PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Gramatiky LALR(1)

2011 David Beer

Anotace

Gramatiky $LALR(1)$ jsou deterministické bezkontextové gramatiky, pro které lze sestrojit efektivní syntaktický analyzátor s lineární složitostí závislé na délce vstupního řetězce. V úvodu jsou shrnuty metody syntaktické analýzy pro deterministické bezkontextové jazyky, zejména LR metody. Dále se text podrobněji zabývá $LALR(1)$ metodou pro její dobrý poměr mezi sílou a efektivitou. V textu je popsána konstrukce $LALR(1)$ syntaktického analyzátoru, tak jak se používá v překladačích. Výsledkem je GUI aplikace realizující konstrukci syntaktického analyzátoru pro zadanou bezkontextovou gramatiku a simulující jeho činnost.

Děkuji vedoucímu této práce, RNDr. Arnoštovi Večerkovi, za jeho čas a trpělivost při zodpovídání nespočet dotazů.

Obsah

Seznam obrázků

1. Syntaktická analýza LR jazyků

Deterministické bezkontextové gramatiky jsou důležitou podmnožinou bezkontextových gramatik, obzvlášť pro počítačové zpracování. Na jednu stranu jsou silné, jelikož dokáží popsat velké množství jazyků. Například většina programovacích jazyků jsou deterministické bezkontextové jazyky. Na druhou stranu se dají rozumně algoritmizovat. Jsme pro ně schopni sestrojit efektivní syntaktický analyzátor (angl. parser) s lineární časovou i paměťovou složitostí. Zároveň jsme schopni proces sestrojení syntaktického analyzátoru zautomatizovat a vytvořit tak program, který pro danou deterministickou bezkontextovou gramatiku sestrojí její syntaktický analyzátor. Sestrojený syntaktický analyzátor dokáže při zpracovávání vstupu zjistit druh syntaktické chyby a umožňuje její sémantické zpracování.

Syntaktické analyzátory bývají mimo jiné součástí překladačů a interpretů. Syntaktické analyzátory provádějí syntaktickou analýzu, která nastupuje po úvodní lexikální analýze zdrojového kódu. Syntaktická analýza kontroluje správnost syntaxe a vytváří datovou reprezentaci kódu, nejčastěji strom.

Jednou z metod syntaktické analýzy deterministických bezkontextových jazyků je metoda LR. Název pochází z anglického „**L**eft to right“ a „**R**ight parse“. Znamená to, že čteme vstup zleva doprava a vytváříme pravé odvození. Tato metoda prochází derivační strom zdola nahoru, tedy od listů ke kořenu. Syntaktickým analyzátorem pro LR metodu je deterministický zásobníkový automat s jediným stavem, automat tedy pracuje pouze se svým zásobníkem. Během čtení vstupu si automat na zásobníku vytváří historii „kudy šel“ a na základě této historie, načteného symbolu a případně dodatečné informace, kterou si automat vytvořil během své konstrukce, se rozhodne „kudy se vydat dál“.

Dalšími metodami jsou například precedenční metoda nebo LL metoda. Hlavní rozdíl mezi LL metodou a LR metodou je, že LL metoda vytváří levé odvození a derivačním stromem prochází shora dolů, tedy od kořene k listům.

LR metoda syntaktické analýzy je výhodná zejména proto, že pro každý deterministický bezkontextový jazyk lze sestrojit LR syntaktický analyzátor. To pro ostatní metody zdaleka neplatí. Navíc je tento analyzátor stejně efektivní jako analyzátor pro ostatní metody.

V dalším textu se předpokládá znalost základních pojmů teorie formálních jazyků a automatů, které lze najít například v [1, 2, 3, 4]. Dále, jelikož se budu zabývat pouze deterministickými bezkontextovými gramatikami, budu pod pojmem gramatika myslet deterministická bezkontextová gramatika.

1.1. Činnost LR automatu

Syntaktickým analyzátorem pro LR metodu je deterministický zásobníkový automat. Tento automat budu dále označovat jako LR automat. Během syntaktické analýzy vstupního řetězce LR automat provádí dvě operace. Těmito

operacemi jsou přesun (angl. shift) a redukce (angl. reduction).

Přesun posouvá LR automat o jeden vstupní znak dopředu. Jedná se o situaci, kdy se automat nachází v kontextu odvozování pomocí určitého pravidla na určité pozici tohoto pravidla a ze vstupu očekává znak, který odpovídá terminálu na této pozici v pravidle. Pokud je na vstupu onen znak, automat se v pravidle posune o jednu pozici dál za terminál a přepne se do jiného kontextu. Pokud na vstupu není očekávaný znak, automat vstup odmítne, jelikož pozná, že tento vstup nemůže být z gramatiky odvozen. Pouze při přesunu se automat posune ve zpracovávání vstupu dál.

Redukce vrací LR automat v odvozování o jednu úroveň zpět. Jedná se o situaci, kdy se automat dostal v odvozovaném pravidle na jeho konec a může tedy podle něj redukovat, protože ví, že právě odvodil kus věty, která odpovídá pravé straně tohoto pravidla. Při redukci se automat neposouvá ve vstupu dál, pouze přepíná kontext.

▷ Příklad 1.1. Mějme jednoduchou gramatiku s počátečním symbolem S a s následujícími pravidly:

1. $S \to E + E$
2. $E \to x$

Činnost automatu pro vstupní řetězec $x + x$ zachycují následující kroky:

1. Na začátku automat ví, že pokud má být vstup odvoditelný z gramatiky, pak jej musí být schopen odvodit ze symbolu S.
2. Odvození symbolu S vede na odvození větné formy $E + E$.
3. Odvození větné formy $E + E$ vede nejprve na odvození E.
4. Automat odvozuje E a čeká na vstupu znak x, který tam skutečně je a automat tedy provede operaci přesun x.
5. Nyní se automat nachází v odvozování na konci pravidla $E \to x$ a provede operaci redukce podle pravidla $E \to x$, což jej vrátí do stavu, kde byl v kroku 3, ale za symbol E, který právě odvodil.
6. Automat provede přesun $+$.
7. Automat se pokusí odvodit E.
8. Automat provede přesun x.
9. Automat provede redukci podle pravidla $E \to x$.
10. Automat provede redukci podle pravidla $S \to E + E$.

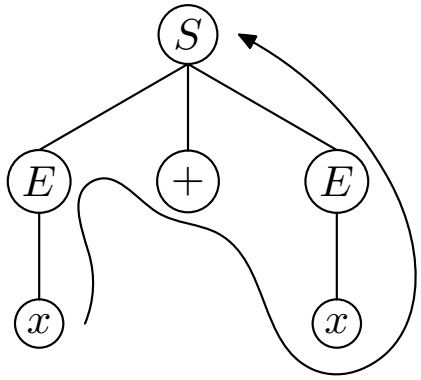
11. Automat úspěšně odvodil symbol S a vstup je vyčerpán. Vstup je automatem přijat.

Ve skutečnosti LR automat provádí pouze operace přesun a redukce. Například kroky 1 až 3 se vůbec neprovedou. Automat se již během konstrukce naučí, že na začátku má očekávat x. Výše uvedená činnost měla sloužit pro lepší pochopení. Skutečná činnost automatu je následující:

1. Přesun x.
2. Redukce podle pravidla $E \to x$.
3. Přesun $+$.
4. Přesun x.
5. Redukce podle pravidla $E \to x$.
6. Redukce podle pravidla $S \to E + E$.
7. Přijetí vstupu.

Kdyby automat na vstup dostal řetězec $x + y$, pak by tento řetězec zamítl v kroku 4, kde by na vstupu bylo y, ale automat očekával x. □

Během syntaktické analýzy se automat pokouší sestavit derivační strom. Jak lze z příkladu vidět, automat derivační strom sestavuje zdola nahoru. Nejprve redukuje koncové symboly x na symboly E a poté redukuje větnou formu $E + E$ na počáteční symbol. Průchod derivačním stromem je zachycen na obrázku 1.



Obrázek 1. Průchod derivačním stromem

Pokud se automatu povede derivační strom sestavit, pak vytvořil nejpravější odvození věty z gramatiky. K tomuto odvození se lze dostat přes redukce, které automat během analýzy provedl. Redukce musíme ovšem projít pozpátku, jelikož

analýza probíhá zdola nahoru. Odvození tedy dostaneme tak, že začneme od počátečního symbolu a projdeme redukce pozpátku a pro každou redukci použijeme pravidlo, podle kterého redukce proběhla, na expanzi nejpravějšího neterminálu. Pro předchozí příklad je tedy $S => E + E => E + x => x + x$ nejpravějším odvozením.

1.2. Konflikty v LR automatu

LR automat v každém kroku syntaktické analýzy provede operaci přesunu nebo redukce, popřípadě vstup přijme nebo zamítne. Může se ale stát, že automat má v určitém kroku více možností co provést. Například může provést přesun i redukovat nebo redukovat pomocí více pravidel. Tyto situace nazýváme konflikty v LR automatu. Jedná se o dva typy konfliktů. Konflikt přesun/redukce a konflikt redukce/redukce. Ke konfliktu přesun/přesun nemůže dojít (viz kapitola 1.2.3.).

1.2.1. Konflikt přesun/redukce

Jedná se o konflikt, kdy v daném kontextu může automat přesouvat nebo redukovat. Příkladem může být známý konflikt, který se nazývá „dangling *else*“ (v překladu „visící *else*“). Tento konflikt se týká programovacích jazyků, které syntakticky nijak neukončují podmíněné větvení pomocí příkazu if, tedy například jazyka C a jazyků z něj odvozených. Uvažujme část gramatiky s následujícími pravidly:

1. $SELECTION \rightarrow if\ COND\ STATEMENT$
2. $SELECTION \rightarrow if\ COND\ STATEMENT\ else\ STATEMENT$

Pravidlu 1 se říká „krátký if“ a pravidlu 2 „dlouhý if“. Automat po přesunu if, redukci na $COND$ a redukci na $STATEMENT$ neví, zda má pokračovat přesunem $else$ nebo redukcí na $SELECTION$. Je nutné si uvědomit, že ani přítomnost $else$ ve vstupu nám situaci nevyřeší. Uvažujme následující kus kódu v jazyce C:

```
if (x > 0)
    if (x == y)
        y = 2*x;
  else
      y = x;
```

Větev $else$ je úmyslně odsazena tak, že nenaznačuje ke kterému if patří. Má se do y přiřadit hodnota x, pokud je x kladné a různé od y nebo pokud není x kladné? Pokud dáme přednost „krátkému if“, pak automat redukuje vnitřní if a větev $else$ tak bude patřit k vnějšímu if. Pokud dáme přednost „dlouhému if“, pak

automat přesune *else* a větev *else* tak bude patřit k vnitřnímu if. V jazyce C lze samozřejmě uvést blokový příkaz, který jednoznačně určuje, kde celý blok začíná a kde končí, nicméně i v uvedeném případě se syntaktický analyzátor obsažen v kompilátoru jazyka C musí nějak zachovat.

Nabízí se dva způsoby řešení konfliktu na úrovni gramatiky, jak uvádí [8].

Úprava syntaxe jazyka

Zavede se syntax, která jednoznačně určuje rozsah příkazu if.

1. $SELECTION \rightarrow if\ COND\ STATEMENT\ end$
2. $SELECTION \rightarrow if\ COND\ STATEMENT\ else\ STATEMENT\ end$

Zde klíčové slovo *end* jednoznačně určuje, kde který if končí. Tento přístup používá například MATLAB. Nevýhodou tohoto přístupu je méně příjemná syntax jazyka pro programátora.

Omezení dlouhého if

Omezíme „dlouhý if“ tak, že bude moci obsahovat pouze příkaz, který nepoužívá „krátký if“. Upřednostníme tak „dlouhý if“ a tedy přesun před redukcí.

1. $SELECTION \rightarrow if\ COND\ STATEMENT$
2. $SELECTION \rightarrow if\ COND\ INNERSTM\ else\ STATEMENT$
3. $INNERSTM \rightarrow INNERSELECTION$
4. $INNERSELECTION \rightarrow if\ COND\ INNERSTM\ else\ INNERSTM$

Tento přístup používá například jazyk C. Výhodou tohoto přístupu je, že nemusíme zasahovat do syntaxe jazyka. Nevýhodou ovšem je množství nových pravidel v gramatice, která syntaktický analyzátor komplikují. Budeme totiž muset pro všechna pravidla, popisující rozvoj neterminálu $STATEMENT$, která končí na $STATEMENT$, přidat odpovídající verze těchto pravidel s levou stranou $INNERSTM$ a končící na $INNERSTM$. To se týká například pravidel popisujících rozvoj příkazu *while*, for a mnohých dalších. Toto řešení například zvětší syntaktický analyzátor jazyka C o zhruba 14%.

Jak lze vidět, oba popsané postupy mají své nevýhody. V praxi se konflikt přesun/redukce neřeší na úrovni gramatiky, ale na úrovni syntaktického analyzátoru pomocí priorit pravidel. Stejného efektu jako při omezení „dlouhého if“, navíc bez jakéhokoliv komplikování syntaktického analyzátoru, lze dosáhnout pomocí upřednostnění „dlouhého if“. Automat jednoduše vždy zvolí přesun a tedy „dlouhý if“. Pro výše uvedený kód platí, že *else* patří k poslednímu if, tedy tak jak to známe z jazyka C.

1.2.2. Konflikt redukce/redukce

Jedná se o konflikt, kdy v daném kontextu může automat redukovat pomocí více pravidel. Uvažujme gramatiku s těmito pravidly:

1. $S \to i \mid iS \mid \epsilon$

Tato gramatika obsahuje konflikt redukce/redukce a to mezi pravidly $S \to i$ a $S \to \epsilon$. Poté co automat ze vstupu načte poslední i, neví zda má rovnou redukovat podle pravidla $S \to i$ nebo nejdřív podle pravidla $S \to \epsilon$ a potom podle pravidla $S \to iS$.

Konflikt redukce/redukce lze většinou snadno řešit úpravou gramatiky a tato úprava navíc často gramatiku zjednoduší. Nejčastěji pomůže odstranění zbytečného pravidla nebo „dosazení“ společného rozvoje některých pravidel. Ve výše uvedeném případě stačí odstranit zbytečné pravidlo $S \to i$.

1.2.3. Teoretický konflikt přesun/přesun

Konflikt typu přesun/přesun v automatu nemůže nastat. Ke konfliktu by teoreticky mohlo dojít pouze v následujících případech:

a) Uvažujme následující dvě pravidla:

   1. $S \to A + B \mid A + C$

   Předpokládejme že automat již redukoval na A a na vstupu nyní máme $+$. V úvahu stále připadají obě pravidla, nicméně v tomto kroku automat nemusí rozhodovat o které z nich jde. Prostě přesune $+$ a bude pokračovat dalším krokem. K rozlišení toho, které pravidlo bude použito, dojde později, případně může později dojít k jinému konfliktu.

b) Uvažujme následující dvě pravidla:

   1. $S \to A + B \mid A - B$

   Předpokládejme že automat již redukoval na A. Potenciální konflikt zde vyřeší přímo vstup. Pokud automat dostane $+$, jedná se o první pravidlo, pokud dostane $-$, jedná se o druhé pravidlo.

c) Uvažujme následující dvě pravidla:

   1. $S \to A + C \mid B + C$

Předpokládejme že automat již redukoval na A nebo na B. Pak tato redukce již rozlišila o jaké pravidlo jde, případně již při ní došlo k jinému konfliktu.

1.3. Základní pojmy

Následující definice jsem převzal z [6] a [1].

Před samotným zpracováním je dobré si vstupní gramatiku upravit. Konkrétně do ní přidáme nový počáteční symbol S' a nové pravidlo $S' \to S$, kde S byl původní počáteční symbol.

Definice 1.1. Mějme gramatiku $G = (N, T, P, S)$. Gramatiku $G' = (N', T, P', S')$, kde $S' \notin N$, $N' = N \cup \{S'\}$, $P' = P \cup \{S' \to S\}$ nazveme *expandovanou gramatikou*.

