



Dokumentace

Implementace překladače imperativního jazyka IFJ23

Tým xeffen00, varianta vv-BVS

Podporovaná rozšíření: FUNEXP

vedoucí Marek Effenberger (xeffen00) 25 %

Samuel Hejníček (xhejni00) 25 %

Dominik Horut (xhorut01) 25 %

Adam Valík (xvalik05) 25 %

Brno, 6. prosince 2023

Obsah

1	Práce v týmu	2
1.1	Rozdělení práce mezi jednotlivé členy týmu	2
1.2	Organizace práce	3
2	Implementace překladače	3
2.1	Lexikální analýza	3
2.2	Syntaktická analýza	3
2.3	Sémantická analýza	4
2.4	Zpracování výrazů	4
2.5	Generování kódu	4
2.6	Tabulka symbolů	5
2.6.1	Uložení tabulek symbolů do hierarchického stromu	5
2.7	Datové struktury	6
2.7.1	Dynamický řetězec	6
2.7.2	Fronta	6
2.7.3	Zásobník tokenů	6
2.7.4	Jednosměrně vázaný seznam	6
2.7.5	Obousměrně vázaný seznam	6
2.8	Rozšíření	6
2.8.1	FUNEXP	6
3	Přílohy	7
3.1	Diagram konečného automatu	7
3.2	Legenda diagramu konečného automatu	8
3.3	LL gramatika	9
3.4	LL tabulka	11
3.5	Gramatika výrazů	12
3.6	Precedenční tabulka	12

1 Práce v týmu

1.1 Rozdělení práce mezi jednotlivé členy týmu

Marek Effenberger

- Návrh automatu pro lexikální analýzu
- Návrh LL gramatiky
- Návrh interní reprezentace kódu
- Implementace syntaktického a sémantického analyzátoru
- Generování cílového kódu

Adam Valík

- Návrh automatu pro lexikální analýzu
- Návrh LL gramatiky
- Implementace tabulky symbolů
- Implementace syntaktického a sémantického analyzátoru
- Generování cílového kódu

Dominik Horut

- Implementace lexikálního analyzátoru
- Návrh gramatiky výrazů a precedenční tabulky
- Implementace syntaktické a sémantické analýzy výrazů
- Generování cílového kódu výrazů
- Dokumentace

Samuel Hejníček

- Návrh automatu pro lexikální analýzu
- Implementace lexikálního analyzátoru
- Návrh gramatiky výrazů a precedenční tabulky
- Implementace syntaktické a sémantické analýzy výrazů
- Generování cílového kódu výrazů

1.2 Organizace práce

Před samotným začátkem vývoje projektu se tým rozdělil na dvě skupiny po dvou členech - první se zaměřila na tvorbu gramatiky, syntaktického a sémantického analyzátoru a následné generování cílového kódu, druhá nejprve implementovala lexikální analyzátor a poté syntaktickou analýzu výrazů a příslušné generování cílového kódu přes datový zásobník. Při vývoji byl použit verzovací systém Git a komunikace probíhala skrze platformu Discord, spoléhali jsme se však na osobní setkání členů týmu.

2 Implementace překladače

Princip implementace překladače stojí na syntaxí řízeném překladu s jednopřechodovou analýzou zdrojového kódu. Za běhu analýzy se vytváří seznam instrukcí kódu IFJcode23, který je po ukončení běhu rekurzivního sestupu vytisknut na standardní výstup. Pro interní reprezentaci hierarchické struktury kódu je použit strom s jednotlivými tabulkami symbolů.

2.1 Lexikální analýza

Lexikální analyzátor je implementován v souboru `scanner.c` a skládá se ze čtyř pomocných funkcí a hlavní funkce `get_me_token()`, která implementuje jeho vnitřní logiku. Dochází k načítání znaků ze standardního vstupu a na základě toho k přechodu do odpovídajícího stavu. Funkce načítá znaky do okamžiku, kdy se automat dostane do koncového stavu. V takovémto případě je navržena hodnota typu `token_t`, která obsahuje důležité atributy získaného tokenu.

Pokud je načten identifikátor, zjišťuje se, zda se nejedná o klíčové slovo. U víceřádkových řetězců se kontroluje správné odsazení všech řádků řetězce vůči ukončujícímu řádku. V případě, že sekvence znaků nevede ke koncovému stavu, dojde k lexikální chybě. Komentáře, stejně jako bílé znaky, jsou ignorovány. U vnořených komentářů je pak navíc kontrolován odpovídající počet otevíracích (`/`*) a uzavíracích (`*/`) znaků.

