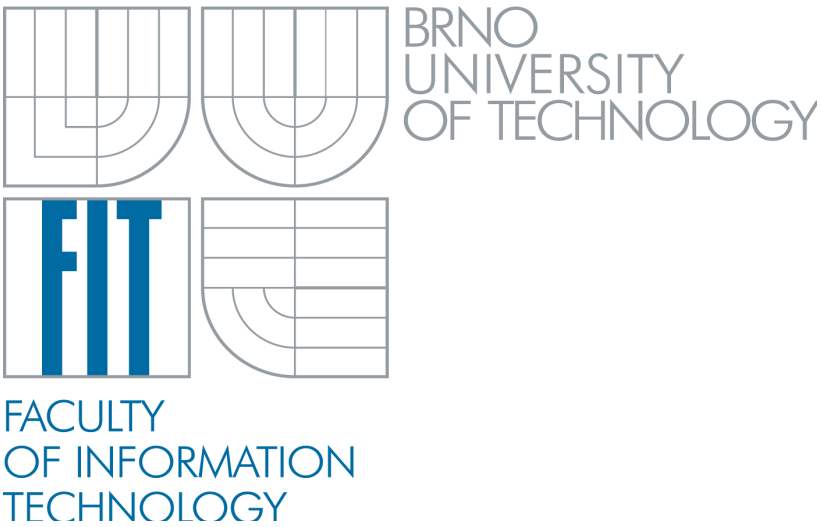



Dokumentácia k projektu pre predmety IFJ a IAL

# Implementácia interpretu imperatívneho jazyka IFJ14

Tým 074, varianta b/1/II
Implementované rozšírenia: *ELSIF*


| | |
|---|---|
| Kačmarčík Martin – 25% – vedúci týmu | *xkacma03* |
| Nedelka Roman – 25% | *xnedel07* |
| Jurdová Natália – 25% | *xjurdo00* |
| Skalník Marek – 25% | *xskaln03* |

# Obsah

1.Úvod.....3
1.1 Zadanie.....3
2.Implementácia.....3
2.1 Lexikálna analýza.....3
2.2 Syntaktická analýza.....4
2.2.1 Syntaktická analýza zhora-nadol.....4
2.2.2 Precedenčná analýza.....4
2.2.3 Sémantická analýza.....4
2.2.4 Generovanie kódu.....5
2.3 Interpret.....5
2.4 Použité algoritmy.....6
2.4.1 Triedenie a vyhľadávanie.....6
2.4.2 Tabuľka symbolov.....6
2.5 Rozšírenia.....6
2.5.1 ELSIF.....6
3.Práca v týme.....7
3.1 Organizácia.....7
3.2 Rozdelenie úloh.....7
4. Záver.....7
4.1 Metriky.....7
4.2 Použitá literatúra.....7
5. Prílohy.....8
5.1 Konečný automat.....8
5.2 Precedenčná tabuľka a pravidlá.....9
5.3 LL-gramatika.....10

# 1.Úvod

Táto dokumentácia popisuje implementáciu interpretu imperatívneho jazyka *IFJ14*, ktorý je tvorený podmnožinou jazyka *Pascal*.

Projekt sa dá rozdeliť do 3 častí, z ktorej každá bude podrobne popísaná:

- lexikálny analyzátor
- syntaktický a sémantický analyzátor
- interpret

## *1.1 Zadanie*

Program je implementovaný podľa **varianty b/1/II**, teda na vyhľadanie podreťazca v reťazci je použitá metóda *Boyer-Moore*, metóda triedenia vstavanej funkcie *sort* je *Quick sort* a tabuľka symbolov je implementovaná pomocou *hashovacej tabuľky*.

# 2.Implementácia

V tejto časti je detailne popísaná implementácia 3 hlavných častí programu (lexikálna analýza, syntaktická analýza, interpret), použitých algoritmov z predmetu IAL podľa varianty zadania a nami zvolených rozšírení.

