Dokumentácia tímového projektu IFJ 2020

Tím 051, Varianta I

08.12.2020

Vypracovali:

Peter Rúček	**xrucek00**
Rebeka Černianska	xcerni13
Marek Miček	xmicek08
Matej Jurík	xjurik12

Obsah

1 Úvod

Cieľom projektu bolo naprogramovať prekladač zdrojového jazyka IFJ20, ktorý je zjednodušenou verziou programovacieho jazyka GO. Výsledný program má formu konzolovej aplikácie, ktorá načíta zdrojový program zo štandardného vstupu, preloží ho do cieľového jazyka IFJcode20 a vypíše ho na štandardný výstup. V prípade výskytu akejkoľvek prekladovej chyby program vracia chybový kód.

2 Návrh a implementácia

Program pozostáva z viacerých modulov, ktoré spolu úzko spolupracujú. Tieto moduly sú predstavené a popísané v tejto kapitole.

2.1 Lexikálna analýza - Skener

Prvým krokom práce na projekte bolo implementovať lexikálnu analýzu. Jej úlohou je transformovať postupnosť znakov zo štandardného vstupu na takzvané tokeny. Platné tokeny nadobúdajú podľa daného vstupu vlastný typ a dáta.

Skener (modul implementujúci lexikálnu analýzu) je zostavený ako deterministický konečný automat. Špecificky, tento automat je implementovaný vo funkcii `get_next_token` ako nekonečný cyklus, v ktorom sa nachádza príkaz switch operujúci nad stavom automatu.

Pokiaľ znak zo vstupu vyhovuje danému stavu, v ktorom sa automat nachádza, pokračuje sa v načítavaní ďalších znakov, pokiaľ sa nenačíta jeden celý platný token. Potom z funkcie vraciame výsledný token na ďalšie spracovanie.

Pokiaľ by ale znak zo vstupu spôsobil prechod do stavu chyby (momentálny znak spolu s jeho predošlými tvoria neplatný token), funkcia navráti hodnotu symbolizujúcu neplatný token a zdrojový program nebude preložený.

Typ tokenu sa vyberá z množiny platných tokenov pre jayzk IFJ20. Špeciálny prípad vznikne, keď sa zo vstupu načíta neplatný token pre tento jazyk. Vtedy je typ tokenu neplatný token a jeho dáta sú definované ako `NULL`.

Stavy skenera nerozlišujú identifikátor od kľúčového slova, a preto zisťujeme či je identifikátor v skutočnosti kľúčové slovo tak, že ho pred návratom z funkcie porovnáme s kľúčovými slovami jazyka IFJ20 a po prípadnej zhode navrátime z funkcie token typu kľúčové slovo.

Dáta tokenu sa ukladajú ako dátový typ `union`, v ktorom je aktívna položka:

- Reťazec, pokiaľ je typ tokenu identifikátor alebo reťazec
- Vyčíslenie kľúčového slova, pokiaľ je token typu kľúčové slovo
- Celé číslo (integer), ak je token typu celé číslo
- Reálne číslo (double), ak je token typu reálne číslo

2.2 Syntaktická analýza

Základná syntaktická analýza (mimo výrazov) sa riadi podľa LL gramatiky a pravidlami LL tabuľky metódou rekurzívneho zostupu. Pre takmer každé pravidlo máme definovanú náležitú funkciu, ktorá ho aplikuje. Tokeny, ktoré syntaktická analýza spracováva sú zaobstarávané funkciou `get_next_token` zo skenera, ktorá je k tomu obalená makrom `GET_TOKEN`.

2.3 Syntaktická analýza výrazov

Analýza výrazov sa vykonáva separátne v module `expressions`. Implementovaná je pomocou precedenčnej tabuľky ako precedenčná analýza. Keďže majú operácie sčítania a odčítania rovnakú asociativitu a prioritu, vložili sme ich spolu do jedného riadku a stĺpca. To isté platí pre operácie súčinu a delenia, taktiež pre relačné operátory, ktoré sú v tabuľke symbolizované skratkou `RO`.

