Dokumentace

Implementace překladače imperativního jazyka IFJ22

Tým xbarta50, varianta 1

Štěpán Bárta	**xbarta50**	**25%**
Roman Janota	xjanot04	25%
Kryštof Paulík	xpauli08	25%
Tomáš Valík	xvalik04	25%

Obsah

1 Práce v týmu

1.1 Rozdělení práce mezi jednotlivé členy týmu

- Štěpán Bárta
  - návrh automatu pro lexikální analýzu
  - implementace lexikálního analyzátoru
  - dokumentace
- Roman Janota
  - návrh automatu pro lexikální analýzu
  - implementace syntaktického analyzátoru
  - syntanktická analýza shora dolů rekurzivním sestupem
  - generování cílového kódu
- Kryštof Paulík
  - návrh automatu pro lexikální analýzu
  - implementace syntaktického analyzátoru
  - precedenční analýza zdola nahoru
- Tomáš Valík
  - návrh automatu pro lexikální analýzu
  - implementace lexikálního analyzátoru
  - implementace tabulky symbolů
  - generování cílového kódu

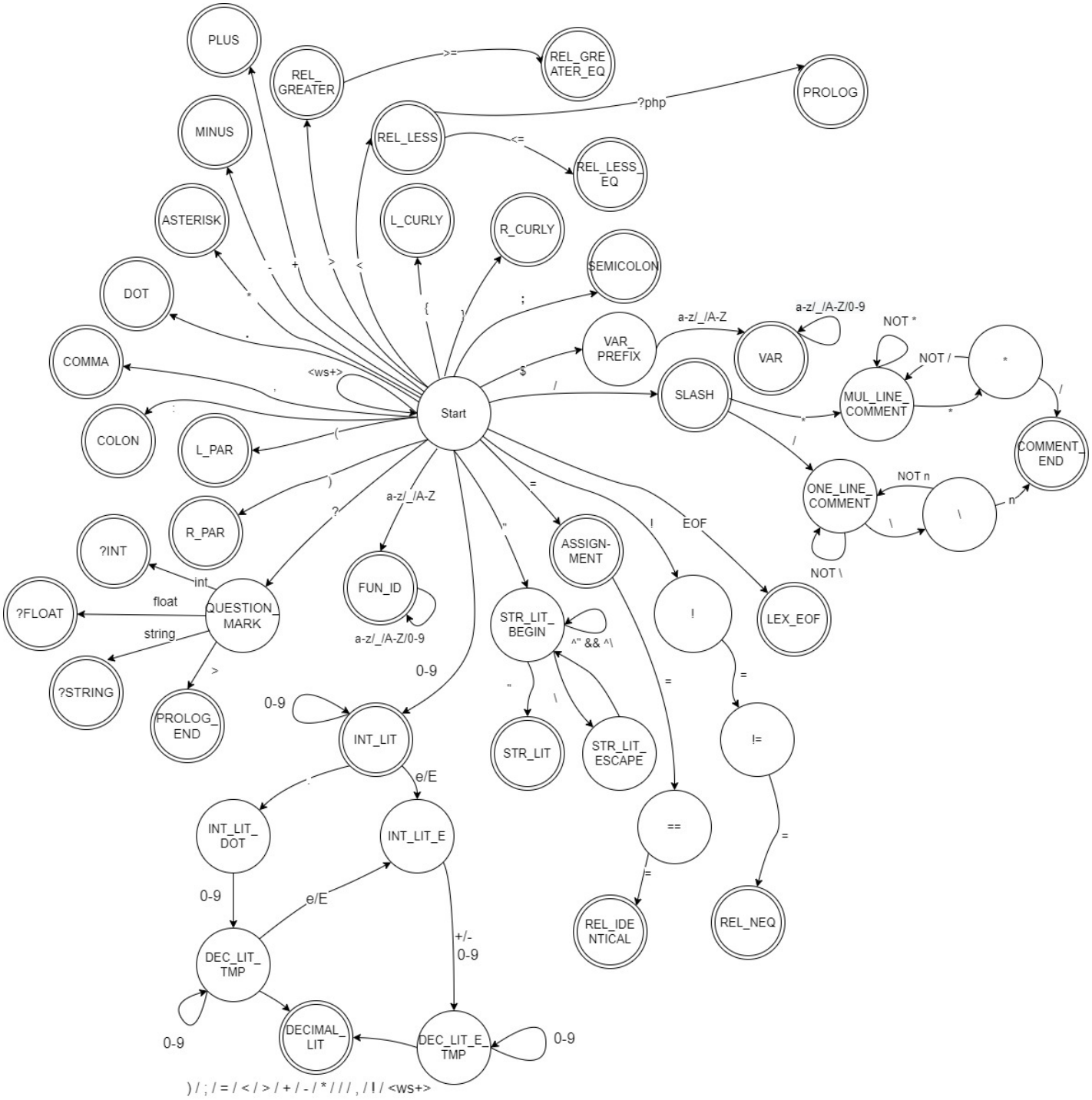
2 Lexikální analýza

Lexikální analyzátor je implementován v souborech **lexical_analysis.c** a **lexical_analysis.h**. Jeho nejdůležitější částí je funkce **getToken()**, ve které je uložen současný a minulý stav. Pomocí cyklu while, který běží dokud není možné vygenerovat token, se načítají jednotlivé znaky ze vstupu a ty se společně se současným stavem posílají do funkce **getNextState()**, která na základě současného stavu a znaku na vstupu rozhodne, zda je znak v současném stavu validní, pokud ano, změní současný stav a pokud ne, hlásí lexikální chybu.
Pokud je možné vygenerovat token, posíláme minulý stav do funkce **makeLexeme()**, která změní typ tokenu. Pokud načte identifikátor, voláme funkci **funIdToKeyword()**, která zkontroluje, zda se nejedná o jedno z klíčových slov, a pokud ano, změní typ tokenu.
Názvy stavů FSM jsou definovány enumerací `state` a názvy typů tokenů `lex_types` v souboru lexical_analysis.h.

Struktura lexeme obsahuje:

- typ tokenu
- jeho identifikátor
- hodnotu podle typu
  - int
  - float
  - string

2.1 Diagram konečného automatu



