Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Syntaktická analýza shora-dolů

2015

Antonín Haas

Vedoucí práce:
RNDr. Arnošt Večerka

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Antonín Haas
Název práce:	Syntaktická analýza shora-dolů
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2015
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	RNDr. Arnošt Večerka
Počet stran:	35
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Antonín Haas
Title:	Top-down syntax analysis
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2015
Study field:	Computer Science, full-time form
Supervisor:	RNDr. Arnošt Večerka
Page count:	35
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Cílem práce bylo sestavit aplikaci provádějící a demonstrující deterministickou syntaktickou analýzu shora-dolů. Výsledná aplikace demonstruje jednotlivé kroky analýzy a zároveň slouží jako ucelená část pro možné rozšíření na celý překladač.

Synopsis

The aim of this bachelor's thesis was to create an application demonstrating top-down syntax analysis. The result is an application with graphical interface demonstrating each step of analysis. This application can be used as a library for complete parsing program.

Klíčová slova: bezkontextové gramatiky; LL(1) gramatika; zásobníkový automat; QT knihovna

Keywords: context free grammar; grammar LL(1); pushdown automaton; QT library

Děkuji vedoucímu práce za spolupráci a rady při vývoji aplikace i psaní tohoto dokumentu.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam vět

Seznam zdrojových kódů

1 Úvod

Tato práce se zabývá syntaktickou analýzou metodou shora-dolů. Jsou v ní uvedeny základní pojmy potřebné k pochopení dané problematiky. Dále jsou popsány součásti syntaktického analyzátoru potřebné ke konstrukci a běhu zásobníkového automatu, který následně analýzu provádí. Popis aplikace společně s technickou dokumentací umožňuje pochopení implementace pro případná rozšíření a dokumentuje očekávaný vstup a výstup.

Pro vývoj aplikace byly stanoveny následující nároky:

- Demonstrace deterministické syntaktické analýzy shora-dolů.
- Vstup pravidel bezkontextové gramatiky.
- Kontrola, zda je vstupní gramatika LL(1).
- Zobrazení průběhu analýzy vstupního slova sestaveným automatem.

Překladač převádí napsaný program ve zdrojovém jazyce do cílového jazyka. Jednou z fundamentální části překladače je syntaktický analyzátor, který má podobu modulu, bez kterého se překladač neobejde. Při syntaktické analýze je sestavován derivační strom, který odpovídá překládanému programu. Existují dva zásadně odlišné algoritmy pro konstrukci derivačního stromu.

V prvním případě se provádí syntaktická analýza směrem shora-dolů, což znamená, že se postupuje od kořene stromu směrem k listům. Tato metoda se označuje jako LL, tedy vstup se zpracovává zleva doprava (první L) a vytváří se nejlevější derivace (druhé L). V druhém případě se analýza provádí směrem zdola nahoru a označuje se LR (vstup se zpracovává zleva doprava, ale vytváří se nejpravější derivace).

K popisu syntaxe se využívají bezkontextové gramatiky, které jsou vhodné pro popis syntaxe běžného programovacího jazyka. Regulérní gramatika naopak vhodná není. S její pomocí například nelze popsat fakt, že počet otevíracích závorek musí být stejný jako počet závorek uzavíracích.

2 Překladač

Dle typu cílového programu lze rozdělit překladače na:

- Kompilátor: překladač, kterému je vstupem program ve vyšším programovacím jazyce (C, C++, FORTRAN, PASCAL,...) a výstupem je strojový jazyk, nebo jazyk symbolických instrukcí.
- Interpret: pouze interpretuje zdrojový program pro zadaná vstupní data. Vytváří tedy pouze vnitřní reprezentaci programu, kterou lze považovat za výstupní jazyk (například skriptovací jazyky: PYTHON, RUBY, PHP, JAVA SCRIPT; shelly operačních systémů: BASH a další shelly unixových systémů, příkazový řádek Windows; HTML a další).
- Hybridní překladač: generuje následně interpretovaný mezikód nezávislý na operačním systému (JAVA nebo .NET).

Proces překladu se rozděluje na několik fází, které mohou být striktně odděleny, nebo vzájemně provázány:

- Lexikální analýza
- Syntaktická analýza
- Sémantická analýza
- Optimalizace kódu
- Generování výstupního kódu nebo interpretování

Do lexikálního analyzátoru vstupuje celý zdrojový program. Ten je převáděn do podoby srozumitelné dalším částem překladače. Vstup se transformuje na posloupnost symbolů (atomů), což jsou nejmenší logické části, kterým lze přiřadit význam (například <klíčové slovo>, <celé číslo>, <relační symbol rovnost>, atd.). Ignorují se části vstupu, které nemají pro další překlad význam (například přebytečné mezery, komentáře nebo konce řádků).

Syntaktický analyzátor vytváří strukturu překládaného programu, nejčastěji derivační strom. Skládá symboly vygenerované lexikálním analyzátorem do příkazů, bloků příkazů, definic proměnných nebo funkcí a dalších struktur.

Sémantický analyzátor přiřazuje význam skupinám symbolů získaných při syntaktické analýze. Kontroluje, zda je použitá proměnná deklarována (u některých programovacích jazyků není vyžadováno) a jestli je použitá správně vůči jejímu deklarovanému typu; kontroluje, zda není deklarovaná proměnná již deklarována a uloží potřebné informace jako název, typ, počáteční hodnota aj.

Optimalizátor kódu zajišťuje použití co nejméně pomocných proměnných; aby se v cyklu zbytečně několikrát nevyhodnocoval tentýž výraz, který se při průchodu cyklem nemění.

Generátor cílového programu nebo interpretace je fáze, ve které program prochází bez chyb sémantickou analýzou, případně optimalizací. Vytváří se kód buď v jazyce symbolických instrukcí, nebo přímo v jazyce stroje.

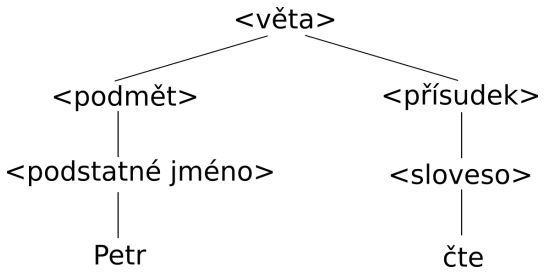
3 Základní pojmy

Následující část textu popisuje základní teoretické znalosti potřebné k pochopení problematiky konstrukce automatu pro syntaktickou analýzu. Přehled zahrnuje pojmy gramatika, formální jazyk, rozdělení gramatik a konečné automaty. Zvláštní pozornost je věnována bezkontextovým gramatikám a zásobníkovému automatu, které slouží jako základní aparát v dalších kapitolách.

3.1 Gramatiky

Struktura věty přirozeného jazyka se definuje gramatickými pravidly. V přirozeném jazyce považujeme za základní jednotku písmeno, ze kterého vytváříme slova a věty. Za abecedu označíme množinu slov, ze kterých se skládá struktura věty. Analogicky pro terminologii formálních jazyků tedy můžeme tvrdit, že symbol odpovídá slovu a řetězec větě.

Jednotky jazyka definujeme pomocí dalších jednotek. V přirozeném jazyce se jedná o stavbu věty, definovanou prvky jako například podmět a přísudek. Podmět pak může být podstatné jméno a přísudek sloveso. Příklad věty: Petr čte.



Obrázek 1: Gramatická struktura věty přirozeného jazyka.

Gramatická pravidla, pomocí kterých je možné definovat jednotky, mohou vypadat následovně[1]:

- <věta> → <podmět><přísudek>
- <podmět> → <podstatné jméno>
- <podstatné jméno> → Petr
- <přísudek> → <sloveso>
- <sloveso> → čte

[1]Pro odlišení základních jednotek a jednotek definovaných dalšími jednotkami se používají speciální rozlišovací symboly. Jedním z běžných způsobů jsou symboly <, >.

