Dokumentácia projektu

IMPLEMENTÁCIA PREKLADAČA IMPERATÍVNEHO JAZYKA IFJ20

Tím 096, varianta II

Rozšírenie FUNEXP

2.12.2020

Členovia tímu:

- Veronika Vengerová (xvenge01) (vedúci) 25%
- Matej Viskupič (xvisku01) 25%
- Tomáš Kučma (xkucma00) 25%
- Martin Voščinár (xvosci00) 25%

I. OBSAH

I. OBSAH 2
II. ÚVOD 3
III. NÁVRH A IMPLEMENTÁCIA 3
1. Lexikálna analýza 4
2. Syntaktická analýza 4
3. Tabuľka symbolov 5
4. Sémantická analýza 5
5. Generovanie kódu 6
IV. ROZŠÍRENIE 6
V. PRÁCA V TÍME 7
Rozdelenie práce 7
VI. ZÁVER 7
VII. ZDROJE A TABUĽKY 7
Zdroje 7
(1) Konečný automat 8
(2) LL-gramatika 9
(3) LL-tabuľka 10
(4) Precedenčná tabuľka 11

II. ÚVOD

Táto dokumentácia približuje a vysvetľuje postup práce pri tvorbe prekladača jazyka IFJ20. Dokumentácia sa skladá z popisov jednotlivých častí prekladača, tabuliek a grafov pre lepšiu názornosť, a popisu práce v tíme.

III. NÁVRH A IMPLEMENTÁCIA

Produkt sa skladá z častí:

- Lexikálna analýza,
- Syntaktická analýza,
- Sémantická analýza,
- Tabuľka symbolov,
- Generovanie cieľového kódu.

Pričom sémantickú analýzu a generovanie cieľového kódu sme pri implementácií spojili a všetky časti medzi sebou navzájom komunikujú.

1. Lexikálna analýza

Lexikálna analýza je v logickej fáze štruktúry prekladača na prvom mieste, preto aj pri implementácií sme ňou začali.

Hlavnou úlohou lexikálneho analyzátora je rozdeliť vstup na základné bloky = lexikálne jednotky (lexémy), ktoré sú reprezentované tokenmi. Na získavanie tokenov využívame deterministický konečný automat ***(1)***.

V implementácií sme využili na vyjadrenie tokenu štruktúru so vstavaným zväzom na uloženie hodnôt tokenov. Token sa získava pomocou funkcie `get_token()` volaním v syntaktickej analýze. Na získanie tokenu potrebujeme ukazovateľ do vstupného súboru a parameter `eolF`, ktorý určuje, či je odriadkovanie ignorované (`EOL_IGNR`), alebo brané ako token (`EOL_RET`, pre odhalenie chyby v odriadkovaní). S takto získaným tokenom ďalej pracujú aj ostatné časti prekladača.

2. Syntaktická analýza

Syntaktická analýza sa stará o určenie správnosti alebo nesprávnosti vstupných údajov na základe gramatiky. Rozdelili sme ju do 2 častí: spracovanie jazykových konštrukcií (analýza zhora nadol pomocou LL gramatiky) a analýza výrazov (analýza zdola nahor pomocou precedenčnej syntaktickej analýzy).

Syntaktický analyzátor si pomocou funkcie `get_token()` získa token, skontroluje, či nejde o lexikálnu chybu a ak nie, tak pokračuje v syntaktickej analýze na základe pravidiel ***(2)***. Syntaktický analyzátor taktiež volá aj funkcie sémantickej analýzy, ktoré postupne vykonávajú sémantickú analýzu a generovanie kódu.

Pri návrhu pravidiel sme sa narazili na problém pri pravidle pre príkaz priradenia, kde identifikátor nasledujúci za '=' môže patriť výrazu alebo volaniu funkcie. Tento problém sme vyriešili získaním ďalšieho tokenu v prípade, že funkcia s daným identifikátorom nebola ešte definovaná. Okrem toho sme sa pokúsili dodržiavať LL(1) gramatiku. V pravidlách nám vystupuje aj token pre ukončovač riadku, no iba v miestach, kde je vyžadovaný jazykom IFJ20.