Obrázek 1: Diagram konečného automatu

3 Syntaktická analýza

Syntaktickou analýzu tvoří dvě hlavní části a to analýza shora dolů za pomocí rekurzivního sestupu a zdola nahoru užitím metody precedenční analýzy.

3.1 Syntaktická analýza shora dolů

K implementaci rekurzivního sestupu byla využita LL gramatika a současně i tabulka popisující tuto gramatiku.
Řídili jsme se syntaxí řízeným překladem, a proto tělem celého překladače je funkce **synt_parse()**.
Tato funkce nejdříve zkontroluje, jestli je prolog programu syntakticky v pořádku a následně i celý jeho zbytek a to voláním funkce **rule_statement_list()**.
Pro skoro každé pravidlo z LL gramatiky existuje funkce, která je rekurzivně volána.
Informace potřebné jak pro syntaktickou, tak i z části pro sémantickou analýzu, jsou uchovávány v struktuře **compiler_ctx**, která udržuje momentální stav překladače.
Naopak informace potřebné pro generování jsou ukládány v struktuře **generator**, a generování probíhá přímo z kódu za běhu, bez mezikroků.

3.2 Syntaktická analýza zdola nahoru

Pro zpracování výrazů jsme použili metodu syntaktické analýzy zdola nahoru s využitím precedenční tabulky a zásobníku.
Funkce **expression_parse()** postupně ukládá tokeny na zásobník a pomocí precedenční tabulky určuje vztah mezi tokenem na vrcholu zásobníku a tokenem na vstupu.
Podle toho buď před již zmíněný token vkládá symbol shift a nebo volá funkci **reduce()**.
Tato funkce nejprve díky pomocné funkci **test_rule()** zjistí, které pravidlo použít, následně pomocí další pomocné funkce **test_semantics()** testuje základní sémantiku a volá příslušné funkce pro generování výsledného kódu.
Následně podle určeného pravidla redukuje tokeny na zásobníku.
Pokud žádné z pravidel neodpovídá tokenům na zásobníku, funkce končí a vrací příslušnou syntaktickou chybu.
Pokud na zásobníku zůstane jediný neterminál a na vstupu se objeví příslušný ukončující token, syntaktická analýza proběhla vpořádku a funkce vrací řízení rekurzivnímu sestupu.

