VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

PROJEKTOVÁ DOKUMENTÁCIA

Implementácia prekladača imperatívneho jazyka IFJ20

Tým 098, varianta II

Harvan Mário	xharva03	25 %
Martiček Juraj	xmarti97	25 %
Belko Erik	xbelko02	25 %
Šlesár Michal	xslesa01	25 %

9. decembra 2020

Obsah

1 Úvod

Cieľom dokumentácie je opísanie implementácie a postupu pri riešení projektu v predmetoch IFJ a IAL. Základ projektu bola implementácia prekladača (program v jazyku C). Prekladač načíta zdrojový kód zo zdrojového imperatívneho jazyka IFJ20. Jazyk IFJ20 je podmnožinou jazyka GO. Prekladač preloží načítaný kód do cieľového jazyka IFJcode20 a vypíše na štandardný výstup alebo sa program ukončí s danou chybovou hláškou.

2 Práca v tíme

S prácou na prekladači sme začali pomerne skoro kvôli rozsiahlosti projektu. Prácu sme si delili rovnomerne počas riešenia projektu. Na úlohách sme väčšinou pracovali vo dvojiciach alebo jednotlivo. Využili sme princípy metodiky Scrum a prácu sme si rozdelili na týždňové sprinty.

2.1 Komunikácia v tíme

Keďže sme sa kvôli momentálnej situácii s pandémiou nemohli stretávať museli sme naplno využiť online komunikačné kanály. Na komunikáciu v tíme sme využili Discord, na ktorom sme si vytvorili server. Bol rozdelený na viacero kanálov a skladov materiálov pre jednotlivé časti projektu. Využili sme najmä voice kanály pre rýchlejšiu a pohotovejšiu komunikáciu pri riešení problémov.

Každý týždeň sme telefonovali, zhodnocovali vykonanú prácu a plánovali si úlohy na další týždeň. Na plánovanie úloh sme využili software Jira, a prepojili ho aj s Discordom.

2.2 Verzovací systém

Pre správu súborov sme sa rozhodli používať verzovací systém GitLab. GitLab nám umožnil pracovať na projekte súčasne v tzv. vetvách. Vyplatilo sa nám to aj ak sme sa chceli k niečomu vrátiť a použiť staršiu verziu. Pracovali sme vo vedlajších vetvách a až po schválení úprav ostatnými členmi týmu sme úpravy pridali do hlavnej vetvy vývoja.

2.3 Rozdelenie práce

2.3.1 Harvan Mário

- Vedenie tímu a organizácia práce
- Návrh a implementácia syntaktickej analýzy výrazov
- Návrh a implementácia sémantickej analýzy výrazov
- Implementácia zásobníku

2.3.2 Martiček Juraj

- Tvorba tabuľky pre precedenčnú analýzu
- Implementácia dátového typu Vector
- Tvorba knižnice pre error handling a chybové hlášky
- Tvorba generátora kódu

2.3.3 Belko Erik

- Tvorba pravidiel a tabuľky LL(1) gramatiky
- Návrh a implementácia syntaktickej analýzy (okrem výrazov)
- Návrh a implementácia sémantickej analýzy (okrem výrazov)
- Tvorba dokumentácie

2.3.4 Šlesár Michal

- Návrh a implementácia lexikálnej analýzy
- Návrh a implementácia syntaktickej analýzy (okrem výrazov)
- Návrh a implementácia sémantickej analýzy (okrem výrazov)
- Implementácia string knižnice

2.3.5 Spoločná práca

- Testovanie kódu
- Tvorba prezentácie

3 Návrh a implementácia

Štruktúru projektu, ktorá je popísaná v tejto časti, ako aj použité algoritmy sme zostavili podľa odporúčaní a návrhov z prednášok predmetov IFJ a IAL.

3.1 Lexikálna analýza

Pri tvorbe prekladača sme začali práve lexikálnou analýzou alebo inak povedané scannerom. Scanner načítava znaky zo zdrojového súboru, vyhodnocuje a prevádza ich na tokeny.

