkové soustavě. Můžeme chtít začít tím, že postupně (zleva doprava) přepíšeme všechna čísla na pomocný symbol *cislo*. Tomu odpovídají následující dvě meta-instrukce:

$$((\textit{cislo}\ |[+-()])^*, \textit{cislo}\ 0 \rightarrow \textit{cislo}, .^*),$$
$$((\textit{cislo}\ |[+-()])^*, \textit{cislo}\ 1 \rightarrow \textit{cislo}, .^*).$$

Chceme-li je zapsat pomocí syntaxe použité v CCRA, zapíšeme je takto:

```
^ (cislo | [+-()])* / cislo "0" -> cislo / ;
^ (cislo | [+-()])* / cislo "1" -> cislo / ;
```

Podrobnosti viz oddíl A.4.3, str. 61.

A.2.4 Sémantická hodnota

Ve skutečných aplikacích nás jen málokdy zajímá pouze to, zda-li vstup patří čí nepatří do daného jazyka. Většinou je toto jen podproblém náležící ke zpracování daného vstupu. V předchozím příkladu kalkulačky nás pravděpodobně nebude zajímat pouze, jestli byl zadán syntakticky správný matematický výraz, ale také to, jakou má hodnotu.

To ale znamená, že už v okamžiku, kdy přepisujeme sekvence cifer na pomocný symbol *cislo*, bychom si měli někde ukládat hodnotu, jakou toto číslo má. Z tohoto důvodu, má každý symbol (vstupní i pomocný) ve skutečnosti dvě hodnoty – syntaktickou a sémantickou. Syntaktická hodnota říká, o jaký symbol se jedná, tedy její název. Sémantická hodnota nese zbytek informace, tedy například číselnou hodnotu čísla nebo jméno identifikátoru.

A.2.5 Sémantické akce

Sémantickou hodnotu symbolů na vstupu je nutné (pokud ji chceme používat) vyplnit již v lexikálním analyzátoru (např. `Flex`), ale pro symboly, které "vznikají" přepisem, je potřeba ji určit právě během přepisu. K tomu slouží tzv. akce. Tu může mít každá meta-instrukce. Jedná se o kód (v jazyce C++), který se má vykonat bezprostředně po aplikaci příslušné meta-instrukce. Podrobnosti viz oddíl A.4.3, str. 63.

Smyslem akce je vypočítat sémantickou hodnotu celé syntaktické konstrukce ze sémantických hodnot jejích částí. Představme si, že máme meta-instrukci, která říká, že součet dvou výrazů je také výraz. Za sémantickou hodnotu výrazu můžeme označit přímo jeho hodnotu. Potom pravidlo CCRA, které toto realizuje vypadá následovně:

```
/ expr "+" expr -> expr /
@{ $$1 = $1 + $3; };
```

Akce se zapisuje do složených závorek za pravý kontext a odděluje se od něj symbolem @. Ta v předchozím příkladě říká, že sémantická hodnota prvního ($$1) zapsaného symbolu (*expr*) je součet sémantických hodnot prvního ($1) a třetího ($3) přepisovaného symbolu (v obou případech také *expr*).

A.2.6 Generované soubory

Jako vstup dostává CCRA soubor s gramatikou, výstupem pak jsou zdrojové soubory parseru v jazyce C++. Je důležité rozumět tomu, že CCRA je pouze nástroj na vytvoření zdrojových souborů, které jsou součástí našeho programu, ale jako takový součástí programu není.

Vygenerovaný parser sestává ze čtyř souborů. CCRA.hxx je jakási knihovna obsahující převážně nezbytné definice tříd nezávislých na konkrétním překladači. Nejdůležitějším souborem je "*.cpp". Ten obsahuje tělo překladače. Příslušný hlavičkový soubor ("*.h") je podstatný hlavně pro začlenění parseru do nějakého většího projektu (mimo jiné obsahuje hodnoty symbolů, které jsou nezbytné pro lexikální analýzu). Poslední soubor je datový ("*.ccra") a obsahuje za běhu načítanou část parseru – přechodové tabulky.

Úkolem parseru založeného na restartovacích automatech je nalézt takovou posloupnost přepisů (podle sady pravidel daného jazyka) vstupu, že se v něm objeví přijímací symbol, čímž je výpočet úspěšně ukončen.

Pokud nás nezajímá jen výsledek, ale i ona přijímací posloupnost, je možné si ji udržovat pomocí sémantických akcí.

Abychom ze vstupního řetězce dostali sekvenci vstupních symbolů, musíme použít nějaký lexikální analyzátor, na který parser napojíme. Můžeme ho buď napsat ručně nebo na jeho vygenerování použít nějaký nástroj (např. `Flex`). Chystáme-li se zpracovávat nějaký jednodušší jazyk a bez lexikální analýzy se obejdeme, můžeme použít vestavěný "analyzátor", který pouze vrací znaky, které načetl.

A.2.7 Kroky při používání CCRA

Od popisu gramatiky k funkčnímu překladači je dlouhá cesta, avšak její část obsahující práci s parserem se dá shrnout do několika bodů:

1. Je třeba popsat jazyk ve formátu meta-instrukcí, kterému rozumí CCRA (Podrobnosti viz oddíl A.4, str. 57).
2. Následně pak pro každou meta-instrukci specifikovat akci (v C++).
3. Dále je potřeba pro parser vytvořit lexikální analyzátor a funkci, která bude parser přímo volat (typicky je to přímo *main*).
4. Nakonec je dobré nahradit i funkci `error`, která slouží k hlášení vnitřních chyb parseru, funkcí vlastní.

A.2.8 Struktura zápisu gramatiky pro CCRA

Vstupem pro CCRA je soubor s gramatikou, jehož obecný tvar je následující:

```
%{
   Prolog
%}

CCRA deklarace

%%
   Gramatická pravidla
%%
   Epilog
```

Dvojice znaků `%%`, `%{` a `%}` jsou oddělovače jednotlivých částí.

Prolog je místo určené k deklaraci proměnných a (alespoň dopředné deklaraci) funkcí použitých v akcích. Je zde také možné používat příkazy pro precosesor (jako `#include`, `#define`, apod.). Také je to nejvhodnější místo, kde deklarovat uživatelskou verzi funkcí *lex* a *error*. Zde umístěný kód bude ve vygenerovaných souborech vložen na začátek hlavičkového (*.h) souboru.

CCRA deklarace obsahuje výhradně direktivy pro CCRA jako název generované třídy, které symboly jsou přijímací, apod.

Sekce *Gramatická pravidla* obsahuje seznam meta-instrukcí s jim příslušnými sémantickými akcemi.

A poslední část – *Epilog* – může obsahovat libovolný kód (který bude umístěn na konec hlavního – *.cpp – souboru). Často se právě zde umísťují těla funkcí deklarovaných v *Prologu*.

