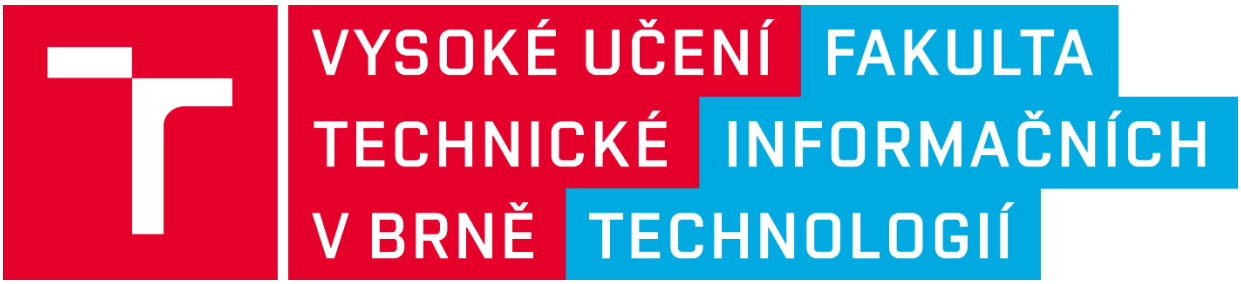


Implementace překladače imperativního jazyka IFJ22

Tým xstipe02, varianta BVS

Rozšíření: neimplementováno

Jiří Štípek	**xstipe02**	**25%**
Štefan Pekník	xpekni01	25%
Matěj Nešuta	xnesut00	25%
Ilia Markelov	xmarke00	25%

Obsah

1. Spolupráce v týmu

Při práci na projektu jsme využívali metodu párového programování. Pro verzování zdrojových kódů jsme používali GitHub. Pro komunikaci jsme zvolili platformu Discord (chat, hovory) a osobní setkání

1.1. Rozdělení práce

1.1.1. Jiří Štípek

- Precedenční analyzátor
- Generování kódu
- Dokumentace
- Automat pro lexikální analýzu

1.1.2. Štefan Pekník

- Lexikální analyzátor
- Syntaktický analyzátor shora-dolů
- Generování kódu
- LL-gramatika
- Automat pro lexikální analýzu
- Precedenční analyzátor