Pre správnu funkcionalitu precedenčnej analýzy využívame vhodnú dátovú štruktúru zásobník. Jednotlivé kroky pri analýze vyberáme podľa vrchného terminálu na zásobníku a momentálneho terminálu na vstupe. Vstup dostaneme získaním nasledujúceho tokenu a jeho nasledujúcou transformáciou na terminál (proces vykonáva funkcia `expr_input`).

2.4 Sémantická analýza

Tabuľky symbolov uchovávame v štruktúre `TNode`. Globálne prístupné sú dva typy tabuliek, a to globálna tabuľka pre identifikátory funkcií (`global_table`) a pole lokálnych tabuliek, ktoré zároveň simulujú zásobník rozsahu videnia (momentálny rozsah videnia je vždy na poslednom indexe tohto poľa) pre identifikátory premenných (`local_table`). Keďže tabuľky symbolov sú implementované ako binárne vyhľadávacie stromy, dáta funkcií a identifikátorov sa nachádzajú až v jednotlivých uzloch týchto stromov – štruktúry `TData`. V týchto štruktúrach uchovávame esenciálne dáta o definovaných identifikátoroch ako napríklad pravdivostné hodnoty určujúce či ide o identifikátor funkcie alebo premennej, dátový typ pre premennú alebo návratové typy pre funkciu. Tieto dáta slúžia na kontrolu existencie identifikátoru a prípadne jeho vlastností.

2.4.1 Sémantická analýza výrazov

Sémantická analýza výrazov prebieha postupne popri krokoch sémantickej analýzy popísanej vyššie. Dátový typ sa nastaví po prvom úspešnom načítaní neterminálu (premennej alebo primitívneho dátového typu) a nasledovne musia byť všetky ďalšie neterminály rovnakého typu. Kontroluje sa tu zároveň využívanie identifikátora `_` a delenie nulou.

2.5 Generovanie cieľového kódu

Popri syntaktickej analýze sa zároveň aj generuje cieľový kód IFJcode20. To je realizované pomocou dynamického reťazca, do ktorého pridávajú patričný kód funkcie z rozhrania generátora kódu. Pokiaľ všetky analýzy úspešne dokončia bez jedinej detekovanej chyby, kód je vygenerovaný na štandardný výstup.

2.5.1 Generovanie návestí

Ku generovaniu návestí dochádza v rôznych prípadoch, predovšetkým pri definícii funkcií, aby sme vedeli kam skočiť, ak dôjde k volaniu danej funkcie. Návestia sa silno využívajú aj pri podmienených príkazoch, kde sa musíme na základe pravdivostnej hodnoty príslušného výrazu rozhodnúť, ktorú časť podmieneného príkazu budeme vykonávať. Ďalším využitím je aj `for` cyklus, kde musíme určiť bloky vygenerovaného kódu na ktoré sa bude skákať, v rámci každej iterácie. Návestia zohrávali kľúčovú rolu aj pri výskyte príkazu `return` v zdrojovom kóde, či pri ošetrení definície premennej vo `for` cykle, aby sme sa vyhli opakovanej definícii premennej.

2.6 Generovanie funkcií

Na úvod treba podotknúť, že sme si všetky vstavané funkcie (okrem funkcie `print`) a ich telo vygenerovali zvlášť. Pri užívateľských funkciách sme postupovali tak, že pri definícii funkcie užívateľom sme si vygenerovali jej návestie, definovali na LF premenné do ktorých sa uloží návratová hodnota funkcie, pričom počet týchto premenných sa rovná počtu návratových hodnôt danej funkcie. Ak došlo k volaniu nejakej funkcie, tak si na TF definujeme toľko premenných, koľko parametrov sa posiela do danej funkcie, a do týchto premenných vložíme hodnoty týchto parametrov. Pri inštrukcii `PUSHFRAME` sa z TF stane LF a volaná funkcia môže s danými parametrami pracovať. Pokiaľ daná funkcia má nenulový počet návratových hodnôt, presunieme do premenných na LF hodnoty nachádzajúce sa v TF v premenných `%retval(i)`, kde i odpovedá indexu danej návratovej hodnoty, čím simulujeme priradenie návratovej hodnoty funkcie do premennej (respektíve návratových hodnôt funkcie do premenných). Funkciu `print` sme generovali iným spôsobom. Pokiaľ ju užívateľ zavolá, na TF sa definuje pomocná premenná do ktorej sa priradí hodnota termu odpovedajúca termu, ktorý posiela do tejto funkcie užívateľ a prostredníctvom inštrukcie `WRITE` vypíšeme hodnotu našej pomocnej premennej.

