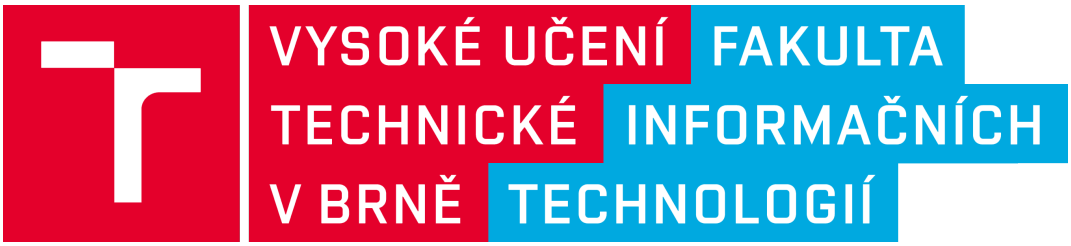


Dokumentace projektu IFJ

Implementace překladače imperativního jazyka IFJ24

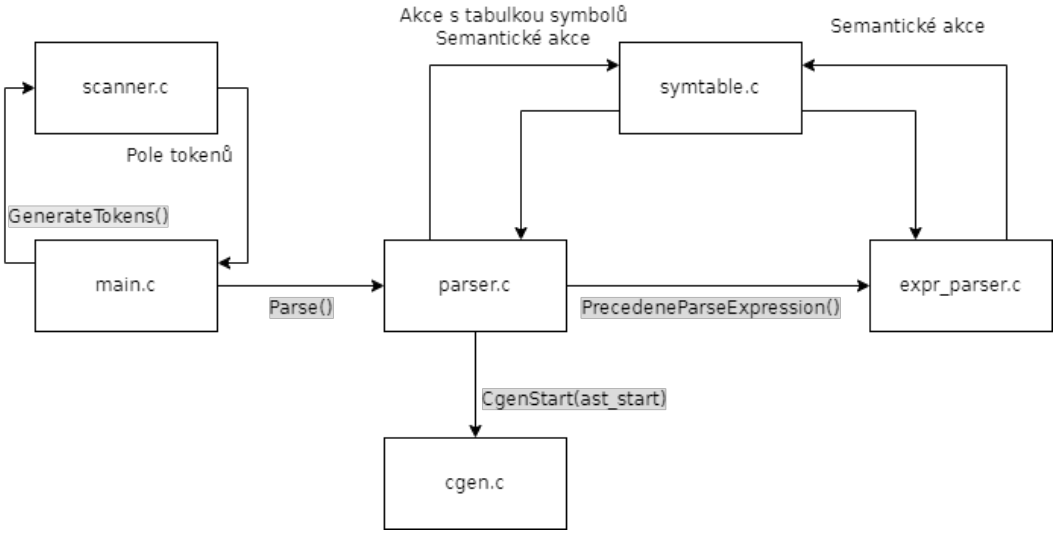
Tým xtikaia00, Varianta - TRP-izp

Jména řešitelů	Rozšíření
Dmitrii Ivanushkin xivanu00 25%	FUNEXP
Danil Tasmassys xtasmad00 25%	
Albert Tikaiev xtikaia00 25%	
Dias Tursynbayev xtursyd00 25%	

Obsah

1 Návrh

1.1 Propojení částí překladače



1.2 Stručný popis

Hlavní funkce nejprve požádá lexikální analyzátor vytvořit pole tokenů. Pak hlavní funkce zavolá syntaktický analyzátor, který z tohoto pole bude tokeny získávat a vytvářet abstraktní syntaktický strom. Když dojde k zpracování výrazu, syntaktický analyzátor zavolá analyzátor výrazů, který pokračuje dokud výraz neskončí. Syntaktický analyzátor a analyzátor výrazů používají a modifikují tabulku symbolů. Po dokončení syntaktický analyzátor zavolá generátor mezikódu, který průchodem AST vygeneruje IFJ24code a program se skončí.