Expandovaná gramatika je ekvivalentní původní gramatice. Pouze jsme před původní gramatiku předsunuli jedno pravidlo. Tento tvar je pro další zpracování výhodný. Často totiž budeme něco odvozovat z „počátečních pravidel“ nebo kontrolovat, zda se neredukuje pomocí „počátečního pravidla“. „Počáteční pravidlo“ je pro gramatiku v tomto tvaru vždy jen jedno a to $S' \to S$, místo několika pravidel tvaru $S \to \alpha$, kde α je libovolná větná forma.

Informaci o tom, zda v určitém kroku přesunout nebo redukovat a podle kterého pravidla tuto operaci provést, budeme reprezentovat pomocí LR položek.

Definice 1.2. Mějme gramatiku $G = (N, T, P, S)$. Trojici $(p, i, \omega) \in P \times \{0, 1, \ldots, n\} \times 2^{(T \cup \{\epsilon\})}$, kde n je délka pravé strany pravidla p, nazveme LR položkou gramatiky G. Speciálně pro $i = n$ trojici nazveme redukční LR položkou gramatiky G. Složky p a i se nazývají *jádro LR položky* a značíme jej $Ker(p, i, \omega) = (p, i)$. Složka ω se nazývá *množina prediktivních symbolů* a značíme ji $LA(p, i, \omega) = \omega$.

LR položka nese všechny potřebné informace k rozhodnutí o tom, jakou operaci provést v určitém kroku analýzy. Složka p je samotné pravidlo, podle kterého operaci provedeme. Složka i určuje pozici (číslovanou od nuly), na které se zrovna nacházíme v pravidle p. V případě redukční položky redukujeme, jinak přesouváme. Množina prediktivních symbolů obsahuje terminály, které nám říkají, že můžeme redukovat podle pravidla p, pokud máme jeden z těchto terminálů na vstupu. Terminál ϵ značí konec vstupu.

Pro větší přehlednost budeme LR položky symbolicky značit pomocí notace, ve které aktuální pozici určuje znak „tečka“ uvedený přímo na pravé straně pravidla. Například položku $(A \to B + B, 1, \{a, \epsilon\})$ budeme značit $[A \to B. + B\,; \{a, \epsilon\}]$ nebo redukční položku $(A \to B + B, 3, \{a, +\})$ budeme značit $[A \to B + B.\,; \{a, +\}]$.

K výpočtu množin prediktivních symbolů budeme potřebovat pomocné množiny $FIRST(\alpha)$.

Definice 1.3. Pro gramatiku $G = (N, T, P, S)$ a větnou formu $\alpha \in (N \cup T)^*$ definujeme $FIRST(\alpha)$ takto:

$$FIRST(\alpha) = \{t \mid \alpha \overset{*}{\Rightarrow} t\beta, t \in T, \beta \in (N \cup T)^*\} \cup \{\epsilon \mid \alpha \overset{*}{\Rightarrow} \epsilon\}$$

Množina $FIRST(\alpha)$ tedy obsahuje ty terminály, které se po úplném odvození z α objeví na začátku odvozené věty, popřípadě může obsahovat ϵ, pokud z α lze odvodit ϵ.

Kontext, ve kterém se při analýze nacházíme, budeme reprezentovat pomocí množiny LR položek. Jednotlivé LR položky v této množině reprezentují, že momentálně odvozujeme pravou stranu pravidla některé z těchto položek a nacházíme se v ní na pozici z položky.

Definici jádra LR položky rozšíříme na množinu LR položek sjednocením přes jádra jednotlivých položek.

Definice 1.4. Mějme množinu LR položek M. Pak

$$Ker(M) = \bigcup_{\forall p \in M} \{Ker(p)\}$$

nazveme jádrem množiny LR položek.

V kapitole 1.1. bylo uvedeno, že automat provede v každém kroku analýzy buď přesun nebo redukci. Představme si, že v některém kroku analýzy odvozujeme například pravou stranu pravidla $A \to aBc$ a nacházíme se na pozici před neterminálem B. Nejsme na konci pravidla ani nenásleduje terminál. Abychom poznali, co máme udělat, potřebujeme všechna odvození z B. Obdobně pokud z B lze odvodit větná forma začínající na neterminál C, budeme potřebovat všechna odvození z C a tak dále. Pokud z původního B nakonec odvodíme větnou formu začínající na terminál, můžeme tento terminál přesunout. Pokud z B odvodíme ϵ, budeme redukovat pomocí ϵ-pravidla, které bylo na konci odvození z B. K tomuto rozhodnutí budeme potřebovat uzávěr množiny LR položek.

Definice 1.5. Mějme gramatiku $G = (N, T, P, S)$. Pro množinu LR položek M definujeme posloupnost množin $Closure_i$, kde $i \in \mathbb{N}_0$, následovně:

$$\begin{aligned} Closure_0 &= M \\ Closure_{i+1} &= Closure_i \cup \{[B \to .\beta\,;\, \kappa(\gamma, \omega)] \mid [A \to \alpha.B\gamma\,;\, \omega] \in Closure_i \wedge B \to \beta \in P\}, \end{aligned}$$

kde nová množina prediktivních symbolů $\kappa(\gamma, \omega)$ je

$$\kappa(\gamma, \omega) = \bigcup_{\forall t \in \omega} FIRST(\gamma t)$$

Množinu

$$Closure(M) = \bigcup_{i=0}^{\infty} Closure_i$$

nazveme *uzávěr množiny LR položek M.*

Jelikož pravidel gramatiky je konečně mnoho, pozic tečky na pravé straně pravidla je konečně mnoho a množina terminálů je konečná a tedy i množin prediktivních symbolů je konečně mnoho (množina prediktivních symbolů je podmnožina množiny terminálů rozšířené o ϵ), je výše uvedený induktivní proces konečný a existuje tedy index i, pro který platí $Closure_i = Closure_{i+1} = Closure_{i+2} = \ldots$ a zároveň platí $Closure(M) = Closure_i$.

Z důvodů, které vyplynou později, budeme potřebovat slučovat množiny LR položek se stejným jádrem.

Definice 1.6. Mějme dvě množiny LR položek M_1 a M_2, pro které platí $Ker(M_1) = Ker(M_2)$. Definujeme *sloučenou množinu LR položek* pro M_1 a M_2 následovně:

$$Merge(M_1, M_2) = \{[A \to \alpha.\beta\,;\, LA(p_1) \cup LA(p_2)] \mid [A \to \alpha.\beta] \in Ker(M_1)\},$$

kde $p_1 \in M_1, p_2 \in M_2 \wedge Ker(p_1) = Ker(p_2) = [A \to \alpha.\beta] \in Ker(M_1)$

Dvě množiny LR položek tedy sloučíme tak, že sloučíme množiny prediktivních symbolů položek se stejným jádrem z obou množin.

Na reprezentaci přechodů mezi kontexty, ve kterých se může LR automat nacházet, použijeme pomocný automat. Důvod proč nepoužijeme přímo LR automat je ten, že v průběhu analýzy se budeme po každé redukci vracet k předchozí „rozpracované části“, a na to stavy automatu nestačí, k tomu potřebujeme zásobník. Nový pomocný automat budeme používat pouze pro průchod k redukci.

Uvažujme například pravidlo $A \to aBc$. Pro odvození A a tedy nalezení kusu věty, která odpovídá větné formě aBc, použijeme pomocný automat. Když se ale dostaneme k neterminálu B, budeme proces potřebovat rekurzivně opakovat. Pro odvození B tedy použijeme jakoby další instanci pomocného automatu. Jakmile pomocný automat přijme část věty jako odvození z B, vrátíme se k instanci pomocného automatu, která odvozuje neterminál A, nyní ve stavu za neterminálem B. Jednotlivé instance pomocného automatu tedy odvozují elementární části věty a přijímají, jakmile je možné redukovat.

Jako pomocný automat, který nazveme *charakteristický LR automat*, bude stačit konečný deterministický automat. Jeho stavy budou kontexty LR automatu, tedy množiny LR položek. Celou stromovitou hierarchii analýzy si LR automat pamatuje na zásobníku.

Ve skutečnosti nebudeme používat instanci pomocného automatu pro každé další odvození. Místo toho rozšíříme přechody pomocného automatu i o neterminály a budeme při redukci procházet pomocným automatem pozpátku, čímž se jakoby vrátíme k dřívější instanci pomocného automatu. Cestu kudy se vrátit má LR automat na svém zásobníku. Pokud tedy odvodíme aBc, vydáme se pozpátku postupně po přechodech c, B, a, čímž se vrátíme jakoby k instanci pomocného automatu, který v některém stavu potřeboval odvodit A, a přejdeme teď už ve směru přechodů po A, protože A jsme právě odvodili.

Definice 1.7. Mějme expandovanou gramatiku $G' = (N', T, P', S')$. Konečný deterministický automat $A = (T \cup N, Q, \delta, Closure(\{[S' \to .S\,; \{\epsilon\}]\}), F)$, kde množina stavů Q je množina množin LR položek, $F \subseteq Q$ je množina $q \in Q$, které obsahují redukční položku, se nazývá „charakteristický LR automat pro gramatiku G'“.

Stavy charakteristického LR automatu jsou množiny LR položek, které reprezentují kontext LR automatu, δ je přechodová funkce mezi těmito kontexty a koncové stavy jsou ty kontexty, ve kterých lze redukovat.

1.4. Gramatiky $LR(k)$

Pokud výsledný LR automat provádějící syntaktickou analýzu pro danou gramatiku dopadne bez konfliktů, pak o takové gramatice říkáme, že je gramatikou $LR(k)$ pro některé $k \in \mathbb{N}_0$. Neformálně řečeno to znamená následující. Mějme posloupnost π_i větných forem pro $i = 0, 1, \ldots, n$, které tvoří pravou derivaci nějaké věty $\alpha \in L(G)$, $G = (N, T, P, S)$. To znamená, že $\pi_0 = S$, $\pi_n = \alpha$, $\pi_i \Rightarrow \pi_{i+1}$ pro $i = 0, 1, \ldots, n-1$ a $\pi_0 \overset{*}{\Rightarrow} \pi_n$. Pokud $\pi_i = \beta A z$ a $\pi_{i+1} = \beta\gamma z$, kde $A \in N$, $\beta, \gamma \in (N \cup T)^*$ a $z \in T^*$, pak jsme schopni na základě nejvýše prvních k symbolů řetězce z (tedy k následujících vstupních symbolů) jednoznačně určit větnou formu γ a neterminál A, ze kterého se odvozuje podle pravidla $A \to \gamma$.

Věta 1.1. *Pro každou gramatiku $LR(k)$, kde $k > 1$, existuje ekvivalentní gramatika $LR(1)$.*

Postup transformace gramatiky $LR(k)$ na gramatiku $LR(1)$ je možné najít v [5]. Transformace je bohužel často vykoupena obrovským nárůstem velikosti této gramatiky. Na druhou stranu, pokud nějaká gramatika není $LR(1)$, pak je jen malá šance, že je $LR(k)$ pro některé $k > 1$. Každopádně to má jeden důležitý důsledek z hlediska LR analýzy. Každý deterministický bezkontextový jazyk je $LR(1)$ jazykem. Z tohoto důvodu nám stačilo zavést množinu prediktivních symbolů a nemuseli jsme zavádět množinu prediktivních řetězců. Pro gramatiku $LR(k)$ jsou totiž prvky této množiny řetězce délky až k. Díky tomu mají syntaktické analyzátory pro $LR(1)$ gramatiky, tedy zásobníkové automaty $LR(1)$, schopnost se při chybě zastavit v analyzovaném vstupu na prvním chybném symbolu, což zachová největší možnou míru kontextu pro ošetření chyby. Zavedeme si tedy speciálně gramatiku $LR(1)$ pomocí již uvedených pojmů, což bude průhlednější z hlediska konstrukce $LR(1)$ automatu.

Definice 1.8. Nechť $A = (T \cup N, Q, \delta, Closure(\{[S' \to .S\,; \{\epsilon\}]\}), F)$ je charakteristický LR automat expandované gramatiky $G' = (N', T, P', S')$. O gramatice G říkáme, že je $LR(1)$, pokud jsou splněny následující dvě podmínky:

1. Pro každou množinu LR položek $M \in F$ a každou její redukční položku $p \in M$ neexistuje přechod $\delta(M, t)$ pro žádný $t \in LA(p)$.

2. Pro každé dvě různé redukční položky $p_1, p_2 \in M$ z množiny LR položek $M \in F$ platí $LA(p_1) \cap LA(p_2) = \emptyset$.

První podmínka říká, že v LR automatu nesmí nastat konflikt přesun/redukce. Pokud by pro některou množinu LR položek $M \in F$ byl definován přechod $\delta(M, t)$ pro některé t, které je obsaženo v $LA([A \to \alpha. ; \omega])$, pak bychom mohli při t na vstupu přesunout t i redukovat podle pravidla $A \to \alpha$, což je konflikt.

Podobně druhá podmínka říká, že v LR automatu nesmí nastat konflikt redukce/redukce. Pokud by totiž množiny prediktivních symbolů dvou redukčních položek $[A_1 \to \alpha_1. ; \omega_1]$ a $[A_2 \to \alpha_2. ; \omega_2]$ ze stejné množiny LR položek obsahovaly shodný symbol t, pak bychom mohli při t na vstupu redukovat podle pravidla $A_1 \to \alpha_1$ i $A_2 \to \alpha_2$, což je opět konflikt.

▷ Příklad 1.2. Uvažujme gramatiku popisující příkaz přiřazení s dereferencí:

1. $S \to L = R \,|\, R$

2. $L \to {}^*R \,|\, i$

3. $R \to L$

Množiny $LR(1)$ položek této gramatiky vypadají následovně:

M_0: $[S' \to .S\,;\, \{\epsilon\}]$
$[S \to .L = R\,;\, \{\epsilon\}]$
$[S \to .R\,;\, \{\epsilon\}]$
$[L \to .{}^*R\,;\, \{=, \epsilon\}]$
$[L \to .i\,;\, \{=, \epsilon\}]$
$[R \to .L\,;\, \{\epsilon\}]$

M_1: $[S' \to S.\,;\, \{\epsilon\}]$

M_2: $[S \to L. = R\,;\, \{\epsilon\}]$
$[R \to L.\,;\, \{\epsilon\}]$

M_3: $[S \to R.\,;\, \{\epsilon\}]$

M_4: $[L \to {}^*.R\,;\, \{=, \epsilon\}]$
$[R \to .L\,;\, \{=, \epsilon\}]$
$[L \to .{}^*R.\,;\, \{=, \epsilon\}]$
$[L \to .i\,;\, \{=, \epsilon\}]$

M_5: $[L \to i.\,;\, \{=, \epsilon\}]$

M_6: $[S \to L = .R\,;\, \{\epsilon\}]$
$[R \to .L\,;\, \{\epsilon\}]$
$[L \to .{}^*R\,;\, \{\epsilon\}]$
$[L \to .i\,;\, \{\epsilon\}]$

M_7: $[L \to {}^*R.\,;\, \{=, \epsilon\}]$

M_8: $[R \to L.\,;\, \{=, \epsilon\}]$

M_9: $[S \to L = R.\,;\, \{\epsilon\}]$

M_{10}: $[R \to L.\,;\, \{\epsilon\}]$

M_{11}: $[L \to {}^*.R\,;\, \{\epsilon\}]$
$[R \to .L\,;\, \{\epsilon\}]$
$[L \to .{}^*R\,;\, \{\epsilon\}]$
$[L \to .i\,;\, \{\epsilon\}]$

M_{12}: $[L \to i.\,;\, \{\epsilon\}]$

M_{13}: $[L \to {}^*R.\,;\, \{\epsilon\}]$

□

V kapitole 2. bude uveden algoritmus pro konstrukci těchto množin $LR(1)$ položek. Lze vidět, že i pro tak malou gramatiku je počet množin $LR(1)$ položek velký. Množiny $LR(1)$ položek tvoří stavy charakteristického $LR(1)$ automatu. Pro středně velkou gramatiku by tak charakteristický $LR(1)$ automat mohl mít tisíce stavů nebo i víc, což je velice paměťově náročné. V praxi to znamená, že $LR(1)$ syntaktický analyzátor by klidně mohl mít stovky megabajtů. Například charakteristický $LR(1)$ automat jazyka C má asi 5000 stavů.