## *2.1 Lexikálna analýza*

Lexikálnu analýzu môžeme v pomyselnom spracovaní programu pre preklad zaradiť na začiatok. Úlohou lexikálnej analýzy je spracovanie textu zo vstupného súboru, ktorý je tvorený zdrojovým textom programu zapísaného v jazyku *IFJ14*. Lexikálny analyzátor číta vstupný text a jednotlivé výrazy/znaky transformuje do tzv. *lexém*, ignoruje komentáre (t.j. text medzi zloženými zatvorkami { a }), biele znaky a odhaľuje lexikálne chyby v zdrojom programe. Každá lexéma je reprezentovaná pomocou tzv. *tokenu*, ktorý sa skladá z 2 zložiek - *state* a *name*.

Zložka tokenu *state* vyjadruje názov stavu v konečnom automate a zložka *name* je použitá len pri tokenoch, pri ktorých je potrebné si pamätať doplňujúce informácie (napr. hodnota celého čísla, názov identifikátoru). V prípade použitia zložky *name* je pre ňu alokovaný v pamäti konštantný priestor o dĺžke 128B a len v prípade reťazca presahujúceho túto dĺžku je tento priestor realokovaný na požadovanú veľkosť. Lexikálny analyzátor vracia načítaný *token* pri volaní funkcie *getNextToken* syntaktickým analyzátorom. Lexikálne chyby odhalené v tejto časti programu su poslané syntaktickej analýze pomocou *tokenu* so stavom *FALSE* a globálnej premennej *errCode*. Lexikálny analyzátor je implementovaný ako konečný automat (viz. 5.1 Konečný automat).

## *2.2 Syntaktická analýza*

### 2.2.1 Syntaktická analýza zhora-nadol

Ak by sa dala lexikálna analýza zaradiť na začiatok, syntaktická analýza (*parser*), ktorá je jadrom celého interpretu by mohla byť pomyselným stredom interpretu. Parser totiž riadi všetky ostatné komponenty interpretu. V našom prípade je riadiaca funkcia *parseru* funkcia *control*, volaná samotným *mainom* a vracia mu tiež typ chyby.

*Parser* po zavolaní inicializuje príslušné premenné, alokuje potrebnú pamäť a zavolá funkciu *zacatek*, ktorá začne prechádzať kód, hľadajúc syntaktické chyby metódou nazývanou *rekurzívny zostup***.** Táto metóda je založená na *LL gramatike*(viz. 5.3 LL-gramatika), pre ktorí bolo nutné vytvoriť pravidlá pre simuláciu *derivačného stromu*, čo je strom, ktorý obsahuje ako listy terminály a ako korene neterminály. Je nutné vytvoriť *LL-tabuľku*, ktorá nám hovorí, aké pravidlo použiť na základe znalostí aktuálneho *tokenu* na vstupe. Pre vytvorenie *LL-tabuľky* sme použili náhradný aplet. *Aplet* zo stránok predmetu IFJ sme nepoužili z dôvodu, že je v ňom možné použiť len určitý počet pravidiel aby bola výstupná *LL-tabuľka* použiteľná.

Vo chvíli, keď je pre *parser* pripravená *LL-tabuľka*, je implementácia samotného *parseru* veľmi jednoduchá. Jedná sa o manuálny prepis *LL-tabuľky* do jazyka C podľa šablóny. Šablóna spočíva v tom, že na začiatku máme premennú typu *boolean*, ktorá značí správnosť celého výrazu, nastavenú na hodnotu 0 (*false*). Vo chvíli, keď na vstupnej páske narazíme na *token*, ktorý sa nachádza v *LL-tabuľke* na riadku pre aktuálne pravidlo, pravidlo aplikujeme. Aplikácia pravidla znamená, že ak máme aktuálne na vstupnej páske terminál (*token*) a v gramatike je tiež *token*, požiadame *scanner* o ďalší *token* alebo ak je v pravidle neterminál, voláme príslušnú funkciu odpovedajúcu danému neterminálu (ktorá ho opäť spracuje). Ak pravidlá prebehnú v poriadku , vracajú hodnotu 1 (*true*) a nastavujú premennú *boolean* tiež na hodnotu 1 (čím označujú daný kus kódu za správny). Ak sa kód nedostane do správneho pravidla, hodnota *boolean* premennej zostane 0 a zistíme chybu. Samotná implementácia *rekurzívneho zostupu* je relatívne jednoduchá, ťažšie bolo pre ňu vytvoriť gramatiku. Z dôvodu, že *LL-gramatika* je pre výrazy neefektívna, použili sme pre syntaktickú analýzu výrazov precedenčnú syntaktickú analýzu *zdola-nahor*, ktorá je volaná *parserom* funkciou *prec_analysis* vždy, keď je na vstupe očakávaný výraz.