2.6.1 Generovanie výrazov

Počas syntaktickej analýzy výrazov sa podľa práve vykonávanej redukcie vygeneruje aj príslušná zásobníková inštrukcia nad jednotlivými neterminálmi (operandmi) daného výrazu. Výsledok výrazu sa ukladá do globálnej premennej `GF@expr_result`.

3 Špeciálne algoritmy a dátové štruktúry

Na implementáciu niektorých častí prekladača sme použili niekoľko špeciálnych dátových štruktúr. V tejto kapitole sú podrobne popísané.

3.1 Dynamický reťazec

Na prácu s tokenmi a generovaným kódom využívame funkcie modulu `dynamicstring`. Tento modul slúži na to, aby sme vedeli uložiť reťazec bez toho, aby sme dopredu poznali jeho dĺžku. Túto vlastnosť potrebujeme využívať často, nakoľko pri načítavaní tokenov nevieme dopredu povedať aký token bude na vstupe a taktiež pri generovaní, nevieme dopredu povedať, aký dlhý výsledný kód bude. Štruktúra, ktorá tento dynamický reťazec tvorí má niekoľko zložiek a to samotný reťazec, teda pole znakov, aktuálna dĺžka a koľko pamäte je pre reťazec vyhradenej. Funkcie ktoré sme v tomto module implementovali sú napríklad: pridávanie znakov, pridávanie celých reťazcov, presun reťazcov medzi tokenmi alebo porovnanie dynamických reťazcov.

3.2 Tabuľka symbolov – binárny vyhľadávací strom

Tabuľku symbolov sme implementovali podľa postupov preberaných na prednáškach predmetu IAL. Náš tým si vybral variantu 1, čiže variantu, kde sa tabuľka symbolov implementuje ako binárny vyhľadávací strom. Každý uzol tohto stromu je reprezentovaný svojím kľúčom (identifikátorom funkcie alebo premennej) na základe ktorého sa potom v strome vyhľadáva, svojou dátovou zložkou, ktorá uchováva metadáta o každej premennej/funkcii, ktorá je v tabuľke symbolov a ukazateľmi na ľavého a pravého syna. Identifikátor premennej má priradené dáta o tom či je už definovaný, či má priradený dátový typ a aký. Pokiaľ ide o funkciu, tak je definovaný aj počet parametrov, ich typy, počet návratových hodnôt a ich typy.

Nad tabuľkou symbolov môžeme vykonávať rôzne operácie. Inicializáciu tabuľky, vyhľadávanie v tabuľke podľa zadaného kľúča , vkladanie nového uzla so svojím kľúčom do tabuľky a samozrejme vymazanie celej tabuľky symbolov. Okrem toho sme tiež rozšírili funkcionalitu tabuľky symbolov o schopnosť vymazať práve jeden uzol so zadaným kľúčom. V prípade, že sa jedná o uzol, ktorý má oboch synov, si najskôr nájdeme najľavejší uzol pravého podstromu uzla, ktorý chceme vymazať, obsah tohto najľavejšieho uzla skopírujeme do uzla, ktorý chceme vymazať a daný najľavejší uzol vymažeme.