Z příkladu 1.2. lze vypozorovat, že některé množiny $LR(1)$ položek jsou velice podobné. Například M_4 a M_{11} mají stejné jádro, liší se pouze v množinách prediktivních symbolů svých položek. Stejně tak M_5 a M_{12}, M_7 a M_{13}, M_8 a M_{10} mají stejná jádra. Pokud ignorujeme množiny prediktivních symbolů, dostaneme $LR(0)$ položky. Počet stavů charakteristického $LR(0)$ automatu pro jazyk C by pak byl asi 350. Takový automat by ovšem měl spoustu konfliktů a nebyl by schopen analýzu provádět.

Gramatiky $LR(0)$, tedy gramatiky, pro které lze sestrojit $LR(0)$ automat bez konfliktů, vedou na mnohem menší $LR(0)$ automaty. Jsou ovšem velice slabé. Málokterá gramatika je vůbec $LR(0)$, natož nějaká reálná gramatika. Například ani miniaturní gramatika se dvěma pravidly $S \to x \mid \epsilon$ není $LR(0)$. Problém je v tom, že jsme odstranily množiny prediktivních symbolů, které nám říkají, že můžeme redukovat pomocí pravidla, ke kterému množina patří, pokud je na vstupu symbol, který v množině leží. Pokud tyto množiny nemáme, musíme se spolehnout čistě na zásobník a hádat, jestli přesouvat nebo redukovat, případně podle kterého pravidla redukovat, jen podle obsahu zásobníku, tedy toho, co jsme ve vstupu již viděli. V uvedené gramatice dojde ke konfliktu, zda přesunout x nebo redukovat podle $S \to \epsilon$. Přitom by se stačilo podívat na vstup a přesouvat v případě, že je na vstupu x, a následně redukovat pomocí $S \to x$, nebo pokud by byl vstup prázdný, tak rovnou redukovat pomocí $S \to \epsilon$. K tomu ale potřebujeme právě množiny prediktivních symbolů.

Na jednu stranu tedy máme silné, ale paměťově náročné $LR(1)$ automaty, na druhou stranu máme sice malé, ale slabé $LR(0)$ automaty. Nabízela by se tedy otázka, zda není něco mezi nimi. Mohli bychom třeba pro každý neterminál určit množinu terminálů, které po něm mohou v gramatice následovat, a redukovat podle pravidla $A \to \alpha$ pouze v případě, že na vstupu následuje symbol z množiny těchto terminálu pro neterminál A. Tato množina terminálů pro neterminál A se nazývá $FOLLOW(A)$. Pro uvedenou gramatiku s pravidly $S \to x \mid \epsilon$ by platilo $FOLLOW(S) = \{\epsilon\}$, takže bychom redukovali podle $S \to \epsilon$ nebo $S \to x$, jen pokud by byl vstup prázdný. To by vyřešilo konflikt přesun/redukce mezi pravidly $S \to x$ a $S \to \epsilon$. Pokud je na vstupu x, přesuneme jej, protože $x \notin FOLLOW(S)$, pokud je vstup prázdný, redukujeme.

Gramatiky, které tímto způsobem vedou na bezkonfliktní automat, se nazývají gramatiky $SLR(1)$. Gramatiky $SLR(1)$ jsou již mnohem silnější než gramatiky $LR(0)$ a přitom $SLR(1)$ automaty jsou stejně velké jako $LR(0)$. Písmeno „S“ v názvu znamená „simple“. Tento postup se skutečně pro jednoduché gramatiky

používá, jelikož je konstrukce $SLR(1)$ automatu jednoduchá.

Bohužel ani $SLR(1)$ gramatiky nejsou dostatečně silné. Například gramatika z příkladu 1.2. není $SLR(1)$ a přitom nevypadá složitě. Pokud na začátku vstupu odvodíme L, to znamená libovolný počet znaků * a i, můžeme buď přesunout $=$ nebo redukovat na R podle pravidla $R \to L$. Z gramatiky lze vidět, že redukovat bychom měli při prázdném vstupu a přesouvat pokud následuje $=$. Jedná se o kontext M_2, jenže zde nemáme k dispozici množinu prediktivních symbolů $LR(1)$ položky $[R \to L. \,; \{\epsilon\}]$, která nám radí redukovat jen při prázdném vstupu, ale množinu $FOLLOW(R) = \{=, \epsilon\}$, která radí redukovat, i když následuje $=$. Výpočet množiny $FOLLOW(R)$ je přitom správný, protože neterminál R se může objevit před $=$ skrze pravidla $S \to L = R$ a $L \to {}^*R$.

Zda redukovat podle $R \to L$ při prázdném vstupu nebo při $=$ na vstupu lze rozlišit tím, jestli neterminálu R přímo předcházel symbol * nebo ne. Pokud ano, pak jde o redukci před $=$, pokud ne, pak jde o redukci za $=$ nebo o samotné R skrze odvození $S \Rightarrow R$. Tento kontext, zda předcházel symbol * nebo ne, množina $FOLLOW(R)$ nezahrne, protože uvažuje R globálně kdekoliv v gramatice. Kontextu, ve kterém konflikt nastane, symbol * nepředchází, takže bychom redukovali jen při prázdném vstupu, jinak přesouvali.

Možné následující symboly ve vstupu v kontextu dosavadního odvozování zahrnují množiny prediktivních symbolů $LR(1)$ položek. Množina prediktivních symbolů $LR(1)$ položky s pravidlem $A \to \alpha$ je vlastně podmnožinou $FOLLOW(A)$, protože některé symboly z $FOLLOW(A)$ nemusí být v daném kontextu možné. Čím je tato množina menší, a tedy čím je v daném kontextu méně možností, tím menší je šance, že v tomto kontextu nastane konflikt. Množiny prediktivních symbolů $LR(1)$ položek jsou v tomto ohledu minimální, množiny $FOLLOW$ ale zdaleka ne.

S cílem snížení počtu množin $LR(1)$ položek jsme ignorovali množiny prediktivních symbolů a dostali tak $LR(0)$ položky. Ztratili jsme tak ale příliš mnoho informací a výsledek nebyl dobrý. Situaci jsme zlepšili zavedením množin $FOLLOW$, ale pořád to nebylo dostatečně dobré. Zkusme místo úplného ignorování množin prediktivních symbolů tyto množiny sloučit v množinách $LR(1)$ položek se stejným jádrem. Například můžeme sloučit množiny M_4 a M_{11}, tak že zanecháme jejich stejné jádro a $LR(1)$ položkám se stejným jádrem sloučíme množiny prediktivních symbolů. Dostaneme tak vlastně původní množinu M_4, protože množina M_{11} ji nijak nerozšířila. Všude, kde se předtím vyskytovala množina M_{11}, se teď bude vyskytovat množina M_4. Samozřejmě tím ztratíme nějakou informaci, konkrétně v kontextu M_{11} se v množinách prediktivních symbolů nevyskytoval symbol $=$, což sloučením „zapomeneme“. Jelikož slučujeme podmnožiny množiny $FOLLOW$, v nejhorším případě dostaneme celou množinu FOLLOW. Pokud ovšem sloučíme dva kontexty, ve kterých nebyl přítomen nějaký symbol, pak nebude přítomen ani ve sloučeném kontextu, takže tento postup je šetrnější ke zvětšování množin prediktivních symbolů a tedy šetrnější ke zvyšování šance konfliktů, než je tomu v případě $SLR(1)$ automatů. Počet kontextů

přitom zůstane stejný.

Gramatiky, pro které slučování množin $LR(1)$ položek vede k automatu bez konfliktů, nazýváme gramatiky $LALR(1)$. Písmena „LA“ jsou zkratkou „look ahead“. Automaty pro gramatiky $LALR(1)$, tedy automaty $LALR(1)$, jsou stejně velké jako $SLR(1)$ automaty. Gramatiky $LALR(1)$ jsou přitom mnohem silnější než gramatiky $SLR(1)$, ve skutečnosti nejsou o mnoho slabší než gramatiky $LR(1)$. Automaty $LALR(1)$ tak tvoří velice výhodný kompromis mezi velikostí a „chytrostí“ a v praxi jsou zdaleka nejpoužívanější. Například jazyk C je $LALR(1)$ a jeho charakteristický $LALR(1)$ automat má pouze asi 350 stavů oproti asi 5000 stavům charakteristického $LR(1)$ automatu.

Při slučování množin $LR(1)$ položek v příkladu 1.2., čímž jsme dostali množiny $LALR(1)$ položek, jsme nic nesloučili s množinou M_2, ve které měl $SLR(1)$ automat konflikt přesun/redukce, takže jsme do této množiny nezanesli konfliktní symbol $=$ a $LALR(1)$ automat tak „nezapomněl“, že v kontextu M_2 může redukovat jen při prázdném vstupu. Gramatika z příkladu 1.2. tedy je $LALR(1)$.

Obecně slučováním množin $LR(1)$ položek nemůžeme zavést konflikt přesun/redukce. Pokud by sloučená množina $LALR(1)$ položek měla konflikt přesun/redukce, pak tento konflikt byl přítomen alespoň v jedné množině $LR(1)$ položek, ze kterých sloučená $LALR(1)$ množina položek vznikla, protože při slučování zvětšujeme pouze množiny prediktivních symbolů, které se uplatňují při redukcích, jádra zůstává stejná. Tento konflikt přesun/redukce tedy musel být přítomen i v automatu $LR(1)$.

Nový konflikt redukce/redukce ale vzniknout může, jak uvádí [7]. Pokud například slučujeme množinu $LR(1)$ položek, ve které se redukuje podle pravidla p_1 při symbolu t_1 a podle pravidla p_2 při symbolu t_2, s množinou $LR(1)$ položek se stejným jádrem, ve které se redukuje podle pravidla p_1 při symbolu t_2 a podle pravidla p_2 při symbolu t_1, pak ve sloučené množině $LALR(1)$ položek redukujeme podle pravidla p_1 při symbolu t_1 i t_2 a podle pravidla p_2 při symbolu t_1 i t_2, což vede na nový konflikt redukce/redukce. Existují tedy gramatiky $LR(1)$, které nejsou $LALR(1)$. Jak ale bylo uvedeno v kapitole 1.2.2., konflikty redukce/redukce je obvykle snadné vyřešit zásahem do gramatiky.

▷ Příklad 1.3. Uvažujme gramatiku s následujícími pravidly:

1. $S \to Aa \mid Ab \mid Ba \mid Bb$
2. $A \to c$
3. $B \to c$

Tato gramatika není $LALR(1)$, přitom je $LR(1)$. Jde o konflikt redukce/redukce mezi pravidly $A \to c$ a $B \to c$, který vznikl sloučením množin $LR(1)$ položek. Konflikt lze vyřešit „dosazením“ terminálu c za neterminály A a B nebo nahrazením jednoho neterminálu druhým, jelikož jsou nerozlišitelné. □

Může se stát, že nepůjde sloučit žádný kontext. V tom případě dostaneme $LR(1)$ automat. Pokud lze slučovat, můžeme zavést nový konflikt redukce/redukce. I když nezavedeme nový konflikt, stále se chování automatu může změnit, protože jsme některé kontexty sloučily a mohli jsme je rozšířit. Vraťme se opět k příkladu 1.2. Zde jsme sloučili 4 dvojice množin $LR(1)$ položek a nezavedli žádný nový konflikt. $LR(1)$ automat pro chybný vstup $i = i =$ vykoná následující kroky:

1. přesun i
2. redukce podle pravidla $L \to i$
3. přesun $=$
4. přesun i
5. chyba, zde lze pouze redukovat podle $L \to i$ při prázdném vstupu, na vstupu je $=$

$LALR(1)$ automat pro stejný vstup vykoná následující kroky:

1. přesun i
2. redukce podle pravidla $L \to i$
3. přesun $=$
4. přesun i
5. redukce podle pravidla $L \to i$
6. redukce podle pravidla $R \to L$
7. chyba, zde lze pouze redukovat podle $S \to L = R$ při prázdném vstupu, na vstupu je $=$

$LALR(1)$ automat vykonal několik redukcí navíc, něž poznal, že vstup nelze z gramatiky odvodit. Sloučily jsme totiž množiny M_5 a M_{12}. Množina M_{12} tvořila kontext odvození L při posledním symbolu i na vstupu, kdežto množina M_5 tvořila kontext odvození L buď při posledním symbolu i na vstupu nebo když následoval symbol $=$. Tuto informaci ovšem automat $LALR(1)$ ztratil. Když se automat $LR(1)$ nacházel v pátém kroku v kontextu odvození L na konci vstupu, věděl, že na vstupu už nesmí nic následovat. $LALR(1)$ automat to ovšem nevěděl, protože byl ve sloučeném kontextu množin M_5 a M_{12}, což je kontext odvození L na konci vstupu nebo před symbolem $=$, což nastalo. Vstup odmítl až později.

Obecně automat $LALR(1)$ může provést několik redukcí navíc oproti automatu $LR(1)$, jelikož může mít některé kontexty obecnější a tím pádem může připustit více možností, ale nikdy se nepřesune dál než automat $LR(1)$. Vlastnost automatů $LR(1)$, která způsobuje, že zastavují na prvním chybném symbolu ve vstupu, tak zůstává i automatům $LALR(1)$.

1.5. Lexikální analýza

Programovací jazyky umožňují vytvořit si vlastní identifikátory, kterými se odkazujeme třeba na proměnou nebo funkci. V jejich jménech bývá povoleno velké množství znaků. Pro gramatiku by to znamenalo velké množství pravidel typu $Letter \rightarrow a \,|\, ... \,|\, z \,|\, A \,|\, ... \,|\, Z$, což by velmi zvětšovalo syntaktický analyzátor a zpomalovalo jeho činnost. Přitom ze syntaktického hlediska je například *myVariable* prostě identifikátor a je jedno, ze kterých písmen se skládá. Pro jazyky, které ve jménech identifikátorů povolují znaky Unicode, by gramatika mohla klidně mít milióny pravidel.

Z tohoto důvodu syntaktické analýze předchází lexikální analýza, která má za úkol přeložit posloupnosti znaků na symboly, se kterými se již pracuje v gramatice. Podobně se překládají i víceznakové terminály jazyka. Například v jazyce C se může klíčové slovo *while* přeložit na symbol `WHILE` a operátor == na symbol `EQUAL`. Přitom symbol `WHILE` a `EQUAL` je nutné skutečně vnímat jako jeden symbol, nejedná se již o posloupnost znaků jako v případě *while* nebo ==. Syntaktický analyzátor tak na vstup dostane jednoduchou posloupnost symbolů a je odstíněn od detailů jejich zápisu.

Nejčastěji se symboly realizují jako symbolické pojmenování čísel. Syntaktický analyzátor s čísly pracuje velmi efektivně a člověku se při zápisu pravidel gramatiky neztratí jejich význam. Mějme například kus kódu v jazyce C:

```
if (i == 0)
    printf("%d", i);
```

Po lexikální analýze by kód mohl vypadat jako tato posloupnost symbolů:

```
IF (IDENTIFIER EQUAL DEC_CONSTANT)
    IDENTIFIER(STRING_CONSTANT, IDENTIFIER);
```

Syntaktický analyzátor mimo jiné sestavuje datovou reprezentaci věty jazyka a k tomu samozřejmě potřebuje rozlišit jednotlivé identifikátory, konstanty a další elementy jazyka. Tuto dodatečnou informaci, že jde například o identifikátor *i* nebo identifikátor *printf*, má analyzátor k dispozici a do datové reprezentace tak například uloží dvojici (`IDENTIFIER`, *i*) nebo dvojici (`IDENTIFIER`, *printf*).