Před implementací jednotlivých částí projektu byly nejprve vytvořeny DKA pro lexikální analýzu a LL-grammatika spolu s precedenční tabulkou pro syntaktickou analýzu. Navrhování struktury abstraktního syntaktického stromu probíhalo když většina projektu už byla rozpracovaná. Všechny definice AST jsou v souboru `ASTnodes.h`.

2 Implementace

2.1 Lexikální analýza

Soubory: `scanner.c/h`

Lexikální analýza je řízena stavovým automatem. Při lexikální analýze ukládá klíčová slova a identifikátory jako token typu `T_ID`. Porovnává řetězce s předdefinovaným seznamem klíčových slov, ve kterém je také definované klíčové slovo pro IFJ `K_IFJ`, a při shodě přiřazuje příslušné hodnoty, například `K_IF` nebo `K_CONST`. Pokud k shodě nedojde, token je označen jako `K_UNKNOWN` a považován za běžný identifikátor. Ostatní lexémy mají příslušející hodnotu typu tokenu ve výčtu `TokenType`.

Při zpracování víceřádkových řetězců skener spojuje řádky do jednoho celku, vytváří řetězec a ukládá ho do hodnoty tokenu včetně escape sekvencí. Dále se snaží rozpoznat co nejdelší sekvenci znaků odpovídající lexému a pro ukládání tokenů využívá dynamicky se rozšiřující pole, které má v odpovídající struktuře `Scanner`.

2.2 Tabulka symbolů

Soubory: `symtable.c/h`

Tabulka symbolů v překladači využívá strategii otevřeného adresování s lineárním probingem pro řešení kolizí hashování. Algoritmus při kolizí vyhledá přímo další volné místo v tabulce pomocí funkce `LinearProbe`, která vypočítá další potenciální index s konstantním krokem `C = 1`.

Hashovací tabulka má statickou velikost 3001 záznamů a využívá hashovací funkci DJB pro mapování hodnot na indexy. Zvolena velikost přináší několik zásadních výhod při organizaci dat. Prvočíselná velikost tabulky minimalizuje riziko kolizí a umožňuje mnohem rovnoměrnější rozptýlení klíčů v paměti.

Tato implementace je klíčová pro analýzu překladače. Existuje pouze jedna tabulka symbolů, která je zapouzdřena v souboru `symtable.c` a je globální. Je importována a inicializována před zahájením syntaktické analýzy. Je naplňována externě pomocí funkce `SymtableAdd`, která na základě klíče a typu symbolu inicializuje `data` a vrací ukazatel na tento prvek.

2.3 Syntaktická analýza

Soubory: `parser.c/h`

Syntaktická analýza založená na LL-gramatice byla implementována metodou rekurzívního sestupu. SA přijímá tokeny pomocí funkce `GetNextToken`, která postupně bere tokeny z pole struktury `Scanner`. Jelikož definice funkce nemusí lexikálně předcházet kódu její volání, implementovali jsme dvouprůchodovou SA.

Při prvním průchodu hledáme podstatné informace z definicí funkcí a naplňujeme jimi tabulku symbolů. Při druhém průchodu začíná tvorba AST, kde jsou uzly předány jako argumenty funkce, také přidáváme informace o definovaných proměnných v jednotlivých funkcích v příslušející položky

tabulky symbolů, které pak obdrží generátor kódu v uzlech AST. Funkce simulují nejdůležitější pravidla LL-gramatiky a na základě současného tokenu rozhodují jestli můžou pokračovat, zavolat další funkce pro simulace jiného pravidla a vytvořit nový uzel AST nebo ukončit program v případě chybné sekvence tokenů. Funkce `ParseExpr` zavolá funkce pro analýzu výrazů, která je založena na precedenční syntaktické analýze.