A.3 Příklady

Nyní si popíšeme příklady programů vytvořených s pomocí CCRA. Programy jsou to velmi jednoduché, ale ukazují vše potřebné pro pochopení práce s CCRA. [1]

A.3.1 context2

S prvním příkladem jsme se částečně už setkali (str. 43). Jedná se o program, který rozhoduje, zda je vstup slovem z jazyka $\{a^n b^n c^n d^n : n \geq 1\}$ a pro ty, které do jazyka patří určí i n. Zdrojový soubor tohoto programu se jmenuje "context2.ra" (přípona ".ra" je sice nepovinná, přesto ji však doporučujeme). Toto je jeho výpis:

context2.ra

```
%{
    #define CCRA_PARAMS_TYPE int
    #define YY_DECL int yylex(int &param)
    YY_DECL;
    #define CCRA_LEX yylex
%}

%Accept accept
```

[1] Všechny tyto příklady byly testovány na systému Windows XP SP2 s využitím Microsoft Visual Studia 2005 a na systému GNU/Linux 2.6.22-p4smp. Všechny potřebné soubory jsou volně ke stažení na internetových stránkách tohoto projektu http://www.assembla.com/spaces/CCRA, případně na přiloženém CD.

```
%ClassName "Example"
%MainGen

%ExternTab "Context2RA.tab.ccra"
%%

^a*/ a b -> X / b* c* d*$ @{
    $$1 = $1 + 1;
};

^a*/ a X b -> a b / b* c* Y d*$ @{
    $$1 = $2;
};

^a* X b* c*/ c d -> Y / d*$ @{
    $$1 = $1 + 1;
};

^a* b* c*/ c Y d -> c d / d*$ @{
    $$1 = $2;
};

^/ X Y -> accept / $ @{
    $$1=$1;
    std::cout<<"Accepted!"<<std::endl;
    std::cout<<"The value of n is "<<$$1<<"."<<std::endl;
    std::cout<<std::endl;
};

%%
```

Deklarace

Deklaracemi se myslí úvodní část kódu po první výskyt značky `%%` (včetně *Prologu*).

Direktiva pre-procesoru `#define` definuje makro `CCRA_PARAMS_TYPE`, čímž určuje, jaký datový typ bude použit pro uchovávání sémantických hodnot symbolů. Tento typ je použit pro všechny symboly. Má-li se pro různé symboly lišit, je nutné použít `struct`, který bude obsahovat všechny použité typy. Není-li toto makro nastaveno, je použit typ `int`.

Makro `YY_DECL` je určeno použitému lexikálnímu analyzátoru, který je generován nástrojem `Flex`. Jedná se o hlavičku funkce, použité k načítání symbolů získaných ze vstupu. Hned na dalším řádku je toto makro vloženo jako dopředná deklarace. To aby mohlo být voláno z vygenerovaného

parseru. Poslední makro – `CCRA_LEX` – je opět název téže funkce. Tentokrát pro potřeby CCRA.

Druhou část sekce deklarace tvoří direktivy CCRA. První z nich říká, že symbol `accept` je symbolem přijímacím. Následuje nastavení názvu vygenerované třídy (parseru) na `Example`. `%MainGen` zařídí, aby byla vygenerována jednoduchá funkce `main`, která pouze spustí parser. A poslední direktiva nastavuje cestu, kam se má uložit (ale hlavně odkud se má načítat) soubor s přechodovou tabulkou.

Gramatická pravidla

Gramatická pravidla již byla vysvětlena (viz oddíl A.2.2, str. 42) stejně jako jejich zápis (viz oddíl A.2.3, str. 43). Je tedy zbytečné zde opakovat již řečené. Zmiňme alespoň opomenutý detail týkající se zjednodušení zápisu kontextu.

Ten je totiž ve většině případů silně lokální záležitostí, v důsledku čehož většina levých kontextů začíná výrazem $.*$, stejně jako jím většina kontextů pravých končí. Z toho důvodu byla zavedena konvence, že je tento podvýraz možné vynechat. (V důsledku čehož je možné, aby byl výraz prázdný, pokud na kontextu nezáleží.) Pokud toto chování není žádoucí, je potřeba použít jeden ze symbolů ^ a `$` v jejich běžném významu, tj. ^ označuje začátek zdrojového textu a `$` jeho konec.

Lexikální analyzátor

Úkolem lexikálního analyzátoru je přepracovat vstup na nízké úrovni na nejmenší možné logické celky a nastavit jejich sémantickou hodnotu. V našem případě má však za úkol pouze vynechání bílých znaků (jako sémantické hodnoty se mají totiž nastavit nuly) a chybová hlášení při výskytu chybného znaku. Na jeho vygenerování byl použit nástroj `Flex`. Toto je výpis souboru "context2.lex", použitého jako jeho vstup:

```
context2.lex
%{
    #include "context2.h"
    #include <iostream>
%}

%option noyywrap nounput
%option nostdinit
%option always-interactive
%option 8bit
```

```
WS        [ \t\f]
NL        \r?\n

%%

%{
    param=0;
%}

"a"   { return CCRA::Example::a;}
"b"   { return CCRA::Example::b;}
"c"   { return CCRA::Example::c;}
"d"   { return CCRA::Example::d;}

{WS}+ { /* ignore */ }

.     { std::cerr<<"Unknown character ('"<<yytext<<"') ignorred!";
        std::cerr<<std::endl;
      }

{NL}  { return 0; }

%%
```

Syntaxí použitou ve `Flexu` se zde nebudeme zabývat. Upozorníme však na pro nás podstatná místa.

Nejpodstatnější je vložení hlavičkového souboru "context2.h", který je jedním ze souborů generovaných CCRA obsahující dvě podstatné věci. Již výše zmiňovanou definici makra `YY_DECL` a deklaraci výčtového typu `TokenValues` definující hodnoty všech symbolů, které jsou vraceny (např. `CCRA::Example::a`). `CCRA` je zde název jmenného prostoru, do kterého je celý parser (z důvodu ochrany před kolizí názvů proměnných) uzavřen.

Kompilace programu

Zbývá popsat, jak celý program zkompilovat. Nejprve je samozřejmě nutné mít nainstalovaný CCRA. (Jak toho dosáhnout viz oddíl A.7, str. 72.)

Následující postup vytvoří spustitelný soubor "context2". Jedná se o příkazy pro systémy linux, ale postup je zachován pod všemi systémy.

```
# Vypis soubory v aktualnim adresari:
$ ls
context2.lex  context2.ra

# Zkompiluj CCRA parser:
$ ccra -o context2 context2.ra
Templates read successfully!
Parsed successfully!

# Znovu vypis soubory:
$ ls
Context2RA.tab.ccra  context2.h    context2.ra
context2.cpp         context2.lex

# Zkompiluj Flex lexikalni analyzator:
$ flex -b -o scanner.cpp context2.lex

# Znovu vypis soubory:
$ ls
Context2RA.tab.ccra  context2.h    context2.ra  scanner.cpp
context2.cpp         context2.lex  lex.backup

# Nyni jiz staci zkompilovat vygenerovane soubory:
$ g++ -Wall -I. -o scanner.o -c scanner.cpp
scanner.cpp: In function ‘int yy_get_next_buffer()’:
scanner.cpp:1005: warning: comparison between signed and unsigned
integer expressions

$ g++ -Wall -I. -o context2 scanner.o context2.cpp

# Znovu vypis soubory:
$ ls
Context2RA.tab.ccra  context2.h    lex.backup
context2             context2.lex  scanner.cpp
context2.cpp         context2.ra   scanner.o
```

Tímto jsme vytvořili spustitelný soubor “context2”. Zde je příklad jeho běhu:

```
$ ./context2
aabbccdd
Accepted!
The value of n is 2.

$
```

A.3.2 palindrom

Dalším příkladem je program, který hledá v textu ukrytý palindrom (číslo, které když se napíše pozadu má stejnou hodnotu) z číslic 0, 1 a 2. Ukrytý je tak, že se za ním může vyskytovat libovolný počet nul a před ním kromě nul i malá a velká písmena anglické abecedy. Formálně bychom mohli jazyk popsat takto:

$$\{u : u = [0a - zA - Z]^* v[012]?v^R 0^*, v \in [012]^*\}.$$

Úkolem programu je zjistit číselnou hodnotu zrcadlené části.

