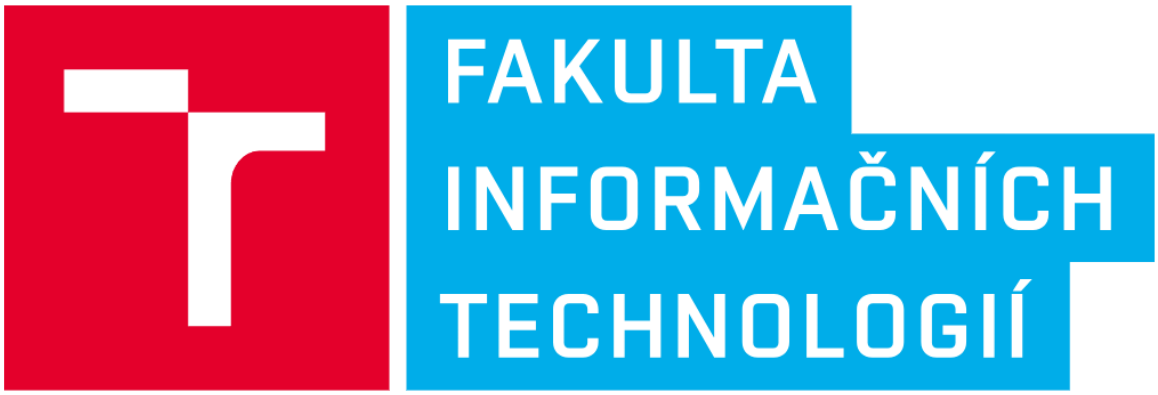


Dokumentácia k projektu z predmetu IFJ a IAL

Implementácia prekladača imperatívneho jazyka IFJ24

Tým xlucnyj00, varianta TRP-izp

Lůčný Jakub(vedúci)	xlucnyj00	25 %
Hrdlička Jakub	xhrdli18	25 %
Tydorová Rebeka	xtydor01	25 %
Ševčík Martin	xsevcim00	25 %

OBSAH

1. Úvod

Cieľom tohto projektu bolo navrhnúť a implementovať prekladač pre jazyk IFJ24, ktorý prekladá zdrojový kód napísaný v tomto jazyku do cieľového jazyka IFJcode24. Tento prekladač je zostavený v programovacom jazyku C, pričom implementácia zahŕňa všetky základné fázy prekladu: lexikálnu analýzu, syntaktickú analýzu, sémantickú analýzu a generovanie výsledného kódu. Projekt bol realizovaný ako tímová práca, pričom jednotliví členovia tímu sa podieľali na rôznych častiach implementácie.

2. Rozdelenie práce v tíme

2.1 Jakub Lůčný (xlucnyj00)

Spoluautor sémantickej analýzy, generovanie kódu a AST, testovanie

2.2 Jakub Hrdlička (xhrdli18)

Navrhnutie štruktúry pre token, pomocná funkcia pre roztriedenie kľúčových slov na určenie typu tokenu v lexikálnej analýze, pomocný kód string_buffer pre dynamické vytváranie stringov v lexikálnej analýze, vytvorenie gramatiky a na základe nej aj syntaktickej analýzy, vytvorenie symtable a k nej potrebné komponenty, ako je symtable_stack a hashtable.

2.3 Rebeka Tydorová (xtydor01)

Lexikálna analýza, spracovanie výrazov

2.4 Martin Ševčík (xsevcim00)

Spoluautor sémantickej analýzy, generovanie kódu a AST

3. Popis implementačného riešenia

Riešenie projektu sa skladá z viacerých častí. Ako taký „main“ celého projektu, ktorý všetko riadi by sa dala nazvať časť syntaktickej analýzy implementovaná v súbore syntakticka_analyza.c. Syntaktická analýza si postupne pýta tokeny od lexikálnej analýzy pomocou funkcie `get_token`, ktorá je implementovaná v súbore lexer.c. Táto časť programu postupne znak po znaku načíta vstup, a rozdeľuje jednotlivé slová a znaky na tokeny definované v súbore token.h. K svojmu chodu používa aj pomocné programy, ako napríklad str_buffer, pre buffer na ukladanie znakov za sebou na vytvorenie slov a viet alebo keyword_check na roztriedenie typov tokenov podľa definovaných kľúčových slov.

