VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

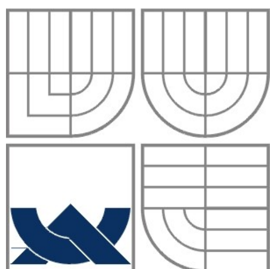
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DETEKCE A ZOTAVENÍ SE Z CHYB PŘI SYNTAKTICKÉ ANALÝZE

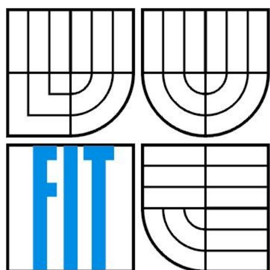
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE VLADIMÍR SÁK
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DETEKCE A ZOTAVENÍ SE Z CHYB PŘI SYNTAKTICKÉ ANALÝZE

DETECTION AND ERROR RECOVERY FOR SYNTACTIC ANALYSIS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE VLADIMÍR SÁK
AUTHOR

VEDOUCÍ PRÁCE Ing. EVA ZÁMEČNÍKOVÁ
SUPERVISOR

BRNO 2010

Abstrakt

Práce se zaobírá detekcí a zotavením se z chyb při syntaktické analýze. Hlavním cílem práce bylo navrhnutí a implementace metody pro detekci chyb a pro zotavení se z těchto chyb. Vytvořená metoda vychází z Hartmannovy metody detekce a zotavení se z chyb. Bylo implementováno i užívatelské rozhraní s využitím multiplatformní knihovny Qt, umožňující jednoduché ovládání programu. Ve výsledku, aplikace vypisuje všechny informace o chybách během analýzy zdrojového kódu.

Abstract

Bachelor's thesis deals with error detection and recovery for syntactic analysis. The main goal of work was to design and implement a method for error detection and recovery. The proposed method is based on the Hartmann method for error detection and error recovery. The user interface, using cross-platform framework Qt was also implemented. As a result, the application prints all of the information about errors while parsing the source code.

Klíčová slova

syntaktická analýza, analýza rekurzivním sestupem, analýza LL, detekce chyb, zotavení se z chyb, Hartmannova metoda, kontextová množina

Keywords

syntactic analysis, recursive descent parsing, LL parser, error detection, error recovery, Hartmann method, context set

Citace

Vladimír Sák: Detekce a zotavení se z chyb při syntaktické analýze, bakalářská práce, Brno, FIT VUT v Brně, 2010

Detekce a zotavení se z chyb při syntaktické analýze

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Evy Zámečníkové. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Vladimír Sák
19. května 2010

Poděkování

Týmto by som chcel poďakovať svojej vedúcej Ing. Eve Zámečníkovej za ochotu a pomoc pri písaní bakalárskej práce. Ďalej by som sa rád poďakoval svojej rodine za ich podporu a trpezlivosť.

© Vladimír Sák, 2010
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah....1
1 Úvod....3
2 Základné pojmy....4
2.1 Abecedy, reťazce a jazyky....4
2.1.1 Abeceda....4
2.1.2 Reťazec....4
2.1.3 Jazyk....4
2.1.4 Gramatika....4
2.1.5 Chomského hierarchia gramatík....4
2.2 Základné pojmy prekladačov....5
3 Štruktúra prekladačov....7
3.1 Lexikálna analýza....7
3.2 Syntaktická analýza....8
3.3 Sémantická analýza....8
3.4 Generátor vnútorného kódu....8
3.5 Optimalizátor....9
3.6 Generátor cieľového kódu....9
3.7 Tabuľka symbolov....9
4 Syntaktická analýza....10
4.1 Modely pre syntaktickú analýzu....10
4.1.1 Bezkontextové gramatiky....10
4.1.2 Derivačný strom....12
4.1.3 Zásobníkové automaty....12
4.1.4 Rozšírený zásobníkový automat....13
4.2 Syntaktická analýza zdola nahor....13
4.2.1 Precedenčná syntaktická analýza....14
4.2.2 Syntaktická analýza LR....14
4.3 Syntaktická analýza zhora nadol....15
4.3.1 LL-tabuľka....16
4.3.2 Analýza rekurzívnym zostupom....17
4.3.3 Nerekurzívna syntaktická analýza....17
5 Detekcia a zotavenie sa z chýb pri syntaktickej analýze....18
5.1 Hartmannova metóda....19
6 Návrh aplikácie....21

6.1 Lexikálna štruktúra jazyka....21
6.1.1 Kľúčové slová....21
6.1.2 Konštanty....21
6.1.3 Identifikátory....22
6.1.4 Operátory....22
6.1.5 Komentáre....22
6.2 Riadiace štruktúry jazyka....23
6.3 Sémantika jazyka....24
6.4 Návrh implemenácie....24
6.4.1 Vývojové prostredie....24
7 Implementácia....25
7.1 Lexikálna analýza – Trieda Scanner....25
7.1.1 Atribúty tokenov – štruktúra TokenValue....25
7.2 Tabuľka symbolov – Trieda HashTable....26
7.2.1 Štruktúra tabuľky symbolov – Štruktúra Htable....26
7.3 Syntaktická analýza – Trieda parser....27
7.3.1 Sémantická analýza....27
7.4 Množiny FIRST a FOLLOW – Trieda Sets....28
7.5 Užívateľské rozhranie....28
7.5.1 Menu aplikácie....29
8 Zhodnotenie dosiahnutých výsledkov....30
9 Záver....31
Literatúra....32
Zoznam príloh....33
Príloha A. CD....33
Príloha B. Konečný automat....34
Príloha C. Pravidlá gramatiky....35
Príloha D. Množiny FIRST a FOLLOW....38

1 Úvod

Syntaktická analýza je jednou z najdôležitejších fáz analýzy a prekladu zdrojového programu. Je to proces, ktorý na základe určitých gramatických pravidiel vytvára derivačný strom pre reťazec, nachádzajúci sa na jeho vstupe. Ak sa pre tento reťazec podarí derivačný strom vytvoriť, reťazec je prijímaný konkrétnym jazykom. V opačnom prípade sa v reťazci vyskytla syntaktická chyba a je potrebné previesť určitú formu zotavenia sa po chybách v priebehu analýzy.

Na začiatku mojej práce sú uvedené niektoré zo základných definícii a pojmov z teórie formálnych jazykov, dôležitých pre pochopenie ďaľšieho textu. V tejto kapitole je vysvetlený aj rozdiel medzi prekladačmi a interpretmi, taktiež popísané rôzne typy prekladačov, resp. kompilátorov.

Prekladač je súbor viacerých logických častí, medzi ktorými sú určité väzby. V každej z týchto častí prebieha určitý druh analýzy a prekladu zdrojového programu napísanom v konkrétnom programovacom jazyku na program cieľový. Tieto súčasti prekladačov sú popísané v kapitole 3.

Existujú 2 prístupy k syntaktickej analýze – syntaktická analýza zhora nadol a syntaktická analýza zdola nahor. Oba z týchto prístupov sa rozlišujú v tvorbe derivačného stromu. Počas syntaktickej analýzy zhora nadol sa derivačný strom vytvára od koreňa postupne až k listom, zatiaľ čo syntaktická analýza zdola nahor vytvára derivačný strom od listov ku koreňu. Proces tvorby derivačného stromu počas syntaktickej analýzy triedy gramatík z neho vychádzajúce sú popísané v nasledujúcej časti mojej práce. Na začiatku tejto časti sú uvedené aj formálne definície základných modelov pre syntaktickú analýzu, ako sú bezkontextové gramatiky, zásobníkové automaty a rozšírené zásobníkové automaty, ktoré majú rovnakú vyjadrovaciu silu.

Okrem analýzy syntaxe zdrojového kódu patrí medzi činnosti syntaktického analyzátora aj detekcia a zotavenie sa z chýb počas analýzy zdrojového textu. Je potrebné previesť čo najlepšiu formu zotavenia kvôli možnosti detekcie čo najväčšieho počtu týchto chýb. Najznámejšou metódou detekcie chýb pri syntaktickej analýze je Hartmannova metóda. Princíp tejto metódy ako aj obecných metód detekcie a zotavenia sa z chýb je vysvetlený v kapitole 5.

V šiestej kapitole je popísaný návrh jazyka využitého na demonštráciu detekcie a zotavenia sa z chýb počas procesu syntaktickej analýzy, ako aj návrh metódy detekcie a zotavenia sa z týchto chýb. Jazyk, zvolený na demonštráciu navrhnutej metódy vychádza z jazyka C++, je vlastne jeho podmnožinou. Metóda detekcie a zotavenia sa z chýb vychádza z Hartmannovej metódy.

Nasledujúca kapitola popisuje implementáciu aplikácie so zameraním sa na jej dôležité triedy. Aplikácia bola vytvorená s využitím multiplatformnej knižnice Qt, ktorá implementuje aj jednoduché užívateľské rozhranie.

Nie všetky syntaktické chyby sa počas procesu syntaktickej analýzy podarí detekovať správne. Dôvodu, prečo tomu tak je, sa venujem kapitola 8. Sú v nej uvedené príklady, ukázaný výstup a sú vysvetlené dôvody, v prípade ktorých sa na výstupe nachádzajú chyby, ktoré by sa tam v prípade úspešnej analýzy predchádzajúcej časti reťazca normalne neobjavili.

V závere bakalárskej práce je zhrnutý celkový dojem z práce ako aj navrhnutý ďalší možný vývoj projektu.

2 Základné pojmy

Na začiatku kapitoly budú vysvetlené základné pojmy z teórie formálnych jazykov, ako sú abeceda, jazyk a gramatika a bude predstavená Chomského hierarchia gramatík. Nasledujúca časť kapitoly sa venuje vysvetleniu pojmov rozdielu medzi interpretom a prekladačom, ako aj popisu niektorých ich typov. Definíce v texte sú prevzaté z [2] a [5].

2.1 Abecedy, reťazce a jazyky

2.1.1 Abeceda

Abeceda je ľubovoľná konečná neprázdna množina elementov, ktoré nazývame symboly.

2.1.2 Reťazec

Nech Σ je abeceda.

- ε je ***reťazec*** nad abecedou Σ.
- Ak x je ***reťazec*** nad Σ a $a \in \Sigma$, potom xa je reťazec nad abecedou Σ.

2.1.3 Jazyk

Nech Σ^* značí množinu všetkých reťazcov nad Σ. Každá podmnožina $L \subseteq \Sigma^*$ je ***jazyk*** nad Σ.
Jazyk L je konečný, ak L obsahuje konečný počet reťazcov, inak je nekonečný.

2.1.4 Gramatika

Generatívna ***gramatika*** je štvorica $G = (N, T, P, S)$, kde

- N je abeceda nonterminálov,
- T je abeceda terminálov, pričom $N \cap T = \emptyset$,
- P je konečná množina pravidiel tvaru $A \rightarrow x$, kde $A \in (N \cup T)^*N(N \cup T)^*$, $x \in (N \cup T)^*$,
- $S \in N$ je počiatočný nonterminál.

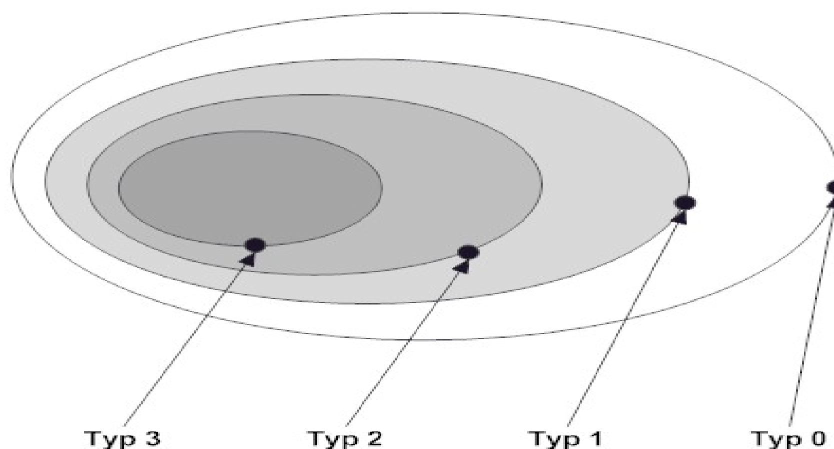
2.1.5 Chomského hierarchia gramatík

Gramatika definovaná v predchádzajúcej kapitole je príliš obecná. Je vhodné vhodným spôsobom upraviť jej pravidlá a vytvoriť niekoľko typov generatívnych gramatík. Tvar pravidiel gramatiky sa teda stane kritériom pri špecifikácii gramatík. K najvýznamnejším takýmto klasifikáciam Chomského hierarchia (*Tabuľka č.1*), ktorá definuje 4 typy gramatík. Na nasledujúcom obrázku (*Obr.1*) je graficky znázornený vzťah medzi týmyto typmi.

Obecne platí pravidlo: ***Typ 3*** $\subset$ ***Typ 2*** $\subset$ ***Typ 1*** $\subset$ ***Typ 0*** .

Typ	*Gramatiky*	*Generované jazyky*	*Tvar pravidiel $A \rightarrow x$*
Typ 0	Neobmedzené	Rekurzívne vyčísliteľné	$A \in (N \cup T)^*N(N \cup T)^*; x \in (N \cup T)^*$
Typ 1	Kontextové	Kontextové	$A \in (N \cup T)^*N(N \cup T)^*;\ x \in (N \cup T)^*, \|A\| \leq \|x\|$
Typ 2	Bezkontextové	Bezkontextové	$A \in N; x \in (N \cup T)^*$
Typ 3	Pravé lineárne	Lineárne	$A \in N; x \in T^* \cup T^*N$

Tabuľka č.1 – Chomského hierarchia gramatík.