2.4 Analýza výrazů

Soubory: `expr_parser.c/h`

Analýza výrazů je implementována SA zdola nahoru a řízena precedenční tabulkou. Používá dva zásobníky: `pushdown`, ve kterém se provádí jednotlivé kroky precedenční SA a sémantické kontroly, `postfix` ve kterém je uložen výraz jako zásobníkový kód. Položky obou zásobníku jsou instance struktury `Expr`. Manipulace s terminály na `pushdown` se provádí pomocí funkce `GetTopTerminal` a `MakeTopTerminalLess`, další kroky precedenční analýzy používají atributy struktury `Expr`.

Všechny položky zásobníkového kódu se vkládají ve funkci `Reduce`, která je volána v případě že hodnota z precedenční tabulky je '>'. Na konci všechny položky z `pushdown` budou přesunuty do zásobníku `output_stack`, který je součástí uzlu `ASTExpression` , v opačném směru, vhodném pro generaci kódu a informace o tom, jestli položku s literálem je zapotřebí při generaci přetypovat. Přesunuty jsou pouze instance struktury `Expr_data`, která obsahuje informace o literálech nebo operacích a je atributem struktury `Expr`.

2.5 Sémantické akce

2.5.1 Využití a modifikace proměnných v `symtable.c/h`

Funkce `SymtableUpdate_isModified` aktualizuje stav modifikace proměnných, a spolu s tímto `SymtableUpdate_isUsed` aktualizuje stav využití proměnné nebo konstanty.

Funkce `SymtableEnterScope` a `SymtableLeaveScope` slouží pro kontrolu nevyužitých proměnných a konstant, chybějící modifikace u proměnných. `SymtableLeaveScope` smaže z TS položky proměnných a konstant, pokud už nejsou v rozsahu platnosti. Tyto manipulace s rozsahámi platnosti se používají nejčastěji ve funkci `ParseBody`.

2.5.2 Kontrola definicí v `symtable.c/h`

Funkce `SymtableAssertFunction` a `SymtableAssertVariable` umožňují získat informace o proměnných a funkcích a zároveň zkontrolovat, zda jsou definované.

2.5.3 Kontrola kompatibility datových typů v `expr_parser.c/h`

Funkce `Reduce` provádí sémantické kontroly při zpracování aritmetických a relačních operátorů odděleně, protože jejich sémantická pravidla se liší. Všechny potřebné informace o datových typech

jednotlivých částí výrazů jsou uloženy na zásobníku `pushdown` jako instance struktury `Expr` a během zpracování jsou vždy aktualizovány.

2.5.4 Ostatní v `parser.c/h`

Funkce `CheckParamTypes` slouží pro ověření datových typů parametrů funkcí, tato funkce využívá jinou funkci `CheckNullableTypes`, která provádí kontroly s datovými typy zahrnující null.

Ostatní sémantické kontroly se provádí v příslušejících funkcích syntaktické analýzy, jako například kompatibilita typu při přiřazení do proměnné nebo chybějící příkaz return ve funkci, atd.

2.6 Generace mezikódu

Soubory: `cgen.c/h`

První myšlenka byla jak zajistit, aby všechny skoky v konstrukcích jako if nebo while směřovaly na správné návěští. K tomu byla vytvořena globální proměnná, která se pokaždé zvyšuje. Toto neřešilo situace, kdy se uvnitř jednoho if nachází další dva if. Doplňující řešení bylo použití zásobníku, který zajistil, že číslo na vrcholu zásobníku odpovídá aktuálně zpracovávanému if.

Dalším problémem byla redeklarace proměnných při každé iteraci smyčky while. Řešení bylo spojeno s parserem. Nyní parser při vytváření AST také zaznamenává, které proměnné se v dané funkci vyskytují, a jejich deklarace probíhá pouze jednou – na začátku funkce.