1.6. Aplikace LALR Acronym Loops Recursively

V této práci vytvořím GUI aplikaci, která se pro zadanou bezkontextovou gramatiku pokusí sestrojit syntaktický analyzátor. Během konstrukce zjistí, zda je zadaná gramatika typu $LALR(1)$. Pokud gramatika není $LALR(1)$, aplikace zobrazí konflikty mezi pravidly, které brání úspěšné konstrukci. Pokud gramatika je $LALR(1)$, aplikace umožní pro uživatelem zadaný vstup simulaci $LALR(1)$ syntaktické analýzy s výstupem činnosti analyzátoru. Aplikace by mohla být použita při studiu syntaktické analýzy zdola nahoru pro demonstraci analýzy.

2. Algoritmy

Konstrukce $LALR(1)$ automatu se skládá ze dvou kroků. Nejprve sestavíme množiny $LALR(1)$ položek a poté z nich vybereme informace nutné pro rozhodování automatu.

2.1. Konstrukce množin $LALR(1)$ položek

Množiny $LALR(1)$ položek sestavíme tak, že začneme od jedné počáteční množiny a pak z dosud vytvořených množin odvodíme další. Tyto množiny budou tvořit stavy charakteristického $LALR(1)$ automatu, který bude reprezentovat přechody mezi nimi.

Pro slučování $LR(1)$ položek v $LALR(1)$ metodě lze použít dva postupy. Buď nejprve vytvořit všechny množiny $LR(1)$ položek a poté sloučit ty se stejným jádrem nebo slučovat postupně v průběhu vytváření nových množin $LALR(1)$ položek. Zvolil jsem postup postupného slučování, protože nevyžaduje konstrukci velkého množství množin $LR(1)$ položek a je tedy efektivnější. Má ovšem také jednu nevýhodu. Jelikož budeme množiny slučovat „za běhu“, musíme pro každou sloučenou množinu znovu odvodit množiny z ní vzniklé a to rekurzivně, protože sloučení může změnit množiny prediktivních symbolů v položkách odvozených množin. Následující algoritmus jsem převzal z [6].

Algoritmus 2.1. (Konstrukce množin $LALR(1)$ položek)
Vstupem je expandovaná gramatika $G' = (N', T, P', S')$. *Výstupem je množina množin* $LALR(1)$ *položek* Q *a přechodová funkce* δ *charakteristického* $LALR(1)$ *automatu* $A = (T \cup N, Q, \delta, Closure(\{[S' \to .S\,;\,\{\epsilon\}]\}), F)$. *Algoritmus používá pomocnou množinu dosud nezpracovaných množin* $LALR(1)$ *položek* W.

1. *Na začátku definujeme* $Q \stackrel{def}{=} \{Closure(\{[S' \to .S\,;\,\{\epsilon\}]\})\}$ *a* $W \stackrel{def}{=} Q$. *Přechodová funkce* δ *není definovaná pro žádný bod definičního oboru. Opakujeme kroky 2 a 3, dokud platí* $W \neq \emptyset$.

2. *Zvolíme množinu položek* $M \in W$ *a provedeme* $W \stackrel{def}{=} W \setminus \{M\}$. *Pro každý symbol* $a \in N \cup T$ *vyskytující se v některé položce z* M *za tečkou označíme*

$$M_a = Closure(\{[A \to \alpha a.\beta\,;\,\omega] \mid [A \to \alpha.a\beta\,;\,\omega] \in M\}),$$

   tedy M_a *vznikne z* M *přesunem symbolu* a.

3. *Pro každou množinu* M_a *provedeme jednu z následujících možností:*

   (a) *Pokud pro nějaké* $M' \in Q$ *platí* $Ker(M') = Ker(M_a)$, *pak provedeme sloučení* $M_a \stackrel{def}{=} Merge(M_a, M')$. *Dále, pokud* $M_a \neq M'$, *pak nahradíme původní množinu* M' *novou množinou* M_a, *tedy* $Q \stackrel{def}{=} (Q \setminus \{M'\}) \cup \{M_a\}$,

$W \stackrel{def}{=} (W \setminus \{M'\}) \cup \{M_a\}$ *a ve všech bodech, kde má* δ *hodnotu* M'*, tuto hodnotu změníme na* M_a*. Původní množina* M' *zanikla.*

(b) Jinak provedeme $Q \stackrel{def}{=} Q \cup \{M_a\}$ *a* $W \stackrel{def}{=} W \cup \{M_a\}$*.*

Dále definujeme $\delta(M, a) \stackrel{def}{=} M_a$*.*

Začneme tedy od počáteční množiny $LALR(1)$ položek a nové množiny vytváříme z již existujících posouváním tečky v pravidlech a to pro každý symbol zvlášť. Pokud vytvoříme množinu, která má společné jádro s některou již existující, pak je sloučíme. Pokud sloučení způsobí rozšíření množiny prediktivních symbolů některé položky, pak musíme sloučenou množinu znovu zpracovat. Celé slučování, což je hlavním přínosem $LALR(1)$ metody, je pokryto bodem 3a.

Množina koncových stavů F charakteristického automatu obsahuje ty stavy z Q, které obsahují redukční položku. Množinu F ale přímo nepotřebujeme. Při vybírání potřebných informací ze stavů budeme potřebovat vědět, o kterou redukční položku konkrétně jde. Nestačí informace, že stav obsahuje nějakou redukční položku. Nemusíme tedy evidovat, které stavy patří do množiny F.

2.2. Rozkladová tabulka

Nyní jsme připraveni na syntaktickou analýzu. Ještě předtím je ale dobré si vybrat skutečně potřebné informace z množin $LALR(1)$ položek, protože bychom je během analýzy opakovaně procházeli a je neefektivní v nich neustále prohledávat položky pro informaci, zda přesouvat nebo redukovat a při kterém symbolu. Navíc položky s tečkou před neterminálem jsou pro syntaktickou analýzu již nepotřebné, hrály roli jen při konstrukci množin. Informace z množin $LALR(1)$ položek si tedy uložíme do rozkladové tabulky, ve které řádky reprezentují množiny a sloupce terminály včetně ϵ pro označení konce vstupu. V buňce tabulky je uložena informace o akci, kterou by automat měl při analýze provést v daném stavu pro daný terminál na vstupu. Následující algoritmus jsem převzal z [1].

Algoritmus 2.2. (Vytvoření rozkladové tabulky)
Vstupem algoritmu je expandovaná gramatika $G' = (N', T, P', S')$ *a množina množin* $LALR(1)$ *položek* Q*. Výstupem je rozkladová tabulka* RT *s řádky reprezentující stavy charakteristického automatu a sloupci reprezentující terminály z* $T \cup \{\epsilon\}$*. Obsahem buněk tabulky jsou akce* $LALR(1)$ *automatu. Pro každou množinu položek* $M \in Q$ *vykonáme následující kroky:*

1. *Pokud* $[A \to \alpha.t\beta] \in Ker(M)$*, kde* $t \in T$*, pak* $RT(M, t) \stackrel{def}{=}$ *přesun.*
2. *Pokud* $[S' \to S.] \in Ker(M)$*, pak* $RT(M, \epsilon) \stackrel{def}{=}$ *přijetí.*
3. *Pokud redukční položka* $[A \to \alpha. ; \omega] \in M$*, kromě speciální situace v bodě 2, a zároveň* $t \in \omega$*, pak* $RT(M, t) \stackrel{def}{=}$ *redukce podle pravidla* $A \to \alpha$*.*

4. *V ostatních případech* $RT(M,t) \stackrel{def}{=}$ *zamítnutí.*

Rozkladová tabulka tedy abstrahuje od $LALR(1)$ položek. Pokud jsme ve stavu M, na vstupu je symbol t a M obsahuje položku s tečkou před terminálem t, pak přesouváme. Pokud taková položka v M není a t patří do množiny prediktivních symbolů jedné redukční položky, pak redukujeme podle pravidla v této položce. Ve speciálním případě, kdy je vstup vyčerpaný a mohli bychom redukovat podle pravidla $S' \to S$, pak vstup přijmeme, protože jsme jej právě celý odvodili. Jde jen o speciální případ redukce, nicméně musíme zastavit analýzu, proto si zde uložíme akci přijetí. Pokud ani jedna z předchozích situací nenastala, pak jsme načetli symbol, který se v daném kontextu nemůže nacházet, a vstup tedy není z gramatiky odvoditelný, proto jej zamítneme.

Za povšimnutí stojí, že body 1 a 3 se vzájemně nevylučují. V tom případě by se ale jednalo o konflikt přesun/redukce. Podobně může bod 3 platit pro více různých redukčních položek. V tom případě ale nastal konflikt redukce/redukce. Pomocí rozkladové tabulky tedy lze snadno zjistit konflikty v automatu tak, že budeme předefinovávat již definovanou buňku v tabulce.

2.3. Syntaktická analýza

$LALR(1)$ automat se během syntaktické analýzy podle rozkladové tabulky rozhoduje co udělat, a podle přechodové funkce δ charakteristického $LALR(1)$ automatu přepíná kontext. Následující algoritmus jsem převzal z [1].

Algoritmus 2.3. (Syntaktická analýza $LALR(1)$ automatu)
Vstupem algoritmu je gramatika $G = (N, T, P, S)$*, její* $LALR(1)$ *automat, přechodová funkce* δ *charakteristického* $LALR(1)$ *automatu, rozkladová tabulka* RT *a řetězec* $w \in T^*$*. Výstupem je přijetí či zamítnutí řetězce* w*. Algoritmus čte po jednom symbolu ze vstupu, dokud vstup buď nepřijme nebo nezamítne. Aktuální symbol na vstupu označíme* t*. Pokud je vstup prázdný, pak je* $t = \epsilon$*. Aktuální vrchol zásobníku označíme top.*

1. *Na začátku je na zásobníku pouze* $Closure(\{[S' \to .S\,;\,\{\epsilon\}]\})$.
2. *Průběžný krok opakujeme, dokud buď nepřijmeme nebo nezamítneme* w.
   (a) *Pokud* $RT(top,t) =$ *přesun, pak na zásobník uložíme* $\delta(top,t)$ *a posuneme se ve vstupu na další symbol.*
   (b) *Pokud* $RT(top,t) =$ *redukce podle pravidla* $A \to \alpha \in P$*, pak ze zásobníku odebereme* $|\alpha|$ *prvků a poté na zásobník uložíme* $\delta(top,A)$.
   (c) *Pokud* $RT(top,t) =$ *přijetí, pak přijmeme řetězec* w.
   (d) *Jinak zamítneme řetězec* w.

3. Implementace

Pro implementaci aplikace jsem zvolil objektový přístup a jazyk C++. Pro implementaci GUI jsem zvolil Qt framework. Aplikaci jsem vytvořil ve vývojovém prostředí MS Visual Studio, ve kterém jsem také vytvořil instalátor pro operační systém MS Windows. Pro tyto prostředky jsem se rozhodl, protože s nimi mám nejvíce zkušeností a neměl jsem žádný silný důvod volit něco jiného.

Aplikaci jsem řešil multiplatformně a poskytuji verze pro operační systémy MS Windows a Linux.

3.1. Zadání gramatiky

Pro zadání gramatiky jsem zvolil Backus-Naurovu formu pro její rozšířenost. V této notaci se neterminály zapisují uzavřené mezi znaky `<` a `>` a terminály se zapisují uzavřené buď do uvozovek nebo apostrofů. Terminál může obsahovat libovolné znaky včetně bílých, pouze pokud obsahuje znak `"` nebo `'`, je nutné zvolit ten zápis, který nepoužívá tento znak. Symbol ϵ se zapisuje jako „prázdný terminál“ tedy jako `""` nebo `''`. Neterminál může obsahovat libovolný znak včetně bílých s výjimkou znaku `>` z důvodu nejednoznačnosti zápisu. Pravidlo v nejobecnější podobě vypadá následovně:

```
nonterminal ::= exp1 | exp2 | ... | expN
```

Část `nonterminal` značí neterminál na levé straně pravidla bezkontextové gramatiky. Levá a pravá strana pravidla je oddělena symbolem `::=`. Části `exp1`, `exp2`, ..., `expN` značí pravé strany pravidel tedy expanze neterminálu `nonterminal` a skládají se z posloupností terminálů a neterminálů. Symbol `|` odděluje jednotlivé expanze. Příkladem pravidla v Backus-Naurově formě by tedy mohlo být:

```
<výraz> ::= <výraz> '+' <výraz> | <výraz> '*' <výraz>
```

Gramatiku lze do aplikace zadat dvěma způsoby. Buď ručně přímo v aplikaci po jednotlivých pravidlech nebo načtením souboru, kde se na každém řádku vyskytuje jedno pravidlo a symbol `->` před neterminálem označuje počáteční symbol.

Při ručním zadávání se během psaní pravidla ověřuje, zda pravidlo „dává smysl“. Dokud pravidlu aplikace nerozumí, tlačítko pro přidání je neaktivní a aplikace tak neumožní pravidlo přidat. Cílem tohoto chování je uživateli naznačit, že pravidlo není v pořádku. Jinak je ale přidávání pravidel velmi shovívavé a jakmile pravidlo dává alespoň trochu smysl, aplikace jej přijme s tím, že ignoruje veškeré části, kterým nerozumí.

Seznam pravidel se chová jako množina pravidel a proto přidáním již existujícího pravidla nedojde k jeho duplikování. Navíc aplikace slučuje všechna pravidla

se stejnou levou stranou do jednoho složeného pravidla. Pokud tedy uživatel přidá pravidlo s levou stranou, pro kterou jsou již definována jiná pravidla, pravidlo se připojí na konec složeného pravidla.

3.2. Datová reprezentace

Základním stavebním kamenem je objekt reprezentující symbol. Ten se skládá ze jména symbolu a informace, zda se jedná o terminál či neterminál. Ze symbolů se skládá větná forma, což je posloupnost symbolů.

Pravidlo jsem reprezentoval jako dvojici skládající se ze jména symbolu na levé straně pravidla (v pravidlech bezkontextových gramatik je na levé straně vždy jen jeden neterminál) a větné formy na pravé straně pravidla. $LALR(1)$ položka se skládá z pravidla, pozice tečky a množiny prediktivních symbolů.

Objekt reprezentující gramatiku se skládá z množiny neterminálů, množiny terminálů, pole pravidel a mapování neterminálů na pravidla s tímto neterminálem na levé straně, pro rychlé zjištění pravidel s daným neterminálem na levé straně. Pravidla jsou očíslována podle svého indexu v poli. Gramatika je od svého vzniku expandovaná, proto není třeba udržovat počáteční symbol, protože ten je vždy S'.

Objekt reprezentující syntaktický analyzátor je asociován s gramatikou, pro kterou vznikl. Syntaktický analyzátor dále obsahuje množinu množin $LALR(1)$ položek, přechodovou funkci δ charakteristického $LALR(1)$ automatu, rozkladovou tabulku a množinu konfliktů. Množiny $LALR(1)$ položek jsou očíslovány a dále se pracuje jen s jejich čísly. Funkce δ je reprezentována jako mapování bodů aktivní domény funkce na její hodnoty. Konflikt je reprezentován dvojicí čísel pravidel, která jsou v konfliktu, a typem konfliktu.

Pokud se budeme dívat na buňky rozkladové tabulky s hodnotou zamítnutí jako na prázdné buňky, pak je tabulka řídká. V daném kontextu totiž lze přesouvat nebo redukovat typicky jen při několika málo symbolech na vstupu. Z tohoto důvodu je tabulka reprezentována jako mapování souřadnic neprázdných buněk na jejich hodnoty. Například pro jazyk C má tabulka asi 33 000 buněk, ale jen asi 6300 z nich jsou neprázdné. I když uložení hodnoty řídké tabulky zabere trojnásobek místa kvůli ukládání souřadnic, stále se ušetří podstatná část tabulky. Akce v tabulce jsou reprezentovány číslem. Číslo odpovídající některému pravidlu z gramatiky znamená redukci podle tohoto pravidla. Akci přesunu a přijetí jsou přiřazeny speciální hodnoty mimo tento rozsah.

