

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÝCH TECHNOLOGIÍ



Dokumentácia k projektu do predmetu IFJ a IAL
Implementácia interpreta jazyka IFJ16

Tím 026, varianta B/1/II

11. decembra 2016

Členovia tímu:

Dávid Bolvanský (xbolva00) - vedúci tímu, 20%

Juraj Dúbrava (xdubra03), 20%

Tamara Krestianková (xkrest07), 20%

Martin Marušiak (xmarus07), 20%

Veronika Svoradová (xsvora01), 20%

Rozšírenia:

SIMPLE, BOOLOP

OBSAH

ÚVOD.....	2
1 IMPLEMENTÁCIA INTERPRETA JAZYKA IFJ16	3
1.1 Lexikálna analýza	3
1.2 Syntaktická analýza (bez spracovania výrazu).....	3
1.3 Syntaktická analýza (spracovanie výrazu)	3
1.4 Sémantická analýza	4
1.5 Interpret.....	4
2 VSTAVANÉ FUNKCIE.....	5
2.1 Algoritmy do predmetu IAL	5
2.1.1 Implementácia radenia (Quick sort).....	5
2.1.2 Implementácia vyhľadávania podreťazca v reťazci (Boyer-Moore)	5
2.2 Implementácia tabuľky symbolov	5
2.3 Vstavane funkcie pre načítanie literálu a výpis termov	5
2.4 Vstavane funkcie pre prácu s reťazcom.....	6
3 PRÁCA V TÍME	6
4 ZÁVER.....	7
5 REFERENCIE	7
6 PRÍLOHY	8

ÚVOD

Dokumentácia popisuje implementáciu interpreta jazyka IFJ16, ktorý je veľmi zjednodušenou podmnožinou jazyka Java SE . Jeho implementáciu sme rozdelili do 4 hlavných častí, ktoré budú bližšie popísané v ďalšej časti dokumentácie:

1. Lexikálny analyzátor
2. Syntaktický analyzátor
3. Sémantický analyzátor
4. Interpret

Zvolili sme si variantu b/1/II, ktorá obsahovala zadanie na implementáciu vyhľadávania podreťazca v reťazci s použitím algoritmu Boyer-Moore, implementáciu radenia s využitím Quick sort algoritmu a implementáciu tabuľky symbolov pomocou tabuľky s rozptýlenými položkami. Jednotlivé implementácie týchto algoritmov nájdete v kapitole 2.

Súčasťou dokumentácie sú aj tri prílohy, ktoré obsahujú diagram konečného automatu špecifikujúceho lexikálny analyzátor, LL gramatiku a precedenčnú tabuľku, ktoré sú jadrom syntaktického analyzátora.

1 IMPLEMENTÁCIA INTERPRETA JAZYKA IFJ16

1.1 Lexikálna analýza

Lexikálny analyzátor (scanner) je implementovaný v súboroch scanner.c a scanner.h. Je to jediná časť celého interpreta, ktorá pracuje priamo so zdrojovým súborom. Jeho úlohou je načítať zdrojový kód, odstrániť zbytočné časti ako sú biele znaky a komentáre, a finálne previesť lexémy na tokeny. Token sa skladá z dvoch častí: typ a atribút. Typ tokenu sa určí aplikovaním konečného automatu (viď príloha č.1) na postupnosť znakov načítaných zo vstupu.

Pre podporu nekonečne dlhého reťazca sme vytvorili pomocnú knižnicu, ktorá je implementovaná v súboroch strings.c a strings.h, a ktorá nám umožňuje pracovať s potenciálne nekonečným reťazcom. Ak prijatá postupnosť znakov nie je platná pre žiadne pravidlo, dochádza k lexikálnej chybe pri spracovávaní zdrojového textu. Lexikálny analyzátor poskytuje funkciu `get_next_token`, ktorá je následne využívaná syntaktickým analyzátorom.

1.2 Syntaktická analýza (bez spracovania výrazu)

Syntaktický analyzátor (parser) kontroluje množinu pravidiel, ktorá určuje prípustné konštrukcie daného jazyka. Základné možné konštrukcie jazyka IFJ16 a prídavné konštrukcie, ktoré vyplynuli z voľby rozšírení SIMPLE a BOOLOP sú definované v LL gramatike (viď príloha č. 2). Samotná syntaktická analýza je riešená metódou rekurzívneho zostupu a je jadrom celého interpreta a spolupracuje s ostatnými časťami interpreta.