Dále o vestavěných funkcích, kde pro každou z nich existuje odpovídající funkce v mezikódu, kromě funkce ifj.substring a ifj.strcmp. Funkce ifj.strcmp byla udělána jako porovnání dvou řetězců pomocí relačních operátorů a příslušných skoků na návěští s návratovou hodnotou. V ifj.substring byla použita konkatenace znaků z původního řetězce do výsledného řetězce. Při testování cyklu while byla odhalena jiná chyba. Pro každý výraz byla deklarována pomocná proměnná, a toto je chyba redeklaraci, pokud je uvnitř jednoho cyklu while další. K řešení byly využity pomocné proměnné umístěné v globálním rámci, a zároveň bylo upraveno jejich použití ve vestavěných funkcích.

Změna taky byla kvůli rozšíření FUNEXP. Museli jsme změnit způsob volání funkcí a zpracování výrazů. Pro správnou implementaci byly zavedeny pomocné parametry–indikátory funkcí. Protože když je funkce volána jako parametr jiné funkce, takové volání vyžaduje jiný postup.

2.7 Rozšíření FUNEXP

Pro použití funkcí ve výrazech přidali jsme `union` jako atribut struktury `Expr_data`, který buď obsahuje token nebo uzel `ASTFuncCall`. Pro použití výrazu jako parameter funkce uzel `ASTParamCall` obsahuje jiný uzel `ASTExpression`, který obsahuje tento výraz.

Výrazy v parametrech funkcí fungují částečně. Pokud výraz parametru funkce obsahuje také funkci, a ta obsahuje jako parametr například proměnnou, dojde k chybě interpretace. Kvůli tomu, že generátor mezikódu v okamžik, kdy se snaží předat proměnnou nemá k dispozici lokální rámec, ve kterém je tato proměnná definována.

3 Použité datové struktury a algoritmy

3.1 Zásobník

Soubory: `stack.c/h`

Datová struktura "zásobník"(stack) byla implementována na základě slajdů z kurzu IAL a obsahuje klasické pomocné funkce `InitStack`, `PushStack`, `PopStack`, `TopStack` a `IsEmptyStack`. Používá se pro generování kódu a v analýze výrazů. Je implementován jako spojený seznam, ve kterém jednotlivé prvky obsahují ukazatel na další prvek a ukazatel na data dopředu neznámého typu.

3.2 Dynamický řetězec

Soubory: `jm_string.c/h`

Tato datová struktura implementuje dynamický řetězec s jeho délkou a maximální kapacitou. Přidávání jednotlivých znaků zajišťuje funkce `PushChar`, která kontroluje kapacitu a v případě potřeby provádí realokaci paměti. Pro porovnávání řetězců slouží funkce `StringEquals`, která kontroluje délku a obsah řetězce. Uvolnění paměti je zajištěno funkcí `FreeString`, která bezpečně odstraní všechny alokované části řetězce.

3.3 Správa alokace paměti

Soubory: `memory.c/h`

Pro bezpečné manipulace s pamětí implementovali jsme funkce, které kontrolují paměťové chyby a uchovávají všechny alokované prostředky. Pro uchování alokovaných prostředků se používá dynamické pole. Pro bezpečné ukončení programu používáme funkci `InvokeExit`, která než se program ukončí pomocí funkce `exit` dealokuje všechny prostředky.

3.4 Dynamické pole

Soubory: `array.c/h`

Dynamické pole se hlavně používá pro správu paměti, a kvůli konfliktu mezi moduly paměti a dynamického pole jeho funkce nejsou paměťové bezpečné. Ale také se používá pro uchování seznamu definovaných proměnných ve funkci, kde jsou použity paměťové bezpečné funkce z `memory.c` jako `InvokeInitVarsArray` a `InvokeAddVarsArray`. Abychom vyhnuli duplikací jmen byla vytvořena funkce `ArrayFindStr`, která bude hledat právě řetězec v poli.

4 Práce v týmu

4.1 Vývojový cyklus a způsob práce v týmu

Na začátku semestru jsme se domluvili o rozdělení nejdůležitějších částí projektu, které jsou zvýrazněny v tabulce tučným písmem, nezvýrazněny jsou části, kde členové buď spolupracovali, nebo pouze opravovali. Návrh jednotlivých částí projektu probíhal na online schůzkách, další implementační detaily jsme řešili textovou komunikací. Jako verzovací systém jsme použili GitHub, ve kterém jsme využili možnost současné práce v jednotlivých větvích.