Tedy vstup `aq0bc000Z120021000000` program příjme a vypíše `120`. Je-li palindrom liché délky, zahrne se do výstupu i prostřední cifra, tedy pro vstup `021012` bude výstup `210`.

Zdrojový soubor ("palindrom.ra") je relativně delší. Rozebereme si ho proto po částech.

Deklarace

Program začíná deklarační částí:

palindrom.ra:1

```
%{
    struct params{
        long long expVal, expMul;
    };
    #define CCRA_PARAMS_TYPE params

    unsigned usrGetToken(params &);
    #define CCRA_LEX usrGetToken

    bool usrGetNextPosHeur(
        std::set<CCRA::RewritePos> &idxset,
        CCRA::RewritePos &miPos
    );
    #define CCRA_USER_NEXT_POSITION_HEURISTIC usrGetNextPosHeur
%}
%Accept cele
%ClassName "PalindromRA"
%ExternTab "PalindromRA.tab"
%MainGen

%%
```

Nejprve je pomocí makra `CCRA_PARAMS_TYPE` nastaven datový typ pro uchovávání sémantických hodnot symbolů na typ `params` (proč vysvětlíme později). Následuje nastavení funkce na načítání symbolů pomocí makra `CCRA_LEX` na funkci `usrGetToken`, jejíž dopředná deklarace se musí v této části také objevit (alespoň prostřednictvím vložení z jiného souboru pomocí direktivy `#include`). Vzhledem k tomu, že se jedná o kód, který je vkládán do generovaného hlavičkového souboru, není vhodné tuto funkci na tomto místě i implementovat.

Makro na posledním řádku *Prologu* nastavuje heuristickou funkci. Restartovací automat je obecně nedeterministický model. Ale protože počítače pracují deterministicky, je nutné tento nedeterminismus simulovat. Pomocí tohoto makra lze určit vlastní funkci, která bude ze seznamu míst, kde je možné kterou meta-instrukci aplikovat, vybírat to, které se nyní použije. Je-li heuristická funkce dobře napsána, může výpočet výrazně zrychlit.

Direktivy za *Prologem* říkají popořadě následující:

- přijímací symbol se jmenuje `cele`,
- vygenerovaná třída parseru se bude jmenovat `PalindromRA`,
- tabulky parseru budou vygenerovány do (a načítány ze) souboru "PalindromRA.tab" a
- bude vygenerována i funkce `main` volající parser.

Gramatická pravidla

Myšlenka zpracování je následující. Nejprve uhádneme, kde je střed palindormu, a poté budeme z obou stran odmazávat cifry, dokud takto nesmažeme celé číslo.

Co se týče sémantiky, budeme si ukládat do značky v prostředku hodnotu zatím smazané části. Ale protože nová cifra se vždy přidává doleva, bylo by nutné vždy znovu počítat, kolik cifer už jsme smazali, abychom ji mohli "přičíst". Z toho důvodu je sémantická hodnota ukládána do struktury obsahující navíc kromě zmiňované hodnoty (`expVal`) i hodnotu kterou je potřeba přenásobit příští cifru před přičtením (`expMul`).

Meta-instrukce odpovídající hádání středu palindromu vypadají tedy následovně:

```
palindrom.ra:2
^([01a-zA-Z]|C2)*/ "0" "0" -> S /([01]|C2)*$ @{
    $$1.expVal = 0;
    $$1.expMul = 10;
};
^([01a-zA-Z]|C2)*/ "1" "1" -> S /([01]|C2)*$ @{
    $$1.expVal = 1;
    $$1.expMul = 10;
};
^([01a-zA-Z]|C2)*/ C2 C2 -> S /([01]|C2)*$ @{
    $$1.expVal = 2;
    $$1.expMul = 10;
};
^([01a-zA-Z]|C2)*/ "0" -> S /([01]|C2)*$ @{
    $$1.expVal = 0;
    $$1.expMul = 10;
};
^([01a-zA-Z]|C2)*/ "1" -> S /([01]|C2)*$ @{
    $$1.expVal = 1;
    $$1.expMul = 10;
};
^([01a-zA-Z]|C2)*/ C2 -> S /([01]|C2)*$ @{
    $$1.expVal = 2;
    $$1.expMul = 10;
};
```

Ve skutečnosti by mohli být všechny levé i pravé kontexty těchto meta-instrukcí prázdné. Tímto se ale výpočet značně urychlí, zabraňujeme totiž vzniku více středů (značek `S`). První tři meta-instrukce odpovídají hledání středu u palindromu sudé délky, druhé tři pak u těch délky liché.

Další tři meta-instrukce odpovídají onomu odmazávání stejných cifer z obou stran od středu. Poslední meta-instrukce pak zajišťuje přijetí v okamžiku, kdy již byl odmazán celý palindrom.

```
palindrom.ra:3
/ "0" S "0" -> S / @{
    $$1.expMul=10*$2.expMul;
    $$1.expVal=$2.expMul*$1.expVal+$2.expVal;
};
/ "1" S "1" -> S / @{
    $$1.expMul=10*$2.expMul;
    $$1.expVal=$2.expMul*$1.expVal+$2.expVal;
```

```
};
/ C2 S C2 -> S / @{
    $$1.expMul=10*$2.expMul;
    $$1.expVal=$2.expMul*$1.expVal+$2.expVal;
};
^[0a-zA-Z]*/S -> cele /"0"*$ @{
    $$1.expVal = $1.expVal;
};

%%
```

Epilog

Epilog je místo, kam se většinou umísťují implementace v *Prologu* deklarovaných funkcí. V našem případě to jsou tedy funkce `usrGetToken`, `usrGetNextPosHeur` a `main`.

První z nich dělá to, že ze standardního vstupu načte znak a ten předzpracuje následovně. Pokud se jedná o znak konce řádky, tak vrátí konec vstupu (tj. nulu). Pokud se jedná o cifru nula, jedna nebo dva, tak nastaví sémantickou hodnotu a daný znak vrátí. Stejně tak vrátí načtený znak pokud se jedná ještě o jiný symbol (např. písmeno). Jako ukázka toho, jakým způsobem je možné kombinovat definované symboly přímo se znaky, je pro znak '2' definován symbol `C2` (jeho definování spočívá v tom, že je uveden v některém z pravidel).

Funkce `usrGetNextPosHeur` má za úkol určit (při možnosti volby), která meta-instrukce (a na jakém místě) bude použita. Náš jazyk tak, jak jsme ho popsali, umožňuje nejednoznačnost pouze při hledání středu palindromu. Proto jsme zvolili strategii takovou, že vždy vrací prostřední prvek seznamu. Ten je totiž setříděn hierarchicky podle pozice, a poté podle pořadí meta-instrukcí.