Datový typ `token_t` reprezentuje token načtený lexikálním analyzátozem a je implementován v souboru `scanner.h`. Obsahuje mimo jiné datový typ tokenu a jeho hodnotu.

2.2 Syntaktická analýza

Pro syntaktickou analýzu jsme zvolili metodu rekurzivního sestupu. Na základě LL gramatiky jsme vytvořili funkce v souboru `parser.c` reprezentující určité podmnožiny gramatiky. Analyzátor začíná funkcí `parser_parse_please()`, která inicializuje potřebné pomocné datové struktury `instruction_list`, `callee_list`, `queue_t` a `forest_node`.

Zavoláním funkce `prog()` začne syntaktická a sémantická analýza vstupního kódu. Analyzátor využívá funkci `peek()`, která dostane další token od lexikálního analyzátoru pro predikci postupu v rekurzivním sestupu a uloží jej do ukazatele na token `token_buffer`. Současně je využívána funkce `get_next_token()`, pomocí které se načte aktuální token (z `token_buffer` nebo přímo od lexikálního analyzátoru) a následně na základě jeho typu se provede přechod mezi funkcemi reprezentujícími syntaktická pravidla dána gramatikou. Je-li načten neočekávaný token, dojde k syntaktické chybě.

2.3 Sémantická analýza

Sémantická analýza probíhá souběžně se syntaktickou v souboru `parser.c`. Při průchodu syntaktickým analyzátozem využíváme pomocných datových struktur pro uchovávání dat, která se po konci hlavního běhu syntaktického analyzátoru vyhodnocují.

Mezi hlavní funkce sémantické analýzy řadíme `callee_validation()`, která porovnává nahromážděná data o veškerých volání funkcí ze vstupního kódu, následně porovnává správné jméno, počet argumentů a typovou kompatibilitu návratové funkce a předávaných argumentů s uchovanými daty v příslušném uzlu definice funkce ve `forest_node`.

Následně pomocí funkce `return_logic_validation()` je rekurzivně procházen hierarchický strom a ověřeno, zda-li je v jednotlivých blocích ve funkci klíčové slovo `return` užito na správném místě. Zbytek sémantické analýzy probíhá přímo v jednotlivých funkcích rekurzivního průchodu.

2.4 Zpracování výrazů

Zpracování výrazů je implementováno v souboru `expression_parser.c` a je voláno syntaktickým analyzátozem skrze funkci `call_expr_parser()` pokaždé, když je potřeba vyhodnotit výraz. Vyhodnocení pak probíhá pomocí precedenční syntaktické analýzy. K tomu využívá zásobník tokenů, kde ihned po inicializaci vloží terminál `$` a pomocí precedenční tabulky a gramatiky výrazů celý výraz vyhodnocuje, dokud na zásobníku nezůstane pouze terminál `$` a výsledek výrazu.

V rámci sémantické analýzy je výsledek podroben kontrole, zda odpovídá očekávanému datovému typu, který byl předán pomocí parametru funkce. Během vyhodnocování výrazu navíc probíhá sémantická kontrola typů pomocí funkce `check_types()`.

2.5 Generování kódu

Během syntaktické analýzy programu je v jednotlivých funkcích syntaktického analyzátoru volána funkce `inst_init()`, která vkládá do obousměrně vázaného seznamu jednotlivé instrukce v odpovídajícím pořadí. Výsledný generovaný kód dělíme na tři druhy rámců. Hlavní rámec reprezentující globální tělo vstupního programu, lokální rámce reprezentující funkce a dočasný rámec pro předávání argumentů a výsledné hodnoty.

Pro dosažení korektního výsledku generujeme unikátní jména pro jednotlivé proměnné, která korelují s jejich rámcovou reprezentací ve struktuře `forest_node`. Instrukce jsou ukládány především kvůli cyklu `while`. Pomocí funkce `inst_list_search_while()` tak lze při deklaraci proměnné vyhledat aktuálně nejvnější návěští vnořených cyklů a posunout deklaraci před něj. Obdobně řešíme smazání instrukce ukládání návratové hodnoty funkce u funkcí bez návratové hodnoty.