Obr. č. 1 – Vzťahy medzi typmi gramatík Chomského hierarchie

2.2 Základné pojmy prekladačov

Kompilátor – Program, prekladajúci program napísaný vo vyššom programovacom jazyku na ekvivalentný program napísaný v nižšom programovacom jazyku.

Assembler – Program, ktorý prekladá program napísaný v strojovom jazyku (assembler) na ekvivalentný strojový kód.

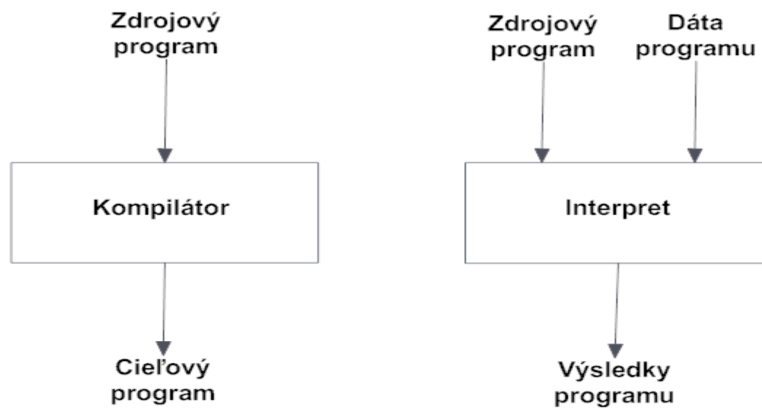
Dekompilátor – Program, prekladajúci program napísaný v nižšom programovacom jazyku na ekvivalentný program napísaný vo vyššom programovacom jazyku.

Disassembler – Program, ktorý prekladá strojový kód na ekvivalentný program napísaný v strojovom jazyku (assembler).

Prekladač – Program, prekladajúci zdrojový program (napísaný v zdrojovom jazyku) na ekvivalentný cieľový program (napísaný v cieľovom jazyku). Jeho hlavnou nevýhodou je, že pri každej, hoci aj drobnej zmene zdrojového programu je nutné opäť ho preložiť do programu cieľového, čo môže pre niektoré väčšie programy trvať celkom dlhú dobu.

Interpret – Program, ktorý je schopnýnielen preložiť, ale aj priamo vykonať program napísaný v zdrojovom jazyku. Medzi jeho nevýhody patrí, že pri každom prevedení zdrojového programu je nutné previesť svoje inštrukcie na ekvivalentné inštrukcie strojového kódu. Táto činnosť totiž spomaľuje činnosť prevedenia programov.

Interpretovaný kompilátor **–** Kompromis medzi prekladačom a interpretom. Zdrojový program je preložený na ekvivalentný cieľový program, obsahujúci inštrukcie vo vhodnom formáte, zrozumiteľnom pre interpret, pomocou ktorého sa cieľový program vykoná

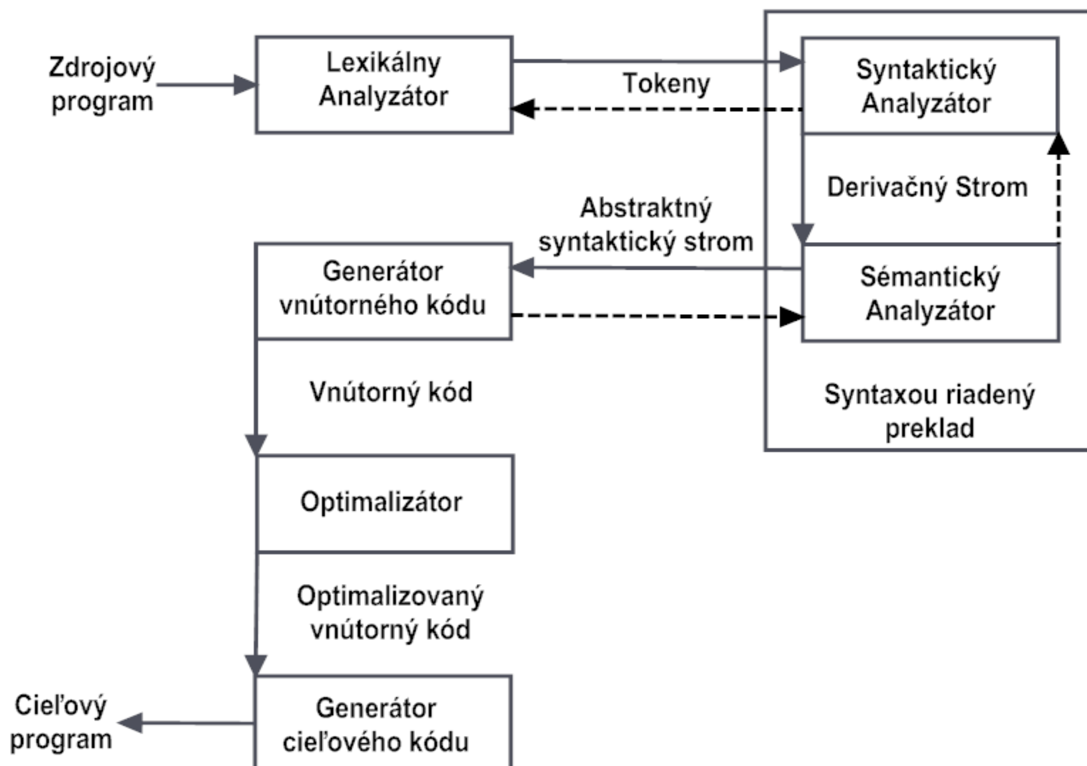


Obr.2 –Interpret vs. Prekladač

Pozn. V ďaľšom texte sa bude pod pojmom prekladač považovať ako prekladač tak aj interpret.

3 Štruktúra prekladačov

V tejto časti práce budú predstavené jednotlivé časti prekladačov a stručne bude popísaná ich činnosť. Prekladač sa skladá z viacerých logických častí (*Obr. 3*), medzi ktorými sú určité väzby. Medzi najhlavnejšie časti analýzy zdrojového textu programu patria procesy lexikálnej (scanner) a syntaktickej (parser) analýzy, ktoré sú medzi sebou úzko prepojené. Pri popise činností konkrétnych častí prakladača som vychádzal z [1] a [2].



Obr. 3 – Štruktúra prekladača

3.1 Lexikálna analýza

Preklad programu začína fázou lexikálnej analýzy, tzv. *scannera*. Lexikálny analyzátor číta zdrojový text programu znak po znaku a transformuje ho na prúd tzv. *tokenov,* ktoré predstavujú jednotlivé lexikálne jednotky daného jazyka (*lexémy*). Každý token je určitého typu, čo sa využíva pri syntaktickej analýze a môže obsahovať aj atribúty, napr. názov, prípadne hodnotu, ktoré sa ukladajú do tabuľky symbolov. Medzi tokeny patria identifikátory, kľúčové slová, operátory, konštanty, atď.

Ďalšou z činností lexikálneho analyzátora je aj odstraňovanie bielych znakov a komentárov, a prípadná komunikácia s tabuľkou symbolov. Lexikálny analyzátor je implementovaný na základe deterministického konečného automatu.

3.2 Syntaktická analýza

Druhou fázou prekladu je proces syntaktickej analýzy, tzv. parser. Syntaktický analyzátor na vstupe obdrží prúd tokenov, získaný počas lexikálnej analýzy, a v súlade s definíciou syntaxe konkrétneho jazyka vytvára derivačný strom, znázorňujúci gramatickú reprezentáciu vstupného programu. Ak sa derivačný strom pre daný vstupný prúd podarí vytvoriť, zdrojový program je zapísaný syntakticky správne.

Syntaktická analýza je teda fázou prekladu, ktorá využíva gramatickú štruktúru na pomoc pri analýze zdrojového programu a generovaní programu cieľového. Syntaktickej analýze sa podrobnejšie venuje kapitola 4.

3.3 Sémantická analýza

Sémantický analyzátor spracováva informácie z derivačného stromu, vytvoreného pomocou syntaktickej analýzy ako aj informácie z tabuľky symbolov na kontrolu sémantickej správnosti zdrojového programu. Taktiež zhromažďuje informácie o typoch premenných a ukladá ich do tabuľky symbolov, prípadne do abstraktného syntaktického stromu, pripravených na použitie v nasledujúcej časti prekladu.

Dôležitou činnosťou sémantického analyzátora je kontrola typov a deklarácii identifikátorov, počas ktorej prekladač zisťuje, či je každý identifikátor deklarovaný, prípadne či má každý operátor správny operand, napr. väčšina definícii programovacích jazykov požaduje index poľa ako celočíselnú konštantu, takže prekladač musí zahlásiť chybu, ak sa na jeho indexovanie použije konštanta desatinná.

Ak je kontrola sémantických pravidiel vykonávaná v rámci syntaktickej analýzy, hovoríme, že sa jedná o syntaxou riadený preklad. Syntaktický analyzátor teda okrem kontroly syntaktických pravidiel riadi aj prevádzanie sémantických akcí ako aj konštrukciu abstraktného syntaktického stromu. Tento princíp sa využíva vo väčšine prekladačov.

3.4 Generátor vnútorného kódu

Pri preklade zdrojového programu na program cieľový môže byť program prekladačom preložený do jedného alebo viacerých typov vnútorného kódu. Aj syntaktické stromy bežne používané pri syntaktickej analýze (derivačný strom) ako aj sémantickej analýze (abstraktný syntaktický strom) sú jednou z foriem vnútornej reprezentácie programu.

Po procese syntaktickej a sémantickej analýzy, väčšina prekladačov explicitne generuje tzv. *vnútorný kód* (najčastejšie 3-adresný), ktorý je reprezentáciou programu v nižšom programovacom jazyku, príp jazyku strojovom. Tento kód sa dá považovať ako program pre abstrakný počítač.

Táto fáza prekladu prebieha najmä kvôli prenositeľnosti a nezávislosti programu na strojovom jazyku konkrétneho počítača, ako aj kvôli jednoduchosti prípadnej optimalizácie. Vnútorný kód by mal byť jednoduchý, ľahko vytvoriteľný ako aj ľahko preložiteľný do cieľového jazyka.

Vnútorný kód môže byť aj konečným produktom prekladu, teda kódom cieľovým v interpretačných prekladačoch, ktoré vygenerovaný kód priamo interpretujú. Generovanie vnútorného kódu obvykle prebieha taktiež v rámci procesu syntaxou riadeného prekladu.

3.5 Optimalizátor

Optimalizácia vnútorného kódu patrí medzi voliteľné fázy prekladu. Závisí na strojovom jazyku príslušného počítača. Pokúša sa upraviť vnútorný kód na viac efektívny. Efektívny kód zvyčajne znamená rýchlejší, ale môžu byť požadované aj iné parametre, ako sú napr. kratší kód, alebo cieľový kód konzumujúci menej pamäťových prostriedkov, prípadne energie.

Optimalizácia sa doslovne nechápe ako nájdenie najlepšej varianty, niektoré optimalizačné varianty mäžu dokonca viesť k zhoršeniu vlastností vnútorného kódu. Optimalizačné algoritmy najčastejšie vykonávajú:

- šírenie konštanty,
- šírenie kopírovaním,
- elimináciu mŕtveho kódu.

3.6 Generátor cieľového kódu

Poslednou etapou v procese prekladu zdrojového kódu je generovanie kódu cieľového. V tejto fáze je vnútorný kód prekladaný na cieľový program. Jeho výstup závisí na strojovom jazyku príslušného počítača. Vyberajú sa pamäťové miesta pre dáta, určujú sa mechanizmy prístupu k nim a vyberajú sa registre, v ktorých bude prebiehať výpočet. Následne sa inštrukcie vnútorného kódu prekladajú na postupnosť inštrukcií strojového jazyka (assembleru). Najkritickejším miestom generovania cieľového kódu je priradenie premenných do registrov.

3.7 Tabuľka symbolov

Podstatnou funkciou prekladača je aj zaznamenávanie si mien premenných používaných v zdrojovom programe a zbieranie informácii o rôznych atribútoch týchto premenných. Tieto atribúty môžu poskytovať informácie o mieste alokovanom pre premennú, jej názve, type, rozsahu platnosti, hodnoty a v prípade mien procedúr a funkcií aj počet a typ ich vstupných parametrov, metódach predávania parametrov (hodnotou alebo adresou) a ich návratového typu.

Tabuľka symbolov je abstraktná dátová štruktúra obsahujúca záznam o každej premennej aj s jej atribútami. Táto dátová štruktúra by mala byť navrhnutá tak, aby bol prístup ku každej jej zložke dostatočne rýchly a taktiež aby bolo rýchle aj získavanie alebo ukladanie dát. Väčšinou je implementovaná formou hašovacej tabuľky alebo binárneho vyhľadávajúceho stromu.

4 Syntaktická analýza

Syntaktická analýza patrí k najdôležitejším fázam analýzy a prekladu zdrojového kódu na kód cieľový. Je to proces prekladača, počas ktorého sa na základe gramatických pravidiel vybraného programovacieho jazyka určuje, či daný reťazec spĺňa alebo nespĺňa podmienky tohto jazyka. K tomu využíva postupnosť lexikálnych symbolov (*tokenov*) zo vstupu, získanú ako výsledok lexikálnej analýzy. Ak sa počas syntaktickej analýzy vyskytnú nejaké chyby, prekladač ich zahlási a predvedie určitú formu zotavenia tak, aby mohol pokračovať v činnosti a odhaliť ďaľšie prípadné chyby.