Funkce `main` pouze vytvoří instanci parseru (proměnná `ra`), který spustí (metoda `parse`). Návratovou hodnotu (`ret`, což je seznam symbolů z pravé strany poslední aplikované meta-instrukce – v našem případě se jedná tedy pouze o symbol `cele`) potom vytiskne a skončí.

palindrom.ra:4

```
unsigned usrGetToken(params &p){
    char in;
    std::cin>>std::noskipws>>in;
    switch(in){
```

```
        case '0':
        case '1':
            p.expVal=(in-'0');
            return in;
        case '2':
            p.expVal=(in-'0');
            return CCRA::PalindromRA::C2;
        case '\n':
        case '\r':
            return 0;
        default:
            return in;
    }
}

bool usrGetNextPosHeur(
    std::set<CCRA::RewritePos> &idxset,
    CCRA::RewritePos &miPos
){
    std::set<CCRA::RewritePos>::iterator it=idxset.begin();
    for(size_t i=0; i<(idxset.size()/2); ++i, ++it);
    miPos=*it;
    return true;
}

int main(int , char**){
    CCRA::PalindromRA ra;

    std::list<CCRA::TokenStruct<unsigned, params> > ret=ra.parse();
    if(ret.size()!=0){
        std::cout<<"The mirrored number is: ";
        std::cout<<ret.begin()->params.expVal<<std::endl;
    }else{
        std::cout<<"This input was not accepted!"<<std::endl;
    }

    return 0;
}
```

Kompilace

Kompilace tohoto příkladu je ještě jednodušší, než tomu bylo u toho předchozího. Je to tím, že je tvořen pouze jedním souborem.

Postup vypadá následovně:

```
# Vypis soubory v aktualnim adresari:
$ ls
palindrom.ra

# Zkompiluj CCRA parser:
$ ccra -o Palindrom -M palindrom.ra
Templates read successfully!
Parsed successfully!

# Znovu vypis soubory:
$ ls
Palindrom.cpp Palindrom.h PalindromRA.tab palindrom.ra

# Zkompiluj vygenerovane soubory:
$ g++ -I . -o palindrom Palindrom.cpp

# Znovu vypis soubory:
$ ls
Palindrom.cpp Palindrom.h PalindromRA.tab palindrom palindrom.ra

# Otestuj vytvoreny program:
$ ./palindrom
abzABZ011221100000
The mirrored number is: 112
$
```

Všímavý čtenář mohl zaznamenat, že vstupní soubor obsahuje direktivu `%MainGen`, ale zároveň jsme napsali vlastní funkci `main`, což by mělo způsobit kolizi. Toto nastavení by kolizi skutečně způsobilo, kdybychom nepoužili přepínač `-M`, který deaktivuje právě direktivu `%MainGen`.

Na první pohled se může zdát takováto možnost nepřehledná, ale ve skutečnosti může být velmi výhodná, pokud používáme jeden parser ve více projektech. Funkci `main` bychom však museli napsat do jiného souboru (jinak bychom totiž přepínač `-M` museli použít úplně vždy).

A.4 Vstupní soubor CCRA

V tomto oddíle je formálně popsán vstupní soubor programu CCRA.

CCRA zpracovává gramatiku zapsanou do zdrojového souboru pomocí meta-instrukcí ve speciální podobě a generuje třídu v jazyce C++, která daný jazyk rozpoznává.

Koncovka použitá pro zdrojový soubor je podle konvence ".ra".

Zdrojový soubor CCRA sestává ze čtyř částí, které jsou od sebe odděleny speciálními značkami.

```
%{
   Prolog
%}

CCRA deklarace

%%
   Gramatická pravidla
%%
   Epilog
```

V každá ze sekcí je možné používat C++ komentáře jak řádkové (`//`) tak blokové (`/* ... */`).

A.4.1 Prolog

Prolog je nepovinná část uzavřená mezi značkami `%{` a `%}`. Obvykle se zde nacházejí dopředné deklarace funkcí použitých v dalších částech, (extern) deklarace globálních proměnných a příkazy pre-procesoru (zejména `#include` a `#define`).

Jedná se o blok kódu, který je vložen na začátek generovaného hlavičkového souboru. Z toho důvodu je to také vhodné místo na deklaraci datového typu, který je použit pro ukládání sémantické hodnoty.

Makro-přepínače

Prolog je také jediným místem, kde se dají nastavit některé parametry generovaného parseru. (Ve skutečnosti je možné nastavit je zde i nepřímo – vložením jiného souboru direktivou `#include`).

Jedná se o následující makra:

- `#define CCRA_TOKEN_TYPE` *tp*
  Nastaví datový typ použitý pro uchovávání načtených znaků na *tp*. Tento typ musí být ordinální (musí mít definovaný `operator<`). Defaultní hodnota je `unsigned int`.

- `#define CCRA_PARAMS_TYPE` *tp*
  Nastaví datový typ pro ukládání sémantické hodnoty na *tp*. Defaultní hodnota je `char`.

- `#define CCRA_LEX` *fn*
  Způsobí, že program očekává existenci funkce `CCRA_TOKEN_TYPE` *fn*`(CCRA_PARAMS_TYPE&)`, pomocí které jsou načítány vstupní symboly. Pokud funkce není nastavena, je vygenerován kód, který vrací jednotlivé znaky vstupu.

- `#define CCRA_USER_NEXT_POSITION_HEURISTIC` *fn*
  Použitý model parseru (restartovací automat) je nedeterministický a tak je nutné testovat jednotlivé varianty postupně. Toto makro nastaví heuristickou funkci, která v každém okamžiku vybírá z množiny možných aplikací meta-instrukcí tu, která se aplikuje. Funkce musí mít následující hlavičku:
  `bool` *fn*`(std::set<CCRA::RewritePos>&, CCRA::RewritePos&);`

- `#define CCRA_ERR` *fn*
  Zamění standardní chybovou funkci za funkci *fn*, jejíž hlavička by měla mít tvar `void` *fn*`(const char *msg, void *param);`

A.4.2 CCRA deklarace

Jak již název napovídá, tato sekce slouží k zápisu různých deklarací CCRA. Implementovány jsou následující přepínače:

%Accept

Použití:
`%Accept` *sym*

Tento přepínač přidává symbol *sym* do seznamu přijímacích symbolů.

%ClassName

Použití:
`%ClassName` *name*

Jedná se o přepínač, který nastavuje jméno generované třídy (parseru) na *name*. Při vícenásobném uvedení je použit název uvedený u posledního výskytu. Naopak pokud není použit vůbec, nese vygenerovaná třída jméno `RestartingAutomaton`.

%ExternTab

Použití:

```
%ExternTab "path"
```

Pomocí tohoto přepínače je možné určit cestu k souboru (*path*), do kterého bude uložena přechodová tabulka automatu. Důležité ovšem je hlavně to, že na této cestě bude vygenerovaný program daný soubor také hledat. Z toho důvodu je typické použití pouze názvu souboru. Je možné použít i relativní cestu. Jako oddělovač se používá na všech systémech symbol "/". To umožňuje přenositelnost zdrojového souboru mezi systémy. Pokud není přepínač uveden, má vygenerovaný soubor jméno odpovídající názvu generované třídy s přidanou příponou ".ccra".

%MainGen

Použití:

```
%MainGen
```

Uvedení tohoto přepínače způsobí, že je kromě třídy obsahující generovaný parser vygenerována i jednoduchá třída `main`. Ta dělá pouze to, že vytvoří instanci parseru a spustí ho.

%No-lines

Použití:

```
%No-lines
```

Někdy není žádoucí, aby byly ve vygenerovaných souborech použity direktivy `#line`, které způsobují to, že ladící nástroje poznají, že se jedná o vložený kus kódu a odkud je vložený (to umožňuje například umístit do původního souboru breakpoint). V takovém případě lze použít tento přepínač a všechny tyto direktivy budou vynechány.