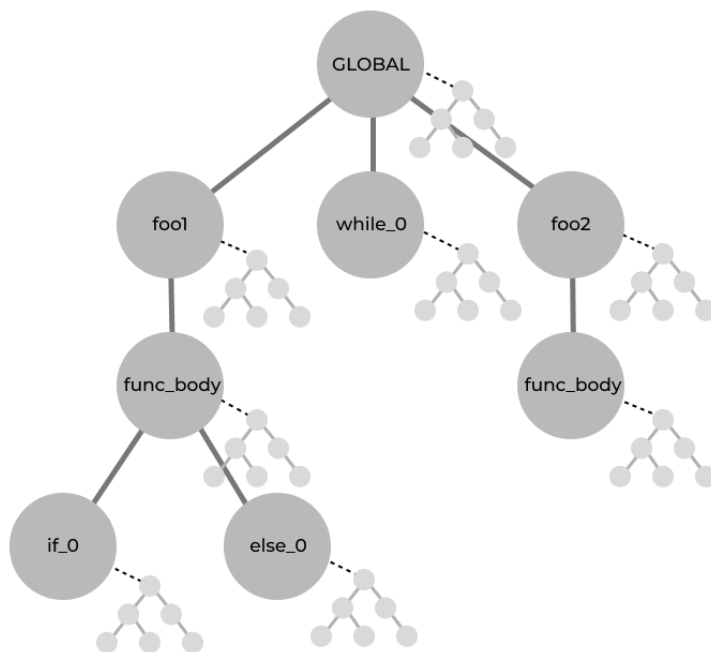
2.6 Tabulka symbolů

Dle varianty zadání jsme implementovali tabulku symbolů v souboru `symtable.c` pomocí výškově vyváženého binárního vyhledávacího stromu. Pro tento účel byly vytvořeny struktury `sym_data` obsahující všechna potřebná data symbolu a `AVL_tree` reprezentující samotný symbol jakožto uzel v binárním stromě.

Tato implementovaná struktura umožňuje ukládat informace o symbolech, jako jsou proměnné, funkce a parametry, do AVL stromu. Každý uzel stromu obsahuje klíč reprezentující jméno symbolu a ukazatel na strukturu `sym_data`, která uchovává specifické informace o daném symbolu potřebné ke správné interní reprezentaci kódu. Výška uzlů stromu je udržována pro zachování vyváženosti stromu a využití efektivních vyhledávacích operací. Tato implementace umožňuje rychlé vyhledávání, aktualizaci a vkládání symbolů do tabulky symbolů.

2.6.1 Uložení tabulek symbolů do hierarchického stromu

V souboru `forest.c` se nachází implementace uložení tabulek symbolů. Tato implementace reprezentuje hierarchickou strukturu kódu pomocí stromu, v němž si každý uzel `forest_node` uchovává ukazatele na svůj rodičovský uzel a na pole potomků. Jednotlivé uzly pak představují rozsah platnosti proměnných, uchovávají si svou tabulku symbolů a další důležitá data v rámci analýzy vstupního kódu. Tato struktura tak umožňuje efektivní vyhledávání symbolů dle relevance rozsahu platnosti, např. rekurzivní hledání napříč uzly a jejich tabulkami symbolů ve funkci `forest_search_symbol()`.



Obrázek 1: Grafická reprezentace hierarchického stromu s jednotlivými tabulkami symbolů

2.7 Datové struktury

2.7.1 Dynamický řetězec

Datová struktura `vector` reprezentuje dynamický řetězec a funkce, které s ním pracují jsou implementovány v souboru `string_vector.c`. Nejvíce je tento typ používán v lexikálním analyzátoru při načítání jmen identifikátorů, klíčových slov a řetězců.

2.7.2 Fronta

V souboru `queue.c` je implementována fronta tokenů. Používá se pro postupné ukládání informací pro následný zápis symbolu do tabulky symbolů. Tyto informace uložené v tokenech ve frontě jsou zároveň využity k sémantické analýze.

2.7.3 Zásobník tokenů

Používá se při vyhodnocování výrazů pomocí precedenční tabulky. Je implementován v souboru `token_stack.c` a kromě klasických operací se zásobníkem jako jsou `push()`, `pop()` nebo `top()`, obsahuje také funkci `stack_top_terminal()`, která navrácí nejvrchnější terminál na zásobníku.