Existujú 2 základné prístupy k syntaktickej analýze – ***Syntaktická analýza zhora nadol*** a ***Syntaktická analýza zdola nahor***. Ako už ich názvy predpovedajú, rozdiel je vo vytváraní derivačného stromu – v prípade analýzy programu zhora nadol vychádzame zo štartovacieho symbolu a snažíme sa postupnou expanziou nonterminálnych symbolov dospieť až k symbolom terminálnym, zodpovedajúcim postupnosti lexikálnych symbolov na vstupe, kým pri analýze programu zdola nahor sa snažíme postupnosť terminálnych symbolov zo vstupu redukovať až na štartovací nonterminál. Uvedeným dvom prístupom zodpovedajú 2 základné triedy gramatík – LR gramatiky (pre analýzu zdola nahor) a LL gramatiky (pre analýzu zhora nadol).

Pravidlá pre syntaktickú analýzu sú popísané pravidlami bezkontextových gramatík. Analyzátor sa na základe týchto pravidiel snaží z reťazca symbolov na vstupe zostrojiť derivačný strom. V prípade, ak sa mu daný strom podarí vytvoriť, jazyk reťazec prijíma. V opačnom prípade je objavená syntaktická chyba a analyzátor musú previesť určitú formu zotavenia a až potom môže pokračovať v analýze vstupného reťazca.

4.1 Modely pre syntaktickú analýzu

Na začiatku kapitoly sa budem venovať formálnym definíciam bezkontextových gramatík, zásobníkových automatov ako aj rozšírených zásobníkových automatov, ktoré sú modelmi pre syntaktickú analýzu. Z hľadiska vyjadrovacej sily sú tieto 3 modely na rovnakej úrovni. Pri definíciach som vychádzal z [2] a [5].

4.1.1 Bezkontextové gramatiky

Bezkontextová gramatika je štvorica $G = (N, T, P, S)$, kde

- N je abeceda nonterminálov,
- T je abeceda terminálov, pričom $N \cap T = \emptyset$,
- P je konečná množina pravidiel tvaru $A \rightarrow x$, kde $A \in N, x \in (N \cup T)^*$,
- $S \in N$ je počiatočný nonterminál.

Generovaný jazyk

Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Jazyk generovaný bezkontextovou gramatikou G, $L(G)$, je definovaný: $L(G) = \{w: w \in T^*, S \Rightarrow^* w\}$.

Bezkontextový jazyk

Nech L je jazyk. L je ***bezkontextový jazyk,*** ak existuje bezkontextová gramatika, ktorá generuje tento jazyk L.

Gramatická nejednoznačnosť

Pri syntaktickej analýze spôsobuje najväčsie problémy nejednoznačnosť bezkontextových gramatík. ***Nejednoznačnosť gramatík*** je definovaná nasledovne:
Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Ak existuje reťazec $x \in L(G)$ s viac ako jedným derivačným stromom, potom G je nejednoznačná. Inak G je jednoznačná.

Nejednoznačný jazyk

Bezkontextový jazyk L je vnútorne ***nejednoznačný***, ak L nie je generovaný žiadnou jednoznačnou bezkontextovou gramatikou.

Priamy Derivačný krok v bezkontextových gramatikách je popísaný nasledovne:
Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Nech $u, v \in (N \cup T)^*$ a $p = A \rightarrow x \in P$ je pravidlo. Potom uAv priamo derivuje uxv podľa pravidla p, zapísané $uAv \Rightarrow uxv\ [p]$ alebo skrátene $uAv \Rightarrow uxv$.

Sekvencia derivačných krokov je definovaná:

- Nech $u \in (N \cup T)^*$. Bezkontextová gramatika G prevedie nula derivačných krokov z u do u; zapisujeme: $u \Rightarrow^0 u\ [\varepsilon]$ alebo zjednodušene $u \Rightarrow^0 u$.
- Nech $u_0, \ldots, u_n \in (N \cup T)^*$, $n \geq 1$ a $u_{i-1} \Rightarrow u_i\ [p_i]$, $p_i \in P$ pre všetky $i = 1, \ldots, n$, čo znamená: $u_0 \Rightarrow u_1\ [p_1] \Rightarrow u_2\ [p_2]\ \ldots \Rightarrow u_n\ [p_n]$. Potom bezkontextová gramatika G prevedie n derivačných krokov $z\ u_0\ do\ u_n$; zapisujeme: $u_0 \Rightarrow^n u_n\ [p_1 \ldots p_n]$ alebo zjednodušene $u_0 \Rightarrow^n u_n$.
- Ak $u_0 \Rightarrow^n u_n\ [\pi]$ pre nejaké $n \geq 1$, potom u_0 derivuje u_n v bezkontextovej gramatike G, zapisujeme: $u_0 \Rightarrow^+ u_n\ [\pi]$.
- Ak $u_0 \Rightarrow^n u_n\ [\pi]$ pre nejaké $n \geq 0$, potom u_0 derivuje u_n v bezkontextovej gramatike G, zapisujeme: $u_0 \Rightarrow^* u_n\ [\pi]$.

Najľavejšia derivácia je definovaná:
Nech $u \in T^*, v \in (N \cup T)^*$ a $p = A \rightarrow x \in P$ je pravidlo. Potom uAv priamo derivuje v nejľavejšej derivácii uxv podľa pravidla p, zapísané $uAv \Rightarrow_{lm} uxv[p]$ alebo skrátene $uAv \Rightarrow_{lm} uxv$.

Najpravejšia derivácia je definovaná:
Nech $v \in T^*$, $u \in (N \cup T)^*$ a $p = A \rightarrow x \in P$ je pravidlo. Potom uAv priamo derivuje v nepravejšej derivácii uxv podľa pravidla p, zapísané $uAv \Rightarrow_{rm} uxv[p]$ alebo skrátene $uAv \Rightarrow_{rm} uxv$.

4.1.2 Derivačný strom

Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Strom je ***derivačný strom*** v G, ak

- každý uzol je ohodnotený symbolom z $N \cup T$,
- koreň je ohodnotený symbolom S,
- ak má symbol aspoň jedného nasledovníka, je ohodnotený symbolom z N,
- ak $b1$, $b2$, …, b_k sú priamymi nasledovníkmi uzlu a, ohodnoteného symbolom A, v poradí zľava doprava s ohodnotením $B_1, B_2, ..., B_k$, potom $A \rightarrow B_1B_2...B_k \in P$.

4.1.3 Zásobníkové automaty

Zásobníkový automat je sedmica $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde

- Q je konečná množina stavov,
- Σ je vstupná abeceda,
- Γ je zásobníková abeceda,
- R je konečná množina pravidiel tvaru $Apa \rightarrow wq$, kde $A \in \Gamma, p, q \in Q, a \in \Sigma \cup \{\varepsilon\}, w \in \Gamma^*$,
- $s \in Q$ je počiatočný stav,
- $S \in \Gamma$ je počiatočný symbol na zásobníku,
- $F \subseteq Q$ je množina koncových stavov.

Konfigurácia zásobníkového automatu M je reťazec $\chi \in \Gamma^*Q\Sigma^*$.

Prechod pri zásobníkovom automate je definovaný nasledovne**:**
Nech $xApay$ a $xwqy$ sú dve konfigurácie zásobníkového automatu M, kde $x, w \in \Gamma^*, A \in \Gamma, p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ a $y \in \Sigma^*$. Nech $r = Apa \rightarrow wq \in R$ je pravidlo. Potom M môže previesť prechod z $Apay$ do $xwqy$ za použitia r, čo zapíšeme ako $xApay \vdash xwqy\ [r]$ alebo zjednodušene $xApay \vdash xwqy$.

Sekvencia prechodov pri zásobníkovom automate je definovaná nasledovne**:**

- Nech χ je konfigurácia zásobníkového automatu. M prevedie nula prechodov z χ do χ; zapisujeme: $\chi \vdash^0 \chi\ [\varepsilon]$ alebo zjednodušene $\chi \vdash^0 \chi$.
- Nech $\chi_0, \chi_1, ..., \chi_n$ je sekvencia prechodov konfigurácii zásobníkového automatu pre $n \geq 1$ a $\chi_{i-1} \vdash \chi_i\ [r_i]$, $r_i \in R$ pre všetky $i = 1, ..., n$, čo znamená: $\chi_0 \vdash \chi_1\ [r_1] \vdash \chi_2\ [r_2]\ ... \vdash \chi_n\ [r_n]$. Potom M prevedie n prechodov z χ_0 do χ_n; zapisujeme: $\chi_0 \vdash^n \chi_n\ [r_1... r_n]$, alebo zjednodušene $\chi_0 \vdash^n \chi_n$.
- *Ak* $\chi_0 \vdash^n \chi_n\ [\rho]$ pre nejaké $n \geq 1$, *potom* $\chi_0 \vdash^+ \chi_n\ [\rho]$.
- *Ak* $\chi_0 \vdash^n \chi_n\ [\rho]$ pre nejaké $n \geq 0$, *potom* $\chi_0 \vdash^* \chi_n\ [\rho]$.

Zásobníkové automaty prijímajú 3 typy jazykov:

- *Jazyk prijímaný prechodom do koncového stavu* je definovaný:
  - $L(M)_f = \{w: w \in \Sigma^*, Ssw \vdash^* zf, z \in \Gamma^*, f \in F\}$
- *Jazyk prijímaný vyprázdnením zásobníku* je definovaný:
  - $L(M)_\varepsilon = \{w: w \in \Sigma^*, Ssw \vdash^* zf, z = \varepsilon, f \in Q\}$
- *Jazyk prijímaný prechodom do koncového stavu a vyprázdnením zásobníku* je definovaný:
  - $L(M)_{f\varepsilon} = \{w: w \in \Sigma^*, Ssw \vdash^* zf, z = \varepsilon, f \in F\}$

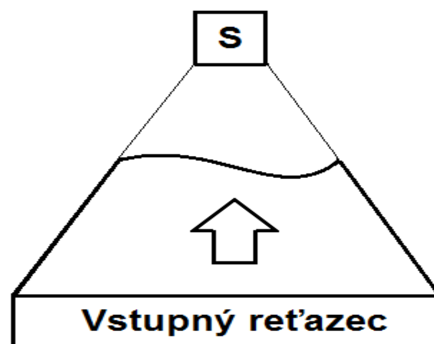
4.1.4 Rozšírený zásobníkový automat

Rozšírený zásobníkový automat je sedmica $M = (Q, \Sigma, \Gamma, R, s, S, F)$, kde

- Q je konečná množina stavov
- Σ je vstupná abeceda
- Γ je zásobníková abeceda
- R je konečná množina pravidiel tvaru $vpa \rightarrow wq$, kde $v, w \in \Gamma^*; p, q \in Q; a \in \Sigma \cup \{\varepsilon\}$
- $s \in Q$ je počiatočný stav
- $S \in \Gamma$ je počiatočný symbol na zásobníku
- $F \subseteq Q$ je množina koncových stavov

4.2 Syntaktická analýza zdola nahor

Pri syntaktickej analýze zdola nahor (*Obr. č.4*), známej tiež ako analýza typu *presun – redukcia*, sa derivačný strom vytvára postupne od listov smerom až ku koreňu. Existujú 2 metódy analýzy zdola nahor – precedenčná syntaktická analýza alebo analýza LR. Obidve metódy vykonávajú 2 typy operácii – redukciu na zásobníku (pri každom z redukčných krokov je nahradený istý podreťazec, ktorý sa zhoduje s pravou stranou prepisovacieho pravidla symbolom na ľavej strane pravidla) a presun symbolu zo vstupu na zásobník.



Obr. č. 4 – Syntaktická analýza zdola nahor

4.2.1 Precedenčná syntaktická analýza

Precedenčná syntaktická analýza využíva precedenčnú tabuľku, v ktorej sú zaznamenané priority jednotlivých operátorov ako aj ich asociativita. Na základe precedenčnej tabuľky sa určí operácia, ktorá sa má pre konkrétny stav vstupného reťazca a reťazca na zásobníku vykonať. Ak po analýze na zásobníku zostane iba štartovací nonterminál danej gramatiky, derivačný strom sa podarilo úspešne vytvoriť, analýza prebehla v poriadku.

Obecný algoritmus precedenčnej syntaktickej analýzy má tvar:

- Vlož na zásobník symbol *$*.
- Hlavný cyklus:
    - Nech *a* je aktuálny vstupný symbol, *b* je najvrchnejší terminálny symbol na zásobníku. Podľa obsahu políčka precedenčnej tabuľky na súradniciach [*b, a*] rozhodni:
        - = – Načítaj symbol *a* zo vstupu a pridaj ho na vrchol zásobníku.
        - < – Nájdi na zásobníku najvrchnejší terminálny symbol *b*. Hneď za tento symbol umiestni na zásobník symbol <. Načítaj symbol *a* zo vstupu a umiestni ho na vrchol zásobníku.
        - > – Nájdi na zásobníku najvrchnejší symbol <. Medzi týmto symbolom a vrcholom zásobníka nájdi pravú stranu istého pravidla *r*. Odstráň túto časť zo zásobníka vrátane symbolu <. Vlož na zásobník ľavú stranu pravidla *r*, príp zapíš na výstup, že bola prevedená redukcia podľa pravidla *r*.
        - Prázdne políčko – syntaktická chyba vo vstupnom reťazci.
- Ak *a = $* a *b = $*, syntaktická analýza prebehla v poriadku, inak preveď ďalšiu iteráciu cyklu.