3.3. Výpočet množin $FIRST$

Z důvodu efektivity jsem tento algoritmus rozdělil na dvě části, viz sekci 3.8.3. První část spočítá množinu $FIRST$ pro neterminál a druhá pro větnou formu tak, že výsledek složí z již známých výsledků pro neterminály vyskytující se v této větné formě. Následující algoritmus jsem převzal z [4].

Algoritmus 3.1. (Výpočet množiny $FIRST$ pro neterminál)
Vstupem algoritmu je gramatika $G = (N, T, P, S)$ a neterminál $A \in N$. Výstupem je množina $FIRST(A)$. Algoritmus používá pomocnou množinu W a pole příznaků computing pro neterminály. Prvky pole jsou na začátku všechny nastaveny na hodnotu $false$.

1. *Na začátku nastavíme $FIRST(A) \stackrel{def}{=} \emptyset$, $computing[A] \stackrel{def}{=} true$ a opakujeme kroky 2–4 pro každé pravidlo $A \to \alpha \in P$. Jakmile zpracujeme všechna pravidla, vrátíme zpět $computing[A] \stackrel{def}{=} false$ a algoritmus končí.*

2. *Nechť je $\alpha = s_1 \dots s_n$. Nastavíme $i \stackrel{def}{=} 0$ a $W \stackrel{def}{=} \{\epsilon\}$. Pokud je $n = 0$, pak pokračujeme krokem 4.*

3. *Nastavíme $i \stackrel{def}{=} i + 1$. Pokud je $s_i \in T$, pak $FIRST(s_i) = \{s_i\}$. Jinak pokud $FIRST(s_i)$ ještě nebylo spočítáno, pak jej buď podle tohoto algoritmu rekurzivně spočítáme, pokud $computing[s_i] = false$, nebo provedeme $FIRST(A) \stackrel{def}{=} FIRST(A) \cup (W \setminus \{\epsilon\})$ a pokračujeme krokem 2 pro další pravidlo, pokud $computing[s_i] = true$. Nastavíme $W \stackrel{def}{=} (W \setminus \{\epsilon\}) \cup FIRST(s_i)$. Opakujeme krok 3, dokud platí $i < n$ a zároveň $\epsilon \in W$.*

4. *Nastavíme $FIRST(A) \stackrel{def}{=} FIRST(A) \cup W$.*

Hledání prvků množiny $FIRST(A)$ je hledání všech terminálů, které se mohou objevit na začátku některého odvození z A, popřípadě ϵ, pokud z A lze odvodit ϵ. $FIRST(A)$ tedy vznikne sloučením $FIRST(\alpha)$ pro všechny větné formy α přímo odvoditelné z A. V rámci větné formy postupujeme po symbolech zleva doprava. Pokud je aktuální symbol terminál, pak jej přidáme do výsledku a s touto větnou formou jsme skončili, protože nás zajímá pouze první neterminál. Pokud je aktuální symbol neterminál, pak na něj použijeme algoritmus rekurzivně a výsledek přidáme do celkového výsledku až na případné ϵ. Pokud by výsledek aktuálního neterminálu obsahoval ϵ, musíme pokračovat dál, protože z tohoto neterminálu lze odvodit ϵ. Pokud takhle dojdeme až na konec větné formy a neskončily jsme ani na posledním symbolu, což tedy znamená, že po cestě byly samé neterminály, z kterých lze odvodit ϵ, pak i z celé větné formy lze odvodit ϵ.

Při výpočtu množin $FIRST$ pro vnitřní neterminály by se mohl algoritmus zacyklit, pokud bychom rekurzivně počítali něco, co již máme rozpočítané. Například pro gramatiku s pravidly $S \to AS \mid a$ a $A \to a \mid \epsilon$ jsou všechny věty odvoditelné z S tvořeny libovolným nenulovým počtem symbolů a, proto $FIRST(S) = \{a\}$. Gramatika ale obsahuje rekurzi a algoritmus by se mohl zacyklit. Proto si výpočet pro neterminál označíme a kdykoliv bychom jej potřebovali rekurzivně provést znovu pro stejný neterminál, tak uložíme dosud spočítanou část kromě případného ϵ a výpočet ořízneme, protože odtud již nemůžeme spočítat nic nového.

Algoritmus 3.2. (Výpočet množiny $FIRST$ pro větnou formu)
Vstupem algoritmu je gramatika $G = (N, T, P, S)$, větná forma $\alpha = s_1 \dots s_n \in (N \cup T)^*$ a množiny $FIRST(s_i)$ pro všechna $i = 1 \dots n$, kde $s_i \in N$. Výstupem je množina $FIRST(\alpha)$.

1. *Na začátku nastavíme $FIRST(\alpha) \stackrel{def}{=} \{\epsilon\}$ a $i \stackrel{def}{=} 0$. Pokud je $n = 0$, pak algoritmus končí.*

2. *Nastavíme $i \stackrel{def}{=} i + 1$. Pokud je $s_i \in T$, pak $FIRST(s_i) = \{s_i\}$. Nastavíme $FIRST(\alpha) \stackrel{def}{=} (FIRST(\alpha) \setminus \{\epsilon\}) \cup FIRST(s_i)$. Opakujeme krok 2, dokud platí $i < n$ a zároveň $\epsilon \in FIRST(s_i)$.*

Množinu $FIRST$ pro větnou formu tedy spočítáme tak, že postupně projdeme symboly větné formy zleva doprava a přidáme do ní obsah již známých výsledků pro neterminály a pokračujeme, pokud jsme přidali i ϵ, nebo přidáme aktuální terminál a skončíme. V každém kroku z výsledku odebíráme ϵ, protože pokud jsme jej v předchozím kroku přidali, pak výpočet nekončí a o výsledku rozhodne aktuální symbol.

3.4. Výpočet uzávěru množiny $LALR(1)$ položek

Algoritmus 3.3. (Výpočet uzávěru množiny $LALR(1)$ položek)
Vstupem algoritmu je gramatika $G = (N, T, P, S)$ a množina $LALR(1)$ položek M. Výstupem je množina $Closure(M)$. Algoritmus používá pomocnou množinu dosud nezpracovaných $LALR(1)$ položek W.

1. *Na začátku nastavíme $Closure(M) \stackrel{def}{=} M$ a $W \stackrel{def}{=} M$. Opakujeme kroky 2 a 3, dokud platí $W \neq \emptyset$.*

2. *Zvolíme $LALR(1)$ položku $r \in W$ a nastavíme $W \stackrel{def}{=} W \setminus \{r\}$. Označíme symbol za tečkou v r jako B. Pokud je tečka v r na konci pravidla, pak je tímto symbolem ϵ. Pokud $B \in T \cup \{\epsilon\}$, pak opakujeme krok 2 pro další položku.*

3. *Položka r je tedy tvaru $[A \to \alpha.B\gamma\,;\,\omega]$, kde $B \in N$ a $\alpha, \gamma \in (N \cup T)^*$. Zavedeme novou množinu prediktivních symbolů $\kappa(\gamma, \omega) \stackrel{def}{=} FIRST(\gamma)$. Pokud $\epsilon \in \kappa(\gamma, \omega)$, pak upravíme $\kappa(\gamma, \omega) \stackrel{def}{=} (\kappa(\gamma, \omega) \setminus \{\epsilon\}) \cup \omega$. Pro každé pravidlo tvaru $B \to \beta \in P$ vytvoříme novou $LALR(1)$ položku $p \stackrel{def}{=} [B \to .\beta\,;\,\kappa(\gamma, \omega)]$ a provedeme jeden z následujících kroků:*

   (a) *Pokud existuje $LALR(1)$ položka $p' \in Closure(M)$ taková, že $Ker(p) = Ker(p')$, pak provedeme $LA(p) \stackrel{def}{=} LA(p) \cup LA(p')$ a $Closure(M) \stackrel{def}{=} (Closure(M) \setminus \{p'\}) \cup \{p\}$. Dále, pokud platí jedna z následujících podmínek:*

i. $p' \in W$

ii. $LA(p') \subset LA(p)$ *a zároveň* $\beta = C\delta$, *kde* $C \in N$ *a* $\delta \in (N \cup T)^*$, *a zároveň* $\epsilon \in FIRST(\delta)$

pak provedeme $W \stackrel{def}{=} (W \setminus \{p'\}) \cup \{p\}$.

(b) Jinak provedeme $Closure(M) \stackrel{def}{=} Closure(M) \cup \{p\}$ *a* $W \stackrel{def}{=} W \cup \{p\}$.

Uzávěr množiny $LALR(1)$ položek tedy vytvoříme tak, že postupně procházíme položky a odvozujeme z nich položky nové. Odvození proběhne tak, že z položky s tečkou před neterminálem vytvoříme pro každé pravidlo s tímto neterminálem na levé straně položku s tečkou na začátku pravé strany tohoto pravidla a patřičně upravíme množinu prediktivních symbolů. Pokud jsme takto vytvořili položku s dosud unikátním jádrem, pak ji musíme také projít. Pokud nově vytvořená položka má společné jádro s některou již existující položkou, pak sloučíme jejich množiny prediktivních symbolů a nahradíme původní položku nově vytvořenou. Uzávěr množiny $LALR(1)$ položek bude tedy obsahovat jen položky s unikátními jádry. Pokud jsme některou položku sloučili, může to mít vliv na položky z ní odvozené, proto musíme sloučenou položku znovu projít. Nemusíme ji ovšem znovu procházet, pokud sloučením nedošlo k rozšíření množiny prediktivních symbolů původní položky nebo pokud v této položce nenásleduje neterminál po tečce a tedy z ní stejně nic nového nelze odvodit nebo pokud se nové symboly v množině prediktivních symbolů nemohou dostat do množin prediktivních symbolů $\kappa(\gamma, \omega)$ odvozených položek.

Ve své implementaci řadím $LALR(1)$ položky podle jejich jader. Dvě položky lišící se pouze v množině prediktivních symbolů jsou tedy z hlediska řazení považovány za ekvivalentní. Po odvození nové položky se ji pokusím vložit do množiny uzávěru. Pokud se položka do množiny vložila, jde o položku s dosud unikátním jádrem. Jinak v množině již je položka se stejným jádrem. Místo vytváření sloučené položky a následné nahrazování původní položky přímo přidám symboly množiny prediktivních symbolů odvozené položky do množiny prediktivních symbolů původní položky. Množinu dosud nezpracovaných položek W reprezentuji pomocí množiny ukazatelů na tyto položky, proto se případná změna při slučování projeví i v množině W. Nemusím tedy při slučování provádět nahrazování původních položek v množině uzávěru a množině W. Algoritmus jsem popsal převážně konstruktivně pro lepší pochopení.

3.5. Konstrukce množin $LALR(1)$ položek

Ve své implementaci algoritmu 2.1. si před konstrukcí množin M_a namapuji položky na symbol následující za jejich tečkou a následně toto mapování projdu a pro každý symbol a vytvořím množinu M_a jako uzávěr položek namapovaných na tento symbol. Nově vytvořenou množinu M_a rovnou zpracuji.

Dále se algoritmus strukturou velmi podobá algoritmu pro výpočet uzávěru množiny $LALR(1)$ položek. Proto jsou i mé implementace těchto algoritmů velmi podobné. Množiny jsou řazeny podle svých jader. Dvě množiny lišící se pouze v množinách prediktivních symbolů svých položek jsou tedy z hlediska řazení považovány za ekvivalentní. Pokud se tedy množina při pokusu o vložení do množiny Q nevloží, sloučím ji s již existující množinou se stejným jádrem. Množina Q tedy obsahuje pouze množiny s unikátním jádrem. Slučování množin opět provádím přímou úpravou původní množiny tak, že do množin prediktivních symbolů $LALR(1)$ položek přidám symboly z odpovídajících položek nově vzniklé množiny. V množině W uchovávám čísla dosud nezpracovaných množin. Funkce δ pracuje také s čísly množin. Znovu mi tak při slučování odpadá údržba množin W a Q a funkce δ.

3.6. Detekce konfliktů

Konflikty detekuji při konstrukci rozkladové tabulky. Pro každou buňku tabulky si pamatuji všechna pravidla, která vedla na definici této buňky, včetně informace, zda šlo o přesun nebo redukci. Po dokončení konstrukce rozkladové tabulky zkontroluji, zda pro některou buňku nebylo definováno více hodnot. V případě více hodnot pro jednu buňku sestavím konflikty a informuji o nich uživatele aplikace.

3.7. Okno automatu

Okno automatu je nemodální. Aplikace tak umožňuje vytvořit více automatů pro stejnou gramatiku nebo více automatů pro různé gramatiky. Chtěl jsem tak uživateli umožnit přímo srovnávat činnost dvou stejných automatů pro různý vstup nebo činnost dvou různých automatů. Proto obsahuje okno automatu gramatiku, pro kterou byl automat zkonstruován.

Po vyhodnocení vstupu je v případě přijmutí sestaveno odvození vstupu z gramatiky. Toto odvození ale může být velmi paměťově i časově náročné. Proto pro činnost automatu s příliš velkým počtem kroků aplikace uživatele varuje o náročnosti sestrojení tohoto odvození a dá mu na výběr, zda jej i přesto chce sestrojit.

3.8. Optimalizace

Po první „intuitivní“ implementaci konstrukce $LALR(1)$ automatu jsem zjistil, že je konstrukce velmi pomalá. Pro testovací gramatiku, která vede na asi o 30% složitější automat než pro gramatiku jazyka C, trvala konstrukce 130 vteřin. Musel jsem tedy optimalizovat. Konstrukce pro testovací gramatiku trvala po optimalizacích 1 vteřinu.

Při hledání úzkého místa aplikace jsem použil profilovací program vývojového prostředí MS Visual Studio. Jak se ale nakonec ukázalo, zdaleka nešlo jen o jedno nebo dvě slabší místa. Celkem jsem provedl asi 10 optimalizací, které až dohromady daly dobrý výsledek. Některé z nich se téměř neprojevily, jiné se naopak projevily hodně, nicméně většina z nich zrychlila konstrukci o 5–10%.

Optimalizace spočívaly zejména ve snižování počtu volání procedur a ve změně datové reprezentace, se kterou lze pracovat rychleji. Následuje popis významnějších optimalizací.

3.8.1. Očíslování symbolů

Profilovací program nejprve ukázal, že nejvíce času se tráví při vkládání nových $LALR(1)$ položek do množin $LALR(1)$ položek při výpočtu jejich uzávěrů a při vkládání množin $LALR(1)$ položek do množiny těchto množin. Množiny mám ovšem reprezentovány pomocí kontejneru s logaritmickou složitostí všech operací a „rychlejší“ kontejner použít nelze, jelikož v nich potřebuji neustále vyhledávat podle jader položek nebo jader množin položek. Musel jsem tedy zrychlit samotnou operaci, která se při vkládání vykoná v logaritmickém počtu závislém na velikosti kontejneru. Touto operací bylo porovnání jader $LALR(1)$ položek, které se převážně skládá z porovnávání symbolů v jejich pravidlech.

V původní implementaci jsem přímo v symbolu uchovával řetězec, který reprezentoval název symbolu v případě neterminálu nebo hodnotu symbolu v případě terminálu. Přitom během konstrukce $LALR(1)$ automatu není potřeba znát přímo obsah těchto řetězců, je potřeba jen od sebe rozlišit různé symboly. Proto jsem symboly očísloval. Skutečný obsah symbolů zná jen gramatika. Pravidla, se kterými se dále pracuje, obsahují jen tyto „číselné“ symboly. Když je třeba znát skutečnou podobu pravidla pro prezentaci uživateli aplikace, gramatika jej přeloží.

Zredukoval jsem tak porovnávání řetězců na porovnávání čísel. Řetězce byly navíc v kódování Unicode. I za předpokladu velmi krátkých jmen symbolů bylo možné předpokládat alespoň trojnásobné zrychlení. Počet těchto porovnávání se navíc pohyboval v řádech statisíců. Přesto se optimalizace téměř neprojevila.