1.1.3. Matěj Nešuta

- LL-gramatika
- Generování kódu
- Automat pro lexikální analýzu

1.1.4. Ilia Markelov

- Tabulka symbolů
- Automat pro lexikální analýzu

2 Návrh překladače

2.1 Lexikální analyzátor

Lexikální analyzátor je implementován v souboru ***scanner.c.***

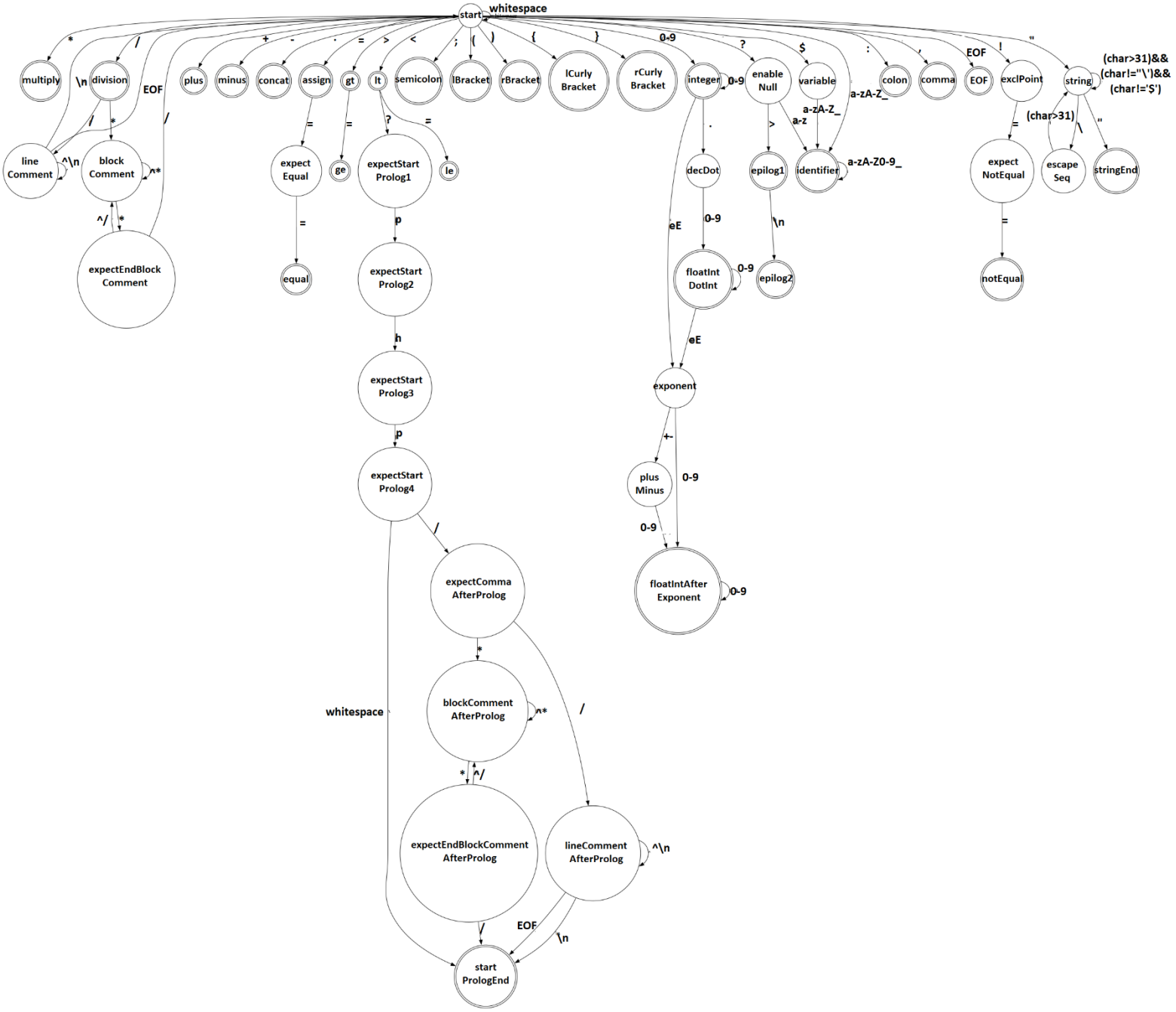
2.1.1 Reprezentace tokenů

Datová struktura obsahující tokeny, které využívá scanner, je definovaná v souboru ***scanner.h.***

2.1.2 Funkce GetLexeme()

Tato funkce je jedna z hlavních funkcí scanneru. Je vždy volána parserem, který vyžaduje následující token. Funkce kontroluje, zdali je prolog na začátku kódu a jestli odpovídá správnému zapsání dle zadání. Dále kontroluje, jestli už se nedošlo na konec kódu. Jakmile se došlo na konec, pošle parseru token ENDOFFILE.

2.1.3 Diagram konečného automatu



2.2 Syntaktický analyzátor

Syntaktický analyzátor je implementován v souboru ***parser.c.***

2.2.1 Funkce UpdateLLfirst(Parser *parser)

Funkce, která aktualizuje příchozí terminál, který načetl parser.

Jako metodu pro analýzu jsme zvolili rekurzivní sestup. Pojmenování funkcí v kódu je vždy ve tvaru rule_X(Parser *parser), kde X odpovídá neterminálu na levé straně LL-tabulky.

2.2.2 LL-gramatika

1. S -> PROG
2. PROG -> START_PROLOG CODE END_PROLOG
3. START_PROLOG -> startProlog function("declare") (function("strict_types") = integer(1)) ;
4. END_PROLOG -> endOfFile
5. END_PROLOG -> endProlog endOfFile
6. CODE -> ε
7. CODE -> INNER_SCOPE CODE
8. CODE -> FUNC_DECLARE CODE
9. BODY -> INNER_SCOPE BODY
10. BODY -> ε
11. INNER_SCOPE -> IF_ELSE
12. INNER_SCOPE -> return RETURN_VALUE ;
13. INNER_SCOPE -> while (EXP) { BODY }
14. INNER_SCOPE -> { BODY }
15. INNER_SCOPE -> FUNC_CALL