Kvôli špecifickým požiadavkám jazyka Java, a teda aj IFJ16 ako jeho podmnožinou, dochádza k parsovaniu v dvoch prechodoch v rámci syntaktického analyzátoru. V prvom prechode získavame tokeny z lexikálneho analyzátoru a zároveň ich ukladáme do poľa tokenov, ktoré je implementované v súboroch token_buffer.c a token_buffer.h. Ak nenastane žiadna chyba počas prvého priechodu, pokračujeme. V druhom prechode získavame tokeny z práve vytvoreného poľa tokenov. Zaviedli sme si logický prepínač na zistenie v ktorom priechode práve sme. Priebežne sa v rámci prvého a druhého prechodu vykonávajú sémantické kontroly a generujú sa inštrukcie na globálnu inštrukčnú pásku alebo na inštrukčnú pásku funkcie. Konštanty v argumentoch pri volaniach funkcií v interpretovanom kóde ukladáme do jednosmerne viazaného zoznamu konštánt, ktorý je implementovaný v memory_constants.c a memory_constants.h, za účelom získania adresy v pamäti pre neskoršie generovanie 3-adresného kódu. Po úspešnej syntaktickej a sémantickej analýze dochádza k samotnej interpretácii kódu.

1.3 Syntaktická analýza (spracovanie výrazu)

Spracovanie výrazov je úloha precedenčnej syntaktickej analýzy implementovanej v súbore expr.c a expr.h. Precedenčná syntaktická analýza je riadená na základe precedenčnej tabuľky (viď príloha č. 3), ktorá popisuje prioritu spracovania jednotlivých operátorov a operandov. Vstupom analýzy sú tokeny zo špeciálneho poľa tokenov vytvoreného pre precedenčnú

syntaktickú analýzu, tokeny získame funkciou `get_next_token_psa`. Jednosmerne viazaný zoznam konštánt využívame aj v tejto fáze a ukladáme sem konštanty použité vo výrazoch.

Spracovávané vstupné tokeny sú počas analýzy ukladané na zásobník a vyhodnocovanie výrazu, ktorý vzniká na zásobníku je riadené precedenčnou tabuľkou. Dôležitou súčasťou spracovania výrazu je kontrola správnosti dátových typov nad ktorými sú uskutočňované jednotlivé aritmeticko - logické operácie a takisto syntaktické kontroly výrazu. V rámci priebežného spracovávaní výrazu prebieha aj generovanie 3-adresného kódu pre interpret.

1.4 Sémantická analýza

Úlohou sémantickej analýzy je preskúmať logický význam jednotlivých výrazov jazyka a zistiť ich platnosť pre daný programovací jazyk. Aj napriek tomu, že jazyk IFJ16 je podmnožinou jazyka Java, obsahuje isté špecifiká, ktorými sa od nej líši. Kľúčovým prvkom je spolupráca s tabuľkou symbolov. Z tejto tabuľky sa získavajú rôzne informácie a zároveň sa kontroluje resp. rozhoduje, či nedošlo k reдекларácii tried, premenných alebo funkcií. Zároveň sa kontroluje či nedošlo k typovej nekompatibilitate pri priradení alebo vo výrazoch a taktiež prípadná nezhoda v počte parametrov funkcie, a rôzne iné sémantické kontroly. Kontroluje sa prítomnosť funkcie `run` v triede `Main`, ktorá je nevyhnutá pre začatie procesu interpretácie.

1.5 Interpret

Interpret vykonáva inštrukcie, ktoré mu boli vygenerované v predošlých fázach. Inštrukciu reprezentujeme prostredníctvom 3-adresného kódu. Jednotlivé inštrukcie za sebou nasledujú v inštrukčnej páske, ktorú reprezentuje jednosmerne viazaný zoznam. Významovo rozlišujeme dva typy páso, globálnu a lokálnu (funkcie). Na globálnej inštrukčnej páske sa nachádzajú priradenia statických premenných. Globálna inštrukčná páska sa začne vykonávať ako prvá, z nej je potom vykonaná inštrukčná páska funkcie `run` z triedy `Main`, ktorá môže vyvolať vykonanie inštrukčných páso d'alších funkcií. Z páso funkcií môžu byť taktiež volané d'alšie pásy funkcií až do ľubovoľného zanorenia. Vykonanie inštrukčnej pásky v rámci inej inštrukčnej pásky je implementované rekurzívne.