4.2.2 Syntaktická analýza LR

Syntaktická analýza LR sa počas svojej činnosti snaží vytvoriť pravú deriváciu vstupného reťazca. Jej výhoda spočíva vo viacerých dôvodoch:

- Je možné vytvoriť analyzátory k rozpoznaniu takmer všetkých možných konštrukcií programovacích jazykov, ktoré sú syntakticky definované bezkontextovými gramatikami.
- Metóda analýzy LR je najobecnejšia známa metóda typu *presun - redukcia* pracujúca bez návratu, pre ktorú je možné detekovať chybu tak rýchlo po jej výskyte, ako je to možné pri prehľadávaní vstupného reťazca zľava doprava.

Analýza je založená na LR tabuľke, zloženej z 2 častí – akčnej časti a prechodovej časti. Najväčšou nevýhodou tejto analýzy je manuálne vytvorenie LR analyzátoru. Práve z tohto dôvodu sa na jeho vytvorenie využíva automatický konštruktor. Existuje viac techník konštrukcie LR analyzátoru:

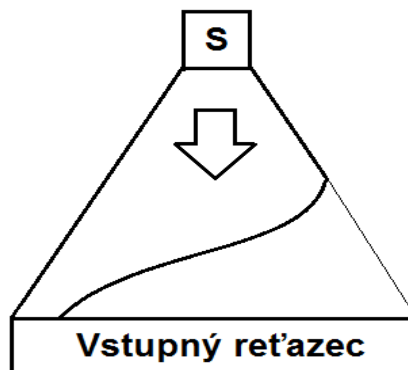
- Metóda pre *SLR gramatiky* – je najjednoduchšia na implementáciu, ale pokrýva najmenšiu triedu gramatík. Môže zlyhať pri vytváraní rozkladových tabuliek pre niektoré gramatiky.
- Metóda pre obecné *LR gramatiky* – je najobecnejšia, ale veľmi náročná na čas a pamäťový priestor.
- Metóda pre *LALR gramatiky* – je kompromisom medzi predchádzajúcimi metódami. Je možné použiť ju pre väčšinu gramatík programovacích jazykov a jednoducho ju implementovať.

Obecný algoritmus pre LR analyzátory má tvar:

- Vlož na zásobník dvojicu symbolov $<\$, q_0>$, nastav aktuálny stav *s* na q_0.
- Hlavný cyklus:
    - Nech *a* je aktuálny vstupný symbol, *s* je aktuálny stav. Podľa obsahu políčka LR-tabuľky akčnej časti na súradniciach [*s, a*] rozhodni:
        - *sq* – načítaj symbol *a* zo vstupu a daj dvojicu $<a, q>$ na zásobník. Aktuálny stav *s* nastav na *q*.
        - *rp* – nech pravidlo s návestím *p* je v tvare $A \rightarrow X_1X_2...X_d$. Potom:
            - Skontroluj, či je prvých *d* symbolov na zásobníku v tvare: $<X_1, ?><X_2, ?> ... <X_d, ?>$, inak nastala chyba.
            - Nech sa ešte hneď pred prvkom $<X_1, ?>$ nachádza prvok v tvare $<?, q>$, kde *q* je nejaký stav. Nastav aktuálny stav *s* na hodnotu, ktorú obsahuje LR-tabuľka prechodovej časti na súradniciach [*q, A*], odstráň symboly $<X_1, ?><X_2, ?> … <X_d, ?>$ zo zásobníka a vlož naň dvojicu $<A, s>$.
        - ☺ – koniec – syntaktická analýza prebehla úspešne.
        - Prázdne políčko – chyba.
- Preveď ďalšiu iteráciu cyklu.

4.3 Syntaktická analýza zhora nadol

Syntaktická analýza zhora nadol (*Obr. č. 5*) vytvára derivačný strom postupne od koreňa smerom k listom. Analýzu zhora nadol je možné popísať ako proces hľadania ľavej derivácie vstupného reťazca, ako aj proces vytvárania derivačného stromu, začínajúci jeho koreňom. Jednou z foriem realizácie tohto procesu je môže byť metóda *pokus-omyl*, keď sa na daný reťazec snažíme postupne aplikovať jednotlivé pravidlá danej gramatiky v snahe vytvoriť derivačný strom daného jazyka. Po neúspechu sa vracia do bodu, z ktorého je možné v analýze pokračovať. Tento rekurzívny postup sa nazýva syntaktická analýza s návratmi, ale veľmi sa nepoužíva, pretože je značne neefektívny, a pre účely prekladu programovacích jazykov nevhodný. Naštastie väčšina bežných konštrukcií programovacích jazykov umožňuje priamočiaru analýzu bez návratov.



Obr.č. 5 – Syntaktická analýza zhora nadol

4.3.1 LL-tabuľka

Najčastejšiu formou realizácie syntaktickej analýzy zhora nadol je vytvorenie LL-gramatiky. LL-gramatika sa vytvára pomocou LL-tabuľky. Konštrukcia LL-tabuľky je pomerne obtiažna, je potrebné definovať si a naplniť nasledujúce množiny:

- Množina ***Empty***: Nech $G = (N, T, P, S)$ je bezkontextová gramatika.
  - $Empty(x) = \{\varepsilon\}$ ak $x \Rightarrow^* \varepsilon$; inak
  - $Empty(x) = \emptyset$, kde $x \in (N \cup T)^*$.
- Množina ***First***: Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Pre každé $x \in (N \cup T)^*$ je definované $First(x)$ ako:
  - $First(x) = \{a: a \in T, x \Rightarrow^* ay; y \in (N \cup T)^*\}$.
- Množina ***Follow***: Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Pre všetky $A \in N$ definujeme množinu $Follow(A)$ ako:
  - $Follow(A) = \{a: a \in T, S \Rightarrow^* xAay, x, y \in (N \cup T)^*\} \cup \{\$: S \Rightarrow^* xA, x \in (N \cup T)^*\}$.
- Množina ***Predict***: Nech $G = (N, T, P, S)$ je bezkontextová gramatika. Pre každé $A \rightarrow x \in P$ definujeme množinu $Predict(A \rightarrow x)$ ako:
  - ak $Empty(x) = \{\varepsilon\}$ potom: $Predict(A \rightarrow x) = First(x) \cup Follow(A)$, inak
  - ak $Empty(x) = \emptyset$ potom: $Predict(A \rightarrow x) = First(x)$.

Algoritmus tvorby LL-tabuľky

- Najprv si vypočítame množiny $First(X)$ a $Empty(X)$ pre každé $X \in N \cup T$.
- Pomocou množín $First(X)$ a $Empty(X)$ pre každé $X \in N \cup T$ určíme pre reťazec $x \in (N \cup T)^+$ množiny $First(x)$ a $Empty(x)$.
- Vypočítame množinu $Follow(A)$ pre každé $A \in N$.
- Ako poslednú vypočítame množinu $Predict(r)$ pre každé pravidlo r, tvaru $A \rightarrow x$.
- Nakoniec skonštruujeme výslednú LL-tabuľku. Tabuľka má záhlavie stĺpcov popísané terminálnymi symbolmi a symbolom $\$$, označujúcim koniec programu a záhlavie riadkov má popísané nonterminálnymi symbolmi.

Transformácia na LL-gramatiku

Na základe vytvorenia predchádzajúcich množín je vytvorenie LL-tabuľky pomerne jednoduché. Ak na jedno miesto tabuľky odkazuje viac ako 1 pravidlo, nejedná sa o LL-gramatiku, preto je potrebné upraviť ju na gramatiku LL. Úprava na LL-gramatiku je možná:

- ***Odstránením ľavej rekurzie*** – Obecne môžeme zľava rekurzívne pravidlo zapísať ako :
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_n \mid \beta_1 \mid ... \mid \beta_m$, kde reťazce β_i nezačínajú nonterminálom A. Takéto pravidlo môžeme prepísať zavedením nového nonterminálu A' ako:
    - $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A'$,
    - $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$.
- ***Faktorizáciou pravidiel*** – Ak začína niekoľko pravých strán A-pravidla tým istým reťazcom terminálnych symbolov, t.j. ak má pravidlo tvar:
  - $A \rightarrow \beta\alpha_1 \mid \beta\alpha_2 \mid \dots \mid \beta\alpha_n$, môžeme previesť ich vytknutie opäť zavedením nového nonterminálu A' s pravidlami:
    - $A \rightarrow \beta A'$,
    - $A' \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$.

- ***Elimináciou Pravidiel*** **–** Niektorým konfliktom sa môžeme vyhnúť tak, že za niektoré nonterminály dosadíme ich pravé strany a tím odstránime z gramatiky pravidlá, spôsobujúce konflikt.
- ***Redukciou množiny Follow*** **–** Ak je pre niektorý nonterminál porušená podmienka, môžeme pridať nový nonterminál, ktorý vedie k zmenšeniu počtu prvkov konfliktnej množiny, príp. k disjunktnosti tejto množiny *FOLLOW* s množinami *FIRST* ostatných pravých strán pravidiel konfliktného nonterminálu.

4.3.2 Analýza rekurzívnym zostupom

Jednou z foriem implementácie syntaktickej analýzy metódou zhora nadol je aj analýza rekurzívnym zostupom. Je to metóda, pri ktorej sa každý nonterminálny symbol gramatiky analyzuje v samostatnej procedúre. Analýza začína volaním procedúry zodpovedajúcej štartovaciemu nonterminálnemu symbolu gramatiky. Z nej sa rekurzívne volajú procedúry symbolizujúce ostatné nonterminály jazyka.

4.3.3 Nerekurzívna syntaktická analýza

Okrem metódy rekurzívneho zostupu existuje aj metóda prediktívnej syntaktickej analýzy, využívajúca vlastný zásobník, na ktorý sa ukladajú, príp. odstraňujú pravé strany pravidiel gramatiky.

Obecný algoritmus nerekurzívnej syntaktickej analýzy má tvar:

- Vlož na zásobní symboly *$S*, kde *S* je štartujúcim neterminálnym symbolom.
- Hlavný cyklus:
  - Nech *a* je aktuálny vstupný symbol, *X* je najvrchnejší symbol na zásobníku
    - $X = \$$ – Ak $a = \$$, analýza prebehla v poriadku, inak sa počas analýzy vyskytla chyba
    - $X \in T$ – Ak $X = a$, prečítaj symbol *a* na vstupe a odstráň symbol *a* zo zásobníku; inak počas syntaktickej analýzy sa vyskytla chyba.
    - $X \in N$ – Ak LL-tabuľka na súradniciach $[X, a]$ obsahuje pravidlo $r{:}X \rightarrow x$, odstráň symbol *X* zo zásobníku a vlož naň reverzovane reťazec *x*; inak nastala chyba počas syntaktickej analýzy.

5 Detekcia a zotavenie sa z chýb pri syntaktickej analýze

Dôležitou činnosťou syntaktického analyzátora okrem prevádzania syntaktickej analýzy je aj detekcia a zotavenie sa po chybách počas analýzy. Aby prekladač v rámci jedného priechodu zdrojovým kódom programu odhalil čo najviac chýb, je potrebné implementovať prostriedky, ktoré dovolia, aby syntaktický analyzátor pokračoval v kontrole správnosti programu aj po výskyte syntaktickej chyby. Bežne používané metódy pri zotavovaní sa z chýb vychádzajú z nasledovného obecného postupu:

- Po odhalení syntaktickej chyby sa vo vstupnom reťazci hľadá miesto (bod zotavenia), od ktorého môže analýza pokračovať v činnosti, pričom sa vynechá určitá časť vstupného prúdu symbolov. Táto časť nie je analyzovaná, preto môže byť zdrojom ďalších chýb. Bod zotavenia je obvykle určený nájdením nejakého symbolu z množiny tzv. kľúčov.
- Syntaktický analyzátor prevedie synchronizáciu podľa pozície nájdeného kľúča v gramatike a pokračuje ďalej v činnosti.

Množina kľúčov musí byť definovaná tak, aby obsahovala, ak sa to dá, len tie symboly, ktorých výskyt v gramatike je čo najjednoznačnejší. Tým je zaistená vyššia spoľahlivosť synchronizácie analyzátoru pri zotavovaní. Ak je však množina kľúčov príliš obmedzená, rastie dĺžka neanalyzovanej časti textu, preskakovaného pri vyhľadávaní kľúča vo vstupnej vete.

Pri zotavení sa po chybách pri syntaktickej analýze zhora nadol sa väčšinou využívajú nasledujúce metódy:

- *nerekurzívna metóda s pevnou množinou kľúčov* – metóda vychádza z dopredu vypočítanej množiny kľúčov. Ku každému z kľúčov je dostupná aj informácia o tom, konkrétne ktorú syntaktickú konštrukciu ukončuje.
- *rekurzívna metóda s pevnou množinou kľúčov* – vylepšenie predchádzajúcej metódy určením množiny kľúčov, ktorými určité syntaktické konštrukcie začínajú. Ak je počas vyhľadávania bodu zotavenia nájdený niektorý z týchto kľúčov, spustí sa analýza vnorenej konštrukcie a po jej ukončení sa pokračuje v zotavovaní. Tým sa obmedzí rozsah neanalyzovaného textu, a môže nastať odhalenie ďalších chýb v zanorených konštrukciách programu.
- *metóda s dynamicky budovanou množinou kľúčov* – pri tejto metóde sa množina kĺúčov vytvára vždy na základe okamžitého kontextu. Jednou z metód tejto skupiny je Hartmannova metóda, využívajúca zjednotenie množín *FOLLOW* rozpracovaných nonterminálov.