A.4.3 Gramatická pravidla

Tato část vstupního souboru obsahuje jednotlivé meta-instrukce (včetně jejich případných sémantických akcí).

Syntax meta-instrukcí

Každá meta-instrukce sestává z několika částí uspořádaných podle tohoto schématu:

$$R_1 \texttt{ / } u \texttt{ -> } v \texttt{ / } R_2 \texttt{ @}\{Sem\} \texttt{ ;}$$

R_1 i R_2 jsou regulární jazyky popsané pomocí speciální formy regulárních výrazů. u a v jsou seznamy symbolů. Toto pravidlo říká, že sekvenci symbolů u je možné zjednodušit do podoby v, pokud to, co se nachází před u patří do jazyka R_1 a to co se nachází za u patří do jazyka R_2.

Až na u může být každá složka prázdná. Pokud je prázdná sémantická akce *Sem*, tak lze vynechat i její uzávorkování – "`@{}`".

Na libovolném místě, kde se může vyskytnout mezera, se smí nacházet i tabulátor nebo dokonce odřádkování. To může významně přispívat k přehlednosti zdrojového souboru.

Přepis

Pokud je meta-instrukce aplikována, dochází k přepisu sekvence symbolů u na sekvenci symbolů v. Zde je nutné být velmi opatrný. Je nutné, aby byla posloupnost v v jistém smyslu jednodušší než původní u. Důvodem je potřeba, aby byl běh nad libovolným vstupem konečný. V základní definici je za onu míru jednoduchosti brána délka posloupnosti, tedy je požadováno aby u bylo delší než v. Mnohdy je však tento požadavek zbytečně silný, znemožňuje totiž přepis symbolu na symbol obecnější (např. číslo → výraz).

Proto byla pro implementaci zvolena silnější forma – zmenšující restartovací automaty. V tomto modelu je každému symbolu přiřazeno kladné číslo – váha – a podmínka je taková, že aplikací meta-instrukce se musí součet vah symbolů snížit.

Tyto váhy však k výpočtu nejsou vůbec potřeba, slouží pouze k ověření konečnosti výpočtu. Navíc existují ještě silnější formy restartovacích automatů, ve kterých se nepožaduje snižování součtu vah po každém kroku, ale až po n krocích. Z těchto důvodů neprobíhá v tomto smyslu žádná kontrola, a tedy není po uživateli požadováno ani zadávání vah.

Kontext

Jak již bylo řečeno, levý a pravý kontext jsou zapisovány ve formě modifikovaných regulárních výrazů. Modifikovaných z toho důvodu, aby

bylo možné pracovat nejen se znaky, ale i se symboly (ať už získanými v předřazené lexikální analýze nebo aplikací některé meta-instrukce).

Podporovány jsou následující konstrukce regulárních výrazů:

- *RE1* | *RE2*
  – disjunkce jazyků *RE1* a *RE2*,
- *RE1 RE2*
  – zřetězení jazyků *RE1* a *RE2*,
- *RE*{*m*,*n*}
  – zřetězení jazyka *RE* *m*- až *n*-krát se sebou samým; neuvedení *m* znamená jeho hodnotu 0, neuvedení *n* znamená jeho hodnotu nekonečno,
- *RE*{*n*}
  – ekvivalent zápisu *RE*{*n*,*n*},
- *RE* *
  – ekvivalent zápisu *RE*{*0*,},
- *RE* +
  – ekvivalent zápisu *RE*{*1*,},
- *RE* ?
  – ekvivalent zápisu *RE*{0,1},
- (*RE*)
  – regulární jazyk *RE* (zavedeno z důvodu potřeby úpravy priorit),
- [*str*]
  – všechna slova délky jedna ze symbolů (znaků) obsažených v *str*,
- [^*str*] – všechna slova délky jednak, která neobsahují žádný symbol (znak) obsažený v *str*,
- .
  – všechna slova délky jedna,
- *id*
  – regulární jazyk sestávající z jediného slova délky jedna – *id*, kde *id* je název symbolu,
- "*str*"
  – regulární jazyk sestávající z jediného slova – *str*, kde *str* je řetězec symbolů (znaků) tohoto slova,

- ^
  – levá zarážka – pokud toto není první znak R_1, může být ignorován libovolně dlouhý úsek na začátku vstupu tak, aby zbytek po přepisovaný výraz u, patřil do jazyka R_1, a tedy bylo možné danou meta-instrukci aplikovat,

- $
  – pravá zarážka – obdoba levé zarážky, ale jedná se o R_2 a o úsek na konci vstupu.

Sémantická hodnota, sémantická akce

V typické aplikaci není až tak zajímavé zjištění, že je daný vstup syntakticky správný. Typicky totiž tento vstup chceme dále zpracovat – přeložit, interpretovat apod. CCRA používá systém převzatý z atributových gramatik. Každému symbolu je možné přiřadit jeho význam (sémantickou hodnotu) do proměnné, kterou si pro tyto účely drží. Princip je takový, že je možné určit kód, který se vykoná v okamžiku aplikace dané meta-instrukce. Tomuto kódu se říká sémantická akce. Smyslem sémantické akce je nastavit sémantickou hodnotu všem symbolům ve v, přičemž by se k tomu nemělo využívat jiných informací, než jsou sémantické hodnoty symbolů v u. Je možné využívat ještě globální proměnné, ale to může být zdroj ohromných problémů s ohledem na potenciální back-tracking.

Takto však není možné nastavit sémantickou hodnotu vstupních symbolů. Pokud se jedná o řetězcovou konstantu (např. “+”), tak to vůbec ničemu nevadí, protože je veškerá informaci o takovém symbolu v pravidle, kde je použit. Totéž platí pro pojmenované vstupní symboly vzešlé z lexikální analýzy, pokud jsou tyto symboly sémanticky ekvivalentní (je-li např. možné použít pro násobení symbol “*” i “.” a lexikální analýza je oba zpracovává na symbol `MUL`). Pokud však tyto symboly sémanticky ekvivalentní nejsou (např. symbol `cislo` pro všechna čísla), je třeba nastavit jejich sémantickou hodnotu. To se dá udělat pouze v okamžiku, kdy je tato hodnota známa – v lexikální analýze. Viz oddíl A.5.

Sémantická akce (*Sem*) se zapisuje do složených závorek a od pravého kontextu příslušné meta-instrukce se odděluje znakem ‘@’. Jedná se o kód v C++ rozšířený o možnost odkazovat se na sémantické hodnoty jednotlivých symbolů pravidla. Odkaz na symboly v seznamu u má tvar `$`n, kdežto odkaz na symboly v seznamu v má tvar `$$`n. V obou případech značí n pořadové číslo symbolu v příslušném seznamu (číslováno od 1). Není-li datový typ použitý pro ukládání sémantické hodnoty (jeho nastavování viz oddíl A.4.1, str. 58) skalární (pole, třída apod.), nečiní to

žádné problémy (např. `$1[3]`, `$13.x` nebo `$7.getX()`).

Uveďme si jednoduchý příklad. Mějme meta-instrukci, která říká, že součet dvou výrazů je výraz s tím, že jako jeho sémantickou hodnot si budeme držet hodnotu tohoto výrazu. Kód může potom vypadat následovně:

```
/ vyraz "+" vyraz -> vyraz / @{ $$1 = $1 + $3; };
```

Někdy nastávají situace, že podle sémantiky lze určit, že aplikace dané meta-instrukce je nesmyslná. V takovém okamžiku je velmi výhodné mít možnost její aplikaci ještě zamítnout. K tomu slouží makro-příkaz `CCRA_REJECT`.

