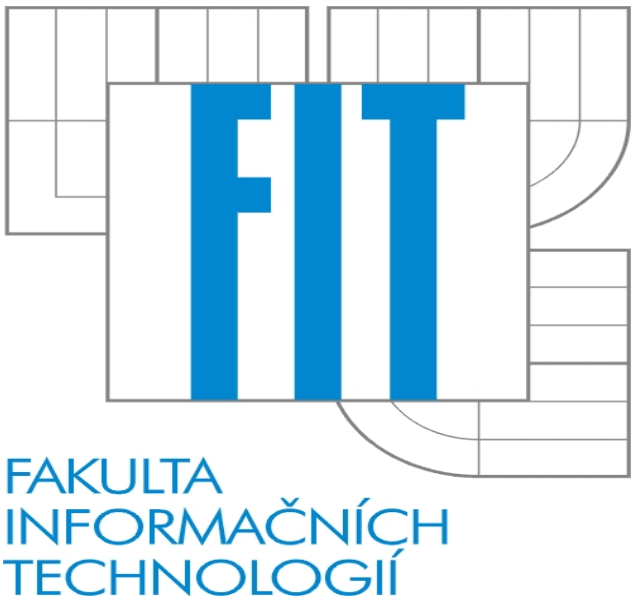



Dokumentácia ku projektu
**Interpret jazyka IFJ2009** pre predmety IFJ / IAL
Tým 6, varianta b/4/II


| | | |
|---|---|---|
| Barjak Maroš | xbarja00 | 25% |
| Haviar Martin | xhavia01 | 25% |
| Henzl Jozef | xhenzl04 | 25% |
| Herich Martin | xheric00 | 25% |


**Výpis implementovaných rozšírení :**

- program spracuje aj riadkové komentáre typu // komentár ...
- konštanty v dvojkovej, osmičkovej a šestnásťkovej sústave
- zjednodušený podmienkový výraz if - then bez časti else
- cyklus repeat – until
- pri nájdení chyby program vypíše aj riadok na ktorom sa chyba nachádza

V Brne dňa 13.12 2009 FIT VUT

# Obsah

# 1. Úvod

Tento dokument opisuje implementáciu imperatívneho jazyka IFJ2009. Tento projekt bol zadaný v predmete Formální jazyky a prekladače na Fakulte Informačních Technologií v Brne. Postupne prejdeme všetkými fázami riešenia daného problému. Od návrhu ,cez popis rozhrania, ktoré sme použili, rozdelenia práce v tíme až po samotnú implementáciu jedntlivých častí a zadaných algoritmov.

# 2. Analýza a návrch riešenia problému

## 2.1 Práca v týme, rozhranie

Súčastou každého skupinového projektu je potrebné v prvom rade vytvoriť tím, ktorý bude fungovať. Vyvarovať sa chybám a nevziať medzi seba neschopných členov. V našom prípade sme zostavili ten náš podľa pravidla "Kvalita je lepšia než kvantita" a práve preto sme v tíme štyria členovia a nie piati. Pri práci viacerých ľudí na jednom projekte je potrebné zvoliť si určité zásady a systém. Ako rozhranie pre správu zdrojových kódov sme zvolili DVCS Mercurial na serveri bitbucket.org poskytovaný pre nekomerčné účely zadarmo. Spoločne s verzovacím systémom je na stránkach k dispozícii tiež rozhranie pre sledovanie chýb (bugtracker). Každý člen má ma na starosti svoj modul na ktorom pracuje. Po dokončení a odladení chýb nahrá zmeny v zdrojových kódoch do centrálneho repozitára. Všetky tieto zmeny sú evidované a zreteľne viditeľné na webovom rozhraní. Tento systém sa nám v konečnom dôsledku oplatil a nemali sme žiadne vačšie problémy s jeho používaním. Taktiež sme cez server komunikovali a neboli nutné časté tímové stretnutia. Jednotlivé moduly sme testovali samostatne, je to oveľa jednoduchšie ako testovať celý program nakoniec. Vtom prípade by sme chyby hľadali len veľmi tažko. Ku každej časti sme si dopísali vlastnú main funkciu, v ktorej sme skúšali či program pracuje ako má pracovať.