Scanner je implementovaný ako deterministický konečný automat. Hlavnou funkciou scanneru je funkcia `scanner_get_token`, ktorá spracováva načítané znaky a prevádza ich na štruktúru `token`. Štruktúra `token` sa skladá zo štruktúr `tokenType` pre typ tokenu a `tokenValue` pre hodnotu tokenu. Typmi tokenu môžu byť `EOF`, `EOL`, identifikátor, kľúčové slovo, prázdny token, čiarka, zátvorky, čísla, reťazce, operátory a daľšie znaky, ktoré sú povolené v jazyku IFJ20. Hodnota tokenu je `union`, a podla typu tokenu to môže byť reťazec, ak je typ tokenu identifikátor alebo reťazec, int64 alebo double, ak sa jedná o typ tokenu integer číslo alebo float číslo, a nakoniec typ klúčového slova ak bol typ tokenu kľúčové slovo.

Celý scanner sa skladá z dlhej série `if`-ov vo `while` cykle. Každý `if` zodpovedá jednému stavu v konečnom automate. Ak scanner práve nájde znak ktorý nezodpovedá jazyku IFJ20 vráti chybu 1. Funkcia `scanner_get_token` je ukončená úspešne ak je z načítaných znakov hotový jeden token. Scanner končí s načítavaním tokenov pokiaľ je ukončený chybou alebo ak už prečíta celý zdrojový súbor.

Scanner je implementovaný v súbore `scanner.c` a využíva hlavičkový súbor `scanner.h`.

3.2 Syntaktická analýza

Po implementácii lexikálnej analýzy sme začali pracovať na syntaktickej. Syntaktická analýza je jedna z najdôležitejších častí prekladača, keďže kontroluje syntax zdrojového kódu. Syntax je kontrolovaná na základe LL(1) gramatiky, ktorú sme si zostavili ešte pred implementáciou syntaktickej analýzy. Pre syntaktickú analýzu sme si vybrali spôsob rekurzívneho zostupu. Syntaktická analýza je spolu so sémantickou analýzou implementovaná v súbore `parser.c`. Parser prechádza zdrojový kód dvakrát, kvôli sémantickej analýze. Syntaktická analýza začína funkciou `parse` a postupne cez funkciu `ruleProgram` rekurzívne zostupuje podľa pravidiel LL(1) gramatiky a porovnáva podla nich tokeny. Tokeny získava pomocou funkcie `load_token` od scanneru alebo zo zásobníku ak sa tam nejaké nachádzajú. Zásobník sme pri implementácii použili kvôli pravidlám s epsilonom. Ak parser narazí na nejakú syntaktickú chybu počas rekurzívneho zostupu, prekladač sa ukončí s chybou 2. Parser využíva aj hlavičkový súbor `parser.h` v ktorom je okrem iného aj štruktúra `ParserData`, v ktorej sú všetky dáta pre syntaktickú aj sémantickú analýzu. Sú to napríklad `bool isFirstScan` pre zistenie, či sa práve jedná o prvý prechod alebo iné využívané štruktúry ako `Vector *scopes`, `htab_t *table`, `Stack *tokens` a tak ďalej. Osobitnou časťou je syntaktická analýza výrazov, ktorá je implementovaná v `expression.c` a popísaná v časti 3.4.

3.3 Sémantická analýza

Spolu so syntaktickou analýzou je v `parser.c` implementovaná aj sémantická analýza. Sémantická analýza kontroluje sémantické pravidlá podľa zadania a na to je potrebný dvojitý prechod zdrojového kódu. Parser prechádza zdrojový kód dvakrát, preto je potrebné vstup uložiť, my sme zvolili dynamické pole `dynamicArr`. Pri prvom prechode parser kontroluje identifikátory, ukladá si funkcie a následne pri druhom prechode sa skontroluje či sú všetky volané funkcie už definované, či neprichádza k redefinícii, alebo aj to či v zdrojovom kóde nechýba funkcia main. Sémantická analýza kontroluje okrem iného aj správne typy, počet a návratové hodnoty funkcii, to či sú pri priradeniach premenných premenné už vopred definované. Na ukladanie premenných sme využili systém vytvárania scopov(tabuľka symbolov), k ich zanorovaniu a prístupu k nim sme použili štruktúru `Vector`. Posledný prvok vo Vectore je scope v ktorom sa sémantická analýza práve nachádza, to pomáha sémantickej analýze zisťovať, či sú premenné definované v lokálnom alebo v niektorom z vonkajších scopov. Po prechode a kontrole daného scopu sa scope z Vectoru odstráni. Implementácia tabuliek symbolov je v `symtable.c` a popísaná v časti 4.1. Sémantická analýza je teda so syntaktickou implementovaná v rámci rekurzívneho zostupu. Sémantická analýza využíva pomocné funkcie z `semantic_analysis.h`, ktoré využíva aj sémantická analýza výrazov, ktorá je implementovaná v `expression.c` a popísaná v časti 3.4.