5.1 Hartmannova metóda

Každému syntakticky správne vytvorenému programu analyzovanému syntaktickým analyzátorom zodpovedá príslušný derivačný strom. Pri syntaktickej analýze metódou rekurzívneho zostupu sa derivačný strom vytvára postupným vyvolávaním procedúr odpovedajúcim jednotlivým nonterminálom gramatiky a ich prevádzaním. Výskyt prípadnej syntaktickej chyby predstavuje z hľadiska syntaktického analyzátora situáciu, keď v určitej fáze budovania derivačného stromu v ňom nie je ďalej možné pokračovať. Začlenenie prostriedkov pre zotavenie sa po chybe pomocou Hartmannovej metódy predpokladá, že:

- sa pri výskyte chyby ukončí vytváranie podstromu derivačného stromu s tým, že daný podstrom je považovaný za správne vytvorený.
- sa pri výskyte chyby preskočia všetky symboly medzi chybou a koncom fráze zodpovedajúcej uzavretému podstromu derivačného stromu.

Snahou dobrého zotavenia sa po výskyte chýb je uzavretie čo najtesnejšieho podstromu, teda preskočenie čo najmenšieho počtu symbolov zo vstupu, ale zároveň aj odhalenie čo najväčšieho počtu chýb, ktoré sa v reťazci nachádzajú. Pretože preskakované tokeny nie sú analyzované, môžu byť zdrojom ďalších syntaktických chýb, ktoré sa nám bohužiaľ odhaliť nepodarí.

V každom kroku analýzy sa vytvára derivačný podstrom pre určitý počet nonterminálov, a tieto podstromy sú do seba vnorené. Každému rozpracovanému derivačnému podstromu priradíme množinu symbolov, nazývanou *CONTEXT(A)*, ktorá je zjednotením množín $FOLLOW(A_i)$ všetkých nonterminálov, ktoré majú v okamihu analýzy nonterminálu *A* rozpracovaný derivačný podstrom, vrátane množiny *FOLLOW(A).* Ak vznikne v priebehu vytvárania derivačného podstromu chyba, musí dojsť k zotaveniu.

Množina *CONTEXT(A)* je vlastne dynamicky buduvanou množinou kľúčov, ktoré využívame pri hľadaní bodu zotavenia. Po objavení chyby sa preskočia všetky symboly na vstupe, ktoré sa v nej nenachádzajú. Pretože všetky symboly z množiny *CONTEXT(A)* sú zároveň aj prvkami apoň jednej množiny *FOLLOW* pre jednotlivé vnorené nonterminály, je zabezpečené, že sa počas zotavenia sa z chyby preskočí čo najmenší možný počet symbolov zo vstupu a podarí sa objaviť čo najbližší možný bod zotavenia v danom kontexte. Zároveň sa musí uzavrieť analýza všetkých nonterminálov, v ktorých množinách FOLLOW nie je nastavený vstupný symbol obsiahnutý. Posledným nonterminálom, ktorého analýza sa uzavrie, je nonterminál, v ktorého množine FOLLOW sa bod zotavenia nachádza.

Pri analýze metódou rekurzívneho zostupu môže dojsť k odhaleniu syntaktickej chyby v nasledujúcich prípadoch:

- na vstupe sa nachádza iný symbol ako sa očakával, alebo
- na základe vstupného symbolu nie je možné vybrať pravú stranu žiadneho pravidla – tento prípad sa zisťuje bezprostredne pri vstupe do procedúry analyzujúcej konkrétny nonterminál.

Pri analýze rekurzívnym zostupom pri detekcii syntaktických chýb a zotavení volané procedúry implementujúce nonoterminály obdržia ako vstupný parameter aktuálnu kontextovú množinu kľúčov. Následne sa pri ich analyzovaní ich vnútorných procedúr vypočíta nová kontextová množina. Na začiatku každej z procedúr sa pred voľbou varianty v nonterminále sa zistí, či aktuálny symbol na vstupe aj zodpovedá niektorej z pravých strán pre práve analyzovaný nonterminál.

Výpočet kontextovej množiny symbolov X_i na pravej strane pravidla $A \rightarrow X_1X_2...X_iX_{i+1}...X_k$ spočíva v rozšírení súčasne kontextovej množiny $CONTEXT(A)$ o symboly, ktoré sa stanú kľúčmi pre nasledujúci analyzovaný terminálny alebo nonterminálny symbol. Na výbere kontextových množín podstatne závisí kvalita zotavenia, ktorá sa prejavuje nielen počtom odhalených skutočných chýb, ale (v opačnom slova zmysle) aj počtom hlásených zavlečených chýb.

Pri výpočte množín kontext sú možné napr. nasledujúce prístupy, ktoré sa zvyčajne zvyknú kombinovať:

- Kontextovú množinu nonterminálov X_i vždy rozšírime o prvky množiny $FOLLOW(X_i)$, t.j. $CONTEXT(X_i) = CONTEXT(A) \cup FOLLOW(X_i)$, $X_i \in N$, zatiaľ čo kontextovú množinu terminálnych symbolov ponecháme pôvodnú, t.j. $CONTEXT(X_i) = CONTEXT(A)$, $X_i \in \Sigma$.
- Kontextovú množinu symbolov X_i (terminálneho aj nonterminálneho) vždy rozšírime o symboly, ktorými môže začínať zostávajúca časť reťazca na pravej strane pravidla, t.j. $CONTEXT(X_i) = CONTEXT(A) \cup (FIRST(X_{i+1}...X_k) \setminus \{\varepsilon\})$.
- Kontextovú množinu symbolov X_i rozšírime o symboly ležiace vo $FIRST$ všetkých nasledujúcich symbolov v pravidle, t.j.

$$CONTEXT(X_i) = CONTEXT(A) \cup (\bigcup_{j=i+1}^{k} FIRST(X_j) \setminus \{\varepsilon\}).$$

Viac informácii o metódach popísaných v predošlej kapitole nájdete v [2] a [4].

6 Návrh aplikácie

V tejto časti práce je popísaný návrh jazyka vytvoreného na demonštráciu a otestovanie metódy detekcie a zotavenia sa z chýb pri syntaktickej analýze, ako aj návrh samotnej metódy. Hlavný princíp navrhnutej metódy detekcie a zotavenia sa z chýb vychádza z Hartmannovej metódy. Počas syntaktickej analýzy metódou rekurzívneho zostupu obdrží každá procedúra analyzujúca nonterminálny symbol gramatiky na svojom vstupe kontextovú množinu, pozostávajúcu z množín FOLLOW, prípadne FIRST všetkých rozpracovaných nonterminálov.

Na predvedenie funkčnosti a otestovanie metódy detekcie a zotavenia sa z chýb pri syntaktickej analýze som si zvolil jazyk, ktorý je podmnožinou jazyka C++ a má jeho podobné vlastnosti. Daný jazyk som si vybral kvôli ľahkému pochopeniu jeho syntaktických a lexikálnych pravidiel, ako aj kvôli tomu, že jazyk C++ je jedným z najrozšírenejších programovacích jazykov súčasnosti.

6.1 Lexikálna štruktúra jazyka

6.1.1 Kľúčové slová

Medzi kľúčové slová daného programovacieho jazyka patria: *main, if, else, while, do, for, cin, cout, cerr, return, true, false,* ako aj názvy, príp vlastnosti jednotlivých typov – *int, long, short, char*, *signed, unsigned, float, double, string*, *bool* a *const*. Medzi kľúčové slová ďalej patria aj názvy operácii, ktorých význam je ekvivalentný ich znakovému znázorneniu. Sú to: *and, or, bitand, bitor, xor, and_eq, or_eq* a *xor_eq*.

6.1.2 Konštanty

Konštanty rozdeľujeme na celočíselné, desatinné, znakové a reťazcové. Celočíselné konštanty môžu byť zapísané vo viacerých číselných sústavách – dvojkovej, osmičkovej, desiatkovej alebo šestnástkovej.

Príklady konštánt:

- `123, +1506, -123` – celočíselné konštanty v desiatkovej sústave,
- `077, 0564` – celočíselné konštanty v osmičkovej súsatve,
- `1110b, 0111b` – celočíselné konštanty v dvojkovej sústave,
- `0xFF, 0xab0` – celočíselné konštanty v šestnástkovej sústave,
- `0.23, 1e34, .504` – desatinné konštanty,
- `“Ahoj“, “Ja\nSom”` – reťazcové konštanty,
- `'a', '\t'` – znakové konštanty.

6.1.3 Identifikátory

Identifikátory zodpovedajú regulárnemu výrazu: [*_A-Za-z*][*_A-Za-z0-9*], teda identifikátor musí začínať písmenom alebo znakom „_“, nasledovaným ľubovoľným počtom písmen, číslic alebo znakov „_“. Identifikátor nemôže začínať číslicou.

6.1.4 Operátory

Operátory jazyka majú rovnaké vlastnosti ako ekvivalentné operátory jazyka C++, teda majú rozdielnu prioritu, príp. asociativitu (*Tabuľka č. 2*). Medzi operátory daného jazyka patria:

```
=     +=    -=    /=    %=    *=    >>=   <<=   |=    ^=    &=
||    &&    |     &     ^     ==    !=    >     <     <=    >=
>>    <<    +     -     *     /     %     ++    --    !     ~
( )   ,
```

<table>
<tr><th>Priorita</th><th>Operátory</th><th>Popis</th><th>Asociativita</th></tr>
<tr><td>1</td><td>( )</td><td>Zátvorky</td><td>→</td></tr>
<tr><td>2</td><td>++ -- ! ~</td><td>Inkrementácia, dekrementácia, negácia</td><td>←</td></tr>
<tr><td>3</td><td>* / %</td><td>Násobenie, delenie, modulo</td><td>→</td></tr>
<tr><td>4</td><td>+ -</td><td>Sčítanie, odčítanie</td><td>→</td></tr>
<tr><td>5</td><td>>> <<</td><td>Bitové posuny</td><td>→</td></tr>
<tr><td>6</td><td>< > <= >=</td><td>Operátory porovnávania</td><td>→</td></tr>
<tr><td>7</td><td>== !=</td><td>Operátory rovnosti</td><td>→</td></tr>
<tr><td>8</td><td>&</td><td>Bitový súčin</td><td>→</td></tr>
<tr><td>9</td><td>^</td><td>Exkluzívny bitový súčet</td><td>→</td></tr>
<tr><td>10</td><td>|</td><td>Bitový súčet</td><td>→</td></tr>
<tr><td>11</td><td>&&</td><td>Logický súčin</td><td>→</td></tr>
<tr><td>12</td><td>| |</td><td>Logický súčet</td><td>→</td></tr>
<tr><td>13</td><td>= /= -= += %= *=
>>= <<= &= ^= |=</td><td>Operátory priradenia</td><td>←</td></tr>
<tr><td>14</td><td>,</td><td>Operátor čiarka</td><td>→</td></tr>
</table>

Tabuľka č. 2 – Priorita a asociativita operátorov

6.1.5 Komentáre

Navrhnutý jazyk, podobne ako jazyk C++ rozlišuje komentáre blokové ako aj komentáre riadkové:

```
// toto je riadkový komentár
/* Toto je blokový komentár
   cez dva riadky */
```

6.2 Riadiace štruktúry jazyka

Navrhnutý jazyk rozlišuje nasledujúce riadiace štruktúry, podobné ako v jazyku C++:

- `return výraz ;` – ukončí vykonávanie programu s návratovou hodnotou.
- `if ( výraz ) { príkaz 1;... }` – podmienený príkaz; príkazy počnúc príkazom 1 sa vykonajú, iba ak je splnená podmienka (výraz > 0).
- `if ( výraz ) { prikaz1;... } else { prikaz2;... }` – ak je splnená podmienka (výraz > 0), vykonajú sa príkazy počnúc príkazom 1, inak sa vykonajú príkazy počnúc príkazom 2.
- `while ( výraz ) { príkaz1;... }` – cyklus, v ktorom sa príkazy počnúc príkazom 1 vykonávajú vždy, ak je splnená podmienka (výraz > 0). Ak je výraz na začiatku inicializovaný na nulu, príkazy sa vôbec nevykonajú.
- `do { príkaz1;... } while ( výraz ) ;` – podobný ako predchádzajúci príkaz, ale vykoná sa minimálne raz, pretože podmienka sa testuje až po vykonaní príkazov počnúc príkazom 1.
- `for ( výraz1 ; výraz2 ; výraz 3 ) { príkaz1;... }` – počítateľný cyklus, príkaz 1 sa vykoná len určitý počet-krát.
- `cout << príkaz1 [ << … << endl];` – výpis dát na štandardný výstup.
- `cerr << príkaz1 [ << … << endl];` – výpis dát na štandardný chybový výstup.
- `cin >> premenná [ >> … >> ];` – načítanie dát zo štandardného vstupu.
- `;` – prázdny príkaz.
- `výraz;` – výraz.

6.3 Sémantika jazyka

Navrhnutý jazyk kontroluje typy identifikátorov, prípadne konštánt pri analýze výrazov. V prípade nesúlade typov ohlási chybu. Pri deklarácii identifikátorov je umžnená aj ich definícia.

Deklarácia identifikátorov môže mať tvar:

- `string a;` - deklarácia premennej *a* typu string.
- `unsigned c;` - deklarácia premenej *c* typu unsigned integer.
- `const int k = 1;` - deklarácia konštanty *k* typu integer a jej inicializácia na hodnotu 1.
- `double b = .5;` - deklarácia premennej *b* typu double a jej inicializácia na hodnotu 2.5.
- `char w('a');` - deklarácia premennej *w* typu char a jej inicializácia na hodnotu 'a'.