# 3. Vlastná implementácia jednotlivých častí

## 3.1 Lexikálny analyzátor

Hlavnou funkciou lexikálnej analýzy je rozdelenie zdrojového kódu na jednotlivé lexémy, ktoré reprezentuje pomocou tokenov. Mimo toho tiež L.A. preskakuje všetky biele znaky a komentáre. L.A. je spracovaná v module *scanner.c* podľa postupov ktoré sme preberali na prednáškach. Je založená a implementovaná na princípe konečného deterministického automatu uvedeneho na konci dokumentácie v prílohách. V hlavičkovom module scanner.h sú nadefinované všetky tokeny jazyka IFJ2009. Každý má jedinečnú hodnotu typu integer podľa ktorej daný token rozpoznáme. Syntaktický analyzátor postupne volá hlavnú funkciu *getNextToken()* ktorá vracia typ lexému ,teda hodnotu daného nadefinovaného tokenu. V parametri funkcie vracia atribút tokenu. Ten môže byť aj prázdny. Napríklad ak sa jedná o číslo typu integer funkcia vráti hodnotu ktorá reprezentuje v hlavičkovom module *scanner.h* konštanta typu integer a v parametri funkcia vráti vlastnú skutočnú hodnotu tochto čísla. Parameter funkcie je typu *string_t* . Jedná sa o typ ktorý nám pomáha spracovávať potencionálne nekonečne dlhé reťazce, je nadefinovaný v module *str.c*.

V lexikálnej analýze tiež načítavame vstupný súbor z ktorého čítame pomocou cyklu while po jednotlivých znakoch zdrojový kód až pokým nedojdeme na koniec súboru. Máme vytvorených toľko stavov koľko existuje tokenov a ešte aj pomocné stavy naviac. Obecne máme stavy ked prečítame jednoduchý lexém ako napr. plus mínus krát a podobne. Vtedy je to veľmi jednoduché a môžeme vrátiť daný typ tokenu. Tak tiež niečo ako priradenie,alebo porovnanie nám žiadny problém nespôsobí. Na komentáre máme pomocný stav do ktorého sa dostanem, dočítam komentár do konca a vrátim sa na začiatok ako by ani nič neprišlo, tento typ lexému je pre nás nezaujímavý. Pri prečítaní čísla už musíme dávať vačší pozor pretože potrebujeme rozlíšiť čísla typu integer či double a taktiež može dôsť ku chybe ,napr. písmeno v strede čísla. V takýchto prípadoch vracia lexikálny analyzátor *WRONG_LEXEM,* chybu v lexikálnej analýze. Pri identifikátoroch musíme myslieť nato, že sa môže jednať o kľúčové slovo. Preto tento reťazec porovnávame s nadefinovanými kľúčovými slovami a vrátime keyword alebo ak zhoda nenastala vrátime identifikátor. Najvačší problém lexikálnej analýzy predstavovali reťazce. Pri implementácii sme pri skoro každej zmene narážali na nové chyby. Výsledná podoba je jednoduchá a zahŕňa len dva stavy. Využili sme funkciu fgetc() pomocou ktorej sme načítavali vždy dopredu ďalší znak. Tento spôsob sme zvolili pretože sme potrebovali efektívne spracovať a previesť escape sekvencie a dvojité apostrofy ktoré sa mohli vyskytnúť v reťazci. Daná sekvencia sa previedla pokiaľ načítavané znaky boli správne a číslo danej escape sekvencie bolo v rozsahu 1 až 31 podľa zadania. Kedže v sekvencii sa mohli vyskytnúť aj nuly, pričom sme ich nebrali v úvahu , napríklad '#009 ' v reťazci je korektná escape sekvencia a bere sa ako by tam nuly ani neboli. V kóde sme ich preskočili jednoduchým cyklom do while. V opačnom prípade ak niekde nastala chyba vracal sa chybný lexém. V jednotlivých stavoch bolo plno maličkostí ktoré bolo
treba ošetriť ako napríklad situácia ak by v reťazci nastal koniec suboru a podobne. Na záver sme implementovali aj niektoré rozšírenia . Náš program spracuje aj riadkové komentáre typu //komentar … Spracuje tiež okrem bežných čísiel jazyka ifj2009 aj čísla binárne, hexadecimálne a čísla v osmičkovej sústave ktoré automaticky prevedie do desiatkovej sústavy.