A.4.4 Epilog

Epilog je (podobně jako *Prolog*) blok kódu v jazyce C++. Tento je však vložen na konec generovaného hlavního ("*.cpp") souboru. Proto je předurčen k umístění implementací funkcí v části *Prolog* deklarovaných.

A.5 Rozhraní parseru

Aby byl vygenerovaný parser lépe použitelný, existuje řada míst, která se dají propojit s dalším kódem. Zde je jejich popis.

A.5.1 Vstupní a pomocné symboly

Pro syntaktické celky v CCRA se používá označení symboly. V programu jsou reprezentovány jako prvky výčtového typu. Symboly je možné rozdělit do dvou skupin na symboly vstupní a pomocné.

Vstupní symboly (v oblasti překladačů obvykle označované jako terminály) jsou ty, u kterých se předpokládá, že se mohou vyskytnout na vstupu parseru. Tedy například znak '`+`', ale pokud je použit lexikální analyzátor, tak třeba i `cislo` pro sekvenci číslic ve vstupním souboru (protože na syntaktické úrovni jsou si všechna čísla ekvivalentní). Pomocné symboly (v oblasti překladačů obvykle označované jako neterminály) jsou pak ty, u kterých se takový výskyt nepředpokládá. Tedy například `vyraz` pro algebraický výraz.

A.5.2 Lexikální analýza

Často nastává situace, kdy je potřeba předřadit parseru lexikální analyzátor (scanner). Toho lze dosáhnout v *Prologu* pomocí makra `CCRA_LEX`, kterým se určí jeho hlavní funkce (viz oddíl A.4.1, str. 58). Pokud je použit generátor scannerů – `Flex`, je to funkce `yylex`.

Chování této funkce musí být takové, že při každém zavolání vrátí kód dalšího symbolu v pořadí ze vstupu. Pokud byl dosažen konec vstupu, je očekáván kód 0. Je také umožněno předávat znaky ze vstupu beze změny (tj. jako kód symbolu je předáván jeho ascii kód).

V případě, že makro `CCRA_LEX` není definováno, je načítán standardní vstup po znacích (včetně bílých znaků). Je nutné ho tedy použít i tehdy, pokud lexikální analýza není nutná, ale zpracován nemá být standardní vstup, nýbrž soubor. Poté funkce může vypadat následovně.

```
unsigned fileGetToken(CCRA::Parser::ParamsType&){
    char c;
    my_ifstream>>c;
    if(my_ifstream.eof()) return 0;
    return c;
}
```

S tím, že `my_ifstream` je globální proměnná typu `std::ifstream` nastavená v okamžiku spuštění parseru na začátek zpracovávaného souboru (tj. soubor je s ní asociován a otevřen).

Pokud je však vhodné použít scanner, je nutné používat pro symboly stejné kódy, jaké jsou použity v parseru. Pro tyto účely je uvnitř vygenerované třídy parseru definován veřejný výčtový typ `TokenValues`. Například je-li třída parseru pojmenována `Parser`, tak se kód symbolu `CISLO` vrátí pomocí `CCRA::Parser::CISLO`. Zde je `CCRA::` nutné uvádět, pouze pokud je kód umístěn mimo jmenný prostor, do kterého je celý parser vždy uzavřen – `CCRA`.

Dalším důležitým úkolem scanneru je nastavení sémantické hodnoty jednotlivých načtených symbolů. Obzvlášť pokud se tato ztrácí. Zpracovává-li například scanner každou sekvenci číslic tak, že předá symbol `CISLO`, tak je z pravidla nutné tuto číselnou hodnotu nastavit jako sémantickou hodnotu tohoto symbolu. K tomu slouží povinný parametr volané funkce, který je předávaný referencí a je právě typu jaký je určeno, že má být použit pro udržování sémantické hodnoty (tedy defaultně `char`, jinak hodnota makra `CCRA_PARAMS_TYPE` – viz oddíl A.4.1, str. 58). Hod-

nota tohoto parametru je totiž v parseru použita pro nastavení sémantické hodnoty načteného symbolu.

A.5.3 Použití parseru

Začlenění vygenerovaného parseru do projektu a jeho spuštění je velmi snadnou záležitostí. Parser je tvořen jednou třídou, jejíž název je možné měnit přepínačem `%ClassName`. Použití sestává pouze ze dvou kroků. V prvním je třeba vytvořit proměnnou daného typu. V konstruktoru proběhne inicializace (včetně načtení souboru externích tabulek). Druhým krokem je zavolání samotného procesu parsování.

Pokud je veškerá práce se sémantikou celého vstupu ošetřena v přijímacích meta-instrukcích, pak je toto vše, co je potřeba udělat (a je to přesně to, co dělá případná generovaná funkce `main`). Pokud však toto ošetřeno není, je potřeba sémantiku předat a dále zpracovat. Pro tyto účely je chování parseru nastaveno tak, že vrací seznam symbolů vytvořených v poslední aplikované (tedy přijímací) meta-instrukci, včetně jejich sémantiky. Toto propojení je realizováno datovým typem `CCRA::TokenStruct`, obsahujícím položku `value` pro kód symbolu a položku `params` pro jeho sémantickou hodnotu.

Byly-li například nastaveny následující konfigurační hodnoty:

- `#define CCRA_TOKEN_TYPE tok`,
- `#define CCRA_PARAMS_TYPE sem`,
- `%ClassName Parser`,

je možné použít k vypsání vrácených symbolů včetně jejich sémantické hodnoty následující kód:

```
CCRA::Parser parser;

std::list<CCRA::TokenStruct<tok,sem> > ret=parser.parse();

if(ret.size()=0){
    std::cout<<"This input was not accepted!"<<std::endl;
}else{
    for(std::list<CCRA::TokenStruct<tok,sem> >::const_iterator
        it=ret.begin(); it!=ret.end(); ++it
    ){
        std::cout<<it->value<<" - "<<it->params<<std::endl;
    }
}
```

A.5.4 Problémy nedeterminismu

Protože restartovací automaty jsou nedeterministický model, je nutné výpočet nějakým způsobem "determinizovat". CCRA k tomu využívá prohledávání do hloubky. Aby však bylo možné výpočet urychlit, lze v každém okamžiku vybrat z možných větví výpočtu tu, která má být prozkoumána, případně i všechny nabízené větve zamítnout.

Pokud uživatel toto rozhodování nechá bez zásahu, je větev výpočtu vybírána tak, že dříve se zkoumá dřívější místo přepisu (měřeno podle pozice prvního přepisovaného symbolu). Pokud je možné na tomto místě aplikovat více meta-instrukcí, je použita dříve ta, která je ve vstupním souboru definována dříve. Není-li toto chování žádoucí, změna se provede definováním makra `CCRA_USER_NEXT_POSITION_HEURISTIC`, jehož hodnota je nastavena na název funkce implementující požadované chování.

Tato funkce má povinný tvar hlavičky:

```
bool fn(std::set<CCRA::RewritePos>&, CCRA::RewritePos&)
```

Návratová hodnota říká, jestli se ze všech přípustných aplikací meta-instrukcí, podařilo vybrat tu, která se použije. V prvním parametru je předána množina těchto přípustných aplikací. Do druhého (výstupního) parametru se kopíruje ta, která byla vybrána. K práci s přípustnými aplikacemi je určena struktura `RewritePos`, která má dvě položky: pořadové číslo meta-instrukce (`mi`) a pozice, na které její aplikace začíná (`pos`). Obě tyto položky jsou typu `size_t`.