3.8.2. Čísla pravidel v $LALR(1)$ položkách

Jelikož se předchozí optimalizace téměř neprojevila i přes tak vysoký počet výskytů, bylo zřejmé, že problém spíše spočíval v počtu porovnání symbolů než v rychlosti tohoto porovnání. Symboly se porovnávají hlavně při porovnávání pravidel, u kterého opět není důležitý samotný obsah pravidla, je potřeba jen rozlišit různá pravidla. V $LALR(1)$ položkách jsem ukládal celá pravidla. Přitom pravidla byla od začátku očíslována kvůli odkazu na pravidlo v rozkladové tabulce při redukci, což jsem přehlédl. Toto přehlédnutí jsem tedy napravil ukládáním čísla pravidla v $LALR(1)$ položkách místo celého pravidla.

Zredukoval jsem tak porovnávání pravidel a tedy porovnávání více symbolů na porovnávání čísel. Při konstrukci je ale potřeba znát i obsah pravidla. Často je potřeba se podívat na symbol za tečkou. Pravidla jsou ovšem v gramatice uložena v poli a číslo pravidla je indexem do tohoto pole, proto je zpomalení při tomto „nahlédnutí“ do pole pravidel zanedbatelné oproti zredukování počtu porovnávání symbolů. Tato optimalizace zrychlila konstrukci asi o 15%.

Mohlo by se zdát, že očíslování symbolů po tomto využití čísel pravidel v $LALR(1)$ položkách ztratilo smysl. I když je to částečně pravda, stále se symboly objevují v množinách prediktivních symbolů, v argumentech přechodové funkce a souřadnicích rozkladové tabulky, kde optimalizace zrychluje vkládání a vyhledávání hodnot.

3.8.3. Předpočítání množin $FIRST$ pro neterminály

Po předchozí optimalizaci trávila konstrukce nejvíce času výpočtem množin $FIRST$. Tyto výpočty je nutné provádět během výpočtu uzávěru množiny $LALR(1)$ položek. Původně jsem počítal množiny $FIRST$ pro každou větnou formu vždy od začátku. Z algoritmu je patrné, že se během tohoto výpočtu počítají množiny $FIRST$ pro neterminály rekurzivně do hloubky. Přitom se během celé konstrukce může stát, že se množina $FIRST$ pro některý neterminál spočítá více než tisíckrát, což je velmi neefektivní. Proto před samotnou konstrukcí nejdříve spočítám množiny $FIRST$ pro všechny neterminály a výsledek si uložím. Množiny $FIRST$ pro větné formy potom stačí spočítat kombinací výsledků pro neterminály nacházející se v těchto větných formách. Zredukoval jsem tak potenciálně velmi hluboké rekurze na jejich první úroveň. Tato optimalizace zrychlila konstrukci asi dvojnásobně.

V pamatování množin $FIRST$ by šlo jít dál a uložit si ji pro každé jádro $LALR(1)$ položky, protože i stejná jádra se mohou vyskytnou na více místech. Počet unikátních jader je ale mnohem větší než počet neterminálů, protože pro každý neterminál bude přítomno alespoň jedno pravidlo a pro pravidlo bude existovat více pozic teček. Naopak počet výskytů jednoho jádra je mnohem menší něž počet výskytů jednoho neterminálu, protože výskyt jednoho jádra bude typicky znamenat výskyt více neterminálů. Navíc pro jedno pravidlo bude existovat více jader lišících se jen v pozici tečky, takže i přesto, že jádra budou jiná, neterminály v nich budou stejné. I když jsem to nezkoušel, myslím si, že by tato další optimalizace nepřinesla velké zrychlení a naopak by velmi zvýšila paměťovou náročnost.

3.8.4. Implicitní sdílení množin prediktivních symbolů

Po předchozí optimalizaci trávila konstrukce nejvíce času při plnění množin prediktivních symbolů. Tyto množiny se plní při slučování položek se stejnými jádry, při odvozování nových položek při výpočtu uzávěrů množin a při odvození nových množin $LALR(1)$ položek. Kromě slučování položek se přitom jedná

o kopie celých množin. Při výpočtu uzávěru množin z jedné položky odvodíme obecně více položek, ale všechny budou mít stejnou množinu prediktivních symbolů. Stejně tak nové množiny položek vznikají posunutím tečky v položkách, čímž vznikne nová položka, která ale má pořád stejnou množinu prediktivních symbolů.

Nabízí se tedy tyto množiny sdílet. Samozřejmě se mohou později množiny změnit vlivem slučování, ale měřením jsem zjistil, že až 90% procent množin bylo kopií jiné množiny. Sdílení jsem implementoval jako implicitní, množiny se tedy sdílí tak dlouho, jak jen to je možné. Každá množina má počítadlo položek, které ji sdílí. Položka, která bude sdílenou množinu potřebovat upravit, si ji zkopíruje a sníží počítadlo sdílení původní množiny. Tato optimalizace zrychlila konstrukci asi čtyřnásobně.

3.8.5. Množina prediktivních symbolů jako bitové pole

Nyní konstrukce trávila nejvíce času ve slučování množin prediktivních symbolů. Množiny jsem měl opět implementovány pomocí kontejneru s logaritmickými složitostmi operací. Symboly jsou ovšem očíslovány a jejich očíslování tvoří souvislou posloupnost. V množinách prediktivních symbolů tak ve skutečnosti ukládám, která čísla z nějakého rozsahu jsou v množině přítomna. Navíc počet terminálů je po vytvoření gramatiky znám a během konstrukce se nemění. Dalo by se tedy místo této množiny terminálů ukládat pole vlajek, kde i-tá pozice značí, zda se i-tý terminál z rozsahu v množině nachází. Sice bych tak v každé množině zbytečně ukládal i informaci o tom, že tam některý terminál není, ale na každý terminál mi stačí pouze jeden bit.

Počet terminálů v reálných gramatikách většinou nepřeroste 128. Například gramatika pro jazyk C obsahuje 83 terminálů. Pro reprezentaci množiny pomocí bitového pole mi tak bude typicky stačit 20 bajtů, 16 bajtů pro samotné pole bitů a 4 bajty pro ukazatel na toto pole. V předchozí implementaci množin by výskyt již dvou terminálů zabral nejméně 28 bajtů. Každý další neterminál by navíc zabral dalších 12 bajtů. Kontejnery s logaritmickými operacemi se většinou implementují jako binární stromy. Každý uzel stromu tedy má dva 4-bajtové ukazatele a samotnou hodnotu. I kdybych hodnotu omezil na jeden bajt, což by gramatiku limitovalo na maximálně 256 terminálů, by uzel pravděpodobně zabral 12 bajtů z důvodu zarovnávání objektů v paměti na násobky čtyř. Situace by dopadla ještě hůře na 64-bitové architektuře.

Co je ale zajímavější než ušetřené místo je zrychlení operací. Vyhledání symbolu v množině znamená otestování bitu s odpovídajícím indexem, což je konstantní složitost oproti logaritmické. Při slučování množin stačí množiny bitově sečíst. Pro většinu gramatik s počtem terminálů do 128 se tak celé sloučení provede na 4 bitové součty, popřípadě na 2 bitové součty na 64-bitové architektuře. Přitom nezáleží na velikosti množiny. Naopak v původním řešení se slučování dělo vkládáním do stromu a jeho případným vyvažováním a počet těchto vklá-

dání závisel na velikosti množiny. Podobně kopírování množin typicky vede na 4 respektive 2 přiřazení bez ohledu na velikost množiny. Pouze průchod množinou by mohl být pomalejší, jelikož se musí projít všechny terminály a testovat, zda jsou v množině přítomny, oproti projití jen těch terminálů, které v množině skutečně jsou. I když i to je sporné, jelikož množinou v původním řešení se prochází podle seřazení, což není zrovna nejefektivnější průchod stromem. Navíc projít množiny je potřeba jen jednou při konstrukci rozkladové tabulky, kdežto kopírování a slučování se provádí takřka neustále. Tato optimalizace měla na konstrukci největší vliv, zrychlila ji asi osminásobně.

Stejným způsobem reprezentuji i množiny $FIRST$, jelikož množiny prediktivních symbolů z nich vznikají. Dalšího zrychlení se tak dosáhne při výpočtu uzávěru množiny $LALR(1)$ položek v podmínce, zda se nové symboly při sloučení mohou dostat do odvozených položek. Ověření této podmínky se skládá z hledání ϵ v množinách $FIRST$ pro neterminály vyskytující se v pravidle a tato operace se provede v konstantním čase oproti logaritmickému v původním řešení. Platnost podmínky, zda při slučování došlo k rozšíření množiny prediktivních symbolů, lze stanovit porovnáním počtu nastavených bitů před a po sloučení.

Optimalizace očíslování symbolů vzhledem k dosaženému výsledku možná vypadala zbytečně, nicméně jen díky ní jsem mohl reprezentovat množiny terminálů jako bitové pole, protože tato reprezentace vyžaduje, aby terminály tvořily souvislou posloupnost čísel, což řetězce nesplňují.

Implicitní sdílení se zachovalo, jelikož jsem pro bitové pole použil kontejner z Qt framework, který je implicitně sdílen. Je ovšem otázkou, zda je v tomto případě sdílení ku prospěchu, jelikož samotná režie sdílení je pravděpodobně pomalejší než okopírování těch pár bajtů a místa se také asi moc neušetří.

3.9. Testování

Konstrukce $LALR(1)$ automatu je poměrně složitý proces a kvůli složitým datovým typům se špatně ladí. Navíc chyba v kterémkoliv kroku konstrukce se projeví až při samotné syntaktické analýze. Z tohoto důvodu jsem si v průběhu vývoje navrhl 6 testů, které by během konstrukce měly odhalit všechny základní chyby a některé složitější. Kdykoliv jsem během vývoje narazil na „zákeřnou“ chybu, vytvořil jsem pro ní test, aby byla v budoucnu odhalena, pokud by se znovu objevila. Tyto testy kontrolují výpočet množin $FIRST$, konstrukci množin $LALR(1)$ položek, konstrukci rozkladové tabulky, správnou detekci konfliktů a syntaktickou analýzu. Testovací gramatiky jsou z důvodu nutnosti definice mezivýsledků v testech malé, ale snažil jsem se, aby pokryly co největší rozsah chyb, proto jsou některé z nich poměrně složité.

Jako vzory korektních výsledků jsem použil některé řešené příklady z [1] a zejména výstup aplikace ParsingEmu [10] a programu Bison [9]. ParsingEmu zobrazuje jednotlivé mezivýsledky konstrukce, což mi velice pomohlo i při samotném pochopení algoritmů. Bison pro konstrukci generuje velmi detailní výstup.

Pro realizaci testování jsem zvolil primitivní způsob založený na podmíněném překladu a aserci. Definicí symbolické konstanty se aplikace přeloží v testovacím režimu, ve kterém jsou dostupné některé dodatečné metody a data pro účely testování. V tomto režimu do aplikace načtu testovací gramatiky v pevně daném pořadí a aplikace postupně spustí jednotlivé testy. Testy se skládají z mnoha asercí, takže když některá podmínka neplatí, aplikace se v místě této nesplněné podmínky zastaví. Pokud aplikace projde přes všechny testy, pak testy uspěly.

Systém testování jsem navrhoval na začátku vývoje a neplánoval jsem jej v takovém rozsahu, v jakém nakonec dopadl, proto jsou testy poměrně neohebné a mnohé změny v implementaci konstrukce vyžadují rozsáhlé úpravy testů. Hlavním problémem je závislost testů na datové reprezentaci konstrukce.

Pro testování výkonu konstrukce jsem si na internetu našel a upravil některé gramatiky reálných programovacích jazyků, které byly dostupné v syntaxi pro program Bison. Na převod do zápisu gramatiky používaného mojí aplikací jsem si napsal pomocný program. Na otestování syntaktické analýzy pro delší vstup jsem použil svůj starý zdrojový kód.

4. Výsledek

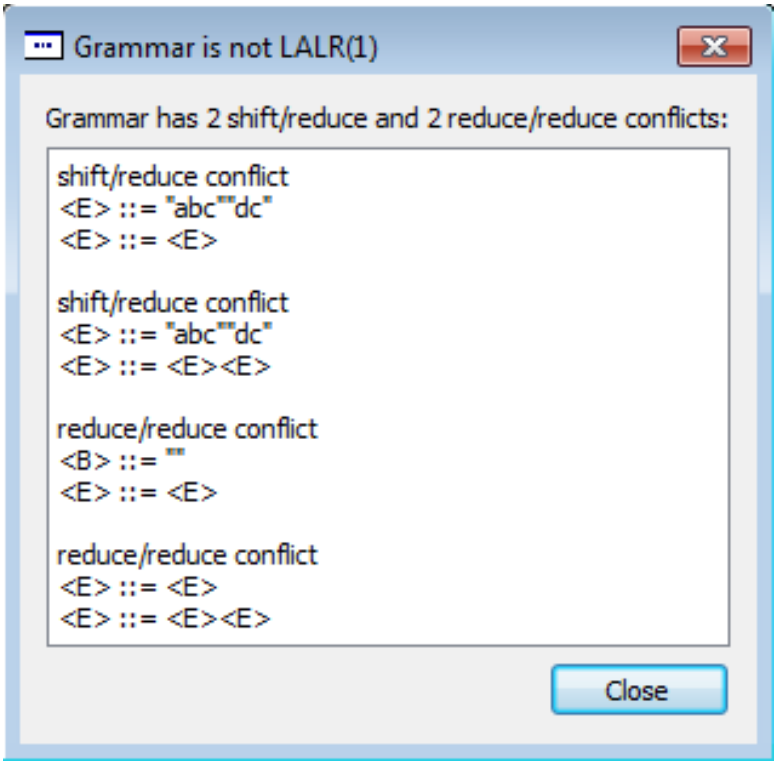
V této práci jsem vytvořil aplikaci, která se pro vstupní bezkontextovou gramatiku pokusí sestrojit syntaktický analyzátor. V případě úspěchu aplikace umožňuje simulovat jeho činnost, jinak jsou uživateli předloženy konflikty a jejich typ.

Gramatiku lze zadat načtením souboru nebo ručně přímo v aplikaci. Obě metody používají Backus-Naurovu formu pro zápis pravidel. Po zadání gramatiky je umožněna její editace, založená na přidávání a odebírání pravidel. Provedené změny lze vracet nebo vrácené změny znovu provést. Upravenou gramatiku lze uložit.