3.4 Syntaktická a sémantická analýza výrazov

Výrazy spracovávame pomocou precedenčnej analýzy. Keď parser narazí na miesto v kóde kde očakáva výraz, zavolá funkciu `expression`, ktorá sa nachádza v súbore `expression.c`.

Funkcia `expression` vracia parseru štruktúru `expResult`, ktorá obsahuje string `result` obsahujúci názov pomocnej premennej v ktorej je uložený výsledok výrazu, bool `isFunc` ak sme narazili na volanie funkcie a ďalšie pomocné premenné. Nespracované tokeny a špeciálny token `TOKEN_DELIMITER` (označuje znak „<“) sa v cykle ukladajú na zásobník `Stack`. Po načítaní tokenu sa zavolá funkcia `precedenceTable`, ktorá nám určuje aká operácia sa má vykonať (shift, reduce, equal). Pokiaľ sa jedná o operáciu reduce, zavolá sa funkcia `reduceByRule`, ktorá vyhodnotí či je výraz syntakticky správny. Po syntaktickej kontrole sa spustí semantická kontrola. Tá kontroluje či sú premenné definované v tabuľke symbolov, taktiež kontroluje dátové typy premenných. Po sémantickej kontrole sa volajú príslušné funkcie pre generovanie kódu. Každý výsledok takto spracovaného výrazu sa uloží do pomocnej premennej. Pokiaľ nastane situácia že sa na zásobníku nachádza iba token `TOKEN_EXPRESSION` a na vstupe je jeden z tokenov, ktoré ukončujú výraz, tak funckia končí, a vráti parseru meno premennej v ktorej je uložený výsledok výrazu.

Môže nastať špeciálny prípad, keď je namiesto výrazu v kóde volanie funkcie, ukončíme funkciu `expression` a nastavíme príznak že sa jedná o volanie funkcie, ktoré spracuje parser.

3.5 Generátor kódu

Pri generovaní výstupného trojadresného kódu je pripravený modul `codegen.c`. Z tohto modulu sú prístupné rôzne funkcie pre generovanie kódu. `parser.h` a `expression.h` volá tieto funkcie podľa daného kontextu.

Ako prvé sa zavolá funkcia `gen_init`, ktorá pripraví zásobníky pre počítanie vnorených `if` podmienok a `for` cyklov. Taktiež na výstup `stdout` vypíše preambulu pre IFJcode20, a deklaruje vstavané funkcie. Generátor kódu sa opiera o vlastnú funkciu `print_i`, pomocou ktorej na štandardný výstup tlačí kód vo forme finálneho reťazca po riadkoch. Každá inštrukcia má svoj ekvivalent vo forme funkcie, ktorú zavolá spomínaný parser a expression. Všetky ostatné premenné potrebné pre generovanie

kódu (ako `scopeVector`), sú predávané týmto funkciám v parametroch, codegen obohatí premenné a symboly o ich prefixy, a mená upraví tak, aby nekolidovali s ostatnými v inom scope.

Podmienky a cykly sú riešené viacerými funkciami ako ich začiatok, následne telo je doplnené jednotlivo inštrukciami, a potom je doplnená o koniec takejto funkcie (napr. pri cykle overenie podmienky, a jump naspäť na začiatok cyklu).