2.7.4 Jednosměrně vázaný seznam

V jednosměrně vázaném seznamu implementovaném v souboru `callee.c` jsou uchovány ukazatele na struktury `callee_t`, které byly využity k sémantické analýze volání funkcí. Seznam umožňuje díky provázanosti snadný průchod napříč všemi položkami.

2.7.5 Obousměrně vázaný seznam

Seznam instrukcí v obousměrně vázaném seznamu je implementován v souboru `codegen.c`. Tento datový typ byl využit za účelem ukládání instrukcí kódu IFJcode23 do seznamu a současné možnosti seznam zpetně procházet a upravovat pořadí instrukcí dle potřeby.

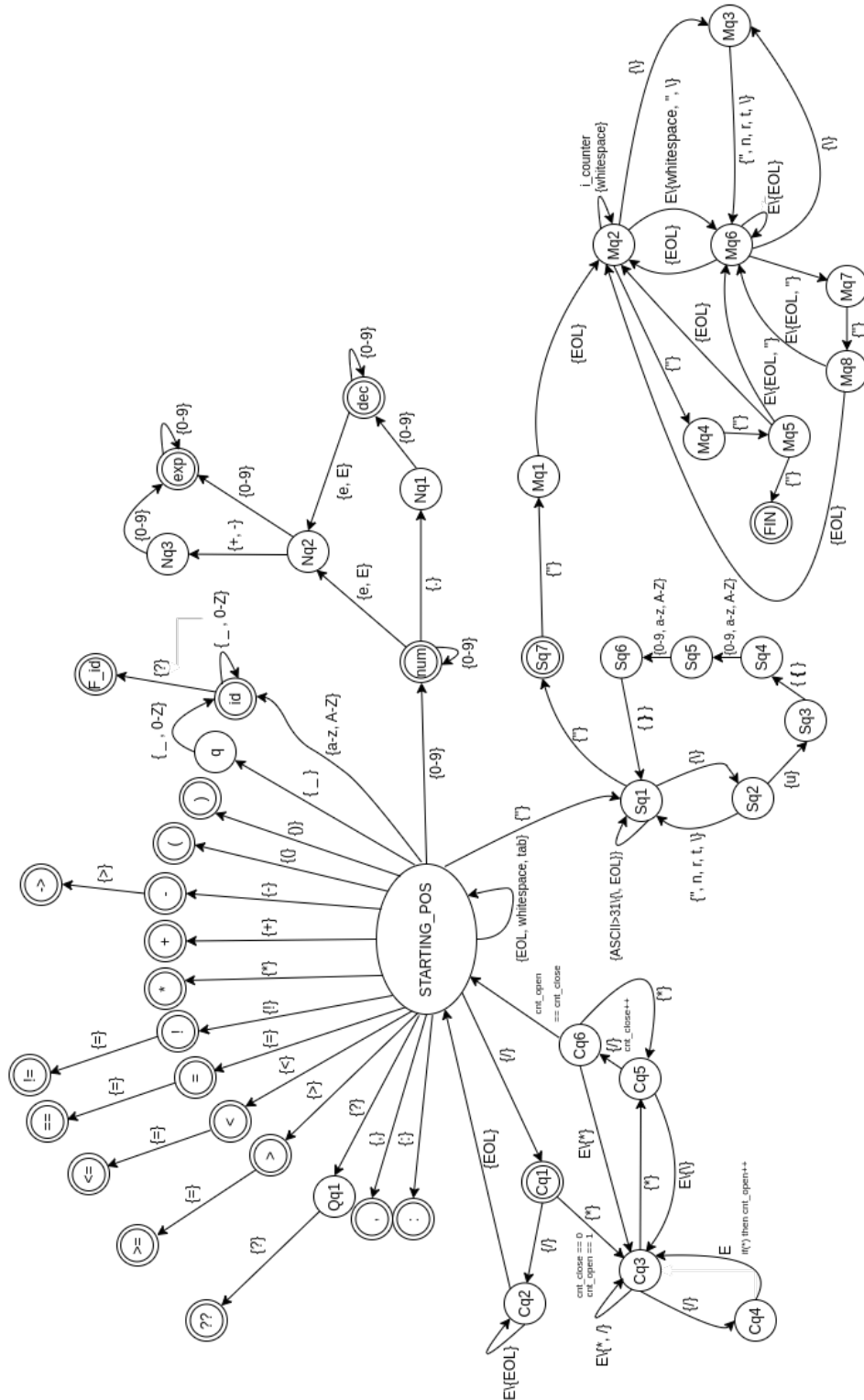
2.8 Rozšíření

2.8.1 FUNEXP

Toto rozšíření uvádíme, ačkoliv s ním naše implementace není plně kompatibilní, avšak návrh i následná implementace gramatiky částečně podporuje specifikaci v rozšíření zadání. Dle pravidel `<arg> -> <exp>` a `<arg> -> id:<exp>` lze za argumenty volání funkce dosadit jakýkoliv výraz, který je následně vyhodnocen a předán volání funkce.




3 Přílohy

3.1 Diagram konečného automatu



Obrázek 2: Diagram konečného automatu

3.2 Legenda diagramu konečného automatu

 S_COLON	 S_LESS_EQ	 S_MINUS	 S_NUM
 S_COMMA	 S_EQ	 S_RET_TYPE	 S_NUM_DOT
 S_QM	 S_EQEQ	 S_LPAR	 S_DEC
 S_DOUBLE_QM	 S_EXCLAM	 S_RPAR	 S_NUM_E
 S_GTR	 S_EXCLAMEQ	 S_UNDERSCORE	 S_NUM_E_SIGN
 S_GTR_EQ	 S_MULTIPLY	 S_ID	 S_EXP
 S_LESS	 S_PLUS	 S_ID_QM	 S_DIVIDE
 S_SL_COM	 S_START_QUOTES	 S_THREE_QUOTES	
 S_NESTED_COM	 S_START_ESC_SEQUENCE	 S_START_MULTILINE	
 S_SLASH	 S_START_HEX	 S_MULTI_ESC_SEQUENCE	
 S_NESTED_CLOSE	 S_FIRST_HEX	 S_ML_END_START	
 S_NESTED_END	 S_SECOND_HEX	 S_PRE_END_MULTILINE	
	 S_END_HEX	 S_IS_MULTILINE	
	 S_STR_EMPTY	 S_ML_EOL_END	
	 S_END_MULTILINE	 S_FAKE_END_MULTILINE	