# 3.2 Syntaktický analyzátor

Syntaktická analýza (parser) predstavuje hlavnú časť interpretu ako celku. Vykonáva syntaxou riadený preklad, ktorý zabezpečuje okrem kontroly syntaxe aj vytvátranie inštrukcií pre interpret už v priebehu analýzy. Logicky sa analýza skladá z dvoch hlavných častí :

a) syntaktická analýza zhora nadol
b) syntaktická analýza zdola nahor

**Syntaktická analýza zhora nadol**

Základom tejto analýzy je LL-gramatika (v prílohe) a z nej zostrojená LL-tabuľka. Gramatika bola v projekte implementovaná metódou rekurzívneho zostupu, kde jednotlivé nonterminály gramatiky predstavujú jednotlivé funkcie parseru. Tieto funkcie používajú na získanie tokenov volanie funkcie lexikálnej analýzy a aplikáciou gramatiky vykonávajú deriváciu až kým nedôjdu k výsledným terminálom.

Podmienený príkaz if-then-else ako aj cyklus while bol implementovaný pomocou inštrukcií podmieneného a nepodmieneného skoku a inštrukcií návestí. Na podmienený skok bola použiá inštrukcia JZ (jump-if-zero), teda nebolo potrebné implementovať inštrukciu NOT.
Súčasťou syntaktickej analýzy je aj sémantická kontrola. Tá sa vykonáva napr. pri deklarácii premenných, priraďovaní, pri funkciách, po vyhodnotení podmienok atď.

**Riešenie epsilon pravidla v if-then-else vetve**

---
Keďže očakávame, že v programe sa môže nachádzať podmienka bez else vetvy (ide o rozšírenie), bolo potrebné zaviesť do gramatiky epsilon pravidlo. V takom prípade je potrebné prečítať jeden token za príkazmi po if-then vetve aby sme zistili, či sa tam vetva else nachádza alebo nie. V prípade že sa tam nachádza tak je riešenie jednoduché, avšak ak tam vetva else nebola, načítali sme jeden token navyše, čo je nežiadúce. Tento stav je v našej implementácii riešený funkciou, ktorá zabezpečí, že sa pri volaní o ďaľší token zopakuje token aktuálny. Táto metóda teda jednoduchým spôsobom zabezpečí, že vetva else nie je v programe povinná.
---

**Syntaktická analýza zdola nahor**

Syntaktická analýza zdola nahor sa používa pri vyhodnotení výrazov. Je implementovaná pomocou precedenčnej tabuľky,
v ktorej je definovaná precedencia tokenov, ktoré sa môžu vyskytnúť vo výraze (binárne operárory, identifikátory, zátvorky a tokeny indikujúce koniec výrazu). S využitím zásobníku simuluje vytváranie derivačného stromu. Tabuľka vracia symbol určujúci vzťah medzi posledným uloženým tokenom a novým tokenom načítaným zo súboru. Symboly sú 4 a majú
nasledovný význam:

**< - nový token uložíme na zásobník a pokračujeme v načítavaní ďalšieho tokenu**
**= - nový token uložíme na zásobník a pokračujeme v načítavaní ďalšieho tokenu**
**> - začneme redukovať výraz na zásobníku podľa stanovených pravidiel**
**0 - nový token a token na zásobníku porušujú syntax, vracia sa syntaktická chyba**

Pravidlá redukcie výrazov:

| | |
|---|---|
| **redukcia identifikátoru:** | **E -> i** |
| **redukcia nonterminálu v zátvorkách:** | **E -> (E)** |
| **redukcia binárnych operácií:** | **E -> E operand E** |

Pri redukcii sa kontroluje sémantika výrazu. V prípade nepovolenej kombinácie typov pre danú operáciu je vrátená sémantická chyba. Analýza výrazu je korektne ukončená,keď na zásobníku zostane počiatočný symbol $ a na vstupe je token bodkočiarky (stredníku) alebo jedno z kľúčových slov : then, else, do, end.

## 3.3 Sémantický analyzátor

Sémantická analýza prechádza tokeny ci skupiny tokenov získané zo syntaktickej analýzy a priraďuje im význam.

Ako už bolo spomínané vyššie ,súčasťou našej syntaktickej analýzy je aj sémantická kontrola. Tá sa vykonáva napr. pri deklarácii premenných, priraďovaní, pri funkciách, po vyhodnotení podmienok a podobne. Taktiež prebieha aj pri redukcii výrazu pri syntaktickej analýze zdola hore.
V prípade nepovolenej kombinácie typov pre danú operáciu je vrátená sémantická chyba.