3.3 Zásobník

Pri precedenčnej syntaktickej analýze výrazov a pri generovaní cieľového kódu využívame zásobník. Implementovali sme základné rozhranie funkcií pracujúcich so zásobníkom ako napríklad: `push`, `pop`, `top`, a ďalšie. Samotné položky, ktoré do neho vkladáme sú celočíselného typu. Výrazy využívajú naviac vyčíslený typ `Terminal_type` pre čitateľnejšiu kontrolu typu terminálu. Pri generovaní je použitý na ukladanie indexov a postupné priraďovanie indexov návestiam a premenným.

4 Práca v tíme

Pred započatím práce na projekte sme si definovali štruktúru projektu a predbežný plán implementácie jeho jednotlivých častí. Prvým krokom, na ktorom sme sa podieľali všetci, bol návrh štruktúry repozitára a predpokladaného rozhrania základných modulov. Potom sme postupovali buď jednotlivo alebo vo dvojiciach na svojej pridelenej práci.

4.1 Rozdelenie práce

Prácu sme si rozdeľovali rovnomerne podľa preferencií a schopností členov. Tabuľka 4.1 popisuje rozdelenie práce medzi členov tímu 051.

Člen	*Práca*
Peter Rúček	Vedenie tímu, štruktúra projektu, návrh rozhrania modulov, lexikálna analýza, syntaktická analýza, syntaktická analýza výrazov, LL gramatika, generovanie
Rebeka Černianska	Dynamický reťazec, testovanie, lexikálna analýza, sémantická analýza premenných, generovanie
Marek Miček	Ošetrenie a hlásenie chýb, tabuľka symbolov, lexikálna analýza, sémantická analýza funkcií, generovanie
Matej Jurík	Korekcia kódu, debug a testovanie, lexikálna analýza, sémantická analýza výrazov, rozšírenie hlásenia chýb, dokumentácia

Tabuľka 1: Rozdelenie práce medzi členov tímu 051

4.2 Správa verzií projektu

Počas vývoja sme používali pre správu verzií Git, pričom sme využívali GitHub na hosting nášho hlavného repozitára. Malé zmeny sme po patričnom testovaní pridávali priamo na hlavnú vetvu, pričom väčšie zmeny, rozsiahle nové implementácie a opravy chýb sme najprv vložili na novú vetvu, nechali ostatných zhodnotiť našu prácu a následne sme ju zjednotili s hlavnou vetvou.