Množina všech gramatických pravidel tvoří gramatiku jazyka, která nám umožňuje generovat věty jazyka. Tyto věty mají důležitou vlastnost – jsou gramaticky správné.

Gramatickými pravidly, gramatikou, se určuje tzv. syntax jazyka. To znamená, že se definuje přípustná struktura vět a základní jednotky jazyka, které je možné na daném místě použít. Významem vět se zabývá sémantika. Pokud libovolná věta splňuje podmínky definované gramatikou, patří do daného jazyka.

Ve formálních jazycích a gramatikách se také pracuje se základními symboly (terminály) a se symboly definovanými pomocí jiných symbolů (neterminály). Zvláštní význam má startovní neterminál. V příkladu přirozeného jazyka je jako startovní symbol použit symbol <věta>. Následuje formální definice gramatiky:

Definice 1 (Gramatika)
Gramatika je struktura G=<N,T,S,P>, kde

N je množina neterminálních symbolů,

T je množina terminálních symbolů,

S je startovní symbol,

P je množina pravidel, která je podmnožinou $(N \cup T)^*$.[1]

Pro zjednodušení zápisu a zvýšení čitelnosti se využívá následující zápis:

1. Neterminální symboly se označují velkými písmeny.
2. Terminální symboly se označují malými písmeny.
3. Slova složená z terminálních a neterminálních symbolů se označují řeckými písmeny.
4. Pravidla se zapisují ve tvaru $\alpha \to \beta$.
5. Pravidla, která mají shodné levé strany $(\alpha \to \beta_1, \alpha \to \beta_2, \cdots, \alpha \to \beta_n)$, lze zkráceně vyjádřit jako $\alpha \to \beta_1|\beta_2|\cdots|\beta_n$, přičemž $\beta_1, \beta_2, \cdots, \beta_n$ se označují jako alternativy pravé strany pravidla.

3.2 Bezkontextové gramatiky

Bezkontextové gramatiky se označují v Chomského hierarchii jako typ 2.
Pro jejich pravidla platí:

Každé pravidlo má tvar $A \to \alpha$, kde $A \in N$ a $\alpha \in (N \cup T)^*$.

3.2.1 Strom odvození

Definice 2 (Strom odvození)

Nechť G=(N,T,P,S) je bezkontextová gramatika. Potom orientovaný, ohodnocený a uspořádaný strom D je strom odvození pro G, pokud splňuje následující podmínky:

1. Každý vrchol je ohodnocený symbolem z $N \cup T \cup \{\varepsilon\}$.
2. Ohodnocení kořenu stromu je S.
3. Vrchol je ohodnocený symbolem z N, pokud má alespoň jednoho následovníka.
4. Jestliže $u_1, u_2, \cdots, u_k$ jsou přímí následníci vrcholu u, který je ohodnocený symbolem $A \in N$, a následovníci jsou ohodnocení zleva doprava $A_1, A_2, \cdots, A_k$, pak musí existovat pravidlo $A \to A_1 A_2 \cdots A_k$.[1]

Obecně může platit, že pro tutéž větu může existovat více derivačních stromů a také různé derivace mohou mít stejný derivační strom.

3.2.2 Levá a pravá derivace

Definice 3 (Levá (pravá) derivace)

Odvození $\alpha_0, \alpha_1, \cdots \alpha_k$, ve kterém se každé přímé odvození $\alpha_i \to \alpha_{i+1}$ pro $0 \leq i$‘k realizuje tak, že ve větné formě α_i přepíšeme první symbol zleva, nazýváme levým odvozením, jestliže přepíšeme první symbol zprava, pak pravým odvozením.[1]

Existuje-li pro derivační strom pouze jedno levé a jedno pravé odvození, gramatika se označuje jako jednoznačná. Nejednoznačná gramatika je taková, kdy pro jedno slovo $w \in L(G)$ existuje víc levých (pravých) odvození. Nejednoznačnost některých gramatik lze vypozorovat již z přepisovacích pravidel. Tento rys je považován za negativní (vede k větám, které mají několik interpretací).

3.3 Gramatiky LL(1)

Zkratka LL(k) znamená, že vstup čteme zleva doprava (první L), vytváříme levý rozklad (druhé L) a při rozhodování potřebujeme znát nejvýše k znaků z nepřečteného vstupu. Silná LL(k) gramatika znamená, že daná gramatika je LL(k) a také splňuje podmínku, že při jejím zpracování stačí pro deterministickou analýzu pouze informace ze vstupu[2]. LL(1) gramatika tedy znamená, že pro deterministickou analýzu stačí vždy jeden symbol. Existuje i LL(0) gramatika, kde pro každý neterminál existuje právě jedno pravidlo, tj. nepotřebuje vstup při rozhodování.

Definice 4 (LL(1) gramatika)
Gramatika je typu LL(1), jestliže každá množina pravidel se stejnou levou stranou $A \to \alpha_1|\alpha_2|\cdots|\alpha_n$ má tyto vlastnosti:

- $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$ pro všechna $i \neq j$.
- Je-li pro nějaké i $\alpha_i \Rightarrow^* \varepsilon$, platí pro všechna $i \neq j$ $FIRST(\alpha_j) \cap FOLLOW(A) = \emptyset$.[2]

První vlastnost znamená, že pokud máme pravidla se stejnou levou stranou, musí být odlišitelná prvním symbolem řetězce, který můžeme po derivaci získat.

Druhá vlastnost znamená totéž, ovšem s ohledem na ε-pravidla, takže se testuje pouze v případě, že pro daný přepisovaný neterminál existuje ε-pravidlo.

Věta 5
Každá LL(1) gramatika je silná LL(1) gramatika.

Důkaz
Vyplývá přímo z definice silné gramatiky (všechny řetězce v množině FIRST jsou jednoznakové) a z definice (obecné) LL(k) gramatiky.[2] □

[2]Kromě informací poskytující pravidla gramatiky.

3.4 Konečné zásobníkové automaty

Zásobníkový automat získáme modifikací konečného automatu. Přidá se dodatečná paměť – zásobník – která je potenciálně nekonečná. To znamená, že v každém kroku zásobník obsahuje konečný počet znaků, ale kdykoliv jej lze rozšířit o konečný počet dalších symbolů. Symboly, které se mohou v zásobníku nacházet, tvoří zásobníkovou abecedu.

Definice 6 (Zásobníkový automat)
Zásobníkový automat P je struktura $P = (Q, T, Z, \delta, q_0, z_0, F)$, kde

Q - konečná množina vnitřních stavů automatu,

T - konečná množina vstupních symbolů - vstupní abeceda,

Z - konečná množina symbolů zásobníku - abeceda zásobníku,

δ - zobrazení $Q \times (T \cup \{\varepsilon\}) \times Z$ do množiny konečných podmnožin $Q \times Z^*$,

q_0 - počáteční stav, $q_0 \in Q$,

z_0 - počáteční symbol zásobníku (dno zásobníku), $z_0 \in Z$,

F - množina koncových stavů, $F \subseteq Q$.[1]

Konfigurace zásobníkového automatu je trojice $(q, w, \alpha) \in Q \times T^* \times N^*$, kde

q - aktuální stav automatu,

w - dosud nezpracovaná část vstupního řetězce,

α - obsah zásobníku (s vrcholem vlevo).

Pokud $w = \varepsilon$, pak se celý vstup zpracoval; pokud $\alpha = \varepsilon$, pak zásobník je prázdný.