Tento imperatívny prístup nám poskytuje vysokú flexibilitu kódu nezávislého od daného kontextu tela funkcií, poprípade kombinácií rôznych štruktúr zadaného jazyka IFJ20.

4 Použité algoritmy a dátové štruktúry

4.1 Tabuľka symbolov

Na implementáciu tabuľky symbolov sme použili tabuľku s rozptýlenými hodnotami. Tabuľku sme implementovali ako pole viazaných zoznamov. To nám umožnilo mať teoreticky neobmedzenú veľkosť tabuľky. Každá položka tabuľky obsahuje dôležité informácie o premennej. Jej dátovy typ, hodnotu, či je konštanta, hodnotu (ak je konštanta) atď. Taktiež môže položka obsahovať informácie o funkcii, a to jej parametre, návratové typy atď.

Hashovaciu funkciu sme použili z literatúry `http://www.cse.yorku.ca/~oz/hash.html` ako vhodnú funkciu pre hashovanie stringov. Každá položka tabuľky obsahuje taktiež hashovací kľúč (meno premennej) a odkaz na ďalšiu položku. Implementovali sme základnú funkciu `htab_init` na inicializáciu prázdnej tabuľky. Funkciu `htab_insert` na vloženie nového prvku do tabuľky. Pokiaľ sa položka už v tabuľke nachádza, funkcia vráti odkaz na prázdnu položku. Vďaka tomu vieme overiť či sa nepokúšame premennú definovať druhý krát. Funkcia `htab_find` nám vráti odkaz na položku v tabuľke.

Jednu tabuľku symbolov používame pre ukladanie informácií o funkciách. Pomocou obslužných funkcií semantickej analýzy vytvoríme novú položku s menom funkcie, a nastavíme jej potrebné informácie ako počet a typ parametrov, návratové typy atď.

Následne vytvárame tabuľku symbolov pre každý scope premenných v programe. Tieto tabuľky si ukladáme do Vectorov. Do tabuľky ukladáme mená premenných, ich dátové typy a ďalšie dôležité informácie.

4.2 Dátový typ Vector

Vector v našom projekte je implementovaný ako dynamické pole, ktoré sa pri naplnení veľkosti zväčší na dvojnásobok svojej veľkosti. Všetky obslužné funkcie na prácu s týmto dátovým typom sú v module `vector.c`.

Pri volaní `vectorInit` sa vektor inicializuje na veľkosť `DEFAULT_VECTOR_SIZE`, veľkosť jednej položky v tejto štruktúre pozostáva z veľkosti `void *`, a teda je to pole ukazateľov na položky.

Modul pozostáva z viacerých obslužných funkcií pre prácu s touto dátovou štruktúrou. Obsahuje klasické funkcie pre prácu s poľom (`vectorGet`, `vectorLength`, `vectorInsert`, `vectorRemove`), ale taktiež operácie podobné zásobníku, ako `vectorPush` a `vectorPop`.

Po dokončení práce s vektorom sa volá `vectorFree`, ktorý dané alokované miesto v pamäti uvolní.