Syntaktická analýza potom tieto jednotlivé tokeny prechádza postupne, a kontroluje ich na prípadné syntaktické chyby. Syntaktická analýza taktiež jednotlivé tokeny po prečítaní ukladá do abstraktného syntaktického stromu, ďalej iba AST, implementovaného v súbore ast. V prípade výrazov si syntaktická analýza zavolá pomocnú funkciu `expression` implementovanú v súbore expressions. Od tohto momentu až po detekciu konca výrazu preberá kontrolu expression. Sám si pýta tokeny, kontroluje ich správnosť a generuje binárny strom daného výrazu. Súbor expressions na svoju funkciu používa aj podporné programy ako sú: btree, bts_stack a string_stack. Následne sa takto vytvorený strom prevedie na postfix a vloží sa do AST.

Syntaktická analýza potom po prečítaní všetkých tokenov zavolá funkcia semantic_analysis implementovaná v súbore sematics. Tu sa prekladaný kód prechádza ešte raz a kontroluje sa na sématické chyby. Prekladaný kód sa ale už nenahráva pomocou funkcie `get_token`, ale načíta sa z AST. Tento druhý prechod kódu zároveň ukladá jednotlivé funkcie, premenné a konštanty prekladaného kódu do tabuľky symbolov implementovanej v súbore symtable. Tabuľka symbolov sa skladá z hashtabuľky implementovanej v súbore hashtable a stacku implementovaného v súbore symtable_stack. Sématická analýza zároveň túto tabuľku symbolov využíva aj na kontrolu sématiky jednotlivých funkcií, premenných a konštánt. Nakoniec syntaktická analýza zavolá funkciu `generate_code` implementovanú v súbore codegen. Tento generátor postupne číta jednotlivé tokeny uložené v AST, a na základe nich generuje výsledný preložený kód.

4. Lexikálna analýza

Lexikálna analýza je implementovaná pomocou konečného automatu, ktorý je v jazyku C reprezentovaný pomocou riadiacej konštrukcie switch. Zo štandardného vstupu načítavame lexikálne jednotky, z ktorých následne vytvárame tokeny. Tokeny v sebe nesú typ a hodnotu, ktoré sú určené na základe aktuálneho stavu lexikálneho analyzátora. Celá lexikálna analýza je zabalená vo funkcii `get_token()`, ktorá na požiadanie vráti práve jeden token.

5. Syntaktická analýza

Syntaktická analýza by sa dala vnímať ako taký main celého prekladača. Syntaktická analýza si postupne načíta tokeny od lexikálnej analýzy a na základe predom definovanej

gramatiky overí syntaktickú správnosť vstupného programu. Správnosť vstupného programu sa overuje pomocou rekurzívneho zostupu, a teda sa program postupne zanára hlbšie a hlbšie do svojich funkcií, ktoré zrkadlia gramatiku. Keď príde na rad nejaký výraz, či už v priradení premennej, podmienke alebo return hodnote, syntaktická analýza predá kontrolu expressions pomocou funkcie `expression`. Syntaktickej analýze sa kontrola nad programom vráti na konci daného výrazu s prvým tokenom za výrazom. Popri syntaktickej analýze sa skontrolované tokeny ukladajú do abstraktného syntaktického stromu, ktorý je neskôr využitý pri sématickej kontrole a generovaní kódu. Po úplnom overení syntaktickej správnosti vstupného kódu syntaktická analýza zavolá ešte funkciu, ktorá spustí sématickú analýzu, následne funkciu, ktorá spustí generáciu kódu a úspešne ukončí program.

5.1 Precedenčná analýza výrazov

Výrazy spracovávame pomocou precedentnej syntaktickej analýzy, ktorá sa riadi precedentnou tabuľkou a súborom bezkontextových pravidiel definujúcich povolené operácie a ich správne kombinácie. Precedentná tabuľka sa používa pre porovnávanie operátora zo vstupu a operátora na vrchole zásobníka. Na základe ich vzťahu v tabuľke sa vykonávajú operácie shift – presun tokenu na zásobník a reduce – aplikácia pravidiel a redukcia výrazu. Pri redukcii sa tokeny a čiastočné výsledky zo zásobníka kombinujú do uzlov binárneho stromu, kde každý uzol reprezentuje operáciu a jeho podstromy zodpovedajú operandom.