Jazyk prevádza aj určité implicitné typové konverzie:

- *char → int → long → float → double,*
- *bool → int,*
- *char → string.*

6.4 Návrh implemenácie

Lexikálna analýza je implementovaná formou konečného automatu (*Príloha B*), ktorý na základe symbolov na vstupe prechádza od štartovacieho stavu do stavu konečného, ktorého výsledkom je návratová hodnota reprezentujúca určitý typ tokenu.

Gramatika jazyka je založená na LL-tabuľke (*Príloha E*). Syntaktický analyzátor je implementovaný metódou rekurzívneho zostupu. Každý z nonterminálnych symbolov gramatiky je analyzovaný v samostatnej funkcií. V prípade syntaktickej chyby nastáva určitá forma zotavenia sa z nej podľa kontextovej množiny, ktorú obdrží na vstupe. Detekcia a zotavenie sa z chýb vychádza z Hartmannovej metódy. Pri vstupe do každej s procedúr analyzúcej nonterminálne symboly jazyka táto procedúra obdrží na svojom vstupe kontextovú množinu, ktorá je zjednotením symbolov množín *FOLLOW*, príp. *FIRST* všetkých rozpracovaných nonterminálov.

6.4.1 Vývojové prostredie

Aplikácia je vytvorená s využitím knižnice Qt. Qt je multiplatformná knižnica od spoločnosti Trolltech, dostupná pod licenciou GPL, využívaná predovšetkým na vývoj programov s grafickým užívateľským rozhraním. Od verzie 4 má však aj podoporu na vývoj ne-grafických aplikácii.

Qt využíva štandardné funkcie jazyka C++, ale je plne objektovo-orientovaná, ľahko rozšíriteľná, ale umožňuje aj pravé programovanie komponent. Na rozdiel od ostatných prekladačov jazyka C++, Qt na obohatenie svojej funkčnosti do značnej miery využíva verziu špeciálneho preprocesoru (*Meta Object Compiler*).

Knižnica Qt je primárne určená pre jazyk C++, ale v určitých prípadoch môže byť využitá aj v iných programovacích jazykoch. Jej hlavnou výhodou je funkčnosť na všetkých hlavných platformách ako aj prenositeľnosť medzi nimi. Viac o knižnici Qt sa nachádza v [6].

7 Implementácia

Aplikácia detekcie a zotavenia sa z chýb počas syntaktickej analýzy bola implementovaná objektovo v jazyku C++ s využitím knižnice Qt. V tomto prostredí bolo vytvorené aj užívateľské rozhranie. Aplikácia je rozčlenená do viacerých tried, ktoré tvoria ucelené logické celky a medzi ktorými sú určité väzby. V kapitole sú popísané najdôležitejšie z týchto tried.

7.1 Lexikálna analýza – Trieda Scanner

Proces analýzy zdrojového kódu programu začína lexikálnou analýzou. Táto analýza je implementovaná formou konečného automatu v triede `Scanner`. Jej hlavnou metódou je metóda s prototypom `Token Get_Next_Token (TokenValue *tv)`, pomocou ktorej komunikuje s objektom triedy `Parser`, implementujúcim syntaktický analyzátor. Táto metóda je vlastne jadrom lexikálnej analýzy. Postupne sa zo vstupného súboru načítavajú symboly a na základe určitých pravidiel sa prechádza zo štartovacieho stavu až do stavu konečného, vytvárajúc pri tom jednotlivé tokeny. V konečnom stave metóda už pozná typ tokenu ako aj jeho atribúty uložené v premennej *tv*, ktorá je typom štruktúry `TokenValue` obsahujúcej všetky dostupné informácie o danom tokene. Táto štruktúra je popísaná v nasledujúcej kapitole.

Metóda `Get_Next_Token()` pri analyzovaní vstupného súboru kontroluje aj výskyt lexikálnach chýb, na ktoré upozorňuje vo forme konkrétnych typov návratovej hodnoty. Medzi lexikálne chyby patria:

- LEXERROR – symbol na vstupe nie je platným symbolom zdrojového programu,
- COMMENTERROR – neukončený blokový komentár na konci programu,
- STRINGERROR – neznámy symbol v escape sekvencií reťazca,
- CHARERROR – viacznaková konštanta,
- DBLERROR – lexikálna chyba pri analyzovaní desatinných konštánt.

Opakom tejto metódy je metóda s prototypom `void Unget_Token (TokenValue *tv)`, ktorá vráti token určený atribútom *tv* späť do vstupného súboru.

7.1.1 Atribúty tokenov – štruktúra TokenValue

Atribúty tokenov sú uložené v štruktúre tvaru:

```
typedef struct tokenvalue
{
    QString          identifier;
    IdentifierValue  value;
    TokenPosition    position;
}TokenValue;
```

kde *identifier* popisuje konkrétny symbol tokenu, *value* obsahuje hodnotu tokenu v prípade konštanty a *position* určuje konkrétnu pozíciu tokenu v zdrojovom súbore - jeho začiatočnú a konečnú pozíciu , ako aj začiatočný a konečný riadok tokenu.

7.2 Tabuľka symbolov – Trieda HashTable

Tabuľka symbolov je implementovaná vo forme hašovacej tabuľky. Štruktúra tejto tabuľky je definovaná je v štruktúre `Htable`, ktorá je popísaná v nasledujúcej podkapitole. Komunikácia s tabuľkou prebieha v rámci syntaktickej analýzy prostredníctvom nasledujúcich metód:

- `bool Lookup(TokenValue *tv, HashItem *hItem)` – metóda v tabuľke symbolov vyhľadá token, určený atribútmi *tv*. V prípade úspechu skopíruje hodnotu nájdenej položky do premennej *hItem*.
- `void Insert(TokenValue *tv, Token types, bool constant, bool defined, bool user, bool usigned)` – metóda vkladá symbol do tabuľky symbolov. Na začiatku najskôr vypočíta tzv. hash kľúča, ktorý určuje pozíciu premennej v hašovacej tabuľke. Pri výpočte hash sa ako kľúč používa názov identifikátoru vkladaného do hašovacej tabuľky. Parametre metódy ďalej určujú, či sa jedná o konštantu alebo identifikátor, či už bola daný identiifikátor definovaný, či sa jedná o užívateľskú alebo pomocnú premennú a či je premenná typu *signed* alebo *unsigned*.
- `void Update(HashItem *hItem)` – metóda na základe atribútov parametra `hItem` vyhľadá daný identifikátor v tabuľke symbolov a v prípade úspechu aktualizuje jeho atribúty.

7.2.1 Štruktúra tabuľky symbolov – Štruktúra Htable

Štruktúra tabuľky symbolov je definovaná nasledovne:

```
typedef struct htable
{
    unsigned size;
    HashList *table;
}Htable;
```

kde *size* určuje veľkosť tabuľky a *table* je ukazateľom na tabuľku symbolov. V mojom prípade som si určil veľkosť tabuľky na hodnotu 1023, ktorá sa mi pri testovaní zdala ako doštatujúca.

Každá položka tabuľky je vo forme lineárneho zoznamu položiek typu `HashItem`, tvaru:

```
typedef struct hashitem
{
    QString key;
    enum token type;
    struct tokenvalue tv;
    bool constant;
    bool defined;
    bool user;
    bool usigned;
}HashItem;
```

kde *key* určuje kľúč, na základe ktorého prebieha vyhľadávanie v hašovacej tabuľke, *type* určuje typ tokenu, *tv* určuje atribúty daného tokenu, *constant* či sa jedná o konštantu, *defined* určuje, či už bola daná premenná definovaná aj definovaná, *user*, či sa jedná o užívateľskú alebo pomocnú premennú a *usigned* určuje, či je premenná typu *signed* prípadne *unsigned*.

7.3 Syntaktická analýza – Trieda parser

Syntaktický analyzátor je implementovaný v triede `Parser` formou metódy rekurzívneho zostupu, teda každý nonterminálny symbol gramatiky je analyzovaný v samostatnej metóde, s názvom podobným názvu tohto nonterminálu. Počas syntaktickej analýzy objekt tejto triedy komunikuje pomocou metódy `Scanner::Get_Next_Token(&tv)` s lexikálnym analyzátorom, od ktorého očakáva ďalší symbol. V prípade neočakávaného symbolu na vstupe sa pomocou kontextovej množiny prevedie zotavenie sa z chyby pomocou metódy:

```
void Parser::Check(QSet<Token> context)
{
      while(!(this->sets.Symbol_In_Set(context, this->tok))){
            Add_Lexical_Error(&tv);
            this->tok = scanner.Get_Next_Token(&tv);
      }
}
```

Táto metóda kontroluje, či sa token, načítaný lexikálnym analyzátorom nachádza v množine context. V prípade, že sa v nej daný token nenachádza, načíta sa prostredníctvom lexikálneho analyzátoru ďalší token, a hodnota pôvoodného tokenu sa zahodí, predtým sa ale skontroluje, či daný token náhodou nesymbolizoval lexikálnu chybu. Táto istá metóda sa volá aj na začiatku pri vstupe do každej z procedúr analyzujúcich nonterminálne symboly gramatiky. Parametrom tejto metódy je kontextová množina, ktorá vzniká zjednotením množín FOLLOW všetkých rozpracovaných nonterminálov.

Metóda, pomocou ktorej syntaktický analyzátor komunikuje s užívateľským rozrhaním je metóda `QString ParseCode(QString file, int language, int action)`, ktorej návratová hodnota je zoznam všetkých chýb, ktoré sa vyskytli behom analýzy zdrojového kódu. Jej parametrami sú *file*, obsahujúci zdrojový text programu, *language*, určujúci jazyk aplikácie a *action*, ktorá určuje aký typ analýzy sa má vykonať. Môžu prebiehať nasledujúce typy analýz zdrojového kódu:

- len lexikálna analýza,
- lexikálna anlýza spolu so syntaktickou,
- kompletná analýza zdrojového kódu (lexikálna, syntaktická aj sémantická analýza).

7.3.1 Sémantická analýza

Pri analýze výrazov počas syntaktickej analýzy môže prebiehať aj kontrola sémantických chýb. Na základe sémantických pravidiel daného jazyka sa kontroluje:

- či bola premenná deklarovaná,
- či bola premenná, používaná pri operáciach aj definovaná
- či môže daná operácia prebehnúť pre premenné konkrétnych typov
- či môže pri operácii nastať implicitná typová konverzia.

Na sémantickú analýzu sa používa zásobník `QStack<HashItem> mystack`, na ktorý sa ukladajú identifikátory, podobne ako v tabuľke symbolov. V prípade, ak počas vyhodnocovania

výrazu nastane nejaká syntaktická chyba, sémantická analýza výrazu by v danom mieste stratila účinnosť, preto sa pre daný výraz skončí.

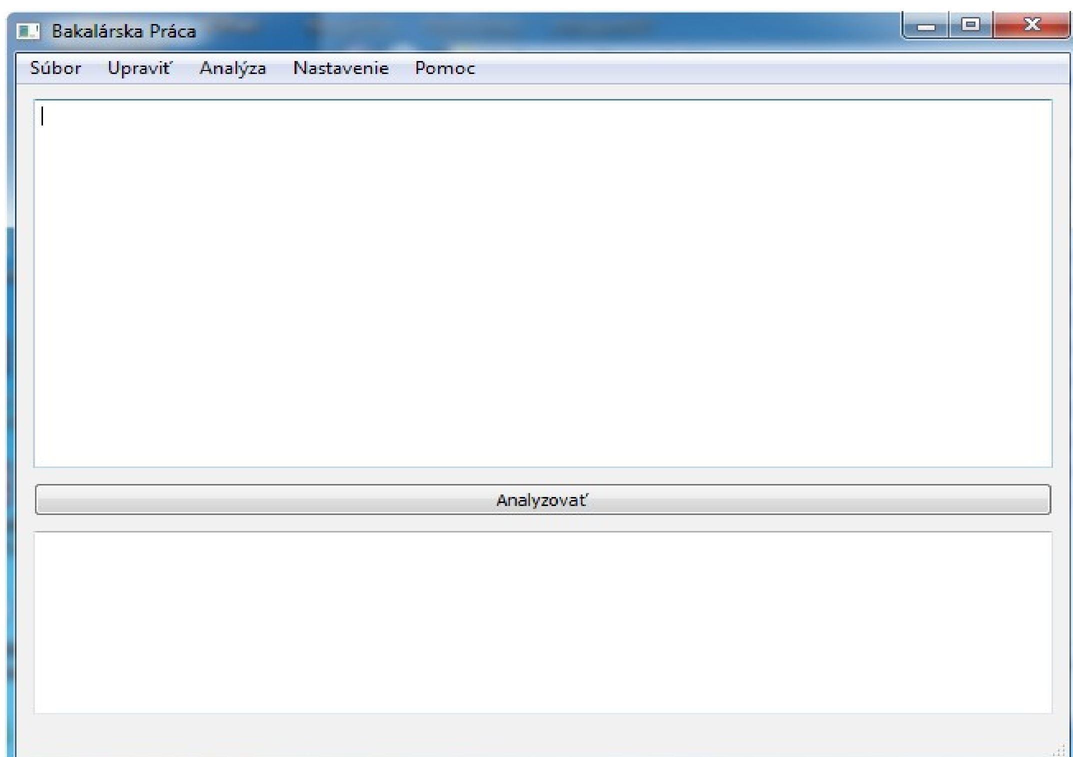
7.4 Množiny FIRST a FOLLOW – Trieda Sets

Trieda Sets obsahuje metódy na prácu s množinami FOLLOW, FIRST každého nonterminálneho symbolu gramatiky. Operácie používajúce tieto množiny sa využívajú pri zotavení sa z chýb pri syntaktickej analýze. Množiny sú typu Qset<Token>, teda obsahujú všetky symboly, požadované na vstupe pri objavení syntaktickej chyby. V konštruktore tejto triedy prebieha ich naplnenie správnymi hodnotami, vypočítanými pomocou pravidiel gramatiky.