## 3.4 Interpret

Interpret ifj2009 používa inštrukcie v trojadresnom kóde. Obecný tvar inštrukcie vyzerá nasledovne : [typ_inštrukcie, adresa1, adresa2, adresa_cieľu]. Samotné inštrukcie sú uložené v poli, ktorého veľkosť sa mení dynamicky podľa potreby. Index inštrukcie poľa slúži zároveň ako adresa, ktorú využívajú inštrukcie skoku JZ(jump if zero) a JMP(jump).

Pri zpracovávaní samotného zoznamu inštrukcií nastaví interpret najprv adresu aktuálnej inštrukcie na 0. Potom prečíta nasledujúcu zo zoznamu a ak nedošiel na koniec zoznamu, inštrukcia sa podľa jeho typu vykoná. Zatím načítame znova a cyklus sa opakuje.

Druhy inštrukcií, ktoré interpret používa ,delíme podľa typu na :

- aritmetické - add, sub, mul, div
- vstupno-výstupné - read, write
- inštrukcie skoku - jmp, jnz
- inštrukcie porovnania - prefix cmp_ : less, great, lesseq, greateq, equal, notequal
- ostatné - sort, find (vstavané funkcie)

Interpret ošetruje dve chyby : chybu delenia nulou a chybu pri čítaní zo štandartného vstupu – napríklad vstup neukončený znakom konca riadku '\n' – 0xa . V týchto prípadoch program vracia chybový kód 4 – vnútorná chyba interpretu.

# 4. Imlementácia zadaných algoritmov

## 4.1 Tabuľka symbolov

Tabuľka symbolov bola implementovaná ako hashovacia tabuľka, s ktorou sa pracuje pomocou funkcie lookup, ktorá zabezpečuje hľadanie aj vkladanie do tabuľky. Obecne do tabuľiek symbolov ukladáme všetky objekty, ktoré sú pri analýze zaznamenané.
Naša tabuľka obsahuje :

- hodnota položky
- názov položky
- typ položky
- identifikátor či sa jedná o normálnu premennú alebo dočasnú konštantu

## 4.2 Merge sort

Pri implementácii vstavanej funckie sort, ktorá vykonáva zoradenie znakov v reťazci podľa ich ordinálnej hodnoty, sme použili radiaci algoritmus Merge sort.

Tento algoritmus je typickým príkladom prístupu "Rozdeluj a panuj". Množina dát je rozdelená na dve približne rovnako veľké podmnožiny, ktoré su naďalej rekurzívne delené na polovice až do podmnožín o veľkosti jedného prvku. Potom nasleduje spájanie a zoraďovanie týchto už zotriedených množín, až kým vo výsledku nedostaneme celú zoradenú pôvodnú množinu. Algoritmus má zložitosť O(n*log(n)) a je stabilný , čo znamená že zachováva vzájomné poradie prvkov z rovnakým kľúčom. Nevýhodou algoritmu je, že funckia pre spájanie potrebuje pre svoju efektívnu implementáciu pomocné pole o veľkosti N. Implementácia našej funkcie sort prebehla bez väčších problémov. Je umiestnená v module *str*.c .

## 4.3 Boyer – Mooreov algoritmus

Tento vysoko výkonný algoritmus sme použili pri implementácii druhej vstavanej funckie find pre hľadanie podreťazca v reťazci.

Algoritmus bol popísaný Bobom Boyerom a J. Stroherom Moorom v roku 1977. Jeho základná myšlienka spočíva v posune vzoru o určitý počet miest doprava, pokým sa nezhoduje s hľadaným textom. Vzor a text sa porovnávajú odzadu. V prípade ak sme pri porovnávaní prišli ku prvému znaku vzoru, vyhľadanie bolo úspešné.
Algoritmus pred samotným vyhľadávaním inicializuje pole znakov, ktorého veľkosť je rovnaká ako veľkosť abeceedy, ktorá sa môže v texte vyskytnúť. Ak sa znak z abecedy vo vzore vyskytuje, je hodnota na indexe ordinálnej hodnoty znaku tabuľky nastavená na vzdialenosť jeho prvého výskytu vo vzore zprava. V opačnom prípade je hodnota v tabuľke nastavená na dĺžku vzoru. Časová zložitosť tejto varianty je O(MN) v najhoršom možnom prípade. N je dĺžka vzoru, M je dĺžka textu.