6. Sémantická analýza

Sémantická analýza prechádza *Abstraktný syntaktický strom* (ďalej už len ako *AST*), ktorý je vytvorený počas syntaktickej analýzy. Pre priechod *AST* je použitá funkcia `next_node()`. Je realizovaná dvojpriechodovo, kedy pri prvom prechode sa uloží do tabuľky symbolov iba deklarácia všetkých užívateľských a vstavaných funkcií a pri druhom prechode sa do tabuľky symbolov ukladajú všetky premenné a konštanty, ktoré potom slúžia pre typovú kontrolu výrazov, priradenie, kontrolu využitia, modifikácie a pod.

Pri druhom prechode sa mimo vyššie zmienených akcií kontroluje aj referencie na nedefinovanú funkciu či premennú, zlý počet alebo typ argumentov pri volaní funkcie, zlý typ alebo nedovolené zahodenie návratovej hodnoty funkcie, chýbajúci alebo prebývajúci výraz v príkaze návratu z funkcie, redefinícia premennej alebo funkcie, možnosť odvodenia typu premennej pri jej definícii bez definovaného typu.

Pre prehľadnejšiu implementáciu bola tiež využitá jedna globálna premenná `current_function_name`, ktorá uchováva názov funkcie, ktorej definícia je práve sémanticky kontrolovaná a slúži na kontrolu typu vráteného výrazu alebo prípadne kontrolu neoprávneného chýbajúceho kľúčového slova return.

Oproti zadaniu naša sémantická analýza umožňuje typové konverzie aj premenných a výrazov s hodnotami neznámymi pri preklade, vďaka tomu, že typová konverzia prebieha dynamicky až počas generovania kódu.

7. Generovanie kódu

Generátor postupuje podobne ako sémantický analyzátor, prechádza *Abstraktný syntaktický strom* pomocou funkcie `next_node()` a na štandardný výstup vypisuje inštrukcie cieľového jazyka *IFJcode24*.

Na začiatku samotného generovania sa vygeneruje kód, ktorý vytvára pomocné premenné v globálnom rámci interpretu.

Po vygenerovaní celého kódu daného *Abstraktným syntaktickým stromom* sa vygeneruje kód vstavaných užívateľských funkcií.

Ku všetkým typovým konverziám dochádza až počas interpretácie. Tzn. počas generovania sa vygeneruje kód, ktorý skontroluje či sú operandy rovnakého typu (*int, float* – typové konverzie medzi inými typmi nie sú dovolené a sú odhalené počas sémantických kontrol) a pokiaľ sa typy nezhodujú, tak sa celočíselná hodnota pretypuje na *float*.

Pre väčšiu prehľadnosť kódu je tu použitá globálna premenná `current_function_name`, ktorá je využitá na generovanie kódu pre dynamickú kontrolu, ktorá zabráni chybe redefinície premennej pri definícii premennej napríklad uprostred *while* cyklu, ktorý sa prechádza viac ako raz.

8. Použité dátové štruktúry

8.1 Tabuľka symbolov

Tabuľka symbolov pozostáva z dvoch základných častí, hashtabuľky a zásobníku. Jednotlivé funkcie, premenné a konštanty sa teda vkladajú priamo do hashtabuľky, každá hashtabuľka sama o sebe reprezentuje jeden scope. V prípade vstúpenia do nového scopu sa aktuálna tabuľka uloží na vrchol zásobníku a vytvorí sa nová tabuľka pre nový scope.
V prípade vystúpenia zo scopu sa naopak zahodí aktuálna tabuľka a ako aktuálna tabuľka sa nastaví tabuľka z vrcholu zásobníku. Hashtabuľka aj zásobník použité v tabuľke symbolov sú dynamické, a teda nemôže prísť k ich pretečeniu.

8.2 Abstraktný syntaktický strom

Implementácia nášho *AST* sa počas vývoja niekoľkokrát menila, nakoniec skončil ako zdegenerovaný binárny strom, ktorý je tvorený iba jednou vetvou. Tzn. všetky uzly majú len pravého syna. Takže v podstate ide o *Jednostranne viazaný zoznam*.

8.3 Dynamický buffer

Dynamický buffer je využitý v lexikálnej analýze pre možnosť ukladania postupne načítaných znakov do slov alebo viet s predom neznámou dĺžkou.