K tomu aby sme mohli volať funkcie, a neprepisovali pritom hodnoty v tabuľke symbolov, slúžia rámce, ktoré sa vytvárajú vždy pred volaním funkcie a zabezpečia uloženie parametrov a vytvorenie lokálnych premenných pre funkciu. Pred vykonaním inštrukcie je potrebné preložiť adresy jednotlivých operandov tak, aby sa v prípade, že ide o lokálnu premennú, odkazovali na príslušnú hodnotu v rámci. Preklad sa uskutočňuje pomocou offsetov, ktorý je pre každú lokálnu premennú v danom rámci jedinečný. Po následnom preklade sa overí inicializácia operandov a následne vykoná inštrukcia.

2 VSTAVANÉ FUNKCIE

2.1 Algoritmy do predmetu IAL

2.1.1 Implementácia radenia (Quick sort) [4]

Quick sort alebo “radenie rozdeľovaním” patrí medzi najrýchlejšie metódy radenia polí. Tento algoritmus funguje na veľmi jednoduchom princípe. Pomocou mechanizmu partition prehodí prvky do dvoch častí poľa tak, že v ľavej časti sú všetky prvky menšie alebo rovné určitej hodnote (pivot) a v pravej časti sú všetky prvky väčšie ako táto hodnota.

V našom prípade sme za pivot zvolili pseudomedián. Pseudomedián sme vyrátali ako sčítanie ľavej a pravej hranice reťazca. Tento súčet sa nakoniec podelí (div) číslom dva:

- pseudomedián = (ľavá hranica + pravá hranica) delené (2).

Po rozdelení poľa na dve časti rekurzívne voláme funkciu pre opätovné radenie jednotlivých častí poľa. Celý algoritmus je uložený v súbore ial.c a ial.h

Priemerná doba výpočtu algoritmu Quick sort je v najlepšom prípade ($O(n \log(n))$), no však pri nevhodnom tvare vstupných dát môže byť časová náročnosť tohto algoritmu až $O(n^2)$. Autorom algoritmu z roku 1962 je Sir Charles Antony Richard Hoare, významná osobnosť v obore teórie a tvorby programov.

2.1.2 Implementácia vyhľadávania podreťazca v reťazci (Boyer-Moore)

Boyer-Moore algoritmus je jedným z najefektívnejších algoritmov pre porovnávanie reťazcov. Používa sa na vyhľadávanie vzoru v inom texte. Funguje na princípe spracovávania znakov sprava doľava. Ak dôjde pri porovnávaní dvoch znakov k nezhode, nastáva posun. Je vhodný pri hľadaní dlhého podreťazca. Čím dlhší je vyhľadávaný vzor, tým väčší počet znakov v texte je možné preskočiť a tým je kratšia doba spracovania. Rýchlosť algoritmu závisí na kardinalite abecedy a opakovaní podreťazca vo vzore. Meraním bolo zistené, že pre dĺžku vzoru väčšiu než 5 sa vykonáva asi 0.3 porovnaní z počtu znakov v prehľadávanom texte.

Pre implementáciu algoritmu sa používajú dve heuristiky - „posun nesprávneho znaku“ (bad symbol shift) a „posun správnej prípony“ (good suffix shift). Algoritmus berie ten z výsledkov dvoch heuristík, ktorý je výhodnejší. My sme použili obe, čím sa zvýšila efektivita algoritmu. Samotnú implementáciu môžete nájsť v súbore ial.c a ial.h.

Autormi algoritmu sú Robert S. Boyer a J Strother Moore z roku 1977. V najhoršom prípade doba behu algoritmu je $O(n+m)$, ale iba v prípade, že vyhľadávaný vzor sa nenachádza v texte. Naopak, ak sa hľadaný vzor nachádza v texte, doba behu algoritmu v najhoršom prípade je $O(n*m)$.