4 Deterministická syntaktická analýza shora-dolů

Syntaktický analyzátor sestavuje syntaktickou strukturu programu. Zjišťuje, jak k sobě patří symboly, které jsme získali při lexikální analýze.

Při syntaktické analýze metodou shora-dolů konstruujeme derivační strom směrem od kořene k listům, zleva doprava, podle levé derivace.

Aby se nemusel v paměti udržovat celý strom, stačí pouze posloupnost čísel pravidel, která se při vytváření použila.

Mechanismus LL(1) parseru je založen na zásobníkovém automatu, který pro bezkontextovou gramatiku G=(N,T,P,S) pracuje pouze se zásobníkem (vrcholem zásobníku) a nevyužívá stavy. Zásobníková abeceda je množina $N \cup T$.

V průběhu analýzy se využívají dvě základní operace:

- Expanze: na vrcholu zásobníku se nachází neterminální symbol A. Expanze probíhá nahrazením tohoto symbolu pravou stranou pravidla gramatiky $A \to X_1 X_2 \cdots X_k$.
- Srovnání: na vrcholu zásobníku se nachází terminální symbol. Tento symbol se porovná se symbolem na vstupu a pokud se shodují, odstraní se ze vstupu i z vrcholu zásobníku.

Automat rozpoznává vstupní slovo, pokud na konci analýzy ukazuje vrchol zásobníku současně na dno.

Předpokládáme, že gramatika je bez levé rekurze. Tato podmínka není pro konstrukci automatu nijak omezující, protože gramatika s levou rekurzí není LL(1) gramatikou.

4.1 Výpočet množiny FIRST

Množina FIRST je definována následujícím předpisem:

Definice 7 (množina FIRST)
$FIRST(\alpha) = \{t | \alpha \Rightarrow^* t\mu, t \in T\} \cup \{\varepsilon | \alpha \Rightarrow^* \varepsilon\}$

Množina FIRST(α) obsahuje všechny terminály, kterými může začínat některý řetězec odvoditelný z α (včetně ε).

Pokud α začíná terminálem, tj. $\alpha = t\mu, t \in T, \mu \in (N \cup T)^*$, nalezení množiny je triviální: $FIRST(\alpha) = \{t\}$.

Začíná-li α neterminálním symbolem, tj. $\alpha = B\mu, B \in N, \mu \in (N \cup T)^*$, musí se uvažovat všechna pravidla s neterminálem B na levé straně $B \to \beta_1 | \beta_2 | \cdots | \beta_k$ a množinu FIRST spočítat rekurzivně: $FIRST(\alpha) = FIRST(\beta_1\mu) \cup FIRST(\beta_2\mu) \cup \cdots \cup FIRST(\beta_k\mu)$.

Dojde-li při rekurzivním výpočtu k situaci, kdy argument je prázdný řetězec, pak $FIRST(\varepsilon) = \{\varepsilon\}$.

Algoritmus pro výpočet množiny FIRST vypadá následovně:

1. $FIRST(\varepsilon) = \{\varepsilon\}$.
2. $FIRST(\alpha) = \{t\}$, pokud $\alpha = t\mu, t \in T, \mu \in (N \cup T)^*$.
3. $FIRST(\alpha) = FIRST(\beta_1\mu) \cup FIRST(\beta_2\mu) \cup \cdots \cup FIRST(\beta_k\mu)$, pokud $\alpha = B\mu, B \in N, \mu \in (N \cup T)^*$.

4.2 Výpočet množiny FOLLOW

Množina FOLLOW je definována následujícím předpisem:

Definice 8 (množina FOLLOW)
$FOLLOW(A) = \{t|S \Rightarrow^* \mu A\nu, \nu \neq \varepsilon, t \in FIRST(\nu)\} \cup \{\varepsilon|S \Rightarrow^* \mu A\}$

Výpočet je vázán na větné formy (tj. derivace z počátečního symbolu gramatiky). Pro výpočet se zavádí pomocná množina H. Ta bude obsahovat již uvažované neterminální symboly, aby se předešlo opakování stejného výpočtu v důsledku rekurze. Y označuje aktuální uvažovaný symbol.

Na počátku se inicializuje H=$\emptyset$ a Y=A. Pokud $Y \in H$ (tj. symbol Y byl již uvažován), výpočet pro aktuální symbol končí, jinak je Y přidáno do H: $H = H \cup \{Y\}$ a $FOLLOW(Y) = \emptyset$.

Je-li Y startovní symbol (tj. Y=S), do množiny FOLLOW se přidá prázdný řetězec: $FOLLOW(Y) = FOLLOW(Y) \cup \{\varepsilon\}$ a dále se uvažují všechna pravidla se symbolem Y na pravé straně $X \to \mu Y\nu$. Mohou nastat dva případy:

- $\nu \neq \varepsilon$, pak $G = FIRST(\nu)$. Pokud $\varepsilon \notin G$, pak $FOLLOW(Y) = FOLLOW(Y) \cup G$. V opačném případě $FOLLOW(Y) = FOLLOW(Y) \cup (G \setminus \{\varepsilon\}) \cup FOLLOW(Y/X)$ [3] a přejde se opět ke kroku testování, zda-li byl symbol Y již uvažován.
- $\nu = \varepsilon$, pak $FOLLOW(Y) = FOLLOW(Y) \cup FOLLOW(Y/X)$.

Z popisu vyplývá, že díky rekurzi se postupuje k počátečnímu symbolu gramatiky.

Algoritmus pro výpočet množiny FOLLOW:

1. $\varepsilon \in FOLLOW(S)$.
2. $FIRST(\beta) \subseteq FOLLOW(B)$, kde $A \to \alpha B\beta$.
3. $FOLLOW(A) \subseteq FOLLOW(B)$, kde $A \to \alpha B\beta$, $\beta \Rightarrow^* \varepsilon$.

[3] FOLLOW(Y/X) znamená, že za aktuální symbol Y se bude považovat symbol X.

4.3 Konstrukce zásobníkového automatu

Překladový automat je modifikací zásobníkového automatu, který v některých ohledech nedostačuje. Automat se rozšiřuje o výstupní pásku a vyžaduje se deterministické rozhodování v případě více možných akcí na zásobníku. V průběhu syntaktické analýzy se vygeneruje v dané gramatice takový terminální řetězec, který odpovídá vstupu a na pásku se zapíší čísla použitých pravidel.

Dle definice LL(1) gramatiky mají pravidla navzájem disjunktní množiny FIRST. To zajišťuje deterministické rozhodování výběru pravidla pro neterminální symbol. Operace expanze říká, že je-li na zásobníku neterminál, nahradí se pravou stranou některého pravidla. Pro pravidlo $A \to \alpha$ to znamená, že se ze zásobníku vyjme neterminál `A` a nahradí se α.

Definice 9 (Překladový automat pro LL(1) překlad)
Překladový automat gramatiky G=(N,T,P,S) typu LL(1) je zásobníkový automat rozšířený o výstupní pásku a definovaný rozkladovou tabulkou.[2]

Rozkladová tabulka je zobrazení $M : (T \cup N \cup \{\#^4\}) \times (T \cup \{\$^5\}) \mapsto \{expand(1), \cdots, expand(n), pop, accept, error\}$, kde:

- Expand(i): i-té pravidlo gramatiky[6] ($A \to \alpha$), na vrcholu zásobníku se nachází neterminál A, na vstupu symbol x a v tabulce záznam M[A,x] = expand(i). Pak automat provede operaci expanzi popsanou v úvodu kapitoly a na výstupní pásku zapíše i (tj. číslo pravidla).
- Pop: operace srovnání.
- Accept: přijetí – vstup se celý přečetl a zásobník je prázdný. Na výstup se zapsal levý rozklad věty.
- Error: chyba ve výpočtu (syntaktická chyba).