# 5. Záver

Počas našej doterajšej doby štúdia na našej fakulte sme si po prvý krát vyskúšali vytvoriť rozsiahlejší projekt. Tento projekt nám priniesol nové skúsenosti, naučili sme sa zostrojiť jednoduchý interpret programovacieho jazyka ifj2009, ako aj sme si vyskúšali tímovú prácu. Dúfame, že tieto cenné skúsenosti využijeme nielen na skúške z predmetu IFJ , ale aj v budúcnosti pri riešení reálnych problémov.

## 5.1 Použitá literatúra

- Prednášky a slajdy predmetu IFJ
- Minuloročné demonštračné cvičenia
- Internet

## 5.2 Metriky kódu

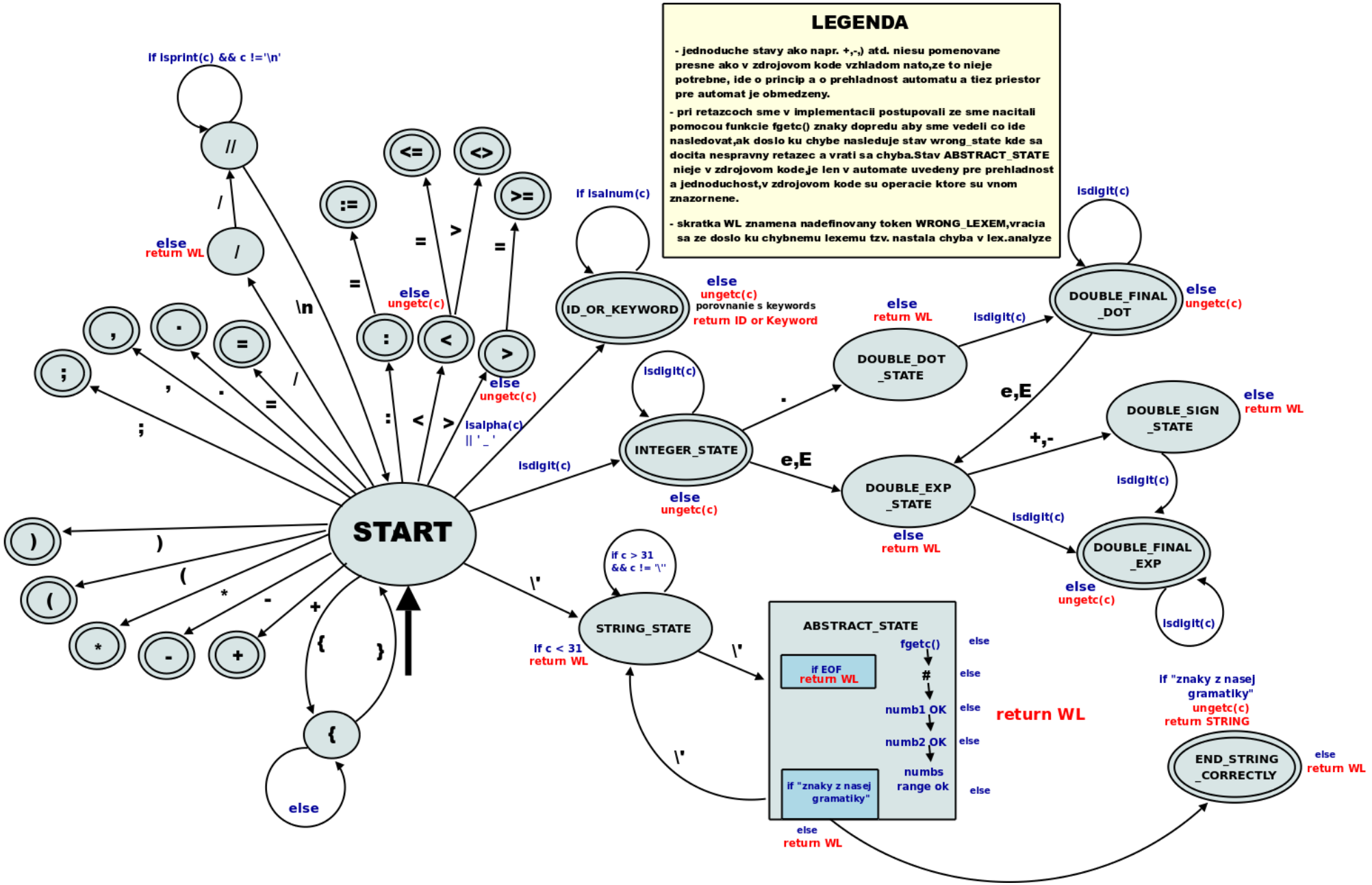
- 3888 riadkov
- 12 719 slov
- veľkosť spustiteľného súboru 43 kB
- 19 súborov (.c , .h , makefile)
- veľkosť projektu 492 kB