Kvôli jednoduchšej implementácii sme zvolili metódu rekurzívneho zostupu, pri ktorej je každý neterminál reprezentovaný vlastnou funkciou. V prípadoch, kedy syntaktický analyzátor narazí na výraz (EXPR) sa využíva analýza postavená na precedenčnej tabuľke ***(4)*** v ktorej sú definované všetky pravidlá na vyjadrenie správnosti výrazu. Pri precedenčnej analýze sme využívali dvojitý zoznam, ktorý nám simuluje zásobník, kvôli jednoduchšiemu prechádzaniu a hľadaniu terminálov. Na orientáciu v tabuľke získavame index tabuľky tokenu pričom tokeny na vstupe predstavujú index stĺpca a terminál na vrchu „zásobníku" index riadku. Rozoznávame tokeny: ' `+ - / * int float64 id > < >= <= != == ( ) string` ' ako ekvivalentné indexy, a všetky ostatné tokeny ako index '$'. Či je daný výraz syntakticky správne zisťujeme pomocou štruktúry „zásobníka", ktorý postupne napĺňame a redukujeme na základe pravidiel až do chvíle kým nezistíme chybu, alebo kým na jeho vrchu nezískame znaky '$' '$' a medzi nimi 1 neterminál, kedy vieme, že výraz mal správny formát.

3. Tabuľka symbolov

Tabuľka symbolov je implementovaná pomocou hashovacej tabuľky(rozsahov platnosti) a zásobníku týchto rozsahov platnosti.

Hashovacia tabuľka je implementovaná ako tabuľka s rozptýlenými položkami s explicitne zreťazenými synonymami. Na získanie kľúča do tabuľky s rozptýlenými položkami sme použili hashovaciu funkciu pre sdbm. Obsahuje jednotlivé prvky, ktorými sú buď premenné alebo funkcie so všetkými ich vlastnosťami.

Každý prvok obsahuje svoj názov, ktorý je použitý ako kľúč do hashovacej tabuľky, a tiež svoj typ, ktorý udáva či sa jedná o premennú alebo funkciu. Ak je prvok typu premenná, `attrType` obsahuje informáciu o type premennej. Prvky typu funkcia obsahujú dva jednosmerne viazané zoznamy, `argList` a `returnList` v ktorých sa nachádzajú informácie o typoch premenných, ktoré funkcia očakáva ako argumenty a jej návratových hodnôt. Prvky sa v týchto zoznamoch nachádzajú v poradí v akom sú očakávané na zásobníku.

Zásobník rozsahov platnosti predstavuje jednotlivé oddelenia, podľa toho kde v programe sa nachádzame. Je implementovaný ako jednosmerne viazaný zoznam, ktorého položky predstavujú jednotlivé hashovacie tabuľky s ich príslušným ID.

4. Sémantická analýza

Sémantická analýza má ako hlavnú úlohu zistiť sémantickú správnosť programu. Kontroluje typy premenných (zhoda typu operandov, správne volanie funkcií…) a deklarácií premenných (premenná ešte nie je deklarovaná, redeklarácia…). Sémantická analýza sa stará aj o kontrolu rozsahu premenných, pri čom využíva tabuľku symbolov a vytvára a ruší nové rozsahy platnosti pre funkcie.

Sémantický analyzátor je implementovaný s pomocou pomocnej premennej `firstPass`, ktorá je nastavená pri prvom prechádzaní programom ako pravda.

Počas prvého prechodu programom sa inicializujú zásobník rozsahov platnosti a zásobníky tokenov a typov a vkladajú sa prvky, listy argumentov a listy hodnôt na vrátenie na zásobníky. Pri prvom prechode sa taktiež generuje kód pre globálne premenné a zisťujú sa chyby typu: viacnásobné definície funkcie, definovanie parametru s rovnakým menom ako má funkcia...

Pri druhom prechode programom dochádza ku generovaní kódu, spracovávaniu rozsahov platnosti a k detekcií zvyšných sémantických chýb.

Na to, aby bol dvojitý prechod možný, sme vytvorili nový typ (presnejšie štruktúru) a niekoľko pomocných funkcií. Tie dokážu vstup načítať do reťazcu a v potrebnej miere napodobňujú prácu s typom `FILE *`, no navyše aj umožňujú návrat na začiatok vstupu (čo pri práci so stdin štandardné `FILE *` funkcie nepodporujú).

5. Generovanie kódu

Generovanie kódu IFJ20code je spojené so sémantickou analýzou. Kód sa ukladá do pomocnej premennej typu `char *` a na výstup sa vypíše až po úspešnom dokončení behu programu.