9. Členenie implementačného riešenia do súborov

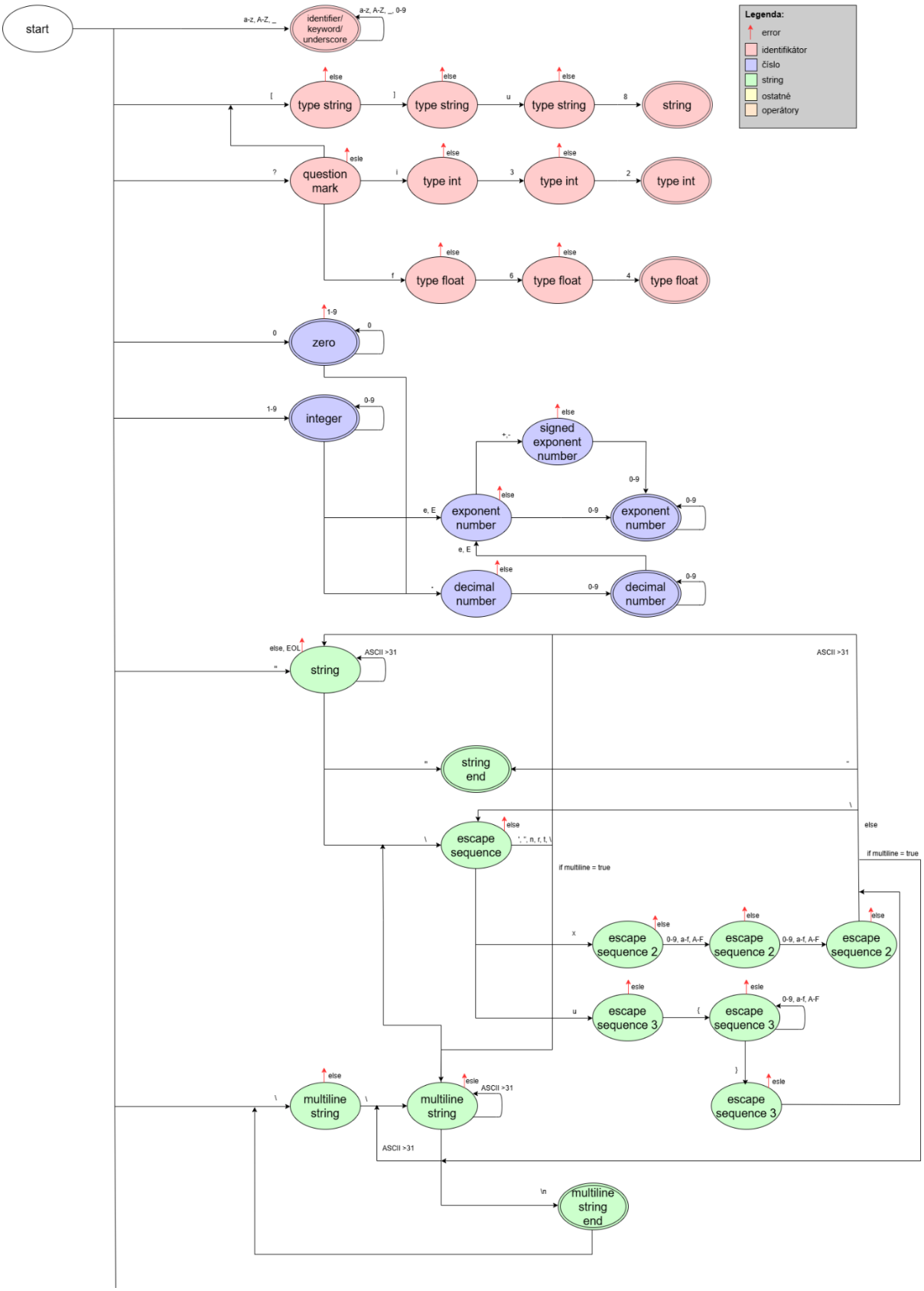
Lexikálny analyzátor	**lexer.c**, lexer.h, token.h
Pomocné štruktúry pre lexikálnu analýzu	str_buffer.c, str_buffer.h
Rozpoznanie kľúčových slov	**keyword_check.c**, keyword_check.h
Syntaktický analyzátor	**syntakticka_analyza.c**
Analýza výrazov	**expression.c**, expression.h
Pomocné štruktúry na analýzu výrazov	btree.c, btree.h, bts_stack.c, bts_stack.h, string_stack.c, string_stack.h
Sémantická analýza	**semantics.c**, semantics.h
Generovanie kódu	**codegen.c**, codegen.h
Tabuľka symbolov	**hashtable.c**, hashtable.h, symtable.c, symtable.h, symtable_stack.c, symtable_stack.h
Abstraktný syntaktický strom	**ast.c**, ast.h