**Na základe programu SLOCCOUNT**

- 2583 riadkov čistého kódu
- predpokladaná doba vývoja jedným programátorom : 6,5 mesiacov
- náklady na vývoj : $ 73 177 (1 279 353 kč)

# Prílohy

- Na daľšej strane model konečného deterministického automatu našej lexikálnej analýzy
- LL – gramatika pre náš syntaktický analyzátor


LEGENDA
- jednoduche stavy ako napr. +,-,) atd. niesu pomenovane presne ako v zdrojovom kode vzhladom nato,ze to nieje potrebne, ide o princip a o prehladnost automatu a tiez priestor pre automat je obmedzeny.
- pri retazcoch sme v implementacii postupovali ze sme nacitali pomocou funkcie fgetc() znaky dopredu aby sme vedeli co ide nasledovat,ak doslo ku chybe nasleduje stav wrong_state kde sa docita nespravny retazec a vrati sa chyba.Stav ABSTRACT_STATE nieje v zdrojovom kode,je len v automate uvedeny pre prehladnost a jednoduchost,v zdrojovom kode su operacie ktore su vnom znazornene.
- skratka WL znamena nadefinovany token WRONG_LEXEM,vracia sa ze doslo ku chybnemu lexemu tzv. nastala chyba v lex.analyze
START
if isprint(c) && c !='\n'
//
/
else return WL
\n
:=
<=
<>
>=
=
>
else ungetc(c)
if isalnum(c)
ID_OR_KEYWORD
else ungetc(c)
porovnanie s keywords
return ID or Keyword
isalpha(c) || '_'
isdigit(c)
INTEGER_STATE
else ungetc(c)
.
DOUBLE_DOT _STATE
else return WL
isdigit(c)
DOUBLE_FINAL _DOT
else ungetc(c)
isdigit(c)
e,E
DOUBLE_EXP _STATE
else return WL
+,-
DOUBLE_SIGN _STATE
else return WL
isdigit(c)
isdigit(c)
DOUBLE_FINAL _EXP
else ungetc(c)
isdigit(c)
\'
if c > 31 && c != '\''
STRING_STATE
if c < 31 return WL
\'
ABSTRACT_STATE
fgetc()
if EOF return WL
#
numb1 OK
numb2 OK
numbs range ok
if "znaky z nasej gramatiky"
else
return WL
\'
if "znaky z nasej gramatiky" ungetc(c) return STRING
END_STRING _CORRECTLY
else return WL
( ) * - + { } ; , . = / : < > { else

+1 <prog> --> <declr> <stat_body> . <EOF>
+2 <declr> --> var <declr_list>
+3 <declr> --> begin
+4 <declr_list> --> id : <type>; <is_declr_end>
+5 <is_declr_end> --> <declr_list>
+6 <is_declr_end> --> begin
+7 <stat_body> --> end
+8 <stat_body> --> <stat_list>
+9 <type> --> string
+10 <type> --> integer
+11 <type> --> double
+12 <stat_list> --> <stat> <is_end>
+13 <is_end> --> ; <stat_list>
+14 <is_end> --> end
+15 <stat> --> readln ( <id_list>
+16 <id_list> --> id <end_id_list>
+17 <end_id_list> --> , <id_list>
+18 <end_id_list> --> )
+19 <stat> --> write ( <expr_list>
+20 <expr_list> --> <expr> <end_expr_list>
+21 <end_expr_list> --> , <expr_list>
+22 <end_expr_list> --> )
+23 <stat> --> begin <is_empty>
+24 <is_empty> --> end
+25 <is_empty> --> <stat_list>
+26 <stat> --> if <expr> then <stat> <if_else>
+27 <if_else> --> else <stat>
+28 <if_else> --> ε
+29 <stat> --> while <expr> do <stat>
+30 <stat> --> id := <inner>
+31 <inner> --> <expr>
+32 <inner> --> sort ( expr )
+33 <inner> --> find ( expr , expr )
+34 <stat> --> repeat <repeat_body> <expr>
+35 <repeat_body> --> <stat> <repeat_end>
+36 <repeat_end> --> ; <repeat_body>
+37 <repeat_end> --> until