### 2.2.2 Precedenčná analýza

Precedenčná analýza pracuje s precedenčnou tabuľkou a pravidlami precedenčnej analýzy (viz. 5.2 Precedenčná tabuľka a pravidlá), a so zásobníkom. Na vstupe vyhodnocuje prichádzajúci *token* a najvrchnejší terminál na zásobníku, pričom podľa precedenčnej tabuľky aplikuje príslušné pravidlo. Riadenie je predané všeobecnému syntaktickému analyzátoru ak je na vstupe *token*, ktorý nemôže byť súčasťou výrazu alebo ak dôjde k chybe v rámci precedenčnej analýzy.

### 2.2.3 Sémantická analýza

Spracovanie sémantickej analýzy sme sa rozhodli zakomponovať priamo do *rekurzívneho zostupu* a precedenčnej analýzy. Bolo nutné implementovať vkladanie funkcií do príslušnej tabuľky symbolov

(globálna alebo lokálna). Pri vkladaní bolo nutné kopírovať každú premennú, správne alokovať pamäť atď. V našom prípade boli niektoré premenné typu *pole* a kopírovanie do pola typu *char* napríklad funkcia *strcpy* efektívne nedokáže. Nakoniec sa nám ale podarilo všetky problémy s pamäťou odladiť a valgrind nám poslušne hlásil „*no leaks are possible*“. Rozpoznanie do akej tabuľky vkladať sa robí na základe príznaku *isLocal*. Zakaždým, keď kód potrebuje pracovať s premennou, skontroluje sa či je premenná v tabuľke symbolov. Ak nie, nastáva chyba 3. Tiež bolo nutné ošetriť redefinície premenných/funkcií. Táto časť bola veľmi jednoduchá, prakticky sa jednalo o vyhľadanie premennej/funkcie pri jej vkladaní. Ak sa pri vkladaní v tabuľke symbolov premenná/funkcia vyskytla, bolo zrejmé, že sa jedná o redefiníciu. Typovú kompatibilitu sme riešili prevažne v precedenčnej analýze, kde sa pri spracovávaní jednotlivých výrazov kontroluje, či sú premenné rovnakého typu. Následne precedenčná analýza vracia *parseru* typ celého výrazu, vďaka čomu *parser* kontroluje typovú kompatibilitu pri priradení. V prípade chyby je nastavená premenná *errCode* na hodnotu 4.

### 2.2.4 Generovanie kódu

Súčasťou *parseru* a precedenčnej analýzy je aj generovanie kódu, kedy je volaná funkcia *IListInsert*, ktorá vloží do lineárneho zoznamu inštrukcií novú inštrukciu. Vkladanie inštrukcií sa riadi gramatikou, vďaka ktorej poznáme akú inštrukciu vložiť.