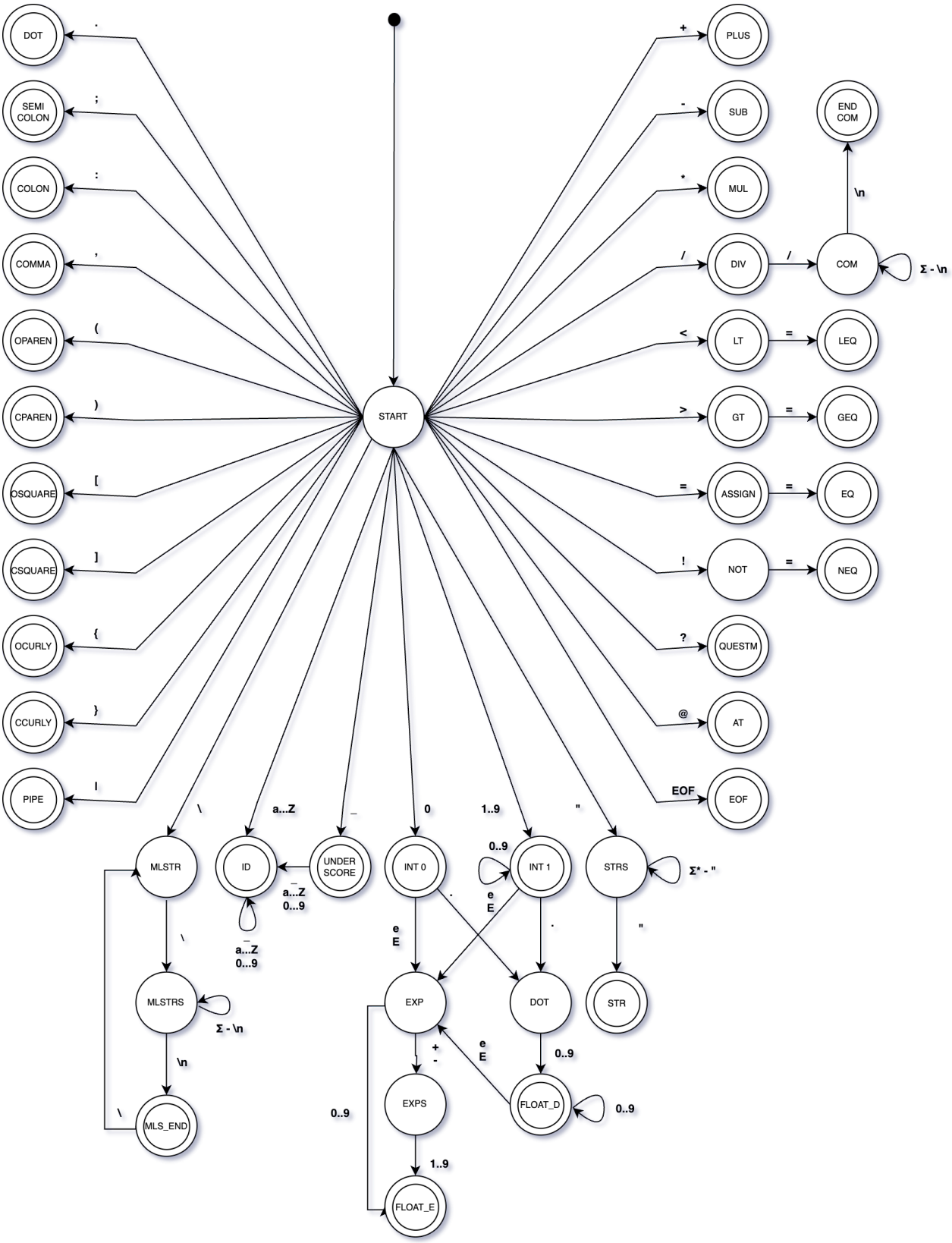
4.2 Testování

Každý člen týmu vytvořil sadu testů alespoň pro svoji implementovanou část a testy s použitím interpretu. Také téměř každý člen týmu vytvořil běžné kódy v jazyce IFJ24 pro testování celého projektu, což bylo velmi užitečné při hledání chyb v různých částech projektu.