Trieda Sets obsahuje aj metódy na zjednotenie 2 množín, metódu na pridanie symbolu do množiny ako aj metódu na zistenie, či sa daný symbol v množine nachádza.

7.5 Užívateľské rozhranie

Užívateľské rozhranie aplikácie je vytvorené s využitím knižnice Qt verzie 4.6.2. Hlavná trieda (okno aplikácie) MainWindow obsahuje 2 textové okná – jedno na zdrojový program, a druhé na výsledok analýzy, a tlačítko, ktorým sa spúšťa analýza zdrojového textu (*Obr. č. 6*). Hlavné okno okrem toho obsahuje aj menu aplikácie, popísané v nasledujúcej podkapitole.



Obr. č. 6 – Hlavné okno aplikácie

7.5.1 Menu aplikácie

Menu aplikácie obsahuje niekoľko položiek, využívaných pri obsluhe, príp. nastavovaní aplikácie.

Položka menu Súbor poskytuje akcie na prácu so súborom so zdrojovým textom - vytvorenie nového súboru, otvorenie súboru so zdrojovým textom, jeho uloženie, ako aj opustenie aplikácie.

Ďalšou položkou menu je položka implementujúca základné operácie so zdrojovým textom aplikácie, ako sú kopírovanie, vystrihnutie alebo vloženie určitej časti textu na miesto určené kurzorom, ale aj operácie späť alebo vpred, ktoré umožňujú pohybovať sa v histórii operácií editácie zdrojového textu.

Pred spustením analýzy je možné nastaviť, aké typy chýb požadujeme detekovať v zdrojovom texte programu. Je možné analyzovať zdrojový text len metódou lexikálnej analýzy, alebo aj metódou syntaktickej, prípadne zisťovať aj sémantické chyby. Samotnú analýzu spúšťame kliknutím myšou na tlačítko na hlavnej ploche alebo stlačením kombinácie tlačidiel `Ctrl + R`.

Aplikácia umožňuje tiež meniť svoj vzhľad ako aj jazyk. Implementovaná je podpora slovenského aj anglického jazyka. Po nastavení jazyka automaticky začína interakcia programu s užívateľom v nastavenom jazyku. Tieto nastavenia aplikácie sa ukladajú do konfiguračného súboru vo forme XML, teda program po opustení a opätovnom spustení aplikácie nastaví svoj vzhľad a jazyk na hodnoty pred opustením aplikácie.

Poslednou položkou menu je Zobrazenie nápovedy, prípadne informácii o aplikácii, ako aj programovej dokumentácie, popisujúcej všetky triedy aplikácie, vygenerovanej pomocou nástroja Doxygen. Táto dokumentácia je však len v jazyku anglickom.

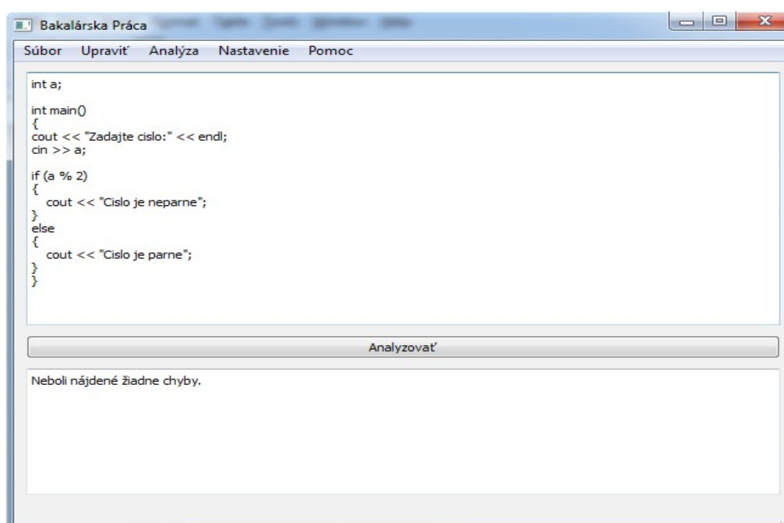
8 Zhodnotenie dosiahnutých výsledkov

Po implementácii aplikácie na detekciu a zotavenie sa z chýb bolo potrebné otestovať, ako daná metóda pracuje – či sa jej podarí detekovať všetky chyby zdrojového textu programu, prípadne či sa jej podarí označiť detekované chyby správne. Nie vždy tomu ale tak je. Dôvodom je výpočet kontextovej množiny pomocou, ktorej nastáva zotavenie sa po chybách počas syntaktickej analýzy.

V tejto kapitole budú ukázané situácie, kedy sa detekcia chýb podarí správne a kedy nie, a bude vysvetlený aj dôvod prečo je tomu tak. Testy budú pre názornú ukážku doplnené snímkami obrazovky aplikácie. Všetky testy vychádzajú z testovacích súborov umiestnených v priečinku `/examples/` na CD nosiči priloženom k správe.

Test č. 1 – Analýza bez chýb

Programy napísané syntakticky správne by nemali ohlasovať žiadne chyby. Napr. taký program na zisťovanie, či je číslo párne resp. nepárne, ak je napísaný syntakticky bey chzby, nemal by zahlásiiť žiadnu chybu.

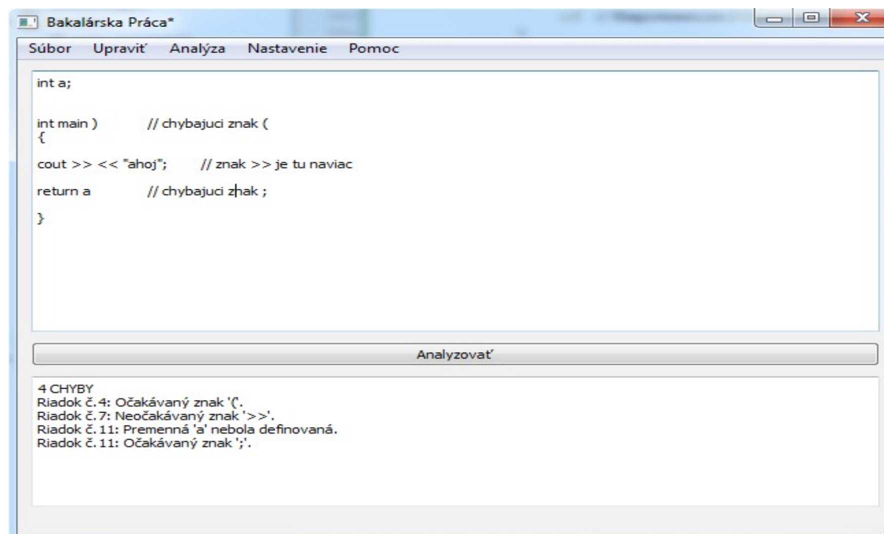


Obr. č. 7 – Aplikácia bez chýb

Test č. 2 – Správne odhalené chyby

Chyba počas procesu syntaktickej analýzy môže vzniknúť z viacerých dôvodov, ale najmä z nepozornosti, keď sa nám na vstupe podarí pridať náhodou o 1 znak viac, prípadne znak vynechať. Navrhnutá metóda zotavenia sa z chýb počíta aj s takouto variantou. V nasledujúcom príklade je ukázané zotavenie sa z oboch druhov týchto chýb. Ako je vidieť na obrázku, aplikácii sa úspešne podarilo odhaliť všetky chyby:

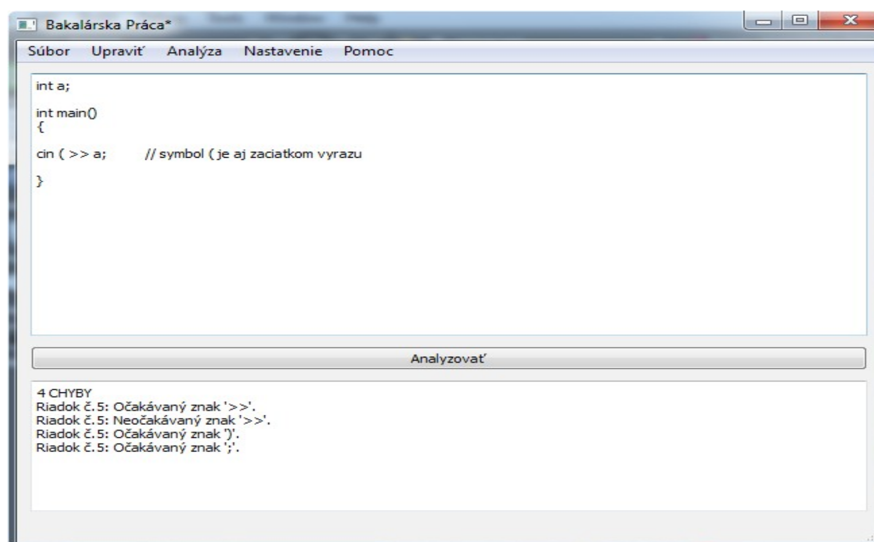
- Chýbajúca zátvorka za terminálnym symbolom *main* – správne je *main()*,
- Znak *'>>'* za symbolom *cout* – znak je tu evidentne naviac,
- Chýbajúci znak *';'* za symbolom *return* – znak *';'* ukončuje všetky príkazy, a napokon
- Premenná *'a'* nebola definovaná – premenná síce bola deklarovaná, ale chýba jej inicializácia, teda nie je isté s akou návratovou hodnotou sa program ukončí.



Obr. č. 8 – Správne odhalenie všetkých zanesených chýb

Test č. 3 – Nesprávne odhalené chyby

Najväčším problémom danej metódy detekcie a zotavenia sa z chýb je práve výpočet kontextovej množiny. Pretože niektoré tokeny sa nachádzajú vo viacerých množinách FIRST a FOLLOW, a teda nie je možné určiť, v ktorom z daných neterminálnych symbolov aplikácie bude pokračovať vykonávanie programu. Jedným z takto problémových programov je program znázornený na nasledujúcom obrázku.



Obr. č. 9 – Nesprávne odhalené chyby

Daná situácia nastala, pretože symbol '*(*' sa nachádza v kontextovej množine analyzovaného výrazu a je považovaný za začiatok výrazu, a tak aplikácia prejde na analýzu daného výrazu, čím vlastne spôsobí, že sa v tomto reťazci odhalia aj zanesené syntaktické chyby z analýzy pôvodného reťazca.

V tejto časti boli spomenuté len niektoré z príkladov, na ktorých je demonštrovaná detekcia metóda detekcie a zotavenia sa z chýb. Ďaľšie príklady sa nachádzajú na priloženom CD nosiči.

9 Záver

Cieľom aplikácie bolo navrhnúť a implemenovať metódu detekcie a zotavenia sa z chýb počas syntaktickej analýzy. Na príkladoch bola predvedená funkčnosť tejto metódy.

K dosiahnutiu daného cieľa bolo potrebné preštudovať princíp fungovania Hartmannovej metódy zotavenie sa z chýb pri syntaktickej analýze a navrhnúť obdobnú metódu. Navrhnutá metóda vychádza z Hartmannovej metódy. Najdôležitejšou časťou návrhu bol správny návrh gramatiky programovacieho jazyka tak, aby spĺňala všetky potrebné kritéria. Navrhnutá gramatika je typu LL a vo svojich pravidlách zahŕňa aj prioritu operátorov pri analýze výrazov. Ďalej bolo potrebné správne určiť množiny FOLLOW a FIRST pre všetky nonterminálne symboly gramatiky, ktoré sa používajú pri výpočte množiny CONTEXT.

V aplikácii bolo navrhnuté aj jednoduché užívateľské rozhranie, poskytujúce užívateľovi jednoduchšiu prácu s aplikáciou. Funkcie tohto rozhrania boli navrhnté tak, aby umožňovali všetky základné operácie so zdrojovým textom programu. Pri návrhu som sa inšpiroval podobnými užívateľskými rozhraniami ostatných vývojových prostredí, prípadne textových editorov.

Možný ďalší vývoj projektu vidím hlavne vo vylepšení pravidiel gramatiky jazyka a jej doplnení, napr. o možnosť deklarácie funkcií, deklarácii lokálnych premenných ako aj niektorých ďalších vlastností jazyka C++. Testy z predchádzajúcej kapitoly ukazujú, že zotavenie sa z chýb neprebehne vždy úplne najideálnejšie. Hlavný problém tohto nedostatku sa nachádza vo výpočte kontextových množín. Aj z tohto dôvodu by sa dalo nadviazať na projekt implementáciou zotavenia sa z chýb inou metódou a porovnaním dosiahnutých výsledkov.

Literatúra

[1] AHO, A. V., et al. *Compilers : Principles, Techniques, and Tools*. 2nd edition. Boston : Pearson Education, 2007. xxiv, 1009 s. ISBN 0-321-48681-1.

[2] MEDUNA, Alexander; LUKÁŠ, Roman. *Výstavba Překladačů VYP : Studijní opora* [online]. FIT VUT v Brně , 2006 [cit. 2010-02-08]. Dostupné z WWW: <https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/VYP-IT/texts/OporaVYP.pdf>.

[3] MEDUNA, Alexander. *Automata and Languages : Theory and Applications*. London : Springer, 2006. xv, 916 s. ISBN 978-1-85233-074-3.

[4] ČEŠKA, Milan; HRUŠKA, Tomáš; BENEŠ, Miroslav. *Překladače*. Skriptum VUT Brno : VUT Brno, 1993. 264 s. ISBN 80-214-0491-4.