Pri generovaní kódu nevyužívame žiaden medzi krok, ako napríklad abstraktný syntaktický strom alebo trojadresný kód. Situácie kedy čisto sekvenčné spracovanie kódu nie je možné sme riešili pomocnými premennými a zásobníkmi. Používajú sa 2 zásobníky, zásobník `tStackToken` ktorý slúži na ukladanie ID tokenov, a zásobník `tStackType`, ktorý reprezentuje spracované ale ešte nevyužité výrazy. V niektorých prípadoch sa tiež využíva generovanie pomocných skokov priamo vo výslednom IFJ20code ktoré umožňujú zmeniť poradie vykonávania príkazov (tento postup sme aplikovali v priradení príkazu cyklu a pri definíciách vo vnútri príkazu cyklu). Kód pre vstavané funkcie je preddefinovaný a generuje sa vždy.

Každá funkcia má v generovanom kóde vlastný rámec. Návestia funkcií sú tvaru `názov_funkcie`. Premenné definujeme na lokálnom rámci v tvare `meno_premennej$ID_tabuľky_symbolov_premennej`. Pri návestiach podmieneného príkazu a príkazu cyklu používame na diferencovanie návestí názov funkcie a ID vrchnej tabuľky symbolov (napr. `$else$ID_vrchnej_tabuľky_symbolov$názov_funkcie`). Na riešenie definícií vo vnútri podmieneného príkazu existujú ešte návestia so značkou špeciálneho indexu `forDefIdx` alebo značkou názvu premennej ktorá sa práve definuje (použitej spolu s inými už spomenutými značkami).

Generovanie kódu výrazov prebieha priamo - pri prečítaní literálu či premennej sa generuje kód ktorý ich zatlačí na zásobník, pri aplikovaní pravidiel pre operácie sa generuje ekvivalentná zásobníková inštrukcia operácie. Kvôli neexistencii zásobníkovej inštrukcie pre zreťazenie, program obsahuje pomocné globálne premenné a zásobníkové zreťazenie simuluje viacerými inštrukciami. Pre relačné operátory “<=”, “>=”, “!=” neexistuje inštrukcia vôbec, ale opäť sa dajú simulovať existujúcimi inštrukciami a následnou negáciou. Informácia o tom, koľko výrazov je momentálne spracovaných a ešte nevyužitých, aj o ich typoch, uchovávame v typovom zásobníku.

IV. ROZŠÍRENIE

Z možných rozšírení, ktoré sme mohli implementovať sme si vybrali FUNEXP, no implementovali sme len jeho časť, konkrétne podporu pre odriadkovanie vo výrazoch a medzi parametrami funkcií, a taktiež podporu pre výrazy v parametroch pri volaní funkcie.

V. PRÁCA V TÍME

Vďaka rozsiahlosti projektu a spoločnej práci viacerých ľudí bolo neoddeliteľnou súčasťou prípravy stanovenie si pravidiel spoločnej spolupráce. Ako verzovací systém sme si určili Git a využívali sme súkromný repozitár na GitHube, čo nám umožňovalo paralelne pracovať na viacerých častiach projektu. Na komunikáciu sme používali prevažne Discord a Messenger.

Rozdelenie práce

- Veronika Vengerová: Lexikálna analýza, Syntaktická analýza výrazov, Dokumentácia
- Matej Viskupič: Tabuľka symbolov, Generovanie kódu
- Tomáš Kučma: Lexikálna analýza, Sémantická analýza, Generovanie kódu
- Martin Voščinár: Syntaktická analýza, Testovanie