4.3 Rozdělení práce v týmu

xivanu00	**Implementace tabulky symbolů**, vytvoření DKA, implementace zásobníku, lexikální analýza, analýza výrazů, sémantická analýza, testování
xtasmad00	**Generace mezikódu**, syntaktická analýza, vytvoření LL-gramatiky, vytvoření AST, testování
xtikaia00	**Syntaktická analýza**, vytvoření LL-gramatiky, implementace dynamického pole, analýza výrazů, sémantická analýza, vytvoření AST, testování
xtursyd00	**Lexikální analýza**, vytvoření DKA, implementace dynamického řetězce, testování

5 DKA


START
DOT
SEMI COLON
COLON
COMMA
OPAREN
CPAREN
OSQUARE
CSQUARE
OCURLY
CCURLY
PIPE
PLUS
SUB
MUL
DIV
LT
GT
ASSIGN
NOT
QUESTM
AT
EOF
END COM
COM
LEQ
GEQ
EQ
NEQ
MLSTR
MLSTRS
MLS_END
ID
UNDER SCORE
INT 0
INT 1
EXP
EXPS
FLOAT_D
FLOAT_E
DOT
STRS
STR
Σ - \n
Σ* - "
a...Z
0...9
0..9
1..9
\n
EOF

6 LL-gramatika

1. <PROLOG> -> const ifj = @ import ("ifj24.zig") ; <FUNC_DECL> <FUNC_DECL_NEXT>
2. <FUNC_DECL> -> pub fn <ID> (<FUNC_PARAMS>) <FUNC_TYPE> <BODY>
3. <FUNC_DECL_NEXT> -> <FUNC_DECL> <FUNC_DECL_NEXT>
4. <FUNC_DECL_NEXT> -> ε
5. <FUNC_PARAMS> -> <PARAM> <PARAM_NEXT>
6. <FUNC_PARAMS> -> ε
7. <PARAM> -> <ID> : <TYPE>
8. <PARAM_NEXT> -> , <PARAM> <PARAM_NEXT>
9. <PARAM_NEXT> -> ε
10. <BODY> -> { <STATEMENT_NEXT> }
11. <STATEMENT_NEXT> -> <STATEMENT> <STATEMENT_NEXT>
12. <STATEMENT_NEXT> -> ε
13. <STATEMENT> -> <ID> <FUNC_VAR>
14. <STATEMENT> -> <FUNC_CALL_IFJ> ;
15. <STATEMENT> -> <VAR_DECL>
16. <STATEMENT> -> <IF_STATEMENT>
17. <STATEMENT> -> <WHILE_STATEMENT>
18. <STATEMENT> -> <RETURN_STATEMENT>
19. <FUNC_VAR> -> <VAR_DEF>
20. <FUNC_VAR> -> <FUNC_CALL> ;
21. <FUNC_CALL> -> (<FUNC_CALL_PARAMS>)
22. <FUNC_CALL_IFJ> -> ifj . <ID> (<FUNC_CALL_PARAMS>)
23. <FUNC_CALL_PARAMS> -> <CALL_PARAM> <CALL_PARAM_NEXT>
24. <FUNC_CALL_PARAMS> -> ε
25. <CALL_PARAM_NEXT> -> , <CALL_PARAM> <CALL_PARAM_NEXT>
26. <CALL_PARAM_NEXT> -> ε
27. <CALL_PARAM> -> <EXPR>
28. <VAR_DECL> -> const <ID> <VAR_TYPE> <VAR_DEF>
29. <VAR_DECL> -> var <ID> <VAR_TYPE> <VAR_DEF>
30. <VAR_TYPE> -> ε
31. <VAR_TYPE> -> : <GEN_TYPE>
32. <VAR_DEF> -> = <EXPR> ;
33. <IF_STATEMENT> ->if (<EXPR>) <NOT_NULLABLE> <BODY> <ELSE_STATEMENT>
34. <ELSE_STATEMENT> -> else <BODY>
35. <WHILE_STATEMENT> -> while (<EXPR>) <NOT_NULLABLE> <BODY>
36. <NOT_NULLABLE> -> | <ID> |
37. <NOT_NULLABLE> -> ε
38. <RETURN_STATEMENT> -> return <RETURN_EXPR> ;
39. <RETURN_EXPR> -> ε
40. <RETURN_EXPR> -> <EXPR>