4.3 Komunikácia v tíme

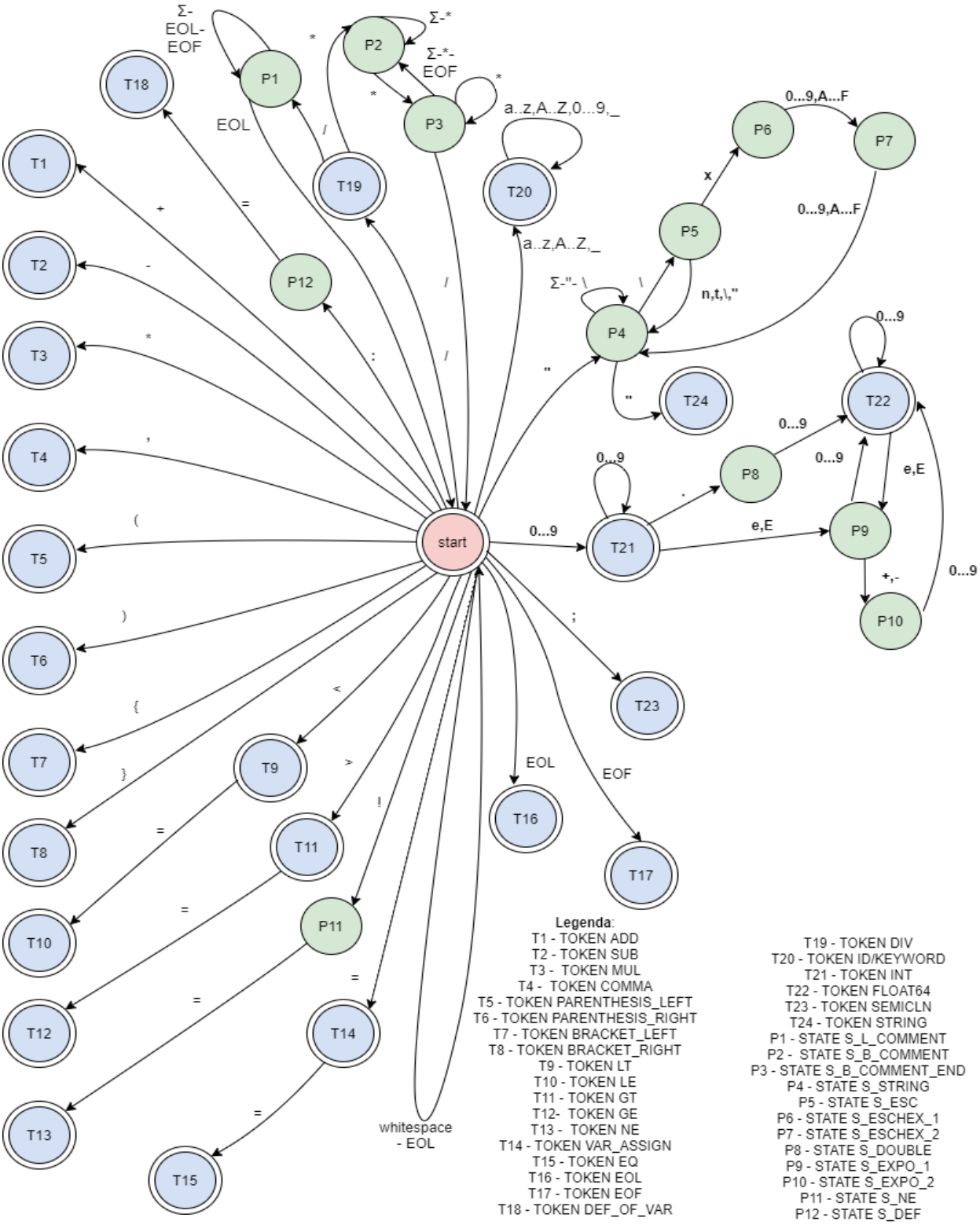
Pre komunikáciu sme používali aplikáciu MS Teams. Náš výber podmienil aj fakt, že túto aplikáciu používame aj na účasť na školských cvičeniach a prednáškach.

Riadili sme sa plánom, podľa ktorého si novinky a oznámenia zasielame pomocou aplikácie Messenger a väčšie problémy riešime cez MS Teams. Dohodli sme sa, že každý piatok a nedeľu sa zúčastníme na tímových stretnutiach, na ktorých sme riešili prácu za daný týždeň, návrhy na úpravu našich riešení a rozdelenie práce na ďalší týždeň.

5 Záver

S projektom takéhoto rozsahu sme veľa skúseností doteraz nemali, takže to v nás zo začiatku vyvolalo obavy a pochybnosti, či sme schopní takýto projekt skompletizovať. No zároveň to pre nás predstavovalo výzvu, ktorú sme boli odhodlaní prijať a popasovať sa s ňou. Prácu sme si snažili rozdeliť tak, aby sa každý člen aspoň nejakým spôsobom podieľal na jednotlivých častiach projektu. Tým pádom boli naše časti na sebe relatívne závislé, čo viedlo k tomu, že si ani jeden člen tímu nemohol dovoliť na svojej časti nepracovať. To jednoznačne pozitívne ovplyvnilo naše tempo implementácie a zároveň sa nám v prípade výskytu chýb podarilo rýchlejšie dôjsť k ich vyriešeniu keďže všetci členovia boli schopní porozumieť princípu fungovania danej časti, kde sa mohla chyba vyskytnúť. Situácia s pandémiou nám relatívne sťažila tvorbu tohto projektu, keďže sme boli izolovaní či už od seba, alebo v istej miere aj od školy. No vďaka pevnému odhodlaniu, pravidelnou komunikáciou medzi jednotlivými členmi a vedomím, že nám tvorba tohto projektu môže do budúcnosti veľa dať, ak to nezoberieme na ľahkú váhu, sme náš prekladač dokončili.