2.2 Implementácia tabuľky symbolov

Základom tabuľky symbolov je tzv. tabuľka s rozptýlenými položkami (tiež známa ako hashovacia tabuľka, ďalej len TRP), ktorá je určená pre našu variantu zadania. TRP je založená na použití poľa, ktoré je primárnym priestorom pre prácu s týmto typom tabuliek. Slúži na ukladanie dvojíc kľúč – hodnota. Kombinuje výhody vyhľadávania podľa indexu (priame adresovanie prvkov = zložitosť $O(1)$) a prechádzanie zoznamu (nízke nároky na pamäť). Dobrá

hashovacia funkcia má zásadný vplyv na výkon TRP. Nami vybraná hashovacia funkcia, ktorú sme použili, je uvedená v referenciách [3]. TRP je implementovaná v súbore `ial.c` a `ial.h`.

Každá trieda má svoju tabuľku symbolov, v ktorej sú uložené informácie o statických (globálnych) premenných a funkciách existujúcich v danej triede. Rovnako ako triedy tak aj funkcie majú svoju tabuľku symbolov, ktorá obsahuje informácie o parametroch/lokálnych premenných a zároveň o ich počte či dátových typov. Štruktúra položky v tabuľke symbolov, ktorá je univerzálna pre uloženie informácií o premennej či funkcii, ako aj funkcie pracujúce s tabuľkou symbolov sú implementované v `symbol_table.c` a `symbol_table.h`.

2.3 Vstavané funkcie pre načítanie literálu a výpis termov

Medzi vstavané funkcie pre načítanie literálu a výpis termov patria nasledovné:

int readInt () ; - funkcia prevedie načítaný reťazec na jedno celé číslo a vráti ho.

double readDouble () ; - funkcia prevedie načítaný reťazec na jedno desatinné číslo a vráti ho.

Pri oboch funkciách nesmie byť vypustený žiaden biely znak, takže formát vstupného reťazca musí odpovedať lexikálnym pravidlám pre daný literál, inak nastane chyba 7.

String readString () ; - funkcia vráti zo štandardného vstupu načítaný reťazec, ktorý je ukončený koncom riadku alebo koncom vstupu.

void print (term/konkatenácia) – ak sa jedná o term, funkcia vypíše hodnotu v danom formáte na štandardný výstup. Pokiaľ je term typu `int` alebo `double`, je najprv automaticky prevedený na reťazec a až potom vypísaný. Parameter, ktorý obsahuje neprázdnu postupnosť termov oddelených operátorom `+` je výraz, ktorý bude najprv vyhodnotený a následne podľa pravidiel pre výpis termov aj vypísaný.

Vyššie uvedené funkcie sú implementované v `builtin.c` a `builtin.h`.

2.4 Vstavané funkcie pre prácu s reťazcom

Medzi vstavané funkcie pre prácu s reťazcom v našom projekte patria nasledovné:

int length (String s) - funkcia vráti dĺžku (počet znakov) reťazca zadaného parametrom `"s"`.

String substr (String s, int i, int n) - funkcia vráti podreťazec zadaného reťazca `"s"`. Hľadaný podreťazec má dĺžku `"n"` a začína na indexe `"i"` zadaného reťazca `"s"`.

int compare (String s1, String s2) - funkcia lexikograficky porovnáva dva zadané reťazce `"s1"` a `"s2"` a vráti celočíselnú hodnotu 0, ak sú reťazce `"s1"` a `"s2"` rovnaké, 1, ak je `"s1"` väčší ako `"s2"` a -1 v ostatných prípadoch.

Funkcie `length`, `substr`, a `compare` sú implementované v `builtin.c` a `builtin.h`.

int find (String s, String search) - funkcia nájde prvý výskyt zadaného podreťazca `"search"` v reťazci `"s"` a následne vráti jeho pozíciu. Ak sa jedná o prázdny reťazec, vyskytuje sa vždy v každom reťazci na indexe 0. V prípade, že zadaný podreťazec `"search"` nie je nájdený, funkcia vráti hodnotu -1. V našom zadaní sme využili metódu Boyer-Moorovho algoritmu, ktorá je podrobne opísaná v časti 2.1.2 a nachádza sa v súboroch `ial.c` a `ial.h`.