41. <FUNC_TYPE> -> void
42. <FUNC_TYPE> -> <GEN_TYPE>
43. <GEN_TYPE> -> <NULLABLE_PART> <TYPE>
44. <NULLABLE_PART> -> ?
45. <NULLABLE_PART> -> ε
46. <TYPE> -> i32
47. <TYPE> -> f64
48. <TYPE> -> []u8
49. <EXPR> -> <VAL>
50. <EXPR> -> (
51. <VAL> -> intval
52. <VAL> -> floatval
53. <VAL> -> <ID> <VAL_FUNC_CONTINUE>
54. <VAL> -> <FUNC_CALL_IFJ>
55. <VAL_FUNC_CONTINUE> -> <FUNC_CALL>
56. <VAL_FUNC_CONTINUE> -> ε
57. <VAL> -> stringval
58. <ID> -> id

Při přechodu pravidlem 49 nebo 50 syntaktický analýzator pokračuje pomocí precedenční syntaktické analýzy pro zpracování výrazů.

7 LL-tabulka

Nonterminal \ Terminal	$	const	var	pub	ifj	while	if	return	else	id	,	(	)	{	}	=	:	?	;	|	i32	f64	[]u8	void	intval	floatval	stringval
PROLOG		1																									
FUNC_DECL				2																							
FUNC_DECL_NEXT	4			3																							
FUNC_PARAMS										5			6														
PARAM										7																	
PARAM_NEXT											8		9														
BODY														10													
STATEMENT_NEXT		11	11		11	11	11	11		11					12												
STATEMENT		15	15		14	17	16	18		13																	
FUNC_VAR												20				19											
FUNC_CALL												21															
FUNC_CALL_IFJ					22																						
FUNC_CALL_PARAMS					23					23		23	24												23	23	23
CALL_PARAM_NEXT											25		26														
CALL_PARAM					27					27		27													27	27	27
VAR_DECL		28	29																								
VAR_TYPE																30	31										
VAR_DEF																32											
IF_STATEMENT							33																				
ELSE_STATEMENT									34																		
WHILE_STATEMENT						35																					
NOT_NULLABLE														37						36							
RETURN_STATEMENT								38																			
RETURN_EXPR					40					40		40							39						40	40	40
FUNC_TYPE																		42			42	42	42	41			
GEN_TYPE																		43			43	43	43				
NULLABLE_PART																		44			45	45	45				
TYPE																					46	47	48				
EXPR					49					49		50													49	49	49
VAL					54					53															51	52	57
VAL_FUNC_CONTINUE											56	55	56						56								
ID										58																	

8 Precedenční tabulka

	*/	+-	id	relational	(	)	$
*/	>	>	<	>	<	>	>
+-	<	>	<	>	<	>	>
id	>	>	-	>	-	>	>
relational	<	<	<	-	<	>	>
(	<	<	<	<	<	=	-
)	>	>	-	>	-	>	>
$	<	<	<	<	<	-	-

relational - značí operace == != < > <= >=
id - značí proměnné, konstanty, literály a volání funkcí(FUNEXP)