4.4 Rozkladová tabulka

Řádky sestavené tabulky obsahují všechny symboly vyskytující se v abecedě zásobníku. Ve sloupcích se nachází všechny znaky vstupní abecedy a symbol konce vstupu $. Pro každé pravidlo gramatiky $A \to \alpha$ se vypočítá množina $FIRST(\alpha FOLLOW(A))$ a v tabulce se v řádku odpovídajícímu neterminálnímu symbolu A doplní expand(i) ve všech sloupcích symbolů obsažených ve vypočítané množině. Pro terminální symboly v tabulce na pozici M[x,x] (pozice, kde terminální symbol ze zásobníku odpovídá vstupnímu symbolu) bude příznak pop. Na pozici M[#, $] se nachází příznak accept. Ostatní buňky jsou označeny příznakem error.

[4]Symbol # je označení pro konec zásobníku.
[5]Symbol $ je označení pro konec vstupního řetězce.
[6]Pravidla gramatiky je třeba očíslovat.

V oblasti tabulky, kde se v řádcích vyskytují terminální symboly, je tabulka vždy stejná (příznak pop), a proto lze v řádcích tabulky uvažovat pouze neterminální symboly. Obdobně není v implementaci použit zápis expand(i), ale pouze číslo odpovídajícího pravidla. Místo odpovídajícího příznaku error se ponechá místo prázdné. Nastane-li případ, že na právě ukládané pozici se již nachází nějaká hodnota, dochází k chybě – zadaná gramatika není LL(1).

5 Uživatelská dokumentace

Aplikace byla vyvíjena pro operační systémy Windows a Linux. Vývoj a testování proběhlo na operačním systému Linux Mint 17.2 „Rafaela“. Testování bylo provedeno i na operačním systému Windows 8.1, Windows 7 a Archlinux.

5.1 Spuštění aplikace

Pro spuštění není vyžadována žádná instalace, stačí spustit soubor dodaný v adresáři **bin/** pro konkrétní operační systém. Grafická verze aplikace pro operační systém Linux byla spuštěna pomocí utility `ldd` a vypsaný seznam knihoven je nakopírován ke spustitelnému souboru. Případné další konflikty jsou ponechány v režii uživatele.

Tabulka 1: Cesta ke spustitelnému souboru pro jednotlivé operační systémy

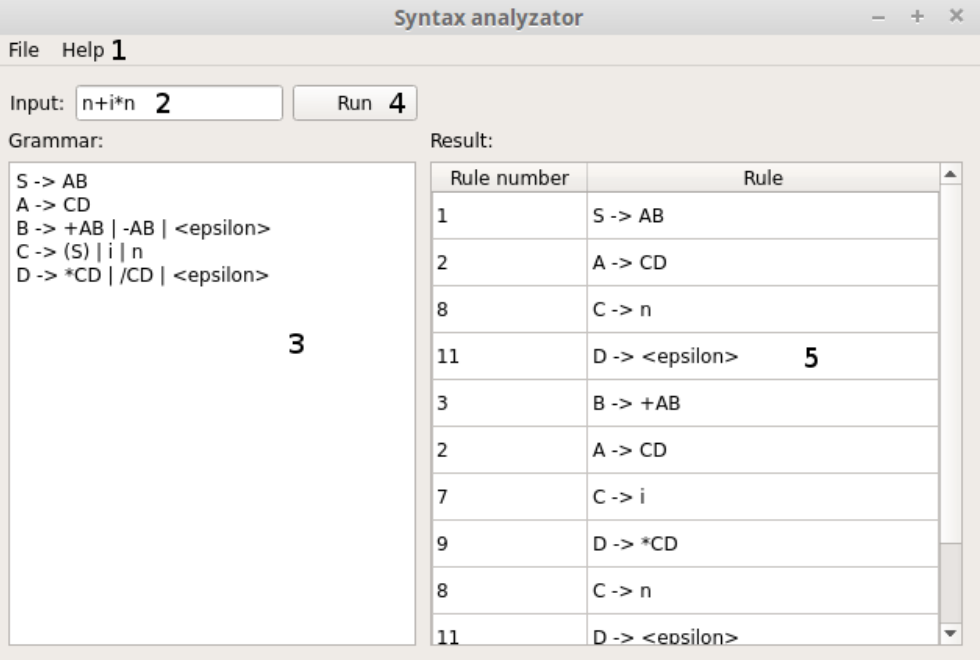
Operační systém	adresář	spustitelný soubor
Windows	bin/Windows/	SyntaxAnalyzator.exe SyntaxAnalyzatorConsole.exe
Linux	bin/Linux/	SyntaxAnalyzator SyntaxAnalyzatorConsole

Spuštění grafické verze aplikace nevyžaduje zadání vstupních parametrů. Výběr gramatiky probíhá klasickým dialogem pro výběr souboru, nebo jej lze zadat ručně. Spuštění konzolové aplikace vyžaduje parametr `-g` následovaný cestou k textovému souboru s gramatikou. Po úspěšném načtení souboru aplikace vyzve k zadání vstupního řetězce a spustí se analýza.

5.2 Popis elementů formuláře

V této sekci jsou popsány hlavní prvky formuláře, včetně ostatních nabídek, které je možné využít:

1. Hlavní nabídka: možnost nahrát gramatiku, uložit gramatiku a ukončit aplikaci.
2. Vstup: vstupní řetězec k analyzování.
3. Editovací okno: zobrazení vybrané gramatiky s možností editace.
4. Spuštění analýzy.
5. Výsledek analýzy: tabulka nejlevější derivace s pořadím použitých pravidel.



Obrázek 2: Grafické rozhraní aplikace

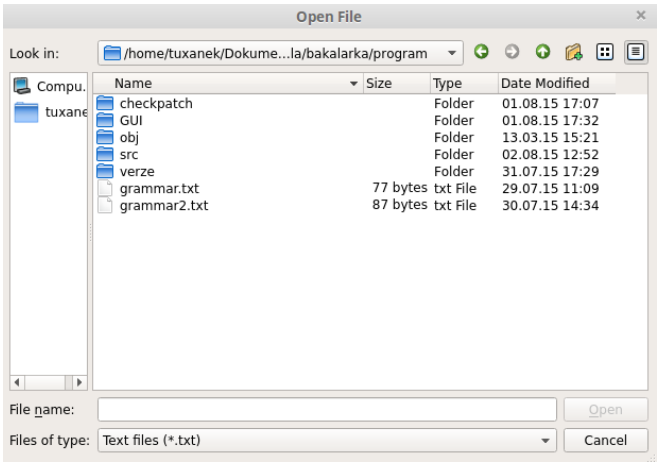
Gramatiku lze načíst ze souboru, nebo jej zadat ručně. Při manuálním zadávání pravidel je důležité dodržet schéma `Neterminální symbol -> pravá strana pravidla`, které je následně porovnáváno s regulárním výrazem. Formát pravidla má uživatel možnost zjistit i v nápovědě. Pravou stranou přepisovacího pravidla se rozumí posloupnost terminálních a neterminálních symbolů[7]. Startovní symbol se označuje `S`. Při ručním zapsání pravidla nezáleží na množství mezer zadaných mezi jednotlivými symboly. Mezery jsou před porovnáním s regulárním výrazem odstraněny. Terminál ε je nutné zapsat ve formě `<epsilon>`. Znak | odpovídá oddělení více pravých stran se stejným neterminálním symbolem na levé straně pravidla. Vyskytnou-li se v gramatice dvě shodná pravidla, jsou sloučena. Každý řádek odpovídá jednomu pravidlu. Jakékoliv nedodržení schématu pravidla se zjišťuje při porovnání s regulárním výrazem a má za následek výpis chybové hlášky `Rule syntax error`[8].