3.3 Tabulka symbolů

Tabulka symbolů byla implementována pomocí binárního vyhledávacího stromu a byla navržena následovně:

- struktura **bs_tree**: obsahuje klíč položky, data, a odkazy na levého a pravého potomka
- struktura **bs_data**: obsahuje buď data funkce nebo proměnné a pravdivostní hodnotu **is_function**, podle níž se rozhodne, o která data se jedná
- struktura **function_data**: uchovává informace o funkci, a to návratový typ, jestli už byla definovaná, typy a počet parametrů
- struktura **variable_data**: uchovává informace o proměnné, a to její typ a jestli už byla definovaná.

Po dokončení návrhu byly nad tímto datovým typem naimplementovány běžné operace a to inicialize tabulky symbolů a dat, vložení a vyhledání dat, vložení vestavěných funkcí a smazání jejího obsahu.
Přístup k tabulce symbolů byl implementován tak, že struktura **compiler_ctx** obsahovala 2 odkazy, jeden na globální a jeden na lokální tabulku symbolů.

3.4 LL gramatika

1. <program> → prolog statement_list EOF

2. <statement_list> → statement statement_list
3. <statement_list> → ϵ

4. <statement> → func ;
5. <statement> → function function_name_T (parameters) : type_T { statement_list }
6. <statement> → if (expr) { statement_list } else { statement_list }
7. <statement> → while (expr) { statement_list }
8. <statement> → return expr ;
9. <statement> → id_T expr ;
10. <statement> → ?>

11. <func> → (func_params)
12. <func_params> → Terminal next_param
13. <func_params> → ϵ
14. <next_param> → , Terminal next_param
15. <next_param> → ϵ

16. <expr> → expr expr_tail
17. <expr> → (expr)
18. <expr> → = expr
19. <expr> → func
20. <expr> → Terminal

21. <expr_tail> → ϵ
22. <expr_tail> → + expr
23. <expr_tail> → - expr
24. <expr_tail> → / expr
25. <expr_tail> → * expr
26. <expr_tail> → === expr
27. <expr_tail> → !== expr
28. <expr_tail> → < expr
29. <expr_tail> → <= expr
30. <expr_tail> → > expr
31. <expr_tail> → >= expr
32. <expr_tail> → . expr
33. <expr_tail> → , expr

3.5 LL tabulka

	prolog	EOF	ε	;	function	function_name_T	(	parameters	)	:	type_T	{	}	if	else	while	
<program>	1																
<statement_list>			3		2		2							2		2	
<statement>					5		4							6		7	
<func>							11										
<func_params>			13														
<next_param>			15														
<expr>							17										
<expr_tail>			21														

	return	id_T	?>	Terminal	,	.	=	+	-	/	*	===	!==	<	<=	>	>=
<program>																	
<statement_list>	2	2	2														
<statement>	8	9	10														
<func>																	
<func_params>				12													
<next_param>					14												
<expr>				20			18										
<expr_tail>					33	32		22	23	24	25	26	27	28	29	30	31

Obrázek 2: LL tabulka

3.6 Precedenční tabulka

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

Obrázek 3: Precedenční tabulka

4 Generování

Při implementaci generátoru cílového kódu jsme zvolili možnost generovat za běhu.
Ze syntaktické analýzy jsou postupně volány příslušné funkce pro generovaní
generator<NazevFunkce>().
Veškeré generované části kódů jsou poté ukládány do dynamicky alokovaných řetězců a ve struktuře generátoru na ně jsou uchovávány jednotlivé ukazatele.
Po doběhnutí celého překládaného programu, jsou všechny řetězce ve správném pořádí vypsány na standardní výstup. Při implementaci smyček a podmínek byly využity inkrementovatelné identifikátory, aby bylo možné správně generovat návěští a nedocházelo tak k jejich duplicitě.
V případě generovaní aritmetických, relačních a řetězcových operátorů jsou také vygenerovány příslušné sémantické kontroly a případně jsou provedeny typové konverze dle zadaní.

5 Členění implementačního řešení

- Lexikální analýza
  - lexical_analysis.c
  - lexical_analysis.h
- Syntaktická analýza
  - shora dolů
    - syntactic_analysis.c
    - syntactic_analysis.h
  - zdola nahoru
    - expression.c
    - expression.h
- Zásobník
  - stack.c
  - stack.h
- Generování cílového kódu
  - generator.c
  - generator.h
- Tabulka symbolů
  - symtab.c
  - symtab.h
- Ostatní
  - compiler.c
  - compiler.h
  - main.c