3.3 LL gramatika

1. $\langle prog \rangle \rightarrow EOF$
2. $\langle prog \rangle \rightarrow \langle func_def \rangle \langle prog \rangle$
3. $\langle prog \rangle \rightarrow \langle body \rangle \langle prog \rangle$
4. $\langle func_def \rangle \rightarrow func\ id(\langle params \rangle) \langle ret_type \rangle \{ \langle local_body \rangle \}$
5. $\langle params \rangle \rightarrow \langle par_name \rangle \langle par_id \rangle : \langle type \rangle \langle params_n \rangle$
6. $\langle params \rangle \rightarrow \varepsilon$
7. $\langle par_name \rangle \rightarrow _$
8. $\langle par_name \rangle \rightarrow id$
9. $\langle par_id \rangle \rightarrow _$
10. $\langle par_id \rangle \rightarrow id$
11. $\langle type \rangle \rightarrow Int$
12. $\langle type \rangle \rightarrow Int?$
13. $\langle type \rangle \rightarrow Double$
14. $\langle type \rangle \rightarrow Double?$
15. $\langle type \rangle \rightarrow String$
16. $\langle type \rangle \rightarrow String?$
17. $\langle params_n \rangle \rightarrow , \langle params \rangle$
18. $\langle params_n \rangle \rightarrow \varepsilon$
19. $\langle ret_type \rangle \rightarrow - > \langle type \rangle$
20. $\langle ret_type \rangle \rightarrow \varepsilon$
21. $\langle local_body \rangle \rightarrow \langle body \rangle \langle local_body \rangle$
22. $\langle local_body \rangle \rightarrow \varepsilon$
23. $\langle body \rangle \rightarrow \langle var_def \rangle$
24. $\langle body \rangle \rightarrow \langle condition \rangle$
25. $\langle body \rangle \rightarrow \langle cycle \rangle$
26. $\langle body \rangle \rightarrow \langle assign \rangle$
27. $\langle body \rangle \rightarrow \langle func_call \rangle$