String sort (String s) - funkcia, ktorá zoradí znaky v danom reťazci tak, aby znak s nižšou ordinálnou hodnotou vždy predchádzal znaku s vyššou ordinálnou hodnotou. V našom projekte

sme využili metódu Quick Sort algoritmu, ktorý je podrobnejšie opísaný v časti 2.1.1 a nachádza sa v ial.c a ial.h.

3 PRÁCA V TÍME

Prácu v tíme sme si rozvrhli nasledovne, s tým, že niektoré časti sa prekrývali, resp. bolo nutné sa na nich spoločne dohodnúť, ako postupovať, aké budú rozhrania, atď.

- **Dávid Bolvanský:** Lexikálna, syntaktická (bez výrazu), sémantická analýza, tabuľka symbolov
- **Juraj Ondrej Dúbrava:** Syntaktická analýza (spracovanie výrazu), tvorba LL gramatiky a precedenčnej tabuľky
- **Martin Marušiak:** Generovanie kódu, návrh 3AK, interpret, hashovacia tabuľka
- **Tamara Krestianková, Veronika Svoradová:** IAL algoritmy, vstavané funkcie, testovanie a dokumentácia

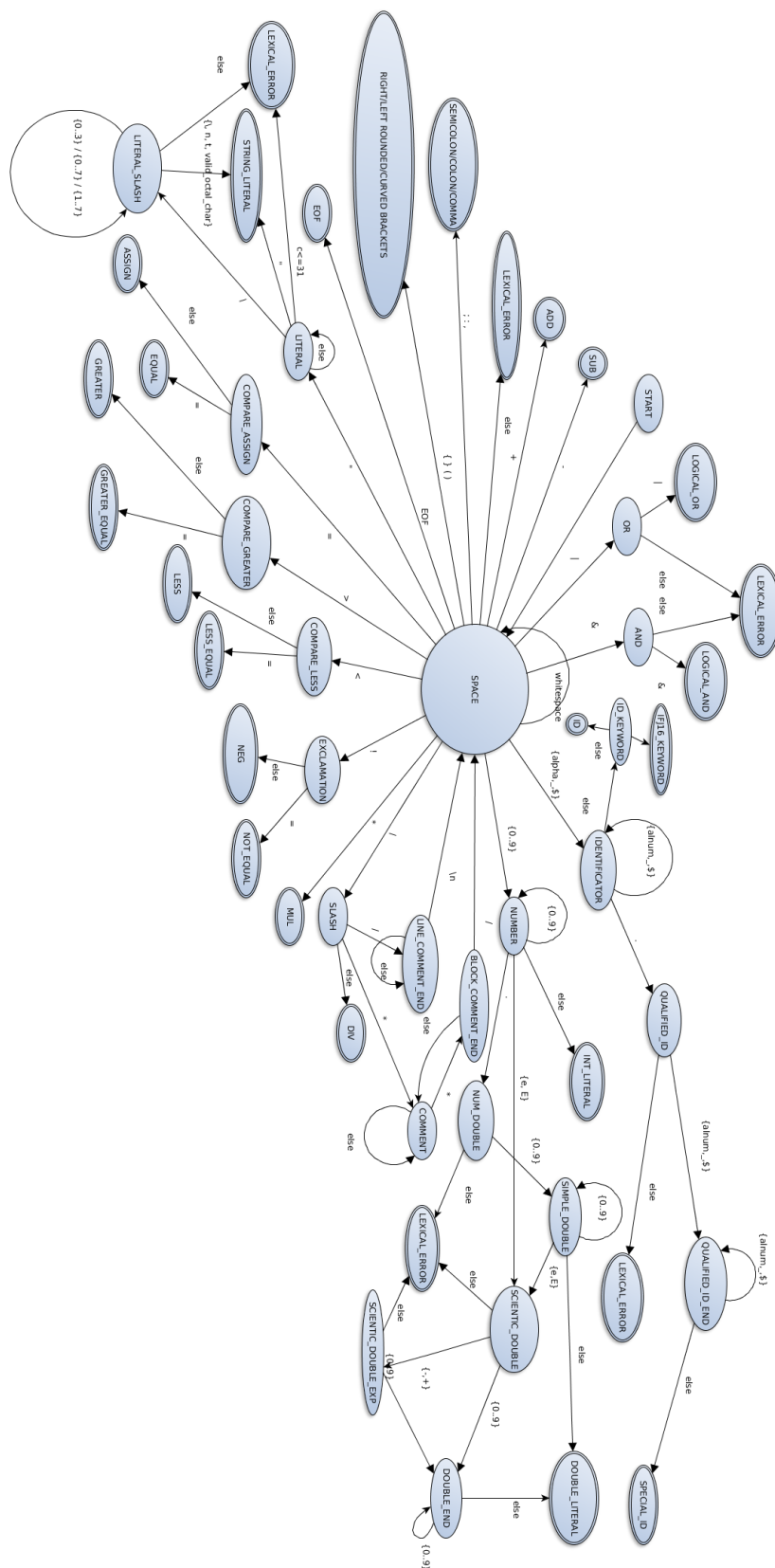
4 ZÁVER

Všetci členovia tímu sa zhodneme, že sme pred týmto projektom nepracovali na komplexnejšom a rozsiahlejšom projekte akým bol práve interpret jazyka IFJ16. Získali sme nie len nové skúsenosti a vedomosti ohľadom tvorby