Dalším problémem, který sebou nedeterminismus přináší je možná nejednoznačnost výpočtu. Na syntaktické úrovni to ještě nevadí – vstup je přijat, pokud existuje alespoň jedna posloupnost aplikací meta-instrukcí vedoucí k přijetí. Existuje-li tedy druhá taková posloupnost, nic to nemění na tom, že je slovo přijato. Na sémantické úrovni je to ale něco jiného, protože se může stát (a v praxi se, bohužel, stává), že každý takový výpočet vede k jiné sémantické hodnotě. Tedy že vstup nemá jednoznačný význam. Né vždy je nutné získat všechny tyto hodnoty, ale obvykle je vhodné dokázat tuto situaci alespoň detekovat.

V CCRA sice není pro tyto potřeby žádná přímá podpora, přesto však lze získat všechny možné sémantické hodnoty vstupu. Dosáhne se toho tak, že se do každé původně přijímací meta-instrukce přidá makropříkaz sémantického odmítnutí – `CCRA_REJECT`. Předtím se však přidá sémantická hodnota do seznamu dosud nalezených sémantických hodnot. Díky tomu je vstup sice nakonec vždy odmítnut, ale v tomto seznamu jsou uloženy sémantické hodnoty všech možných přijímacích výpočtů.

A.5.5 Externí tabulky

Vygenerovaný parser není uložen pouze do souborů s kódem ("*.h" a "*.cpp"). U složitějších jazyků totiž mohou být přechodové tabulky značně rozsáhlé a obzvláště u větších projektů nemusí být úplně žádoucí, aby byly v paměti po celou dobu běhu programu. Proto jsou uloženy do separátního souboru, který je načítán až v konstruktoru daného parseru. (Jméno tohoto souboru je možné nastavit přepínačem `%ExternTab` – viz oddíl A.4.1, str. 58).

Z důvodů snazšího ladění je tento soubor v textovém formátu, obsahuje však pouze (díky formátování pomocí bílých znaků) strukturovaně zapsané konstanty. Gramatika tohoto souboru je následující:

```
externtab ::= poc_metainstrukci sez_metainstrukci ;
poc_metainstrukci ::= CISLO ;
sez_metainstrukci ::= sez_metainstrukci metainstrukce
                  |
                  ;

metainstrukce ::= prefix suffix prepis semantika je_prijimaci ;
prefix ::= regexp ;
suffix ::= regexp ;
    regexp ::= zastavuje poc_stavu sez_stavu ;
    zastavuje ::= LOG_HOD ;
    poc_stavu ::= CISLO ;
    sez_stavu ::= sez_stavu stav
              |
              ;
      stav ::= je_in je_out poc_prechodu sez_prechodu prechod_zb ;
      je_in ::= LOG_HOD ;
      je_out ::= LOG_HOD ;
      poc_prechodu ::= CISLO ;
      sez_prechodu ::= sez_prechodu prechod
                   |
                   ;
      prechod_zb ::= poc_cilu sez_cilu ;
      prechod ::= symbol poc_cilu sez_cilu ;
      poc_cilu ::= CISLO ;
      sez_cilu ::= sez_cilu cil;
               |
               ;
      cil ::= CISLO ;
prepis ::= stara_delka nova_delka sez_symbolu ;
    stara_delka ::= CISLO ;
    nova_delka ::= CISLO ;
    sez_symbolu ::= sez_symbolu symbol
```

```
              |
              ;
    symbol ::= CISLO ;
semantika ::= CISLO ;
je_prijimaci ::= LOG_HOD ;
```

Terminál CISLO reprezentuje běžný zápis čísla v desítkové soustavě (tedy sekvenci číslic "0" až "9" bez úvodních nul). Terminál LOG_HOD reprezentuje číselně logickou hodnotu – true jako 1 a false jako 0.

Významy použitých neterminálů jsou následující:

- externtab
  – počáteční neterminál,
- sez_metainstrukci
  – seznam definovaných meta-instrukcí (délky poc_metainstrukci),
- metainstrukce
  – zápis pro jednu každou meta-instrukci,
- prefix
  – levý kontext dané meta-instrukce,
- suffix
  – pravý kontext dané meta-instrukce zapsaný pozadu a zahrnující i přepisované symboly,
- regexp
  – regulární výraz popsaný nedeterministickým konečným automatem,
- zastavuje
  – určení, zda daný regulární výraz obsahuje zarážku (^, resp. $),
- sez_stavu
  – seznam stavů daného konečného automatu (délky poc_stavu),
- stav
  – popis stavu daného konečného automatu,
- je_in
  – určení, zda je stav počáteční,

- `je_out`
  – určení, zda je stav výstupní,
- `sez_prechodu`
  – seznam přechodů z daného stavu o délce `poc_prechodu`,
- `prechod_zb`
  – přechod z daného stavu přes symboly neuvedené v `sez_prechodu`,
- `prechod`
  – definice jednoho přechodu ze stavu,
- `sez_cilu`
  – seznam stavů (délky `poc_cilu`) dosažitelných přechodem přes příslušný `symbol`,
- `cil`
  – číslo koncového stavu daného přechodu,
- `prepis`
  – definice co se na co přepisuje,
- `stara_delka`
  – počet přepisovaných symbolů,
- `nova_delka`
  – počet nově zapsaných symbolů,
- `sez_symbolu`
  – seznam nově zapsaných symbolů,
- `symbol`
  – kód symbolu,
- `semantika`
  – kód odkazující na přidruženou sémantickou akci,
- `je_prijimaci`
  – určení, zda se jedná o přijímací meta-instrukci.

A.5.6 Chybová hlášení

Jedním z největších problémů v oblasti parserů je zotavení z chyb. Tento problém je v CCRA bohužel ještě umocněn tím, že je výpočet nedeterministický, a tedy není možné v okamžiku, kdy již neexistuje meta-instrukce, kterou by bylo možné aplikovat, rozhodnout o tom, zda se jedná o chybu ve vstupu, nebo zda se pouze výpočet nachází ve slepé větvi.

Z toho důvodu není pro automatické generování chybových hlášení o syntaktické chybě v CCRA žádná podpora. Pokud jsou tato hlášení potřebná je nutné vytvořit speciální meta-instrukce pokrývající právě chybné vstupy s tím, že jejich sémantická akce se skládá pouze z vypsání chybové hlášky a tzv. sémantického odmítnutí – příkazu `CCRA_REJECT`.

Další možností, jak vytvářet alespoň nějaká chybová hlášení je ukládat si co bylo na co přepsáno a v případě, že se vstup nepodaří zpracovat je možné například vypsat stav, kdy byl vstup zjednodušen do maximální možné míry, tedy např. minimalizován počet symbolů nebo třeba maximalizován počet aplikovaných meta-instrukcí.

Přecejen však existují situace, kdy je chybové hlášení vyvoláno uvnitř generovaným kódem. V konstruktoru je otvírán a načítán soubor externích tabulek. Pokud se však tento soubor nepodařilo otevřít, je vyvoláno chybové hlášení: *"Cannot open CCRA tables file!"*. Stejně je tomu i v případě poškození tohoto souboru, v důsledku kterého je meta-instrukci přiřazena sémantická akce s kódem, který neexistuje. Pak je použita zpráva: *"Unknown semantic section – corrupted CCRA tables file!"*. Protože však není vždy žádoucí vypisování těchto zpráv v právě této podobě právě na standardní chybový výstup, je možné napsat vlastní funkci zajišťující hlášení chyb. Její používání se pak nastaví pomocí makra `CCRA_ERR` (viz oddíl A.4.1, str. 58).