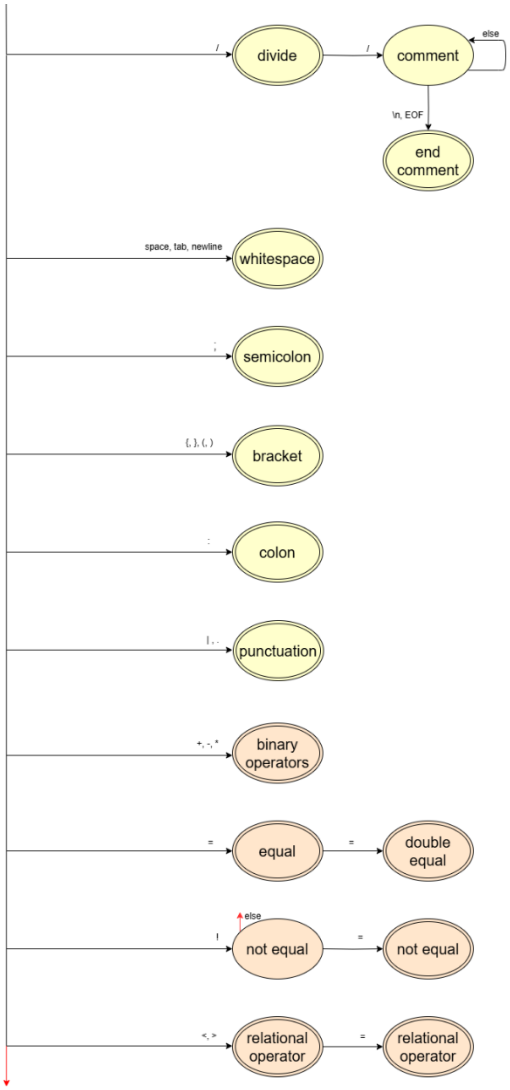
10. Záver

Tento projekt priniesol komplexný pohľad na implementáciu prekladača, od lexikálnej analýzy po generovanie výsledného kódu. Vďaka jasnému rozdeleniu úloh v tíme a dobre navrhnutým dátovým štruktúram sme úspešne vytvorili program, ktorý dokáže spracovať kód v jazyku IFJ24 a preložiť ho do formátu IFJcode24. Výsledkom je funkčný prekladač, ktorý spĺňa požiadavky zadania v rozsahu, aký sme boli schopní dosiahnuť v danom časovom rámci a s dostupnými zdrojmi. Hoci niektoré časti prekladača nefungujú na 100 %, výsledný produkt poskytuje pevný základ pre ďalšie rozšírenie a vylepšenie. Tento projekt nám umožnil získať cenné skúsenosti s návrhom a implementáciou väčších softvérových systémov a prehĺbil naše chápanie základných princípov programovacích jazykov a prekladačov.

11. Tabuľky a diagramy

11.1 Konečný automat pre lexikálnu analýzu




/
divide
/
comment
else
\n, EOF
end comment
space, tab, newline
whitespace
;
semicolon
{, }, (,)
bracket
:
colon
|, .
punctuation
+, -, *
binary operators
=
equal
=
double equal
else
!
not equal
=
not equal
<, >
relational operator
=
relational operator

11.2 LL-gramatika

(1) <CODE> -> <VARIABLE> ID <VARIABLE_CONTINUATION> ; <CODE>
(2) <CODE> -> ID <ID_DEFINING> ; <CODE>
(3) <CODE> -> if (<EXPRESSION>) <WHILE_IF_EXTENSION> { <CODE_SEQUENCE> } else { <CODE_SEQUENCE> } <CODE>
(4) <CODE> -> while (<EXPRESSION>) <WHILE_IF_EXTENSION> { <CODE_SEQUENCE> } <CODE>
(5) <CODE> -> pub fn ID (<PARAM>) <TYPE> { <CODE_SEQUENCE> } <CODE>
(6) <CODE> -> EOF