interpretov/prekladačov, ale aj nové znalosti v programovaní v jazyku C a používaní GITu. Dôležité bolo naučiť sa spolupracovať medzi sebou, akceptovať požiadavky ostatných pri tvorbe rôznych častí interpreta a dohodnúť sa na rozhraniach medzi jednotlivými časťami interpretu. Našou miernou výhodou bolo, že sme mohli rýchlo riešiť problémy, keďže sme od seba nebývali ďaleko a mohli sme osobne vykonzultovať daný problém a tým pádom ho zväčša aj rýchlo vyriešiť. Môžeme už len konštatovať že IFJ si s projektom interpretu pre zvolený jazyk označenie TOP projektu na tejto škole právom zaslúži.

5 REFERENCIE

- [1] Prednášky, podklady k predmetu IFJ
<http://www.fit.vutbr.cz/study/courses/IFJ/public/>
- [2] Prof. Ing. Jan Maxmilián Honzík, CSc. Algoritmy IAL: Študijná opora [online].
Verzia 16-D. 2016-11-30 [cit. 2016-12-5]. Dostupné na adrese:
<https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IAL-IT/texts/Opora-IAL-2016-verze-16D.pdf>
- [3] Hashovacia funkcia [online]. [cit. 2016-12-5]. Dostupné na adrese:
<http://www.cse.yorku.ca/~oz/hash.html>
- [4] Quicksort algoritmus [online]. [cit. 2016-12-5] Dostupné na adrese:
<https://sk.wikipedia.org/wiki/Quicksort>