[5] Přednášky kurzu Formální jazyky a překlladače., FIT VUT v Brně, 2007 [cit. 2010-05-10]. Dostupné z WWW: <https://www.fit.vutbr.cz/study/courses/IFJ/public/materials/index.php>.

[6] *Qt* [online]. 2010 [cit. 2010-05-18]. Online Reference Documentation. Dostupné z WWW: <http://doc.trolltech.com/>.

Zoznam príloh

Príloha A. CD.
Príloha B. Konečný automat lexikálnej analýzy
Príloha B. Pravidlá gramatiky.
Príloha D. Množiny FIRST a FOLLOW.
Príloha E. LL tabuľka.

Príloha A. CD

Priložené CD obsahuje:

- Zdrojové kódy aplikácie
- Binárne súbory aplikácie pre OS Windows aj OS Linux
- Testovacie príklady, demonštrujúce metódu zotavenia
- Programovú dokumentáciu vygenerovanú nástrojom Doxygen
- Help.

Príloha B. Konečný automat

Obrázok č.I – Konečný automat lexikálnej analýzy

Príloha C. Pravidlá gramatiky

1) <prog> → <decl> <statelist> eof
2) <decl> → <ctype> <idmain>
3) <ctype> → ***const*** <utype>
4) <ctype> → <utype>
5) <utype> → ***unsigned*** <itype>
6) <utype> → ***signed*** <itype>
7) <utype> → <type>
8) <itype> → ***int***
9) <itype> → ***char***
10) <itype> → ***short*** <htype>
11) <itype> → ***long*** <htype>
12) <htype> → ***int***
13) <htype> → ***ε***
14) <type> → ***int***
15) <type> → ***double***
16) <type> → ***string***
17) <type> → ***char***
18) <type> → ***bool***
19) <type> → ***float***
20) <type> → ***long*** <htype>
21) <type> → ***short*** <htype>
22) <idmain> → ***id*** <iddecl> <idcomma> <decl>
23) <idmain> → ***main ( )***
24) <iddecl> → **=** <decl_expr>
25) <iddecl> → ***(*** <decl_expr> ***)***
26) <iddecl → ***ε***
27) <idcomma> → ***, id*** <iddecl> <idcomma>
28) <idcomma> → ***;***
29) <statelist> → ***{*** <state> ***}***
30) <state> → <expr> ***;*** <state>
31) <state> → ***;*** <state>
32) <state> → ***ε***
33) <state> → <statelist> <state>
34) <state> → ***return*** <expr>***;*** <state>
35) <state> → ***while (*** <expr> ***)*** <statelist> <state>
36) <state> → ***do*** <statelist> ***while (*** <expr> ***) ;*** <state>
37) <state> → ***if (*** <expr>) <statelist> <elseif> <state>
38) <elseif> → ***else*** <statelist>
39) <elseif> → ***ε***
40) <state> → ***for (*** <forexpr> ***;*** <forexpr> ***;*** <forexpr> ***)*** <statelist> <state>
41) <forexpr> → <expr>

42) <forexpr> → ***ε***
43) <state> → ***cin*** **>>** ***id*** <input> <state>
44) <input> → **>>** ***id*** <input>
45) <input> → ***;***
46) <state> → ***cout*** **<<** <expr> <output> <state>
47) <state> → ***cerr*** **<<** <expr> <output>
48) <output> → **<<** <expr> <output>
49) <output> → ***;***
50) <expr> → <decl_expr> <expr_der>
51) <expr_der> → **,** <decl_expr> <expr_der>
52) <expr_der> → ***ε***
53) <decl_expr> → <or_expr> <decl_expr_der>
54) <decl_expr_der> → **=** <or_expr> <decl_expr_der>
55) <decl_expr_der> → **+=** <or_expr> <decl_expr_der>
56) <decl_expr_der> → **-=** <or_expr> <decl_expr_der>
57) <decl_expr_der> → *= <or_expr> <decl_expr_der>
58) <decl_expr_der> → /= <or_expr> <decl_expr_der>
59) <decl_expr_der> → **%=** <or_expr> <decl_expr_der>
60) <decl_expr_der> → |= <or_expr> <decl_expr_der>
61) <decl_expr_der> → **^=** <or_expr> <decl_expr_der>
62) <decl_expr_der> → **&=** <or_expr> <decl_expr_der>
63) <decl_expr_der> → **>>=** <or_expr> <decl_expr_der>
64) <decl_expr_der> → **<<=** <or_expr> <decl_expr_der>
65) <decl_expr_der> → ***ε***
66) <or_expr> → <and_expr> <or_expr_der>
67) <or_expr_der> → || <and_expr> <or_expr_der>
68) <or_expr_der> → ***ε***
69) <and_expr> → <bit_or_expr> <and_expr_der>
70) <and_expr_der> → ***&&*** <bit_or_expr> <and_expr_der>
71) <and_expr_der> → ***ε***
72) <bit_or_expr> → <bit_xor_expr> <bit_or_expr_der>
73) <bit_or_expr_der> → | <bit_xor_expr> <bit_or_expr_der>
74) <bit_or_expr_der> → ***ε***
75) <bit_xor_expr> → <bit_and_expr> <bit_xor_expr_der>
76) <bit_xor_expr_der> → **^** <bit_and_expr> <bit_xor_expr_der>
77) <bit_xor_expr_der> → ***ε***
78) <bit_and_expr> → <equal_expr> <bit_and_expr_der>
79) <bit_and_expr_der> → | <equal_expr> <bit_and_expr_der>
80) <bit_and_expr_der> → ***ε***
81) <equal_expr> → <lower_expr> <equal_expr_der>
82) <equal_expr_der> → ***!=*** <lower_expr> <equal_expr_der>
83) <equal_expr_der> → **==** <lower_expr> <equal_expr_der>
84) <equal_expr_der> → ***ε***
85) <lower_expr> → <bit_shift_expr> <lower_expr_der>

86) <lower_expr_der> → **<** <bit_shift_expr> <lower_expr_der>
87) <lower_expr_der> → **<=** <bit_shift_plus_expr> <lower_expr_der>
88) <lower_expr_der> → **>** <bit_shift_expr> <lower_expr_der>
89) <lower_expr_der> → **>=** <bit_shift_expr> <lower_expr_der>
90) <lower_expr_der> → ***ε***
91) <bit_shift_expr> → <plus_expr> <bit_shift_expr_der>
92) <bit_shift_expr_der> → **>>** <plus_expr> <bit_shift_expr_der>
93) <bit_shift_expr_der> → ***ε***
94) <plus_expr> → <multiply_expr> <plus_expr_der>
95) <plus_expr_der> → **+** <multiply_expr> <plus_expr_der>
96) <plus_expr_der> → **-** <multiply_expr> <plus_expr_der>
97) <plus_expr_der> → ***ε***
98) <multiply_expr> → <preinc_expr> <multiply_expr_der>
99) <multiply_expr_der> → ***** <preinc_expr> <multiply_expr_der>
100) <multiply_expr_der> → ***/*** <preinc_expr> <multiply_expr_der>
101) <multiply_expr_der> → ***%*** <preinc_expr> <multiply_expr_der>
102) <multiply_expr_der> → ***ε***
103) <preinc_expr> → ***!*** <preinc_expr>
104) <preinc_expr> → **~** <preinc_expr>
105) <preinc_expr> → ++ <preinc_expr>
106) <preinc_expr> → **--** <preinc_expr>
107) <preinc_expr> → <postinc_expr>
108) <postinc_expr> → <final_expr> <postinc_expr_der>
109) <postinc_expr_der> → **++**
110) <postinc_expr_der> → ***--***
111) <postinc_expr_der> → ***ε***
112) <final_expr> → ***id***
113) <final_expr> → ***(*** <expr> ***)***

Príloha D. Množiny FIRST a FOLLOW

Na základe gramatických pravidiel jazyka sa vypočítali množiny FIRST a FOLLOW pre každý z neterminálnych symbolov jazyka.

Množiny FIRST

- FIRST(<prog>) = { char, bool, string, int, double, long, float, short, unsigned, signed, const, }
- FIRST(<decl>) = { char, bool, string, int, double, long, float, short, unsigned, signed, const, }
- FIRST(<ctype>) = {char, bool, string, int, double, long, float, short, unsigned, signed, const,}
- FIRST(<utype>) = { char, bool, string, int, double, long, float, short, unsigned, signed, }
- FIRST(<itype>) = { char, int, long, short, }
- FIRST(<htype>) = { int, }
- FIRST(<type>) = { char, bool, string, int, double, long, float, short, }
- FIRST(<idmain>) = { id, main,}
- FIRST(<iddecl>) = { =, (, }
- FIRST(<idcomma>) = { ,, ;, }
- FIRST(<statelist>) = { {, }
- FIRST(<state>) = { !, ~, ++, --, id, (, ;, {, return, while, do, if, for, cin, cout, cerr, }
- FIRST(<elsif>) = { else, }
- FIRST(<forexp>) = { !, ~, ++, --, id, (, }
- FIRST(<output>) = { ;, <<, }
- FIRST(<input>) = { ;, >>, }
- FIRST(<expr>) = { !, ~, ++, --, id, (, }
- FIRST(<expr_der>) = { , }
- FIRST(<decl_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<decl_expr_der>) = { =, +=, -=, *=, /=, %=, |=, &=, >>=, <<=, ^=, }
- FIRST(<or_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<or_expr_der>) = { || }
- FIRST(<and_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<and_expr_der>) = { && }
- FIRST(<bit_or_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<bit_or_expr_der>) = { |, }
- FIRST(<bit_xor_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<bit_xor_expr_der>) = { ^, }
- FIRST(<bit_and_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<bit_and_expr_der>) = { &, }
- FIRST(<equal_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<equal_expr_der>) = { !=, ==, }
- FIRST(<lower_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<lower_expr_der>) = { <, <=, >, >=, }
- FIRST(<lbit_shift_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<bit_shift_expr_der>) = { >>, }

- FIRST(<plus_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<plus_expr_der>) = { +, -, }
- FIRST(<multiply_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<multiply_expr_der>) = { *, /, %, }
- FIRST(<preinc_expr>) = { !, ~, ++, --, id, (, }
- FIRST(<postinc_expr>) = { id, (, }
- FIRST(<postinc_expr_der>) = { ++, --,}
- FIRST(<final_expr>) = { id, (, }

Množiny FOLLOW

- FOLLOW(<prog>) = { $, }
- FOLLOW(<decl>) = { {, }
- FOLLOW(<ctype>) = { id, main, }
- FOLLOW(<utype>) = { id, main, }
- FOLLOW(<itype>) = { id, main, }
- FOLLOW(<htype>) = { id, main, }
- FOLLOW(<type>) = { id, main, }
- FOLLOW(<idmain>) = { {, }
- FOLLOW(<iddecl>) = { ,, ;, }
- FOLLOW(<comma>) = { char, bool, int, double, string, float, long, short, unsigned, signed, const, }
- FOLLOW(<statelist>) = { eof, !, ~, ++, --, id, (, ;, {, return, while, do, if, for, cin, cout, cerr, else, }, }
- FOLLOW(<state>) = { }, }
- FOLLOW(<elsif>) = { ! , ~, ++, --, id, (, ;, {, return, while, do, if, for, cin, cout, cerr, }, }
- FOLLOW(<forexp>) = {;,), }
- FOLLOW(<output>) = { ! , ~, ++, --, id, (, ;, {, return, while, do, if, for, cin, cout, cerr, }, }
- FOLLOW(<input>) = { ! , ~, ++, --, id, (, ;, {, return, while, do, if, for, cin, cout, cerr, }, }
- FOLLOW(<expr>) = { ;,), <<, }
- FOLLOW(<expr_der>) = { ;,), <<, }
- FOLLOW(<decl_expr>) = { ;, ,,), <<, }
- FOLLOW(<decl_expr_der>) = { ;, ,,), <<, }
- FOLLOW(<or_expr>) = { ,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, }
- FOLLOW(<or_expr_der>) = { ,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, }
- FOLLOW(<and_expr>) = { ,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, }
- FOLLOW(<and_expr_der>) = { ,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, }
- FOLLOW(<bit_or_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&,}
- FOLLOW(<bit_or_expr_der>) = { ,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, }
- FOLLOW(<bit_xor_expr>) = { ,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, }
- FOLLOW(<bit_xor_expr_der>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |,}

- FOLLOW(<bit_and_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^,}
- FOLLOW(<bit_and_expr_der>) = { ,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, }
- FOLLOW(<equal_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &}
- FOLLOW(<equal_expr_der>) = { ,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, }
- FOLLOW(<lower_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, }
- FOLLOW(<lower_expr_der>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, }
- FOLLOW(<lbit_shift_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >= }
- FOLLOW(<bit_shift_expr_der>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >= }
- FOLLOW(<plus_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >=, >>, }
- FOLLOW(<plus_expr_der>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >=, >>, }
- FOLLOW(<multiply_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >=, >>, +, -, }
- FOLLOW(<multiply_expr_der>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >=, >>, +, -, }
- FOLLOW(<preinc_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >=, >>, +, -, *, /, %, }
- FOLLOW(<postinc_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >=, >>, +, -, *, /, %, }
- FOLLOW(<postinc_expr_der>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >=, >>, +, -, *, /, %, }
- FOLLOW(<final_expr>) = {,, ;,), <<, =, +=, *=, -=, /=, %=, |=, &=, ^=, <<=, >>=, ||, &&, |, ^, &, ==, !=, <, <=, >, >=, >>, +, -, *, /, %, ++, --, }