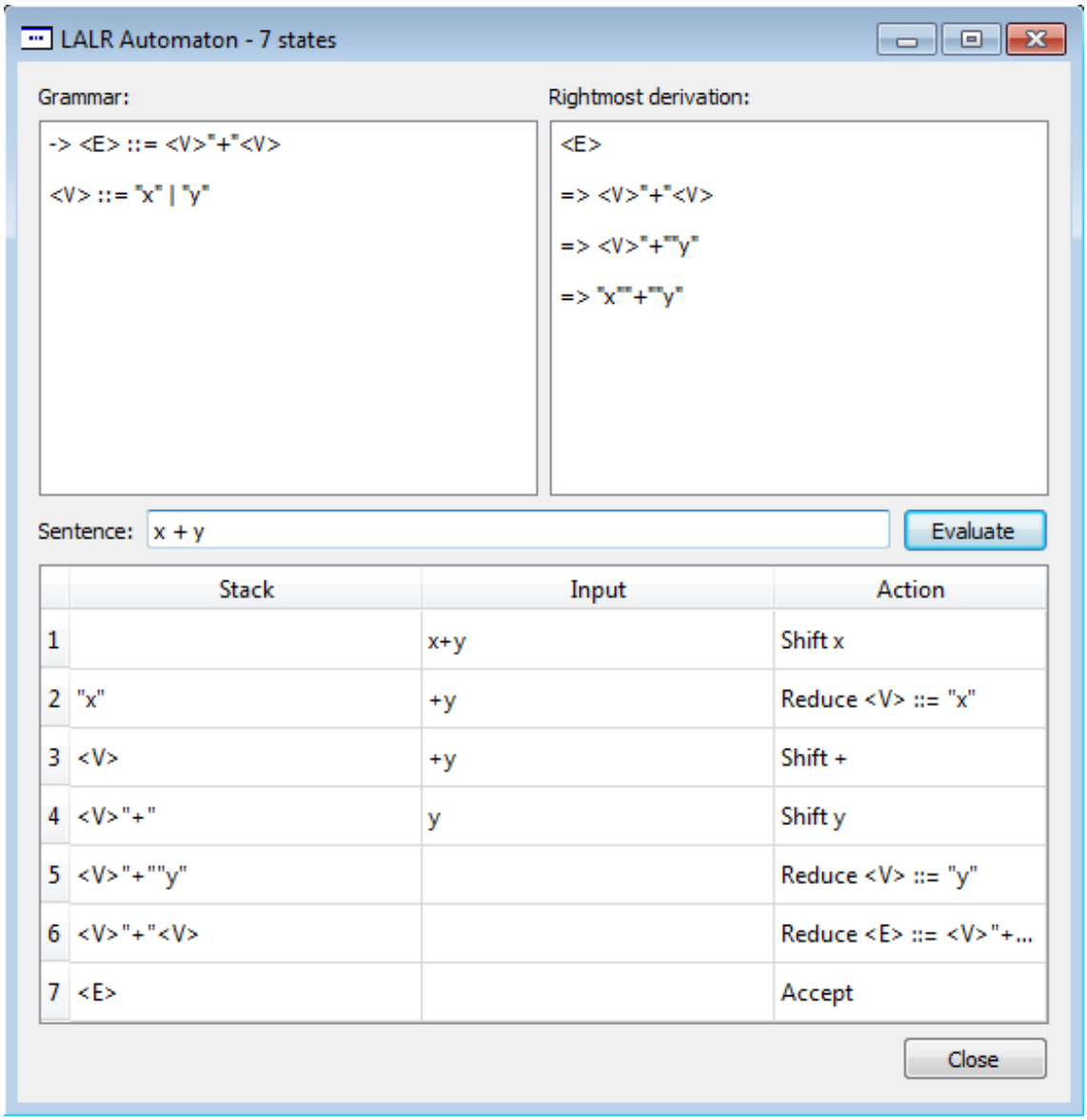
Aplikace umožňuje simulaci činnosti syntaktického analyzátoru pro libovolný vstup. Simulace činnosti je tvořena tabulkou, ve které každý řádek reprezentuje jeden krok analyzátoru. Tabulka obsahuje tři sloupce zobrazující obsah zásobníku, vstup a akci, pro kterou se analyzátor rozhodl. V případě přijetí vstupu je z činnosti analyzátoru sestrojeno nejpravější odvození věty z gramatiky.

4.1. Ukázky výstupu

Na obrázku 2. je zachyceno ohlášení konfliktů pro gramatiku, která není $LALR(1)$. Na obrázku 3. je zobrazena činnost automatu pro jednoduchý vstup.



Obrázek 2. Ohlášení konfliktů



Obrázek 3. Simulace činnosti automatu

4.2. Uživatelská příručka

4.2.1. Zadání gramatiky

Gramatiku lze zadat načtením ze souboru nebo ručním zadáním v okně pro úpravu gramatiky, které je zobrazeno na obrázku 4. Pravidla gramatiky se zadávají v Backus-Naurově formě. Jednotlivé části okna mají následující význam:

1 – Textové pole pro zadání levé strany pravidla
Očekávaným vstupem je jméno neterminálu uzavřené mezi znaky < a >.

2 – Textové pole pro zadání pravé strany pravidla
Očekávaným vstupem je posloupnost terminálů a neterminálů. Neterminály se zapisují uzavřené mezi znaky < a >. Terminály se zapisují uzavřené do uvozovek nebo apostrofů. Jednoznakové terminály (kromě bílých znaků) nemusejí být uzavřené do uvozovek nebo apostrofů. Terminál ϵ lze zapsat jako terminál s prázdným jménem, tedy jako "" nebo ''. Znak | odděluje více pravých stran zadaných pro stejný neterminál na levé straně pravidla. Bílé znaky se ve vstupu ignorují, pokud nejsou součástí terminálu nebo jména neterminálu.

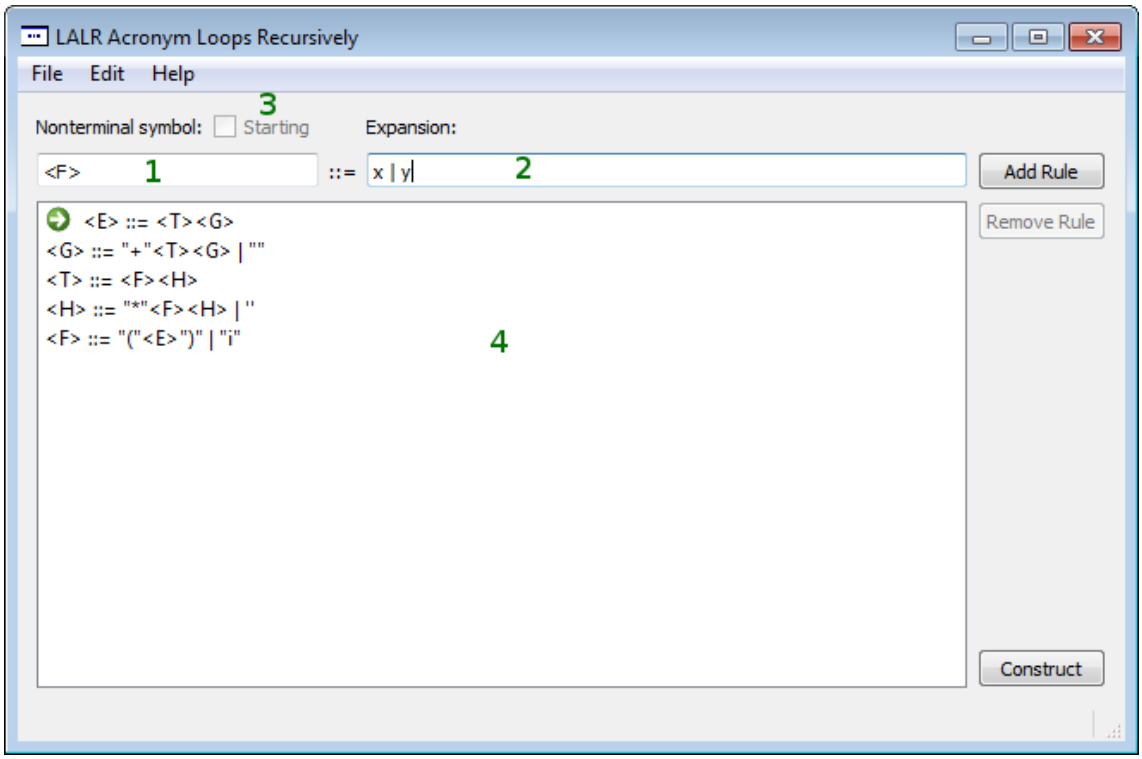
3 – Zaškrtávací pole „Starting“
Slouží pro označení symbolu na levé straně přidávaného pravidla jako počátečního symbolu. Gramatika může obsahovat pouze jeden počáteční symbol, proto je zaškrtávací pole neaktivní, pokud již gramatika počáteční symbol obsahuje. Pro přidání pravidla s novým počátečním symbolem je třeba nejprve z gramatiky odebrat pravidla s dosavadním počátečním symbolem na levé straně.

4 – Seznam pravidel gramatiky
Pravidla se stejnou levou stranou jsou slučována. Počáteční symbol je označen ikonou šipky.

Pravidlo se do gramatiky přidá zapsáním jeho levé a pravé strany do příslušných textových polí a kliknutím na tlačítko „Add Rule“ nebo zmáčknutím klávesy „Enter“ během úpravy levé nebo pravé strany pravidla. Tlačítko „Add Rule“ je neaktivní, pokud je pravidlo zadáno chybně.

Pravidla lze z gramatiky odebrat jejich označením a kliknutím na tlačítko „Remove Rule“ nebo zmáčknutím klávesy „Delete“.

Automat se pro gramatiku sestrojí kliknutím na tlačítko „Construct“. Pro sestrojení automatu je nutné, aby gramatika obsahovala alespoň jedno pravidlo s počátečním symbolem. Tlačítko „Construct“ je proto neaktivní, pokud gramatika neobsahuje ani jedno pravidlo s počátečním symbolem.



Obrázek 4. Okno pro úpravu gramatiky

4.2.2. Okno automatu

Po sestrojení automatu se zobrazí okno (obrázek 5.), ve kterém je možné zadávat vstupy pro automat a sledovat jeho činnost. Jednotlivé části okna mají následující význam:

1 – Gramatika
Zobrazuje gramatiku, pro kterou automat vznikl.

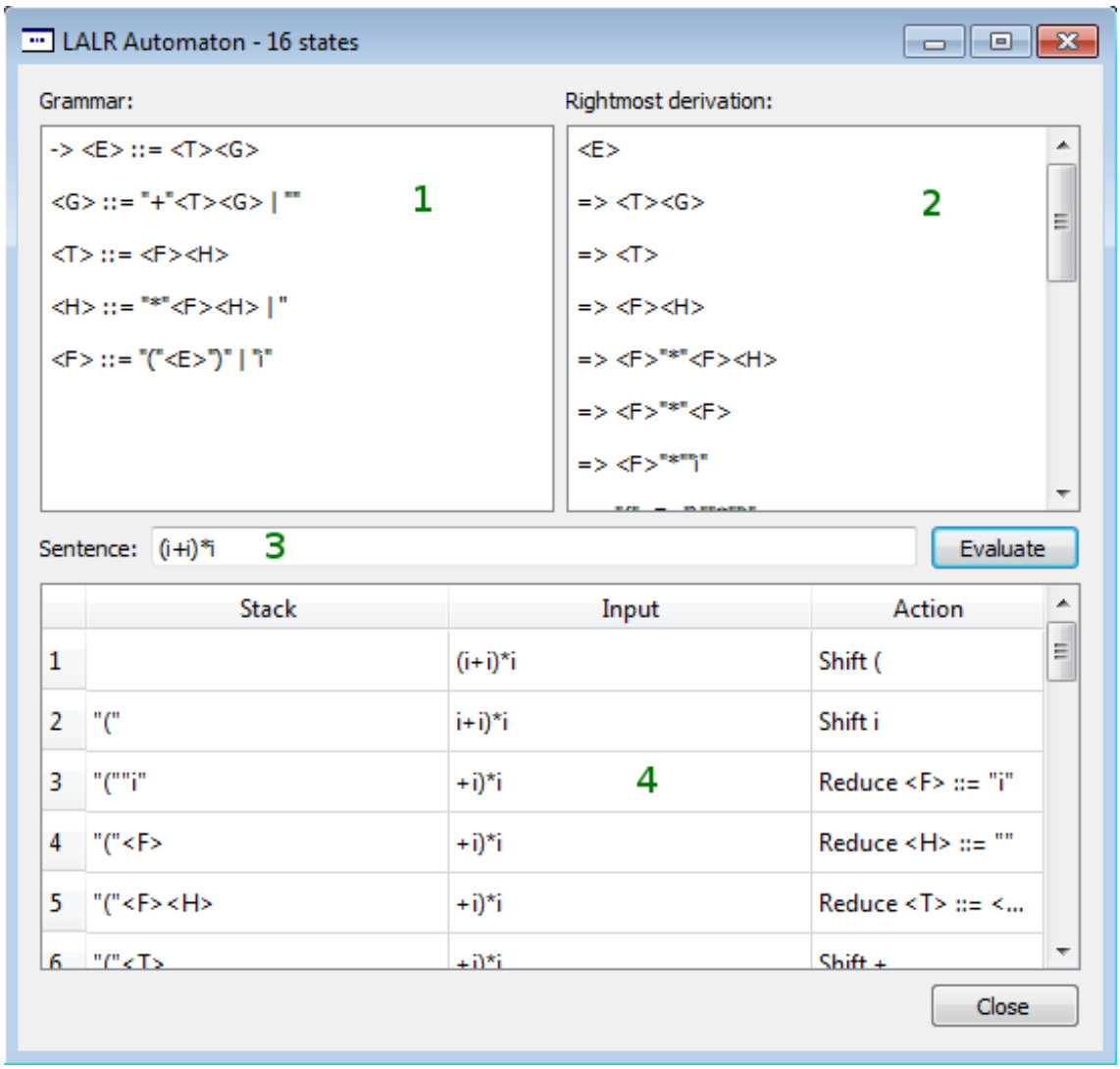
2 – Odvození
Zobrazuje, jak lze zadaný vstup z gramatiky odvodit, pokud jej automat přijal.

3 – Textové pole pro zadání vstupu
Očekávaným vstupem je posloupnost terminálů. Terminály se zapisují uzavřené do uvozovek nebo apostrofů. Jednoznakové terminály (kromě bílých znaků) nemusejí být uzavřené do uvozovek nebo apostrofů. Bílé znaky se ve vstupu ignorují, pokud nejsou součástí terminálu.

4 – Činnost automatu
Každý řádek tabulky značí jeden krok automatu. Sloupec „Stack“ zobrazuje obsah zásobníku automatu v daném kroku. Sloupec „Input“ zobrazuje

dosud nezpracovanou část vstupu v daném kroku. Sloupec „Action“ zobrazuje akci, kterou se v daném kroku automat rozhodl vykonat.

Simulace činnosti automatu se spustí kliknutím na tlačítko „Evaluate“ nebo zmáčknutím klávesy „Enter“ během zadávání vstupu.



Obrázek 5. Okno automatu

5. Diskuze

5.1. Porovnání s ostatními díly

Díky tomu, že proces sestrojení syntaktického analyzátoru lze zautomatizovat, je v dnešní době k dispozici mnoho generátorů syntaktických analyzátorů, které pro zadanou gramatiku vygenerují zdrojový kód syntaktického analyzátoru v některém programovacím jazyce. Pokud si tedy vymyslíte nový jazyk a potřebujete pro něj syntaktický analyzátor, nemusíte jej pracně programovat. Stačí vám vytvořit gramatiku generující tento jazyk, kterou jste již pravděpodobně stejně vytvořili, když jste jazyk navrhovali, a nechat si analyzátor vygenerovat některým z generátorů. Výsledný zdrojový kód si pak upravíte podle vlastních potřeb, například doplněním chybových hlášení při syntaktických chybách. Jedny z nejpoužívanějších generátorů jsou GNU Bison a GOLD Parsing System.

GNU Bison je volně dostupný generátor zpětně kompatibilní s dříve hojně používaným generátorem yacc, který byl dodáván jako součást Unixových systémů. GNU Bison je novější a oproti generátoru yacc poskytuje mnohá vylepšení. Bison načte bezkontextovou gramatiku a pokud to jde, vygeneruje pro ni $LALR(1)$ syntaktický analyzátor v jazyce C, C++ nebo Java. V případě konfliktů vypisuje varovná hlášení a pravidlo, které se rozhodl upřednostnit. Konflikty lze kromě úpravy gramatiky také řešit pomocí uvedení priority nebo asociativity pravidel. Pro gramatiky, které nejsou $LALR(1)$, Bison podporuje GLR syntaktický analyzátor, který dokáže zpracovat nejednoznačné jazyky.

GOLD Parsing System poskytuje grafické vývojové prostředí pro tvorbu syntaktických analyzátorů. Obsahuje editor gramatiky a po vytvoření analyzátoru umožňuje jeho testování. Oproti jiným generátorům, které generují přímo zdrojový kód analyzátoru, GOLD vytvoří analyzátor v podobě nezávislé na programovacím jazyce. Takto vytvořený analyzátor může být testován a upravován. Zdrojový kód analyzátoru se vygeneruje jedním z generátorů pro konkrétní jazyk. Tyto generátory jsou nezávislé na zbytku systému. Právě díky oddělení vytvoření analyzátoru a jeho implementace v konkrétním programovacím jazyce GOLD podporuje výstup ve více než deseti programovacích jazycích a kdokoliv může vytvořit generátory pro další jazyky. GOLD je také volně dostupný.