---------- CANT BE RESOLVED BY LL(1)----------

17. INNER_SCOPE -> EXP ;
18. INNER_SCOPE -> variable = RIGHT_SIDE (variable = RIGHT_SIDE)

---------- CANT BE RESOLVED BY LL(1)----------

21. RETURN_VALUE -> ε
22. RETURN_VALUE -> EXP
24. RETURN_TYPE -> ARG_TYPE
25. RETURN_TYPE -> void
26. FUNC_CALL -> function_id (FUNC_CALL_ARGS) ;
27. FUNC_CALL_ARGS -> ε
28. FUNC_CALL_ARGS -> ARG NEXT_ARG
29. NEXT_ARG -> ε
30. NEXT_ARG -> , ARG NEXT_ARG
31. ARG -> variable
32. ARG -> LITERAL
33. LITERAL -> float_lit
34. LITERAL -> int_lit
35. LITERAL -> null
36. LITERAL -> string_lit
37. RIGHT_SIDE -> FUNC_CALL
38. RIGHT_SIDE -> EXP ;
42. FUNC_DECLARE -> function function_id (FUNC_DECLARE_BODY) : RETURN_TYPE { BODY }
43. FUNC_DECLARE_BODY -> ε
44. FUNC_DECLARE_BODY -> ARG_TYPE variable FUNC_DECLARE_BODY
45. FUNC_DECLARE_BODY -> , ARG_TYPE variable FUNC_DECLARE_BODY
46. ARG_TYPE -> string
47. ARG_TYPE -> int
48. ARG_TYPE -> float
49. IF_ELSE -> if (EXP) { BODY } else { BODY }

2.2.3 LL-tabulka

currently used in project	if	else	while	function	function_id	int	float	string	void	return	"="	startProlog	endProlog	","	"("	")"	"{"	"}"	";"	"."	","	variable	string_lit	float_lit	int_lit	null	endOfFile	$	"+"	"-"	"*"	"/"
S																																
PROG												2																				
START_PROLOG												3																				
END_PROLOG													5														4						
CODE	7		7	8	7					7					7		7					7	7	7	7	7		6				
BODY	9		9		9					9					9		9					9	9	9	9	9		10				
INNER_SCOPE	11		13		15					12					17		14					17/18	17	17	17	17						
RETURN_VALUE															22							22	22	22	22	22		21				
IF_ELSE	49																															
FUNC_CALL					26																											
FUNC_CALL_ARGS																						28	28	28	28	28		27				
NEXT_ARG																					30							29				
ARG																						31	32	32	32	32						
LITERAL																							36	33	34	35						
FUNC_DECLARE				42																												
FUNC_DECLARE_BODY						44	44	44													45							43				
ARG_TYPE						47	48	46																								
RETURN_TYPE						24	24	24	25																							
RIGHT_SIDE					37										38							38	38	38	38	38						
OPERATOR																													50	51	52	53

2.2.4 Konstrukce nepokryté LL-gramatikou

Z důvodu nedeterminismu u pravidel 17 a 18 za použití LL(1) bylo nutno při výběru správného pravidla použít LL(2), kdy se rozhodujeme mezi pravidly dle existence rovnítka za prvním terminálem v případě, že je první terminál "variable".

2.3 Precedenční analýza

Analyzátor pro výrazy je implementován v souboru ***expressionParser.c.***
Jakmile se dojde ke zpracovávání výrazů, a to právě díky LL-gramatice, volá se funkce `rule_EXP()`. Přes tuto funkci se přejde k top-down parseru. Ten začne vyhodnocovat výraz, a to za pomocí precedenční tabulky. Také se k tomu používá funkce `UpdateLLfirst(Parser *parser)`.

2.3.1 Jednosměrně vázaný list

V rámci syntaktické kontroly generujeme strukturu, která se zakládá na klasickém stromu, přičemž kořenem je symbol (může být terminálem či neterminálem), potomci kořene listu jsou zapsány seřazeně v jednosměrném vázaném listu, přičemž každý potomek je opět touto strukturou.

2.3.2 Precedenční tabulka

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

2.4 Generování kódu

Vypisování cílového kódu na standardní výstup se provádí voláním funkcí implementovaných v souboru codegen.c

2.5 Sémantické kontroly

Nejsou implementovány v samostatném souboru, ale provádí se průběžně během parsingu a generování kódu.

2.6 Tabulka symbolů

Tabulka je implementována v souboru symtable.c

2.6.1 BVS

Tabulka symbolů je implementována ve variantě binárního vyhledávajícího stromu (BVS).