A.6 Parsovací algoritmus

Způsob práce parseru neumožňuje interaktivní režim. Prvním krokem je tedy načtení všech symbolů ze vstupu do vnitřních struktur parseru. Následuje rekursivní hledání přijímací sekvence meta-instrukcí.

Toto hledání vypadá tak, že se nejprve najdou všechny možnosti aplikace meta-instrukcí. Tím je vytvořen seznam dvojic (meta-instrukce, místo přepisu). Z tohoto seznamu jsou postupně jednotlivé aplikace vybírány. Zvolená aplikace je provedena a následuje rekursivní zpracování. Pokud je seznam prázdný, znamená to, že se výpočet nachází ve slepé

větvi a je proto vrácen o úroveň výš.

Zbývá popsat, jak probíhá hledání možných aplikací meta-instrukcí. Každá meta-instrukce obsahuje popis dvou regulárních jazyků (levého a pravého kontextu). Tyto jazyky jsou v paměti udržovány v podobě nedeterministických konečných automatů. Automat pro pravý kontext je navíc modifikován tak, že popisuje zrcadlový obraz jazyka pravého kontextu zřetězený se zrcadlovým obrazem přepisované části vstupu. To umožňuje hledat místa, kde je možné aplikovat meta-instrukci tak, že je vstup předložen automatu, který rozpoznává levý kontext, s tím, že se poznačí každá pozice, na které se automat nacházel v přijímacím stavu. Následně se obdobným způsobem pustí automat pro pravý kontext na vstup, ale zprava doleva. Místa, kde je možné danou meta-instrukci aplikovat, jsou pak ta, která byla poznačena v obou bězích. Tento postup je zopakován s každou meta-instrukcí. Hledaný seznam možných aplikací meta-instrukcí je pak vytvořen sjednocením takto nalezených seznamů.

A.7 Instalace CCRA

Díky tomu, že CCRA nevyužívá žádnou knihovnu, je jeho instalace velmi snadná.

Nástroj CCRA sestává z následujících částí:

- ze spustitelného souboru ("CCRA.exe" v systémech Windows, resp. "ccra" v systémech Linux)
- ze souborů "header.tpl" a "body.tpl" – šablon, jejichž vyplněním je získán požadovaný kompilátor a
- ze souboru "CCRA.hxx" – knihovního hlavičkového souboru.

Pokud je pro daný systém k dispozici již zkompilovaná verze programu, sestává jeho instalace pouze z nakopírování všech souborů programu do cílového adresáře. Je však nutné zachovat adresářovou strukturu. Ta může vypadat následovně:

```
CCRA\
    output\
        CCRA.exe
    data\
        header.tpl
        body.tpl
        CCRA.hxx
```

Program se souborem “CCRA.hxx” ve skutečnosti nepracuje, je pouze vkládán do hlavičkových souborů generovaných kompilátorů (příkazem pre-procesoru `#include <CCRA.hxx>`). Z toho důvodu je možné ho libovolně přesunout. Je však nutné zajistit, aby kompilátor vygenerovaného parseru věděl, kde ho má hledat. Jsou tedy dvě možnosti, buď soubor přesunout do adresáře, kde jsou hledány standardní hlavičkové soubory, nebo jeho umístění mezi tyto adresáře přidat. U kompilátoru `g++` se toto zajistí přepínačem `-I`, v prostředích Microsoft Visual Studio (MSVS) pak nastavením položky `Additional Include Directories` ve vlastnostech projektu, resp. přepínačem kompilátoru `/I`.

Pokud zkompilovaný program k dispozici není, stačí se na systémech Linux přepnout do složky s příslušným souborem “Makefile” a spustit příkaz `make`. Tedy za předpokladu, že jsou korektně nainstalovány nástroje `Flex` a `Bison`. Pro systémy Windows je připraven projekt pro MSVS 2005, jehož kompilací je dosaženo za stejného předpokladu stejného efektu.

Může nastat také situace, kdy nejsou k dispozici ani potřebné soubory šablon (“*.tpl”). V takovém případě je nutné na systému Windows nejprve zkompilovat program `Templater`. Nyní stačí spustit skript `makeTemplate.bat`, který ze zdrojových souborů projektu `Template` vytvoří pomocí nástroje `Templater` požadované šablony a umístí je do příslušného adresáře. Program `Templater` je možné zkompilovat i na systému Linux, je však nutné ho volat ručně namísto spouštění uvedeného skriptu.

```
~/CCRA/templater$ g++ -o Debug/Templater.exe Templater.cpp
~/CCRA/templater$ cd ../template
~/CCRA/template$ ../templater/Debug/Templater.exe Template.cpp Temp
late.hpp
~/CCRA/template$ cp CCRA.hxx ../data/CCRA.hxx
~/CCRA/template$ mv Template.hpp.tpl ../data/header.tpl
~/CCRA/template$ mv Template.cpp.tpl ../data/body.tpl
```

Dodatek B

Obsah přiloženého CD

- **CCRA/**
  Program včetně zdrojových kódů.
  - **data/**
    Datové soubory, které jsou součástí instalace programu.
  - **doc/**
    Obrazová dokumentace zdrojového kódu.
  - **examples/**
    Příklady použití CCRA.
    - **context2/**
      Kompilátor jazyka $\{a^n.b^n.c^n.d^n : n \in N\}$
    - **kontext/**
      Kompilátor jazyka $\{a^n.b^n.c, a^n.b^{2n}.d : n \in N\}$
    - **lokanty/**
      Kompilátor jazyka řeckých čísel do 9999.
    - **palindrom/**
      Kompilátor jazyka $\{[0a - zA - Z]^*.u.0^*\}$,
      kde $u \in \{0, 1, 2\}^*$ *je palindrom.*
  - **generated/**
    Vygenerované zdrojové kódy.
  - **include/**
    Hlavičkové soubory.
  - **output/**
    Zkompilovaný projekt.
  - **src/**
    Zdrojové kódy.
  - **template/**
    Pomocný projekt – příklad požadovaného výstupu.
  - **templater/**
    Pomocný projekt – převod zdrojových kódů projektu template na šablony.
- **TEXT/**
  Textová dokumentace včetně zdrojových kódů.

  - **img/**
    Vložené obrázky.
  - **include/**
    Definice formátu.
  - **out/**
    Zkompilovaná dokumentace.
  - **src/**
    Zdrojové kódy jednotlivých kapitol.

- **utils**/
  Použité pomocné skripty.
  - **bisonnicer.bat**
    Převod chybového výstupu nástroje `bison` *do formátu MSVS2005.*
  - **fasttex.bat**
    Usnadnění kompilace dokumentace.
  - **flexnicer.bat**
    Převod chybového výstupu nástroje `yacc` *do formátu MSVS2005.*
  - **makeTemplates.bat**
    Vytvoření šablon CCRA z projektu template.
  - **todos.bat**
    Skript hledající v celém projektu nedodělané věci.
  - **AutomataDumpPainter.ppt**
    Ladicí nástroj generující z textového zápisu grafickou reprezentaci automatu.