4.3 Dynamický reťazec

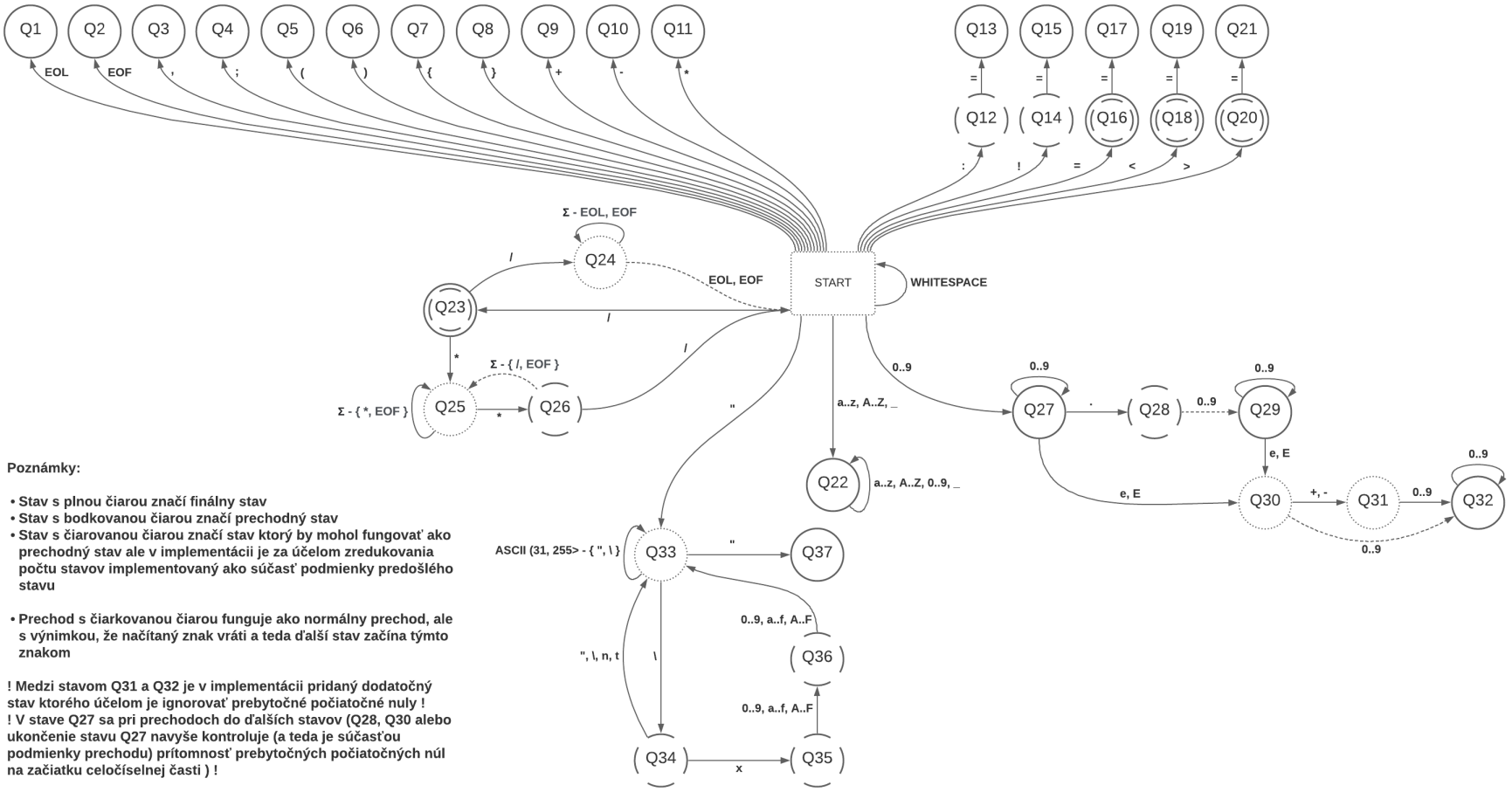
Pri implementácií prekladača sme využili aj dynamický reťazec. S dynamickým reťazcom pracuje string knižnica. Dynamický reťazec je dynamické pole znakov, slúžiace na ukladanie stringov. Pole sa alokuje vždy vo väčších blokoch pamäte, aby sa nemuselo realokovať tak často po každom znaku.

Implementácia sa nachádza v súbore `string.c`, ktorý využíva hlavičkový súbor `string.h`. Funkcia `string_init` vytvorí string. Do stringu je potom možné pridávať, na koniec stringu, ďalší znak alebo ďalší string pomocou funkcie `string_append_string` alebo `string_append_char`. V string knižnici je aj funkcia `string_compare` na porovnávanie stringov, a samozrejme funkcia `string_free` na uvoľňovanie zdrojov.

5 Záver

Projekt bol pre náš tím prínosný, a získali sme veľa skúseností. Zo začiatku sme nevedeli čo všetko nás čaká, ale keď sme postupne nabrali znalosti o tvorbe prekladača z prednášok, vedeli sme čo robiť a vývoj dopadol dobre. S prácou v tíme a komunikáciou sme nemali žiadne problémy. Projekt nám teda v konečnom dôsledku pomohol s pochopením preberanej látky v predmetoch IFJ a IAL.