Moje aplikace neumí vygenerovat syntaktický analyzátor v žádném programovacím jazyce, pouze jej vytvoří v paměti a umožňuje simulaci jeho činnosti. Z tohoto pohledu je moje aplikace podobnější aplikaci ParsingEmu, kterou jsem používal při testování.

Stejně jako moje aplikace i ParsingEmu slouží pro „hraní si“ se syntaktickým analyzátorem a je proto vhodným doplňkem studia problematiky syntaktické analýzy. ParsingEmu navíc oproti mojí aplikaci nenabízí jen konstrukci $LALR(1)$ syntaktického analyzátoru, ale i $LL(1)$, $SLR(1)$ a $LR(1)$ syntaktického analyzátoru. Kromě toho také umožňuje zobrazení množin $FIRST$ a $FOLLOW$ pro neterminály gramatiky a zobrazení rozkladových tabulek a množin položek pro

jednotlivé metody. Pro množiny $LALR(1)$ položek navíc umí zobrazit, z kterých množin $LR(1)$ položek byly sloučeny. Na druhou stranu ParsingEmu vyžaduje zápis gramatiky v notaci používané v literatuře, kde terminály a neterminály jsou jednoznakové a rozlišuje se mezi nimi podle velkých a malých písmen a navíc písmeno e je vyhrazeno pro terminál ϵ. Zápis je tedy sice jednodušší, jelikož v něm není nutné značit co je terminál a co neterminál, ale na druhou stranu je velmi omezující. Pro účel aplikace je ale možná jednoduchost vhodnější než obecnost.

Moje aplikace v případě konfliktů vypíše konfliktní pravidla. ParsingEmu pouze ohlásí počet konfliktů. Ke konfliktním pravidlům se sice dá dopracovat přes rozkladovou tabulku a množiny položek, je to ale pracnější. Moje aplikace dále pro vstup analyzátoru sestrojí odvození, což ParsingEmu neposkytuje. Výkonnost aplikací jsem nemohl porovnat kvůli omezení pojmenování neterminálů velkými písmeny, což znemožňuje zadání gramatiky s více než 26 neterminály. Vzhledem k účelu aplikace se ale asi nejedná o klíčovou věc.

Každopádně jsem nevytvořil nic nového. Pokud potřebujete pro gramatiku vygenerovat syntaktický analyzátor, pak jej z mojí aplikace nedostanete. Pokud si potřebujete během studia syntaktické analýzy něco sami vyzkoušet, pak je ParsingEmu pravděpodobně názornější, snad až na řešení ohlašování konfliktů. Pokud byste si chtěli vyzkoušet činnost analyzátoru pro reálnější příklad, jako třeba kód nějakého programovacího jazyka, pak by vám aplikace mohla posloužit. Toto ale umí například i GOLD ve fázi testování analyzátoru a navíc poskytne i lexikální analyzátor. Snad jen sestrojení odvození jsem nikde jinde neviděl. Nejde ovšem o žádnou převratnou funkcionalitu.

5.2. Nedostatky

Možnosti ruční úpravy gramatiky jsou dosti omezené. Pokud se uživatel splete a zadá některé pravidlo chybně, musí celé pravidlo odebrat a zadat jej znovu. Navíc kvůli slučování pravidel se stejnou levou stranou do jednoho se tak může stát, že uživatel chybně zadá jedno pravidlo, ale odebrat bude muset více pravidel, protože byla sloučena do jednoho. Podobně, když se uživatel splete v označení počátečního symbolu, je nutné odebrat celé pravidlo. Možná, že by bylo vhodnější pro účel úpravy gramatiky poskytnout textový editor, který by tyto nedostatky odstranil. Nevýhodou editoru je ovšem nemožnost zjistit syntaktickou chybu během zadávání pravidel. Pokud je uživatel zadáváním gramatiky příliš frustrován, může si gramatiku vždy vytvořit ve svém oblíbeném textovém editoru a soubor si do aplikace načíst.

Aplikace se pro větší gramatiku může lehce stát nepřehlednou, kvůli výskytu syntaktických prvků Backus-Naurovy formy všude tam, kde se pracuje s pravidly gramatiky. Přehlednost by se určitě zvýšila obarvením terminálů a neterminálů. Nejen že by uživatel rychleji rozlišil mezi tím co je terminál a neterminál, ale díky obarvení by již nemusely být syntaktické prvky Backus-Naurovy formy vůbec přítomny.

Pro zápis gramatiky by bylo vhodnější zvolit zápis, který již používá některý rozšířený program jako například Bison. Moje aplikace by tak byla kompatibilní s mnoha gramatikami, které se vyskytují na internetu právě v zápisu pro Bison nebo yacc. Syntax mojí aplikace je blízká mnoha programům, jelikož často používají některou variantu Backus-Naurovy formy s případným rozšířením o další prvky. Je například podobná aplikaci GOLD. Přímo kompatibilní ale není s žádným programem, co znám. Zápis gramatiky jsem volil hlavně s ohledem na jednoduchost na začátku vývoje v době, kdy jsem ještě nebyl seznámen s nástroji používanými v této oblasti.

Jak lze vidět, moje aplikace má spoustu nedostatků, na jejichž řešení jsem již neměl dost času ani vůle. Troufám si ale tvrdit, že jde spíše o nedostatky uživatelské přívětivosti a že z hlediska funkčnosti je aplikace obstojná.

5.3. Výkon

I přes mnohé provedené optimalizace, které skutečně aplikaci velmi zrychlily, jsem si vědom dalších slabších míst, která by šla zlepšit. Níže jsou popsány další možné optimalizace. Aplikace je ale i bez nich podle mého názoru dostatečně rychlá, takže jsem je z časových důvodů neprovedl.

Symboly jsou reprezentovány jako číslo symbolu a informace o tom, zda jde o terminál nebo neterminál. Na tuto informaci by stačil jeden bit, ale v implementaci používám celý bajt. Navíc z důvodu zarovnávání objektů v paměti na násobky 4 pravděpodobně informace stejně zabere 4 bajty. Pro symbol je tedy potřeba 33 bitů, ale zabere 64 bitů. Navíc porovnání dvou symbolů potřebuje kvůli tomuto jednomu bitu dvojnásobný počet instrukcí procesoru. Šlo by použít jeden bit přímo z čísla symbolu. Symboly by pak zabraly polovinu paměti a hlavně by se dvakrát rychleji porovnávaly. Rozsah možných čísel pro symboly by se tak vlastně rozdělil na poloviny, například spodní polovina pro terminály a horní pro neterminály. Ještě lepší by bylo neterminály číslovat od posledního terminálu nahoru. Tento v nejhorším případě 31 bitový rozsah očíslování při statickém rozdělení na poloviny by byl stále dostatečně velký, jelikož se jedná o více než 2 miliardy hodnot. Pokud by to nějaké gramatice nestačilo, pak by byl vůbec problém takovou gramatiku v paměti uložit natož s ní pracovat v nějakém rozumném čase.

Ve výpočtu uzávěru množin se po sloučení dvou $LALR(1)$ položek testují až tři podmínky. Testuje se, zda sloučení rozšířilo množinu prediktivních symbolů, zda pravá strana pravidla položky začíná neterminálem a zda množina $FIRST$ zbytku této pravé strany obsahuje ϵ. Při konstrukci syntaktického analyzátoru se sestrojí pro každé pravidlo velmi mnoho $LALR(1)$ položek a tento test se tedy provede mnohokrát. Poslední dva testy vedou na průchod pravé strany pravidla, přitom ale oba testy závisí pouze na pravidlu a ne na celé $LALR(1)$ položce. Můžeme si tedy na začátku konstrukce předpočítat a zapamatovat platnost posledních dvou podmínek pro každé pravidlo. Dohledání výsledku je díky očíslování

pravidel pouhý přístup do pole. Zredukujeme tak průchod pravidlem na jeden přístup do pole.

Často se stane, že mnoho množin $LALR(1)$ položek obsahuje pouze jednu položku a to redukční položku. Týká se to až 30–50% množin, jak tvrdí [4]. V rozkladové tabulce se to projeví tak, že všechny definované hodnoty v řádku pro tuto množinu mají stejnou hodnotu. Během analýzy zde tedy buď redukujeme vždy podle stejného pravidla nebo ohlásíme chybu. Z konstrukce množin $LALR(1)$ položek plyne, že tato jediná položka v množině musela vzniknout přesunutím tečky a kopií množiny prediktivních symbolů jiné neredukční položky. Chybu tedy lze během analýzy odchytit již dříve ihned po přesunu posledního symbolu. Pokud nejde o chybu, určitě budeme redukovat podle jednoho konkrétního pravidla bez ohledu na kontext. To by se dalo také zahrnout již do předchozího přesunu. Vytvořili bychom tak novou akci syntaktického analyzátoru a to akci přesun-redukce a analyzátor by při tomto přesunu rovnou i redukoval. Počet stavů charakteristického $LALR(1)$ automatu a počet řádků rozkladové tabulky by se tak snížil o 30–50%. Analyzátor tak ušetří mnoho přístupů do rozkladové tabulky a vkládání prvků na zásobník, které vzápětí stejně odebere.

Idea další optimalizace spočívá v eliminaci řetězových pravidel, což jsou pravidla ve tvaru $A \to B$, kde $B \in N$. Tato pravidla jsou velmi častá a zřetězení těchto pravidel může být velmi hluboké. Vyskytují se při zahrnování konkrétnějších konceptů do obecnějších konceptů. Například příkazem v programovacím jazyce může být příkaz větvení, příkaz cyklu, příkaz vyhodnocení výrazu a další. Příkazem cyklu může být příkaz *for*, příkaz *while* a další. Příkazem větvení může být příkaz *if* nebo příkaz *switch*. Během syntaktické analýzy tak analyzátor například najde část věty redukovatelné na příkaz *if*, redukuje na příkaz *if*, pak na příkaz větvení, pak na samotný příkaz a až tady se stane něco zajímavějšího. V daném kontextu nebylo syntakticky důležité, že šlo o příkaz *if*, ani to, že šlo o příkaz větvení. Důležité bylo pouze to, že šlo o příkaz, přitom analyzátor strávil spoustu času přidáváním prvků na zásobník a jejich následným odebíráním. Z pohledu analýzy jsou tedy řetězová pravidla nevhodná. Mohli bychom je odstranit přímo v gramatice pomocí substituce, ale to by vedlo na značné znepřehlednění gramatiky a mnoho nových pravidel, což by analýzu naopak zpomalilo. Lepší je během konstrukce tyto zřetězené redukce detekovat a zahrnout je do jedné. Často se tak i sníží počet stavů charakteristického $LALR(1)$ automatu.

Rozkladovou tabulku lze komprimovat sloučením řádků a sloupců se stejnými hodnotami. Takových řádků a sloupců ale nebude mnoho. Často se ale budou dva řádky nebo sloupce lišit jen na těch pozicích, kde jeden má hodnotu definovanou a druhý ne. Můžeme zavést pomocnou tabulku úspěchu, která ve svých buňkách ukládá, zda jsou buňky rozkladové tabulky o stejných souřadnicích definovány. Tabulku úspěchu lze efektivně reprezentovat pomocí bitového pole. Potom již nemusíme v rozkladové tabulce uvažovat nedefinované hodnoty, protože ty odhalíme pomocí tabulky úspěchu. Výrazně se tak zvýší počet slučitelných řádků a sloupců. Podle [4] se takto může ušetřit i 90–95% původní velikosti rozkladové

tabulky.

5.4. Možné pokračování

V aplikaci by šlo pokračovat přidáním lexikálního analyzátoru. Momentálně aplikace vyžaduje zadávání vstupu pro syntaktický analyzátor ve formě posloupnosti terminálních symbolů. Nelze tedy například zadat přímo zdrojový kód nějakého programu, uživatel jej musí ručně přepsat na symboly vyskytující se v gramatice.

Jediný možný způsob, jak vyřešit konflikty v gramatice, je momentálně úprava gramatiky. Aplikace by mohla poskytnout řešení konfliktů na úrovni syntaktického analyzátoru pomocí určení priority a asociativity pravidel, tak jak to umožňují generátory syntaktických analyzátorů.

Dále by šlo aplikaci rozšířit na plnohodnotný generátor syntaktických analyzátorů v některém programovacím jazyce. Toto nemalé rozšíření by však svým rozsahem překonalo celou dosavadní aplikaci.

Závěr

$LALR(1)$ metoda je nejpoužívanější metoda syntaktické analýzy deterministických bezkontextových jazyků. Využívá se zejména při syntaktické analýze programovacích jazyků pro svůj dobrý poměr mezi rychlostí a obecností.

Vytvořil jsem aplikaci, která pro vstupní bezkontextovou gramatiku sestrojí syntaktický analyzátor, popřípadě ohlásí konflikty v gramatice. Aplikace umožňuje simulaci činnosti sestrojeného analyzátoru pro libovolný vstup. Aplikaci lze využít při studiu problematiky syntaktické analýzy pro lepší pochopení činnosti analyzátoru.

Dosáhl jsem všech cílů práce. Aplikaci lze dále rozšířit až na případný generátor zdrojového kódu syntaktického analyzátoru.

Conclusions

$LALR(1)$ method is the most used method of parsing deterministic context-free languages. $LALR(1)$ method is mainly used for parsing programming languages for its good ratio between speed and generality.

I have developed an application which constructs parser for input context-free grammar or eventually reports conflicts in the grammar. This application further enables constructed parser simulation for arbitrary input. This application could be used during study of syntactic analysis problems for better understanding of parsing.

I have achieved all goals of this thesis. Application could be further developed into eventual compiler-compiler.

Reference

[1] Ľ. Molnár, M. Češka, B. Melichar, *Gramatiky a Jazyky* (SNTL, 1987).

[2] M. Češka, *Gramatiky a Jazyky*, studijní text (Vysoké učení technické v Brně, 1992).

[3] D. Grune, C.J.H. Jacobs, *Parsing Techniques - A Practical Guide* (Ellis Horwood Limited, 1990).

[4] W.M. Waite, G. Goos, *Compiler Construction* (Springer, 1983).

[5] M.D. Mickunas, R.L. Lancaster, V.B. Schneider, *Transforming LR(k) grammars to LR(1), SLR(1) and (1,1) bounded right-context grammars* (Journal of the ACM, volume 23 issue 3, 1976).

[6] V. Vychodil, *Konstrukce zásobníkového automatu LALR(1)*, studijní text (Univerzita Palackého v Olomouci, 2001).

[7] M. Johnson, *LALR Parsing* [online] `http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/11-LALR-Parsing.pdf` [citováno 12.5.2011].

[8] *If-Then-Else Statement* [online] `http://www.devincook.com/goldparser/doc/grammars/example-if-then-else.htm` [citováno 12.5.2011].

[9] R. Corbett, R.M. Stallman, *Bison* [online] `http://www.gnu.org/software/bison/` [citováno 12.5.2011].

[10] H. Bonaffini, *ParsingEmu* [online] `http://www.supereasyfree.com/software/simulators/compilers/principles-techniques-and-tools/parsing-simulator/parsing-simulator.php` [citováno 12.5.2011].

A. Obsah přiloženého CD

`bin/`
: Instalátor aplikace a samotná aplikace spustitelná přímo z CD.

`data/`
: Ukázková a testovací data použitá pro potřeby obhajoby práce.

`doc/`
: Text bakalářské práce ve formátu PDF a ZIP archiv se zdrojovým textem tohoto dokumentu a všemi potřebnými zdroji.

`src/`
: Kompletní zdrojové texty aplikace se všemi potřebnými zdroji.

`readme.txt`
: Instrukce pro instalaci a spuštění aplikace a požadavky pro její provoz.