VI. ZÁVER

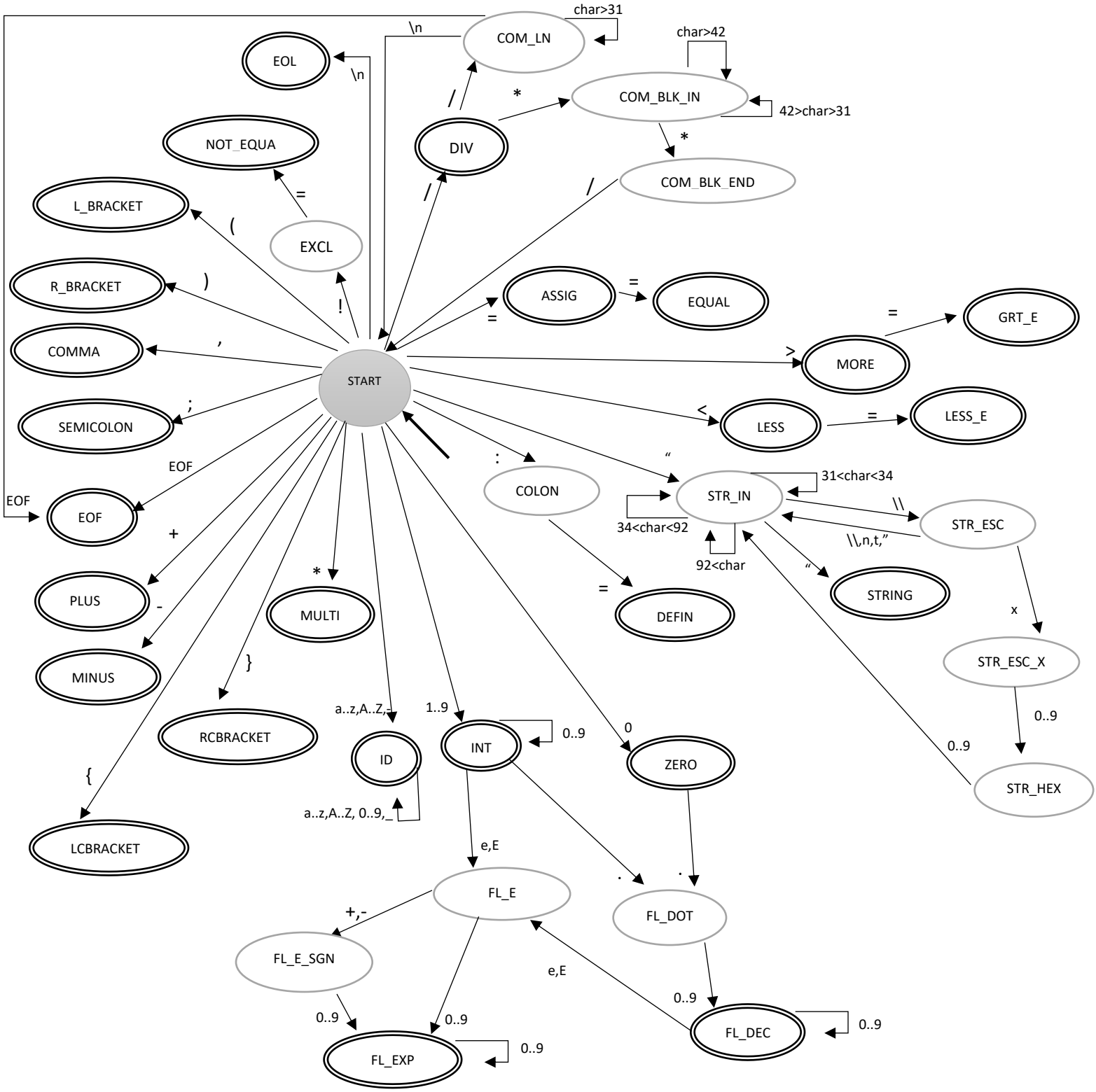
Pri zadaní projektu, na začiatku semestra, sme nedisponovali dostatočnými znalosťami na začatie tvorenia samotného prekladača preto sme tento čas využili na pripravenie si základov na budúcu prácu (založenie repozitára, pripravenie komunikačných kanálov, dohodnutie sa na používaných štandardoch...). S pribúdajúcimi znalosťami z predmetov IAL a IFJ sme postupne začali práce na jednotlivých častiach projektu a rozdeľovaní si práce. Tento postup nám umožnil získať plno nových skúseností z rôznych oblastí softwarového vývoja no najmä o práci samotných prekladačov. Správnosť týchto našich nových skúseností sme si overili pri pokusnom odovzdaní, vďaka ktorému sme boli schopní ešte väčšmi zdokonaliť našu implementáciu.

VII. ZDROJE A TABUĽKY

Zdroje

- Prednášky z predmetov IAL a IFJ
- http://www.cse.yorku.ca/~oz/hash.html
- https://man7.org/linux/man-pages/man3/asprintf.3.html