A Diagram konečného automatu



Poznámky:

- **Stav s plnou čiarou značí finálny stav**
- **Stav s bodkovanou čiarou značí prechodný stav**
- **Stav s čiarovanou čiarou značí stav ktorý by mohol fungovať ako prechodný stav ale v implementácii je za účelom zredukovania počtu stavov implementovaný ako súčasť podmienky predošlého stavu**

- **Prechod s čiarkovanou čiarou funguje ako normálny prechod, ale s výnimkou, že načítaný znak vráti a teda ďalší stav začína týmto znakom**

! Medzi stavom Q31 a Q32 je v implementácii pridaný dodatočný stav ktorého účelom je ignorovať prebytočné počiatočné nuly !
! V stave Q27 sa pri prechodoch do ďalších stavov (Q28, Q30 alebo ukončenie stavu Q27 navyše kontroluje (a teda je súčasťou podmienky prechodu) prítomnosť prebytočných počiatočných núl na začiatku celočíselnej časti) !

B LL-Gramatika

1. <program> -> package id EOL <body> EOF
2. <body> -> <func_n>
3. <func_n> -> <func> <func_n>
4. <func_n> -> eps
5. <func> -> func id (<params>) <ret_types> { EOL <st_list> }
6. <params> -> id <type> <params_n>
7. <params> -> eps
8. <params_n> -> , id <type> <params_n>
9. <params_n> -> eps
10. <type> -> int
11. <type> -> float64
12. <type> -> string
13. <ret_types> -> (<type> <ret_types_n>)
14. <ret_types> -> eps
15. <ret_types_n> -> , <type> <ret_types_n>
16. <ret_types_n> -> eps
17. <st_list> -> <stat> EOL <st_list>
18. <st_list> -> eps
19. <stat> -> id <stat_body>
20. <stat_body> -> <id_n> = <expression> <expression_n>
21. <stat_body> -> := <expression>
22. <stat_body> -> (<call_params>)
23. <call_params> -> <values> <call_params_n>
24. <call_params> -> eps
25. <call_params_n> -> , <values> <call_params_n>
26. <call_params_n> -> eps
27. <values> -> value_int
28. <values> -> value_float64
29. <values> -> value_string
30. <values> -> id
31. <id_n> -> , id <id_n>
32. <id_n> -> eps
33. <expression_n> -> , <expression> <expression_n>
34. <expression_n> -> eps
35. <stat> -> if <expression> { EOL <st_list> } else { EOL <st_list> }
36. <stat> -> for <for_def> ; <expression> ; <for_assign> { EOL <st_list> }
37. <for_def> -> id := <expression>
38. <for_def> -> eps
39. <for_assign> -> id <id_n> = <expression> <expression_n>
40. <for_assign> -> eps
41. <stat> -> return <return_exp>
42. <return_exp> -> <expression> <expression_n>
43. <return_exp> -> eps
44. <expression> -> expression

C Pravidlá pre výrazy

```
E->id
E->VALUE_INT
E->VALUE_STRING
E->VALUE_FLOAT

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
```

D LL-Tabuľka

	package	id	eol	eof	func	(	)	{	}	,	int	float64	string	:=	=	value_int	value_float64	value_string	if	else	for	;	return	expression	$
<program>	1																								
<body>				2	2																				
<func_n>				4	3																				
<func>					5																				
<params>		6					7																		
<ret_types>						13		14																	
<st_list>		17							18										17		17		17		
<type>											10	11	12												
<params_n>							9			8															
<ret_types_n>							16			15															
<stat>		19																	35		36		41		
<stat_body>						22				20				21	20										
<id_n>										31					32										
<expression>																								44	
<expression_n>			34					34		33															
<call_params>		23					24									23	23	23							
<values>		30														27	28	29							
<call_params_n>							26			25															
<for_def>		37																				38			
<for_assign>		39						40																	
<return_exp>			43																					42	

E Precedenčná tabuľka

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