Po zadání cesty pomocí `QFileDialog` je soubor načten a zobrazen v komponentě `QTextEdit`. Gramatiku lze následně upravit (případně upravenou gramatiku uložit). Při ukládání gramatiky se obsah komponenty `QTextEdit` uloží do zvoleného souboru. Opět se pomocí `QFileDialog` zvolí cílový soubor. Nezadá-li uživatel příponu, je automaticky doplněna na `.txt`. Dojde-li během manipulace se souborem k chybě, je to uživateli oznámeno.

Výběr souboru pomocí dialogu je omezen pouze na textové soubory s příponou `.txt`.

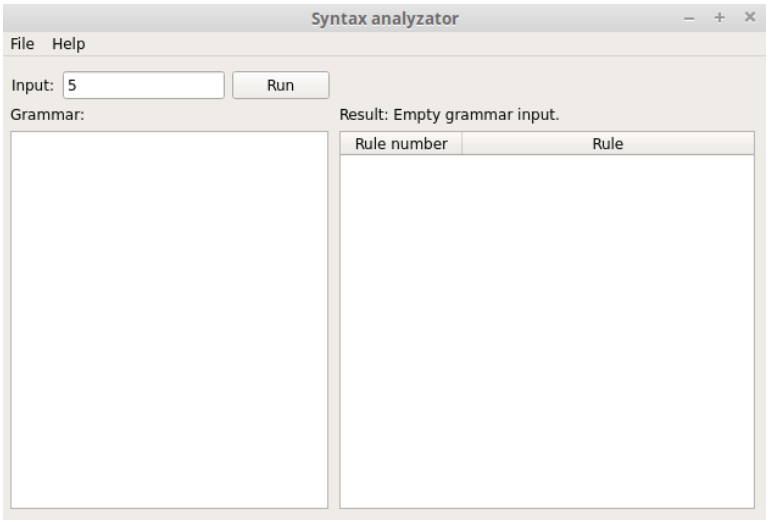
[7] Z pochopitelných důvodů není vhodné zadávat jako pravou stranu dvojici terminálních symbolů ->.

[8] Ošetření proti zadání nekorektního pravidla.



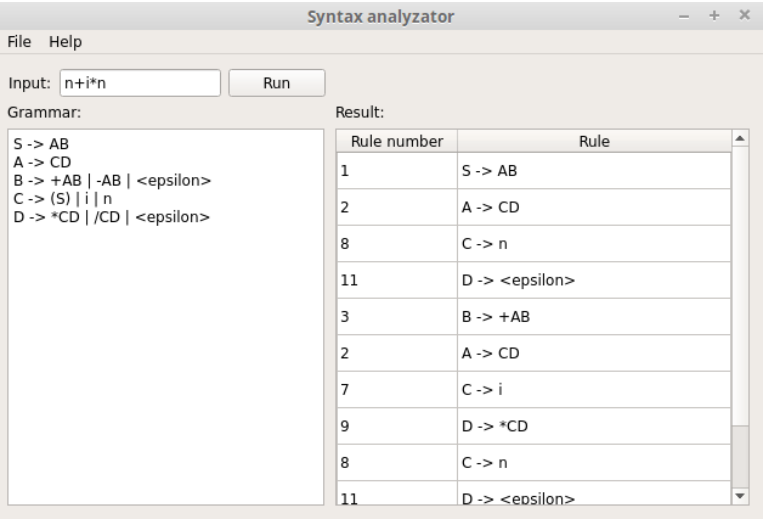
Obrázek 3: Výběr souboru s gramatikou

Ke spuštění syntaktického analyzátoru slouží tlačítko `Run`. Aplikace provede syntaktickou analýzu (viz kapitola 6) vstupu pomocí zadaných přepisovacích pravidel a výsledek se zobrazí v části 5. Pokud během překladu dojde k chybě, označené pole zůstává prázdné a za popisek `Result:` se vypíše chybová hláška popisující problém. Dokončí-li parser úspěšně analýzu, je v sekci 5 zobrazen výsledek analýzy (tj. pořadí použitých přepisovacích pravidel).

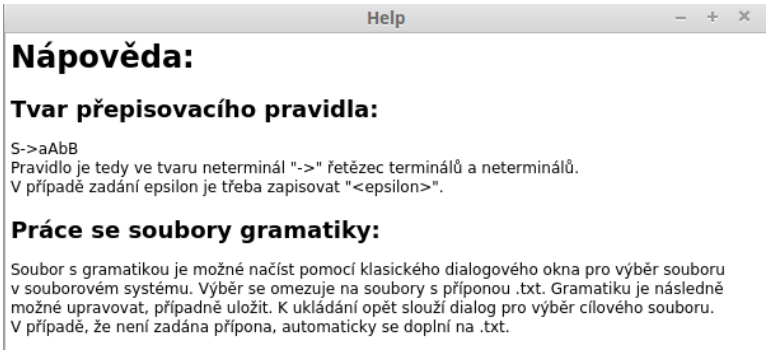


Obrázek 4: Chybná analýza

Uživatel má k dispozici stručnou nápovědu, která popisuje formát pravidla přijímaný aplikací ke zpracování. Dále se popisuje manipulace se soubory (tj. načítání a ukládání) a postup analýzy od načtení gramatiky až k vypsání výstupu.



Obrázek 5: Dokončená analýza



Obrázek 6: Okno s nápovědou

Aplikace nabízí standartní klávesové zkratky:

- Nápověda: F1.
- Načtení souboru s gramatikou: control + L.
- Uložení gramatiky do souboru: control + S.
- Ukončení aplikace: control + Q.

6 Technická dokumentace

Tato část se zabývá prostředím a strukturou samotné aplikace. Slouží k orientaci v kódu pro případné rozšíření.

Pro vývoj aplikace byl využit standard C++11[9]. Aplikace je rozdělena do souborů zpracovávajících gramatiku (`grammar.h` a `grammar.cpp`) a do souborů provádějících analýzu (`parser.h` a `parser.cpp`). Dále se v aplikaci vyskytují soubory pro grafické rozhraní.

Aplikace byla nejdříve vyvíjena jako konzolová, později byla rozšířena o implementaci grafického rozhraní pomocí knihovny Qt.

6.1 Konzolová aplikace

V souboru `main.cpp` konzolové aplikace se nachází funkce pro výpis nápovědy `static void show_usage(void)`, která na výstup vytiskne přehled použitelných přepínačů. Ošetřují se vstupní argumenty a otevře se zadaná gramatika do streamu. Uživatel je vyzván k zadání vstupního řetězce, který se předává společně se streamem gramatiky třídě `Parser`.

```
1  //načtení souboru zadaného parametrem
2  ifstream filestream(grammarfile);
3  if (!filestream.is_open())
4    throw "Failed to load grammar file.";
5  cout << "Grammar file successfully loaded" << endl;
6  //vyžádání vstupního řetězce
7  cout << "insert input:" << endl;
8  getline(cin, input);
9  //spuštění analýzy
10 Parser parser(filestream, input);
```

Zdrojový kód 1: Otevření souboru s gramatikou a žádání o vstupní řetězec

6.1.1 Zpracování gramatiky

Soubory zabývající se zpracováním gramatiky převádí textovou reprezentaci přepisovacího pravidla na strukturu obsahující levou a pravou stranu pravidla.

Pro převod gramatiky existuje třída `Rules`. Metoda `rewrite` s jedním argumentem `&istream` převádí jednotlivá pravidla z textové formy na seznam struktur pravidel `std::vector<RewriteRule>`. Dále pomocí metod `GetTerminals` a `GetNonterminals` s návratovou hodnotou `std::set<std::string>` získáme množinu terminálních a neterminálních symbolů.