Ak nám funkcia *zacatek* vráti hodnotu premennej boolean rovnú 1, vieme že celý kód je správne, pretože sa podarilo odsimulovať derivačný strom a môžeme teda postupovať ďalej. Ak funkcia *zacatek* nevráti hodnotu 1, nastavíme premennú *errCode*, ktorá obsahuje číslo chyby, na číslo 2(čo je kód pre syntaktickú chybu). V prípade, že počas syntaktickej kontroly nastala nejaká chyba, nemá zmysel volať interpret. Ak všetko prebehlo v poriadku, zavolá sa interpret, ktorý sa postará o interpretáciu kódu. Po ukončení interpretu sa vyčistí pamäť a funkcii *main* sa vráti príslušná návratová hodnota chyby(prípadne sa vráti 0, ak všetko prebehlo v poriadku).

## *2.3 Interpret*

Interpret je časť programu, kde sa vykonávajú jednotlivé inštrukcie *3-adresného kódu*, ktorý bol pôvodne navrhnutý tak, aby bolo čo najjednoduchšie ho generovať a vykonávať. Každý príkaz v jazyku *IFJ14* mal k sebe ekvivalentnú inštrukciu. Aritmetické operácie sa riešili pomocou zásobníku a zápisu v *post-fixovej* notácii. Neskôr sa však tento spôsob implementácie zdal neefektívny a veľmi pomalý a to hlavne kvôli tomu, že pri každom načítaní premennej musel interpret vyhľadať, akú premennú má načítať, aký má typ a až podľa toho sa rozhodol, ktorá operácia sa vykoná. Preto bola vymyslená nová sada inštrukcií. Inštrukcie pre načítanie a ukladanie premenných v seba nosia informáciu o type a priamo o adrese k dátam. Potom môžeme jednoducho určiť, aké aritmetické operácie budeme vykonávať a aký bude mať výsledok typ. Z dôvodu, že by bolo veľmi zložité ostatné časti programu, kde sa inštrukcie generujú, prerobiť, tak sa vygenerované staré inštrukcie prečítajú a vygeneruje sa z nich sada nových inštrukcií. Každá premenná sa uloží do tabuľky symbolov a priradí sa jej unikátny index. Neskôr bude tento index použitý do poľa premenných. Toto riešenie nie je príliš efektívne, ale bolo najjednoduchšie pre implementáciu.

Samotný interpret sa skladá zo zoznamu inštrukcií, z poľa kde sú uložené lokálne a globálne premenné, zásobníku a z funkcie, kde sa inštrukcie vykonávajú. Zoznam inštrukcií je implementovaný ako *lineárny zoznam*. Samotných inštrukcií je 150. Inštrukcie na načítanie a ukladanie hodnôt v sebe nesú

informácie o type a indexe do poľa lokálnych alebo globálnych premenných. Inštrukcie, ktoré môžu meniť ukazovateľ na aktuálnu vykonávanú inštrukciu (ako je napr. *while* alebo *if*) v sebe majú uloženú adresu inštrukcie kam skočiť. Pole, v ktorom je uložený zásobník a premenné programu sú alokované staticky na zásobníku programu. Vďaka tomu je prístup k hodnotám veľmi rýchly. Vykonávanie inštrukcie sa realizuje vo funkcii v cykle. Pomocou *switchu* sa rozhodneme, čo sa bude robiť. V prípade volania funkcie sa na zásobník uložia argumenty a rekurzívne sa zavolá funkcia s vykonávaním inštrukcií. Vďaka tomu nemusíme ukladať rámce premenných na nejaký nami vytvorený zásobník, ale využijeme zásobník v systéme a všetko je tak mnohokrát rýchlejšie.

## *2.4 Použité algoritmy*

Táto podkapitola je venovaná popisu nami implementovaných algoritmov podľa varianty zadania b/1/II z pohľadu predmetu IAL.

### 2.4.1 Triedenie a vyhľadávanie

Pre vyhľadávanie podreťazca v reťazci bol použitý *Boyer-Moorov algoritmus* a pre metódu triedenia, ktorú využíva vstavaná funkcia *sort*, bol použitý algoritmus *Quick sort* Inšpiráciou pri implementácii daných algoritmov nám bola študijná opora k predmetu IAL [1], čo spravilo implementáciu značne jednoduchšou.