28. $\langle body \rangle \rightarrow \langle ret \rangle$
29. $\langle body \rangle \rightarrow \varepsilon$
30. $\langle ret \rangle \rightarrow return \langle exp \rangle$
31. $\langle ret \rangle \rightarrow return$
32. $\langle var_def \rangle \rightarrow let\ id \langle opt_var_def \rangle$
33. $\langle var_def \rangle \rightarrow var\ id \langle opt_var_def \rangle$
34. $\langle opt_var_def \rangle \rightarrow : \langle type \rangle$
35. $\langle opt_var_def \rangle \rightarrow \langle assign \rangle$
36. $\langle opt_var_def \rangle \rightarrow : \langle type \rangle \langle assign \rangle$
37. $\langle assign \rangle \rightarrow = \langle exp \rangle$
38. $\langle assign \rangle \rightarrow = \langle func_call \rangle$
39. $\langle func_call \rangle \rightarrow id(\langle args \rangle)$
40. $\langle args \rangle \rightarrow \varepsilon$
41. $\langle arg \rangle \rightarrow \langle exp \rangle$
42. $\langle arg \rangle \rightarrow id : \langle exp \rangle$
43. $\langle args_n \rangle \rightarrow , \langle arg \rangle \langle args_n \rangle$
44. $\langle args_n \rangle \rightarrow \varepsilon$
45. $\langle condition \rangle \rightarrow if \langle exp \rangle \{ \langle local_body \rangle \} else \{ \langle local_body \rangle \}$
46. $\langle condition \rangle \rightarrow if\ let\ id \{ \langle local_body \rangle \} else \{ \langle local_body \rangle \}$
47. $\langle cycle \rangle \rightarrow while \langle exp \rangle \{ \langle local_body \rangle \}$

Pozn. "*exp*" je označení pro výraz

3.4 LL tabulka

	EOF	func	if	while	id	return	let	var	=	_	exp
$\langle prog \rangle$	1	2	3	3	3	3	3	3	3		
$\langle func_def \rangle$		4									
$\langle params \rangle$					5					5	
$\langle par_name \rangle$					8					7	
$\langle par_id \rangle$					10					9	
$\langle type \rangle$											
$\langle params_n \rangle$											
$\langle ret_type \rangle$											
$\langle local_body \rangle$			21	21	21	21	21	21	21		
$\langle body \rangle$	29	29	24	25	27	28	23	23	26		
$\langle ret \rangle$						30					
$\langle var_def \rangle$							32	33			
$\langle opt_var_def \rangle$									35		
$\langle assign \rangle$									37		
$\langle func_call \rangle$					39						
$\langle args \rangle$											
$\langle arg \rangle$					42						41
$\langle args_n \rangle$											
$\langle condition \rangle$			45								
$\langle cycle \rangle$				47							

Tabulka 1: 1. část LL tabulky

	Int	Int?	Double	Double?	String	String?	:	,	->	{	})
$\langle prog \rangle$												
$\langle func_def \rangle$												
$\langle params \rangle$												6
$\langle par_name \rangle$												
$\langle par_id \rangle$												
$\langle type \rangle$	11	12	13	14	15	16						
$\langle params_n \rangle$								17				
$\langle ret_type \rangle$									19	20		
$\langle local_body \rangle$											21	
$\langle body \rangle$												
$\langle ret \rangle$												
$\langle var_def \rangle$												
$\langle opt_var_def \rangle$							34					
$\langle assign \rangle$												
$\langle func_call \rangle$												
$\langle args \rangle$												40
$\langle arg \rangle$												
$\langle args_n \rangle$								43				
$\langle condition \rangle$												
$\langle cycle \rangle$												

Tabulka 2: 2. část LL tabulky

3.5 Gramatika výrazů

1. $E \rightarrow i$
2. $E \rightarrow (E)$
3. $E \rightarrow E + E$
4. $E \rightarrow E - E$
5. $E \rightarrow E * E$
6. $E \rightarrow E / E$
7. $E \rightarrow E < E$
8. $E \rightarrow E > E$
9. $E \rightarrow E \leq E$
10. $E \rightarrow E \geq E$
11. $E \rightarrow E == E$
12. $E \rightarrow E != E$
13. $E \rightarrow E!$
14. $E \rightarrow E ?? E$

3.6 Precedenční tabulka

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

Tabulka 3: Precedenční tabulka