[9] www.cplusplus.com/articles/EzywvCM9/

```
struct RewriteRule {
  std::string left;
  std::string right;
};
```

Zdrojový kód 2: Interní struktura přepisovacího pravidla

```
void Rules::Nonterminals(void)
{
  auto it = rules.begin();
  while (it != rules.end()) {
    nonterminals.insert(it->left);
    ++it;
  }
}
```

Zdrojový kód 3: Metoda pro získání neterminálních symbolů

Převod pravidel probíhá následovně:

- Načtení řádku s pravidlem v textové podobě.
- Odstranění přebytečných mezer a srovnání se vzorem pomocí regex_match.
- Uložení levé a pravé strany pravidla do příslušných proměnných ve struktuře.
- Opakuje se, dokud není vstup prázdný.

Neodpovídá-li pravidlo vzoru, je uživateli oznámeno `Rule syntax error`. Během přepisování pravé strany se ověřuje, nevyyskytuje-li se v řetězci symbol `|`. Pokud ano, pravidlo se podle tohoto symbolu rozdělí do více pravidel se shodnou levou stranou (jednotlivé alternativy pravidel jsou ve vnitřní reprezentaci rozděleny).

```
regex rgx("[A-Z]->([^\\|]+)(\\|[^\\|]+)*");
if (!regex_match(line, rgx))
  throw "Rule syntax error";
```

Zdrojový kód 4: Přijímaný formát přepisovacího pravidla

6.1.2 Soubory parseru

Soubory parseru obsahují třídy First, Folow a Parser. První dvě třídy slouží k výpočtům množin FIRST(4.1) a FOLLOW(4.2) potřebných ke konstrukci rozkladové tabulky.

Obě třídy obsahují metody pro získání příslušných množin FIRST a FOLLOW. Pro účely ladění programu třídy obsahují metody, které umožňují tisk množin na výstup `void PrintSet(void)`. Při vytvoření instance třídy `First` je volána metoda `void ComputeAlpha(std::string)`, která má za úkol vypočítat odpovídající množinu. Obdobně se při vytvoření instance třídy `Follow` volá metoda `void ComputeFollow(std::string)`.

Ve třídě `parser` je nejdůležitější metoda `parse`, která má vstupní argument `&istream`. Před vlastním spuštěním automatu je třeba připravit několik věcí. Volají se metody pro přepsání gramatiky a získání seznamu pravidel v interní reprezentaci. Získají se množiny terminálních a neterminálních symbolů, které jsou vyžadovány pro vytvoření rozkladové tabulky. Následně se zavolá vytvoření tabulky a na konec se spouští automat.

```
1  void Parser::parse(istream& filestream)
2  {
3     //přepis pravidel do vnitřní reprezentace
4     Rules rules_struct(filestream);
5     rules = rules_struct.GetRules();
6     //získání terminálních a neterminálních symbolů
7     terminals = rules_struct.GetTerminals();
8     nonterminals = rules_struct.GetNonterminals();
9     //vytvoření rozkladové tabulky
10    CreateTable();
11    //spuštění automatu
12    RunAutomata();
13 }
```

Zdrojový kód 5: Metoda parse

Metoda `CreateTable` vytváří rozkladovou tabulku pro automat. Dle algoritmu popsaného v kapitole 4.4 vypadá rozkladová tabulka následovně:

- Sloupce jsou terminální symboly a ε.
- Řádky jsou neterminální symboly.
- V příslušné buňce se může vyskytovat číslo pravidla, nebo může být buňka prázdná.

Při vytváření tabulky se prochází každé pravidlo, pro které se vypočítá množina FIRST-FOLLOW. Množina FOLLOW se počítá v případě, kdy lze pravou stranu pravidla zderivovat na ε. Následně se uvažují všechny terminální symboly, které lze vypočítáním $FIRST(\alpha FOLLOW(A))$ získat a vyplní se všechny odpovídající pozice v tabulce na pozicích pro vypočítané terminální symboly a neterminální symbol A.

```
1  //výpočet first-follow pro každé pravidlo
2  for (auto it = rules.begin(); it != rules.end(); ++it) {
3     First first_set(it->right,rules);
4     set<string> first = first_set.GetAlpha();
5     //spočítej follow, lze-li z alpha získat epsilon
6     for (auto i = first.begin(); i != first.end(); ++i) {
7       string member = *i;
8       if (member.compare("<epsilon>") == 0) {
9         Follow follow_set(it->left, rules);
10        set<string> follow = follow_set.GetFollow();
11        for (auto q = follow.begin(); q != follow.end(); ++q) {
12          //pokud se na daném místě nenachází číslo, zapiš do tabulky.
                V opačném případě gramatika není LL(1)
13          if (parsing_table.find(make_pair(it->left, *q)) ==
               parsing_table.end()) {
14            parsing_table.insert(make_pair(make_pair(it->left, *q),
                 rule_number));
15          } else {
16            throw "Grammar is not LL(1).";
17          }
```

Zdrojový kód 6: Vytváření rozkladové tabulky

Pokud při vytváření tabulky dojde k pokusu o zapsání do již zaplněné buňky, je uživateli oznámeno `Grammar is not LL(1).`

Vnitřní reprezentace tabulky se realizuje pomocí `std::map<>`, kde ukládaná hodnota je typu `int` (čislo pravidla) a jako klíč se používá pár terminálního a neterminálního symbolu `std::pair<std::string,std::string>`. Uložení do kontejneru map je velice výhodné pro automat, který může snadno vyhledávat na základě terminálního symbolu na vstupu a neterminálního symbolu na zásobníku pozici v rozkladové tabulce. Na tomto místě se může nacházet odpovídající číslo pravidla, nebo prázdná buňka.

Metoda `RunAutomata` simuluje výpočet deterministického zásobníkového automatu. Dle algoritmů popsaných v kapitole 4 provádí operaci expanze nebo srovnání.

Pro zásobník je použit typ `std::string`, který plně dostačuje. Zásobník je simulován operacemi na začátku řetězce. V případě operace srovnání se odstraní první pozice řetězce, během operace expanze se po odstranění neterminálního symbolu nacházejícího se na první pozici řetězce vloží jako prerfix celá pravá strana pravidla. V inicializační části automatu se na zásobník vloží startovní symbol S a za něj se vloží symbol $ značící dno zásobníku.

```
1  //najdi číslo pravidla v tabulce
2  auto search = parsing_table.find(make_pair(stack_non, input_first));
3  //pokud se na daném místě nachází číslo
4  if (search != parsing_table.end()) {
5    //operace expanze: odebere se neterminál ze zásobníku a vloží se
         pravá strana odpovídajícího pravidla
6    parser_stack.erase(parser_stack.begin());
7    if (rules[search->second].right.compare("<epsilon>") != 0) {
8      parser_stack = rules[search->second].right + parser_stack;
9    }
10   //na výstup zapiš číslo použitého pravidla
11   output.push(search->second);
```

Zdrojový kód 7: Operace expanze

Dojde-li průběh operace srovnání do stavu, kdy terminální symbol na vstupu neodpovídá terminálnímu symbolu na zásobníku, dochází k chybě a uživateli je oznámeno `Input syntax error`. Dojde-li průběh operace expanze do stavu, kdy se na příslušném místě v tabulce nenachází žádné číslo pravidla, je uživateli oznámeno `Input not recognized`. Nedojde-li během činnosti automatu k žádné chybě, je výstupní páska uložena a je možné ji vypsat.