### 2.4.2 Tabuľka symbolov

Tabuľku symbolov sme mali podľa zadania implementovať ako tabuľku s **rozptýlenými položkami** (hash table). Keďže sme mali v zadaní 2. domácej úlohy predmetu IAL túto tabuľku implementovať, použili sme teda náš kód z domácej úlohy. Tabuľka z domácej úlohy bola skvele spracovaná, obsahovala uvoľnenie pamäte, vyhľadanie, aktualizáciu, všetko čo bolo treba. Boli nutné mierne úpravy (napr. uvoľňovanie pamäte bolo značne prerobené, pretože muselo rozlišovať funkcie a premenné), ale nakoniec poslúžil pôvodný kód nadmieru dobre. Veľmi užitočná bola tiež funkcia pre výpis tabuľky, ktorú obsahoval testovací skript. Vďaka tejto funkcii sme si mohli efektívne prezerať obsah našej tabuľky symbolov. Ako veľkosť tabuľky bolo nutné použiť prvočíslo, konkrétne sme zvolili číslo 101. Hashovacia funkcia je rovnaká ako v domácej úlohe a sčíta nominálne hodnoty jednotlivých znakov, ktoré ďalej spracuje funkciou *modulo* (pre "trafenie sa" do intervalu tabuľky). Hashovacia funkcia nie je najideálnejšia, ale pri testovaní sa nám nestalo, že by sa dve premenné „trafili“ do rovnakého indexu tabuľky (pravdepodobne aj z dôvodu pomerne veľkej veľkosti tabuľky), preto sme sa rozhodli ju zachovať.

## *2.5 Rozšírenia*

### *2.5.1 ELSIF*

Toto rozšírenie je obdobou podmieneného príkazu *if.. then.. else..*, čo sa nám implementovalo veľmi jednoducho, pri čom iba musíme testovať, či za zloženým príkazom po *then* nasleduje *else* alebo nie.

# 3.Práca v týme

## *3.1 Organizácia*

Tým sme mali zostavený hneď na začiatku semestru a po vzájomnej dohode sme zvolili vedúceho tímu. Zo začiatku sme mávali stretnutia každý týždeň, kde sme konzultovali zadanie a rozdeľovali si prácu. Postupom času, keď boli rozdelené úlohy, sme zmenili spôsob komunikácie a namiesto pravidelných stretnutí sme začali riešiť problémy cez skype a sociálne siete a začali sme využívať verzovací systém na zdieľanie jednotlivých zdrojových kódov. Používali sme verzovací systém Git cez službu BitBucket. Problémy v rámci projektu sa nám naskytli iba pri implementácii, čo sme ale vždy efektívne vyriešili vďaka dobrej komunikácii v týme. Z organizačného a komunikačného hľadiska sme teda nemali žiadne problémy.

## *3.2 Rozdelenie úloh*

***Nedelka Roman*** – lexikálna analýza, dokumentácia

***Kačmarčík Martin*** – syntaktická analýza, hashovacia tabuľka

***Jurdová Natália*** – precedenčná analýza

***Skalník Marek*** – vstavané funkcie, interpret

# 4. Záver

Projekt bol z časového hľadiska náročný, nad jeho riešením sme strávili približne dva mesiace. K projektu sme od začiatku pristupovali zodpovedne, materiály k prednáškam sme študovali dopredu a mali sme v pláne využiť pokusné odovzdanie, čo sa nám aj podarilo a veľmi pomohlo. Najviac času nám zabralo testovanie a odhaľovanie chýb a spájanie všetkých častí programu dohromady. Bol to náš prvý skupinový projekt a mnohému nás naučil – či už tímovej spolupráci, komunikácii alebo niektorým programátorským zručnostiam(napr. práca s verzovacím systémom). Z celkového hľadiska hodnotíme projekt veľmi pozitívne.