A Diagram konečného automatu použitého v lexikálnej analýze



Obrázok 1: Diagram deterministického automatu pre lexikálnu analýzu

B LL Gramatika

1. <program> -> <prolog> <func>
2. <prolog> -> package "main" EOL
3. <prolog> -> EOL <prolog>
4. <func> -> <func_header> <statement> <func>
5. <func> -> EOL <func>
6. <func> -> EOF

7. <func_header> -> func ID (<header_arg>) <header_ret> { EOL
8. <header_arg> -> eps
9. <header_arg> -> ID <data_type> <header_args>
10. <header_args> -> eps
11. <header_args> -> , ID <data_type> <header_args>

12. <header_ret> -> (<header_reta>)
13. <header_ret> -> eps
14. <header_reta> -> <data_type> <header_rets>
15. <header_reta> -> eps
16. <header_rets> -> , <data_type> <header_rets>
17. <header_rets> -> eps

18. <statement> -> EOL <statement>
19. <statement> -> <if_s> <statement>
20. <statement> -> <for_s> <statement>
21. <statement> -> <return_s> <statement>
22. <statement> -> ID <id>
23. <id> -> <def_of_var> <statement>
24. <id> -> <assignment_s> <statement>
25. <id> -> <fun_call> <statement>
26. <statement> -> }

27. <if_s> -> if <expression> { EOL <statement> else { EOL <statement>

28. <for_s> -> for <def> ; <expression> ; <ass> { EOL <statement>
29. <def> -> eps
30. <def> -> ID := <expression>
31. <ass> -> eps
32. <ass> -> ID <ids> = <ex>

33. <return_s> -> return <val> EOL
34. <val> -> eps
35. <val> -> <expression> <vals>
36. <vals> -> , <expression> <vals>
37. <vals> -> eps

38. <def_of_var> -> := <expression> EOL

39. <assignment_s> -> <ids> = <ex> EOL
40. <ids> -> , ID <ids>
41. <ids> -> eps
42. <ex> -> <expression> <exs>
43. <exs> -> , <expression> <exs>

44. <exs> -> eps

45. <fun_call> -> (<arg>) EOL
46. <arg> -> eps
47. <arg> -> <term> <args>
48. <args> -> , <term> <args>
49. <args> -> eps

50. <data_type> -> int
51. <data_type> -> string
52. <data_type> -> float64

53. <term> -> int_value
54. <term> -> string_value
55. <term> -> float_value
56. <term> -> ID

C LL Tabuľka

	package	func	if	for	return	ID	,	}	(	=	:=	int	string	float64	int_value	string_value	float_value	EOL	EOF ($)	EXPRESSION
program	1																			
prolog	2																	3		
func		4																5	6	
func_header		7																		
header_arg						9														
header_args							11													
header_ret									12											
header_reta												14	14	14						
header_rets							16													
statement			19	20	21	22		26										18		
if_s			27																	
for_s				28																
id							24		25	24	23									
def						30														
ass						32														
return_s					33															
val																				35
vals							36													
def_of_var											38									
assignment_s							39			39										
ids							40													
ex																				42
exs							43													
fun_call									45											
arg						47									47	47	47			
args							48													
data_type												50	51	52						
term						56									53	54	55			

Tabuľka 2: LL tabuľka pre syntaktickú analýzu

D Precedenčná tabuľka

Precedence table		Input token						
		(	)	+-	*/	RO	I	$
Terminal on stack	(	<	=	<	<	<	<	ERR
	)	ERR	>	>	>	>	ERR	>
	+-	<	>	>	<	>	<	>
	*/	<	>	>	>	>	<	>
	RO	<	>	<	<	ERR	<	>
	I	ERR	>	>	>	>	ERR	>
	$	<	ERR	<	<	<	<	ERR

RO = relation operator ERR = syntax error
I = ID, int_value, string_value, float_value
$ = end token / stack bottom

Obrázok 2: Precedenčná tabuľka pre precedenčnú syntaktickú analýzy výrazov