(7) <CODE_SEQUENCE> -> <VARIABLE> ID <VARIABLE_CONTINUATION> ; <CODE_SEQUENCE>
(8) <CODE_SEQUENCE> -> ID <ID_DEFINING> ; <CODE_SEQUENCE>
(9) <CODE_SEQUENCE> -> if (<EXPRESSION>) <WHILE_IF_EXTENSION> { <CODE_SEQUENCE> } else { <CODE_SEQUENCE> } <CODE_SEQUENCE>
(10) <CODE_SEQUENCE> -> while (<EXPRESSION>) <WHILE_IF_EXTENSION> { <CODE_SEQUENCE> } <CODE_SEQUENCE>
(11) <CODE_SEQUENCE> -> <FUNC_EXTENSION>
(12) <CODE_SEQUENCE> -> ε

(13) <VARIABLE_CONTINUATION> -> : <TYPE> = <NEXT_VARIABLE_CONTINUATION>
(14) <VARIABLE_CONTINUATION> -> = <NEXT_VARIABLE_CONTINUATION>

(15) <NEXT_VARIABLE_CONTINUATION> -> <EXPRESSION>
(16) <NEXT_VARIABLE_CONTINUATION> -> ID (<IN_PARAM>)

(17) <ID_DEFINING> -> = <NEXT_ID_DEFINING>
(18) <ID_DEFINING> -> (<IN_PARAM>)

(19) <NEXT_ID_DEFINING> -> ID (<IN_PARAM>)
(20) <NEXT_ID_DEFINING> -> <EXPRESSION>

(21) <WHILE_IF_EXTENSION> -> ε
(22) <WHILE_IF_EXTENSION> -> | ID |

(23) <FUNC_EXTENSION> -> ε
(24) <FUNC_EXTENSION> -> return <RETURN_VALUE> ;

(25) <RETURN_VALUE> -> ε
(26) <RETURN_VALUE> -> <EXPRESSION>

(27) <VARIABLE> -> var
(28) <VARIABLE> -> const

(29) <PARAM> -> ID : <TYPE> <PARAM_CONTINUATION>
(30) <PARAM> -> ε

(31) <PARAM_CONTINUATION> -> , <PARAM>
(32) <PARAM_CONTINUATION> -> ε

(33) <IN_PARAM> -> ID <IN_PARAM_CONTINUATION>
(34) <IN_PARAM> -> <TERM> <IN_PARAM_CONTINUATION>
(35) <IN_PARAM> -> ε

(36) <IN_PARAM_CONTINUATION> -> , <IN_PARAM>
(37) <IN_PARAM_CONTINUATION> -> ε

(38) <TYPE> -> i32
(39) <TYPE> -> f64
(40) <TYPE> -> []u8
(41) <TYPE> -> ?i32
(42) <TYPE> -> ?f64
(43) <TYPE> -> ?[]u8

(44) <TERM> -> string
(45) <TERM> -> int
(46) <TERM> -> float
(47) <TERM> -> NULL

11.3 LL-tabuľka

	var	const	pub fn	if	while	ID	;	,	:	=	|	(	)	{	}	return	expr	i32	?i32	f64	?f64	[]u8	?[]u8	string	int	float	NULL	EOF
<CODE>	1	1	5	3	4	2																						6
<CODE_SEQUENCE>	7	7		9	10	8									12	11												
<VARIABLE_CONTINUATION>									13	14																		
<NEXT_VARIABLE_CONTINUATION>						16											15											
<ID_DEFINING>										17		18																
<NEXT_ID_DEFINING>						19											20											
<WHILE_IF_EXTENSION>											22			21														
<FUNC_EXTENSION>															23	24												
<RETURN_VALUE>							25										26											
<VARIABLE>	27	28																										
<PARAM>						29							30															
<PARAM_CONTINUATION>								31					32															
<IN_PARAM>						33							35												34	34	34	34	
<IN_PARAM_CONTINUATION>								36					37															
<TYPE>																		38	41	39	42	40	43					
<TERM>																								44	45	46	47	

11.4 Precedenčná tabuľka

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