(1) Konečný automat



(1) Konečný automat

(2) LL-gramatika

1. PROG -> package id eol FUNC_LIST eof
2. FUNC_LIST -> ε
3. FUNC_LIST -> FUNC FUNC_NEXT
4. FUNC_NEXT -> ε
5. FUNC_NEXT -> eol FUNC FUNC_NEXT
6. FUNC -> func id (PARAM_LIST) RET_TYPES { eol ST_LIST }
7. PARAM_LIST -> ε
8. PARAM_LIST -> id TYPE PARAM_NEXT
9. PARAM_NEXT -> ε
10. PARAM_NEXT -> , id TYPE PARAM_NEXT
11. RET_TYPES -> ε
12. RET_TYPES -> (TYPE_LIST)
13. TYPE_LIST -> ε
14. TYPE_LIST -> TYPE TYPE_NEXT
15. TYPE_NEXT -> ε
16. TYPE_NEXT -> , TYPE TYPE_NEXT
17. TYPE -> int
18. TYPE -> float64
19. TYPE -> string
20. ST_LIST -> ε
21. ST_LIST -> STMT eol ST_LIST
22. STMT -> IF
23. STMT -> FOR
24. STMT -> return EXPR_LIST
25. STMT -> id CALL_OR_VAR
26. IF -> if EXPR { eol ST_LIST } else { eol ST_LIST }
27. FOR -> for FOR_DEF ; EXPR ; FOR_ASSIGN { eol ST_LIST }
28. FOR_DEF -> ε
29. FOR_DEF -> id VAR_DEF
30. FOR_ASSIGN -> ε
31. FOR_ASSIGN -> id VAR_NEXT VAR_ASSIGN
32. CALL_OR_VAR -> (EXPR_LIST)
33. CALL_OR_VAR -> VAR_DEF
34. CALL_OR_VAR -> VAR_NEXT VAR_ASSIGN
35. VAR_NEXT -> ε
36. VAR_NEXT -> , id VAR_NEXT
37. VAR_DEF -> := EXPR
38. VAR_ASSIGN -> = EXPR_OR_CALL
39. EXPR_OR_CALL -> id (EXPR_LIST)
40. EXPR_OR_CALL -> EXPR EXPR_NEXT
41. EXPR_LIST -> ε
42. EXPR_LIST -> EXPR EXPR_NEXT
43. EXPR_NEXT -> ε
44. EXPR_NEXT -> , EXPR EXPR_NEXT

(3) LL-tabuľka

	package	id	eol	eof	func	(	)	{	}	,	int	float64	string	return	if	else	for	;	:=	=
PROG	1																			
FUNC_LIST				2	3															
FUNC					6															
FUNC_NEXT			5	4																
PARAM_LIST		8					7													
RET_TYPES						12		11												
ST_LIST		21							20					21	21		21			
TYPE											17	18	19							
PARAM_NEXT							9			10										
TYPE_LIST							13				14	14	14							
TYPE_NEXT							15			16										
STMT		25												24	22		23			
IF															26					
FOR																	27			
EXPR_LIST			41				41													
CALL_OR_VAR						32				34									33	34
EXPR																				
FOR_DEF		29																28		
FOR_ASSIGN		31						30												
VAR_DEF																			37	
VAR_NEXT										36										35
VAR_ASSIGN																				38
EXPR_OR_CALL		39																		
EXPR_NEXT			43				43	43		44										

(3) LL-tabuľka

(4) Precedenčná tabuľka

	+	-	/	*	int	float64	id	>	<	>=	<=	!=	==	(	)	$	string
+	>	>	<	<	<	<	<	>	>	>	>	>	>	<	>	>	<
-	>	>	<	<	<	<	<	>	>	>	>	>	>	<	>	>	<
/	>	>	>	>	<	<	<	>	>	>	>	>	>	<	>	>	<
*	>	>	>	>	<	<	<	>	>	>	>	>	>	<	>	>	<
int	>	>	>	>	#	#	#	>	>	>	>	>	>	#	>	>	#
float64	>	>	>	>	#	#	#	>	>	>	>	>	>	#	>	>	#
id	>	>	>	>	#	#	#	>	>	>	>	>	>	#	>	>	#
>	<	<	<	<	<	<	<	#	#	#	#	#	#	<	>	>	<
<	<	<	<	<	<	<	<	#	#	#	#	#	#	<	>	>	<
>=	<	<	<	<	<	<	<	#	#	#	#	#	#	<	>	>	<
<=	<	<	<	<	<	<	<	#	#	#	#	#	#	<	>	>	<
!=	<	<	<	<	<	<	<	#	#	#	#	#	#	<	>	>	<
==	<	<	<	<	<	<	<	#	#	#	#	#	#	<	>	>	<
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	#	<
)	>	>	>	>	#	#	#	>	>	>	>	>	>	#	>	>	#
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	#	#	<
string	>	>	>	>	#	#	#	>	>	>	>	>	>	#	>	>	#

(4) Precedenčná tabuľka