## *4.1 Metriky*

- počet zdrojových súborov: 21
- veľkosť spustiteľného programu(Linux 64bit): 141,1 kB Bytov
- počet git commitov:  129

## *4.2 Použitá literatúra*

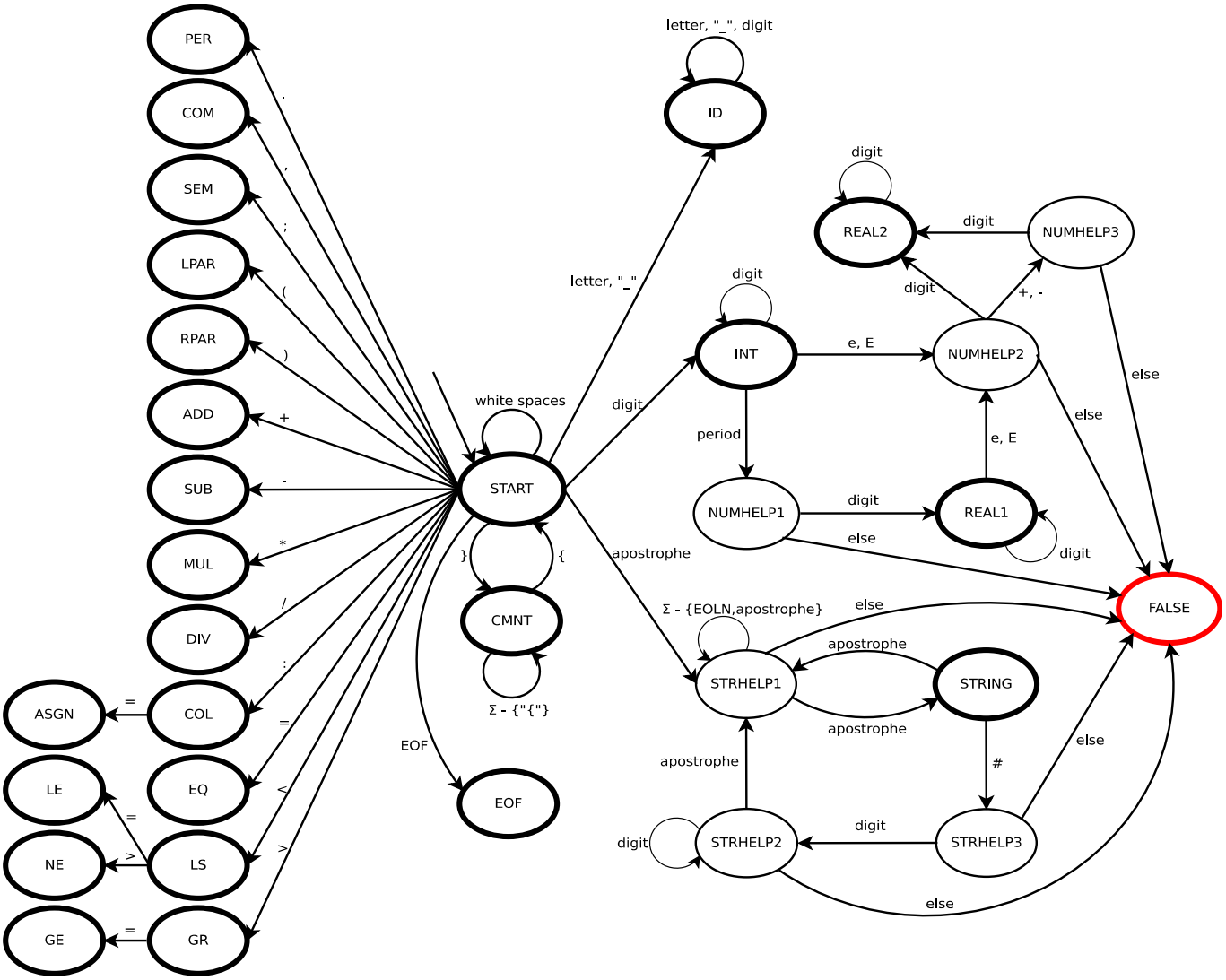
[1] Prof. Ing. Jan M. Honzík, CSc., ALGORITMY - Studijní opora. Verzia: 14-Q, 2014.