Príloha č.1: Diagram konečného automatu



Príloha č.2: LL gramatika

<CLASS-LIST> -> CLASS ID { <CLASS-ELEMENT> } <CLASS-LIST>
<CLASS-LIST> -> EPSILON
<CLASS-ELEMENT> -> STATIC <DECLARATION-ELEMENT> <CLASS-ELEMENT>
<CLASS-ELEMENT> -> EPSILON
<DECLARATION-ELEMENT> -> <PARAM> <DECLARATION>
<DECLARATION-ELEMENT> -> VOID ID <METHOD-DECLARATION>
<DECLARATION> -> ;
<DECLARATION> -> = <E>;
<DECLARATION> -> <METHOD-DECLARATION>
<METHOD-DECLARATION> -> (<PARAM-LIST>) { <METHOD-ELEMENT> }
<ID> -> ID
<ID> -> SPECIAL_ID
<DATA-TYPE> -> INT
<DATA-TYPE> -> DOUBLE
<DATA-TYPE> -> STRING
<DATA-TYPE> -> BOOLEAN
<PARAM-LIST> -> EPSILON
<PARAM-LIST> -> <PARAM> <NEXT-PARAM>
<NEXT-PARAM> -> ,<PARAM> <NEXT-PARAM>
<NEXT-PARAM> -> EPSILON
<PARAM> -> <DATA-TYPE> ID
<CALL-ASSIGN> -> (<PARAM-VALUE>)
<CALL-ASSIGN> -> = <E>
<PARAM-VALUE> -> EPSILON
<PARAM-VALUE> -> <E> <NEXT-PARAM-VALUE>
<NEXT-PARAM-VALUE> -> ,<E> <NEXT-PARAM-VALUE>
<NEXT-PARAM-VALUE> -> EPSILON
<VALUE> -> EPSILON
<VALUE> -> = <E>
<METHOD-ELEMENT> -> <PARAM> <VALUE>; <METHOD-ELEMENT>
<METHOD-ELEMENT> -> EPSILON
<METHOD-ELEMENT> -> <ELEMENT-LIST> <METHOD-ELEMENT>
<ELEMENT-LIST> -> <STATEMENT>
<ELEMENT-LIST> -> { <STATEMENT-LIST> }
<STATEMENT-LIST> -> <STATEMENT> <STATEMENT-LIST>
<STATEMENT-LIST> -> EPSILON
<STATEMENT-LIST> -> { <STATEMENT-LIST> } <STATEMENT-LIST>
<STATEMENT> -> ;
<STATEMENT> -> <ID> <CALL-ASSIGN>;
<STATEMENT> -> IF (<E>) <CONDITION-LIST> <ELSE>
<STATEMENT> -> WHILE (<E>) <CONDITION-LIST>

<STATEMENT> -> RETURN <RETURN-VALUE>;
 <RETURN-VALUE> -> EPSILON
 <RETURN-VALUE> -> <E>
 <ELSE> -> EPSILON
 <ELSE> -> ELSE <CONDITION-LIST>
 <CONDITION-LIST> -> {<STATEMENT-LIST>}
 <CONDITION-LIST> -> <STATEMENT>

Príloha č.3: Precedenčná tabuľka

	+	-	*	/	()	<	>	<=	>=	==	!=	&&		ID	LIT	\$!
+	>	>	<	<	<	>	>	>	<	>	>	>	>	>	<	<	>	<
-	>	>	<	<	<	>	>	>	>	>	>	>	>	>	<	<	>	<
*	>	>	>	>	<	>	>	>	>	>	>	>	>	>	<	<	>	<
/	>	>	>	>	<	>	>	>	>	>	>	>	>	>	<	<	>	<
(<	<	<	<	<	=	<	<	<	<	<	<	<	<	<	<	>	<
)	>	>	>	>		>	>	>	>	>	>	>	>	>				>
<	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	<	>	<
>	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	<	>	<
<=	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	<	>	<
>=	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	<	>	<
==	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	<	>	<
!=	<	<	<	<	<	>	>	>	>	>	>	>	>	>	<	<	>	<
&&	<	<	<	<	<	>	<	<	<	<	<	<	>	>	<	<	>	<
	<	<	<	<	<	>	<	<	<	<	<	<	>	>	<	<	>	<
ID	>	>	>	>		>	>	>	>	>	>	>	>	>			>	
LIT	>	>	>	>		>	>	>	>	>	>	>	>	>			>	
\$	<	<	<	<	<		<	<	<	<	<	<	<	<	<	<		<
!	>	>	>	>	<	>	>	>	>	>	>	>	>	>	<	<	>	<

$\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle - \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle / \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle < \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle > \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle \leq \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle \geq \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle \neq \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle == \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle \&\& \langle E \rangle$
 $\langle E \rangle \rightarrow \langle E \rangle \parallel \langle E \rangle$
 $\langle E \rangle \rightarrow (\langle E \rangle)$
 $\langle E \rangle \rightarrow !\langle E \rangle$
 $\langle E \rangle \rightarrow \text{ID}$
 $\langle E \rangle \rightarrow \text{SPECIAL_ID}$
 $\langle E \rangle \rightarrow \text{TRUE}$
 $\langle E \rangle \rightarrow \text{FALSE}$
 $\langle E \rangle \rightarrow \text{INT_LITERAL}$
 $\langle E \rangle \rightarrow \text{DOUBLE_LITERAL}$
 $\langle E \rangle \rightarrow \text{STRING_LITERAL}$