Pro výstup bylo zamýšleno použití typu `std::string`, který dostačuje požadavkům výstupní pásky, kdy je třeba pouze v průběhu analýzy připisovat číslo použitého pravidla (tj. operace append), avšak pro potřeby grafického rozhraní byla zvolena fronta (`std::queue<int>`), kam se v průběhu činnosti automatu ukládají čísla použitých pravidel gramatiky (operace push).

Na závěr činnosti konzolové aplikace se uživateli zobrazí obsah fronty, který odpovídá pořadí použitých pravidel v průběhu analýzy.

6.1.3 Optimalizace

V původní verzi konzolové aplikace se pravidla ukládala do `std::list<>` a struktura `RewriteRule` navíc obsahovala proměnou `int number`, ve které bylo uloženo číslo pravidla následně použité v rozkladové tabulce. V rámci optimalizací byla přepsána struktura pravidel na `std::vector<RewriteRule>`, který umožňuje konstantní přístup k jednotlivým pravidlům a není třeba prohledávat celou strukturu.

```
1 //operace expanze
2 auto it = rules.begin();
3 while (it != rules.end()) {
4    if (search->second == it->number) {
5       parser_stack = it->right + parser_stack;
6    }
7    ++it;
8 }
9 output.push(search->second);
```

Zdrojový kód 8: Srovnání: operace s pravidly typu std::list<>

```
1 //operace expanze
2 parser_stack = rules[search->second].right + parser_stack;
3 output.push(search->second);
```

Zdrojový kód 9: Srovnání: operace s pravidly typu std::vector<>

6.2 Grafické rozhraní

Qt je multiplatformní knihovna pro vytváření aplikací s grafickým uživatelským rozhraním. Podporuje desktopové (Windows, Linux, OS X) i mobilní platformy (Android, iOS, Windows Phone 8, BlackBerry a další).

Qt je k dispozici pod třemi licencemi:

1. Qt GNU GPL v3.0.
2. Qt GNU LGPL v2.1.
3. Qt Komerční licence pro vývojáře.

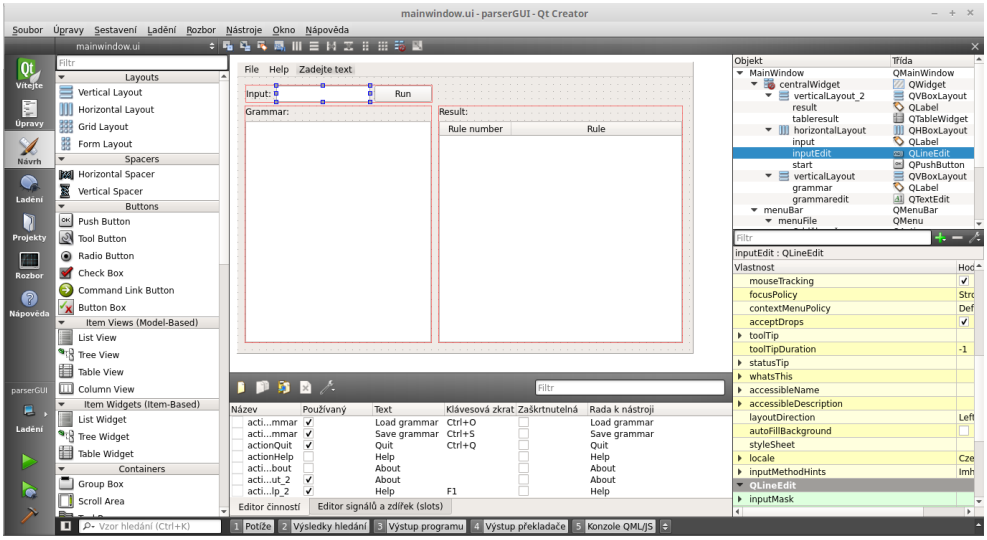
Pro menší projekty je snadné nechat vygenerovat `makefile` pomocí `qmake`:

1. Vytvoření adresáře a uložení zdrojových kódů.
2. Spuštění příkazu `qmmake -project`, který vygeneruje soubor s příponou `.pro`.
3. Spuštění příkazu `qmake` bez parametrů, který vytvoří `makefile`.
4. Spuštění kompilaci příkazem `make`.

Při každém přidání/odebrání je třeba spustit znovu příkaz `qmake -project`.

6.2.1 Tvorba formuláře pomocí Qt Creator

Qt Creator je vývojové prostředí obsahující editor s kompletní podporou C++ (doplňování kódu, zvýraznění syntaxe a další). Integruje do sebe Qt Designer, který slouží pro návrh grafického rozhraní. Widgety se přetahují do okna a následně se mohou nastavovat vlastnosti. Dle potřeby lze propojit požadované signály se sloty.[10]



Obrázek 7: Vývojové prostředí Qt Creator s návrhem formuláře

Navrhnutý formulář se ukládá do souboru mainwindow.ui, který Qt Creator zobrazuje buď formou xml nebo jako grafický formulář s komponentami. Editovat lze pouze v druhém případě.

```
1  <item>
2    <widget class="QPushButton" name="start">
3      <property name="text">
4        <string>Run</string>
5      </property>
6      <property name="flat">
7        <bool>false</bool>
8      </property>
9    </widget>
10 </item>
```

Zdrojový kód 10: XML forma komponenty

[10] http://doc.qt.io/qt-4.8/signalsandslots.html

V souboru `main.cpp` se nachází blok zdrojového kódu, který se stará o vykreslení hlavního okna. Zbytek funkcionality je uložen v souboru `mainwindow.cpp` a příslušném hlavičkovém souboru.

```
1 int main(int argc, char *argv[])
2 {
3   QApplication a(argc, argv);
4   MainWindow w;
5   w.show();
6
7   return a.exec();
8 }
```

Zdrojový kód 11: Funkce main

6.2.2 Funkcionalita komponent

Při vytvoření hlavního okna se propojí signály pro načtení a uložení gramatiky s příslušnými sloty.

```
1 connect(ui->Quit,SIGNAL(triggered()),this,SLOT(close()));
2 connect(ui->Load,SIGNAL(triggered()),this,SLOT(loadGrammar()));
3 connect(ui->Save,SIGNAL(triggered()),this,SLOT(saveGrammar()));
4 connect(ui->About,SIGNAL(triggered()),this,SLOT(showAbout()));
5 connect(ui->Help,SIGNAL(triggered()),this,SLOT(showHelp()));
```

Zdrojový kód 12: Navázání signálů na sloty

Pro načtení a následné zobrazení gramatiky v komponentě `QTextEdit` slouží `QFileDialog`, kterým získáme cestu k souboru. Následně je zvolený soubor otevřen a zobrazen.

```
1 QString fileName=QFileDialog::getOpenFileName(this,tr("Open File"),
     QDir::currentPath(),tr("Text files (*.txt)"));
2 if ( fileName.isNull() == false )
3 {
4   QFile file(fileName);
5   if(!file.open(QIODevice::ReadOnly))
6     showMessage("Failed to load grammar file.");
```

Zdrojový kód 13: Načtení souboru s gramatikou pomocí QFileDialog

Při ukládání gramatiky se obsah komponenty `QTextEdit` uloží do zvoleného souboru. Cestu opět získáme pomocí `QFileDialog`. Ke konverzi z typu `QString` slouží metoda `toPlainText()`. Nezadá-li uživatel příponu, je automaticky doplněna na `.txt`.

Po stisku tlačítka `Run` se vstup z komponenty `QLineEdit` převede z `QString` na `std::string` a gramatika zobrazená v `QTextEdit` se transformuje na `std::istringstream`. Vytvoří se instance třídy `Parser` a dále se aplikace chová jako konzolová verze.
Zobrazení výsledku analýzy závisí na průběhu:

- Dojde-li během výpočtu k chybě, je odpovídající chyba zobrazena v komponentě `QLabel`.
- Je-li analýza úspěšná, výsledkem je fronta použitých pravidel, která se zobrazí jako tabulka v komponentě `QTableWidget`.