# 5. Prílohy

## 5.1 Konečný automat

**Vysvetlivky ku grafu:**

- hrubá elipsa – koncový stav
- tenká elipsa – nekoncový stav
- START - počiatočný stav
- FALSE – chybný stav




Obrázok 1: graf konečného automatu

## *5.2 Precedenčná tabuľka a pravidlá*

i->Integer
i->Real
i->String
i->ID
i->True
i->False
i->Nil
E->E*E
E->E/E
E->E+E
E->E-E
E->E<E
E->E>E
E->E<=E
E->E>=E
E->E<>E
E->E=E
E->(E)
E->i

| | * | / | + | - | < | > | <= | >= | <> | = | ( | ) | i | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| * | > | > | > | > | > | > | > | > | > | > | < | > | < | > |
| / | > | > | > | > | > | > | > | > | > | > | < | > | < | > |
| + | < | < | > | > | > | > | > | > | > | > | < | > | < | > |
| - | < | < | > | > | > | > | > | > | > | > | < | > | < | > |
| < | < | < | < | < | > | > | > | > | > | > | < | > | < | > |
| > | < | < | < | < | > | > | > | > | > | > | < | > | < | > |
| <= | < | < | < | < | > | > | > | > | > | > | < | > | < | > |
| >= | < | < | < | < | > | > | > | > | > | > | < | > | < | > |
| <> | < | < | < | < | > | > | > | > | > | > | < | > | < | > |
| = | < | < | < | < | > | > | > | > | > | > | < | > | < | > |
| ( | < | < | < | < | < | < | < | < | < | < | < | = | < | |
| ) | > | > | > | > | > | > | > | > | > | > | | > | | > |
| i | > | > | > | > | > | > | > | > | > | > | | > | | > |
| $ | < | < | < | < | < | < | < | < | < | < | < | | < | |

Obrázok 2: precedenčná tabuľka

## 5.3 LL-gramatika

<zacatek> -> var  <var-def> <after-var>
<zacatek> -> function <func-def>
<zacatek> -> begin <st-list>
<type> -> integer
<type> -> real
<type> -> string
<type> -> boolean
<term> -> id
<term> -> <type>
<after-var> -> function <func-def>
<after-var> -> begin <st-list>
<after-stat> -> ; <st-list>
<after-stat> -> end <after-end>
<after-end> -> . EOF
<after-end> -> eps
<st-list> -> <stat> <after-stat>
<par-list> -> id : <type>  <param_n>
<par-list> -> epsilon
<param_n> -> ; id :  <type> <param_n>
<param_n> -> epsilon
<after-func> ->  <var-func> begin <func-prikazy>
<next-func> -> begin <st-list>
<next-func>  -> function <func-def>
<after-func> ->  forward ;  function <func-def>
<func-def> ->  id ( <par-list> ) : <type> ; <after-func>
<var-def> -> id : <type> ; <var-def>
<var-def> -> epsilon
<stat> -> id := <expresion>
<stat> -> if <if-exp> then  begin <epsilon-if>
<after-if> -> else begin <st-list>
<after-if> -> epsilon
<stat> -> while <if-exp> do begin <st-list>
<stat> -> readln (id)
<stat> begin <slozeny>
<stat> -> write ( <term-list> )
<expresion> ->PRECEDENCNI-ANALYZA
<expresion> -> id ( <par-list> )
<term-list> -> <term> <term-n>
<term-n> -> epsilon
<term-n> -> , <term> <term-n>
<slozeny> -> <st-list>
<slozeny> -> end
<epsilon-if> -> <st-list> <after-if>
<epsilon-if> -> end <after-if>
<func-prikazy> -> <st-list> end; <next-func>
<func-prikazy> -> end; <next-func>
<var-func> -> epsilon
<var-func> -> var id : <type> ; <var-def>

Obrázok 3: LL-gramatika