Výsledek úspěšné analýzy získáme metodou `GetOutput()` třídy `Parser`, která vrací `std::queue<int>`. Pro výpis do tabulky je třeba získat strukturu přepisovacích pravidel (metoda `GetRulesStruct()`. Z takto získaných dat lze vytvořit tabulku zobrazovanou jako výsledek. V prvním sloupci tabulky jsou uvedena čísla pravidel, která jsme získali z výstupu a ve druhém sloupci se vypisují odpovídající pravidla.

Uživateli je k dispozici stručná nápověda, kterou lze zobrazit po stisku klávesy `F1` nebo v menu `help`. Nápověda je uložena ve formátu html , který se zobrazuje do nově vytvořeného okna.

```
1 QTextEdit *help = new QTextEdit(this);
2 help->setWindowFlags(Qt::Window);
3 help->setReadOnly(true);
4 help->setGeometry(0,0,600,400);
5 help->setWindowTitle("Help");
```

Zdrojový kód 14: Vytvoření okna nápovědy

7 Diskuze

Výsledná aplikace představuje pouze jednu část plnohodnotného parseru. Jedním z možných rozšíření by mohlo být přidání lexikálního analyzátoru.

Nedostatkem výsledné aplikace je uvažovaný vstup pouze jako posloupnost neterminálních symbolů. Uživatel tedy nemá možnost zadat například přímo zdrojový kód programu a musí vstup předem přepsat na symboly vyskytující se v gramatice, což znemožňuje praktické využití aplikace. Vypsáním posloupnosti použitých pravidel však dostatečně demonstruje průběh analýzy a díky tomu může sloužit k edukačním účelům.

Zápis gramatiky je velmi přehledný, jelikož jasně definuje velká písmena jako neterminální symboly a malá písmena jako terminální symboly (pro symbol ε je vyhrazena posloupnost znaků `<epsilon>`. Jako oddělovač levé a pravé strany pravidla se intuitivně užívá symbolu $\rightarrow$, který uživatel zapisuje jako posloupnost symbolů – a >). Tento způsob zápisu je v jistých ohledech omezující, avšak pro účely aplikace s cílem demonstrace průběhu analýzy vyhovuje z důvodu jednoduchosti.

Jelikož se aplikace zabývá analýzou jazyků typu LL(1), nepředpokládá se vstup s levou rekurzí. Zadá-li uživatel gramatiku s levou rekurzí, dojde k pádu programu. Tento nedostatek je možný odstranit transformacemi na LL(1) gramatiku pomocí levá faktorizace nebo odstranění levé rekurze. Transformace gramatiky na LL(1) však nebyly cílem této práce.

Problematika syntaktických analyzátorů je již mnohokrát zpracované téma a proto existuje nespočet aplikací. Většina z nich se však zabývá LALR[11] analýzou. Yacc je aplikace generující syntaktické analyzátory v programovacím jazyce C, GNU Bison umožňuje generování syntaktického analyzátoru v jazycích C, C++ nebo Java. Obě aplikace pracují na principu LALR analýzy.

Při srovnání obou metod syntaktické analýzy nalezneme několik odlišností. Výhoda analýzy metodou zdola-nahoru spočívá v její obecnosti. Existuje více jazyků LR než LL. Metoda shora-dolů nabízí snadnější implementaci, rozkladová tabulka je menší a jelikož se konstruuje derivační strom směrem od kořene k listům, uživateli se snadněji oznamují chyby v průběhu analýzy. Nevýhodou této metody je problém levé rekurze.

[11] Varianta syntaktického analyzátoru pracujícího metodou zdola-nahoru.

Závěr

V rámci bakalářské práce byla vytvořena aplikace demonstrující syntaktickou analýzu metodou shora-dolů pro operační systémy Windows a Linux. Konzolová verze aplikace vyžaduje jeden vstupní parametr, kterým je cesta k textovému souboru přepisovacích pravidel gramatiky. Výstup zachycuje levý rozklad jako posloupnost čísel použitých přepisovacích pravidel. Grafická verze umožňuje uživateli procházet adresářovou strukturu a vybrat požadovaný textový dokument s gramatikou, nebo ji zadat ručně. Výstup se zobrazuje přehledně v tabulce. Kontrola, zda je gramatika LL(1), probíhá při vytváření překladové tabulky. Aplikace tedy splňuje všechny zadané požadavky.

V průběhu vývoje aplikace bylo třeba vyřešit několik konfliktů. Jedním z nich byl převod `std::string` na `QString` zobrazovaný v komponentách grafického rozhraní a opačně. Využilo se metod `QString::fromStdString()` a `std::string::toStdString()`. Dále bylo třeba provést kompilaci grafického rozhraní v operačním systému Windows. Řešením byla instalace Qt pro Windows společně s rozšířením Add-in for Qt5 a následná kompilace ve Visual Studiu.

Tento dokument slouží především jako technická a uživatelská dokumentace pro seznámení uživatele se strukturou projektu, teoretických znalostí týkajících se konstrukce překladového automatu a důležitých součástí syntaktického analyzátoru nutných k vlastní analýze.

Conclusions

Top-down syntax analyzator application has been created as a part of the bachelor's thesis. This application was created for operating systems Windows and Linux and all of the requirements were met. The console version of the application requires a text file containing grammar rules. This file is passed on to the application as an argument. The output shows leftmost derivation as a number sequence of rules that was used in analysis. Graphical interface allows to choose grammar file in the file system. The output is presented in a clearly arranged table.

There were some conflicts to be solved during the development of the application. For example there was a conversion problem between `std::string` (simple text output) and `QString` (text format for graphical interface). There also was a problem with compiling on Windows system. The solution was to install Qt for Windows and Add-in for Qt5 extension. Finally, the application was compiled in the Visual Studio.

This document has been written as an user and technical documentation to acquaint users with the structure of the project and all of the parts used for creating automaton that runs the analysis itself.

A Obsah přiloženého CD/DVD

`bin/`
: Konzolová i grafická aplikace spustitelná přímo z CD/DVD.

`doc/`
: Text práce ve formátu PDF včetně souborů potřebných pro vygenerování PDF dokumentu.

`src/`
: Kompletní zdrojové texty programu.

`readme.txt`
: Instrukce pro spuštění programu.

`data/`
: Ukázková a testovací data použitá v práci a pro potřeby testování práce při tvorbě posudků a obhajoby práce.

Literatura

[1] MOLNÁR, L.; ČEŠKA, M.; MELICHAR, B. *Gramatiky a jazyky*. První vydání. Bratislava: Alfa, 1987. 188 s.

[2] VAVREČKOVÁ, Š.; ÚSTAV INFORMATIKY, Slezská univerzita. *Programování překladačů*. 2008. 218 s. ISBN 9788072484935.

[3] GRUNE, Dick. *Modern compiler design*. Chichester: John Wiley & Sons, 2000. 736 s. ISBN 0471976970.

[4] GRUNE, Dick; JACOBS, Ceriel J. *Parsing techniques: a practical guide*. 2nd ed. New York: Springer, 2008. 662 s. ISBN 0387689540.

[5] SIPSER, M. *Introduction to the theory of computation*. Boston: Thomson Course Technology, 2006. 431 s. ISBN 0-534-95097-3.

[6] MICROSOFT CORPORATION. *MSDN, dokumentace jazyka C++*. Dostupný z: ⟨https://msdn.microsoft.com/en-us/library/3bstk3k5.aspx⟩.

[7] THE QT COMPANY. *Qt Documentation*. Dostupný z: ⟨http://doc.qt.io/⟩.