



Dokumentace projektu z předmětů IFJ, IAL
Tým xpokhv00, varianta TRP

Autoři:

Ivan Onufriienko, xonufr00	- 25%
Vsevolod Pokhvalenko, xpokhv00	- 25%
Oleksii Shelest, xshele02	- 25%
Sviatoslav Pokhvalenko, xpokhv01	- 25%

1.Úvod

Tato dokumentace popisuje návrh a implementaci překladače jazyka IFJ23, který je zjednodušenou podmnožinou jazyka Swift. Verzí naší úlohy byla TRP, ve které bylo nutné implementovat tabulku symbolů pomocí hash tabulky s implicitním zřetěžením položek.

2.Implementace

2.1 Návrh

- Lexikální analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Generátor kódu

2.2 Lexikální analyzátor

- je první částí překladače, která pracuje přímo se vstupem, konkrétně čte lexemy a převádí je na tokeny. Tyto tokeny jsou již předány syntaktickému analyzátoru. Lexikální analyzátor byl implementován pomocí konečného automatu, zpracovává text znak po znaku a podle stavu, ve kterém čtení dokončil, určí typ, řádek a kam je potřeba token hodnotu. Pokud čte znak, který nelze zpracovat, jsou dvě možnosti: pokud dočetl v koncovém stavu, tak právě přečtený znak odešle zpět na vstup, nebo pokud dočetl v nekoncovém stavu, vypíše lexikální chybu. Dále bylo nutné implementovat funkce *peek_token()* pro prohlížení dalšího tokenu, který funguje stejně jako hlavní funkce, ale také ukládá token pro další použití. Řešení této části je v souborech *scanner.c* a *scanner.h*.

2.3 Syntaktický analyzátor

Syntaktický analyzátor, ve zdrojových souborech `parser.c` a `parser.h`, je klíčovou částí překladače (volá všechny ostatní části překladače), která zpracovává vstupní tokeny a vytváří syntaktický strom. Implementace tohoto analyzátoru byla realizována metodou rekurzivního sestupu za pomoci LL-gramatiky a LL-tabulky. Syntaktický analyzátor postupuje podle pravidel definovaných gramatikou a porovnává aktuální stav zásobníku s příchozím tokenem, výrazy jsou zpracovávány precedenční syntaktickou analýzou. Abychom pochopili, zda potřebujeme určit typ proměnné/konstanty sami nebo zda již byla nastavena, potřebovali jsme funkci `peek_token()` k zobrazení dalšího tokenu.

2.4 Sémantický analyzátor

Sémantický analyzátor pracuje společně se syntaktickým, konkrétně kontroluje každou úroveň kódu, aby zjistil, zda došlo k redefinici funkce nebo proměnné/konstanty na jedné úrovni či nikoli, zda je možné funkci na této úrovni definovat a také kontroluje správnost zadání parametrů funkce a jejich použití v této funkci. Jelikož jsme potřebovali implementovat použití jak jména, tak ID proměnné do parametrů funkce, použili jsme spojový seznam.

2.5 Generátor kódu

Generátor kódu je poslední částí naší práce. Zpracovává všechna data přijatá ze všech parserů a na jejich základě začne generovat kód v IFJcode23. Po každé změně byl generátor testován společně s parserem, protože správnost generování kódu závisí na parseru. Integrace generátoru do překladače byla provedena přes globální funkce použité v souborech `parser.c` a `expression.c`.

3. Algoritmy

3.1 Precedenční syntaktická analýza

Výrazy jsou zpracovávány odděleně od syntaktického zpracování, a to v souborech *expression.c* a *expression.h*. Tento algoritmus byl implementován technikou zdola nahoru s využitím precedenční tabulky. Algoritmus byl implementován tak, že funkce *parseExpression* zapisuje tokeny do zásobníku a pomocí tabulky určuje vztah mezi tokenem v horní části zásobníku a právě přečteným tokenem, pokud nebylo nalezeno vhodné pravidlo, vrátí se chyba 2.

3.2 Tabulka symbolů

Pro naše zadání jsme potřebovali implementovat hashovací tabulku, která se nachází v souborech *symtable.c* a *symtable.h*. Tato tabulka je potřebná pro uložení názvů funkcí a proměnných, jejich dat a také na podporu globálního a lokálního scope. Samotné prvky tabulky najdeme pomocí hashe, který nám prozradí místo prvku v tabulce.

3.3 Zásobníky

Zásobník implementovaný v souborech *symstack.c* a *symstack.h* je potřebný pro uložení proměnných a funkcí a jejich dat ve správném pořadí.

3.4 Spojový seznam

Spojový seznam je úložištěm parametrů funkcí, když jsou deklarovány a volány. Implementováno v souborech *linlist.c* a *linlist.h*.

4. Práce v týmu

4.1. Rozdělení práce

Ivan Onufriienko - Návrh konečného automatu, lexikální analýza, dokumentace, chybové hlášky.

Vsevolod Pokhvalenko - Makefile, syntaktická analýza a sémantická analýza, testování jednotlivých částí překladače, Linked List, SymTable, SymStack.

Oleksii Shelest - Generování cílového kódu, testování jednotlivých částí překladače.

Sviatoslav Pokhvalenko - Návrh LL-tabulky, LL-gramatiky, precedenční tabulky, syntaktická analýza a sémantická analýza, prezentace.

4.2 Komunikace

Ke komunikaci a distribuci našeho kódu jsme použili Telegram a GitHub a samozřejmě skutečný svět. Když nebyly problémy s komunikací, nastaly problémy s distribucí vlastního kódu, protože někteří členové týmu si občas nestahovali aktuální verzi projektu a v důsledku toho vznikaly “kuriozity”.

Stav	Popis
Start	initial state
C1	identifier
C2	nillable keyword
C3	minus
C4	plus
C5	multiply

C6	divide
C7	colon
C8	left bracket
C9	right bracket
C10	comma
C13	assignment
C14	greater
C15	less
C16	exclamation mark
C17	not equal
C18	equal
C19	greater or equal
C20	less or equal
C21	left curly bracket
C22	right curly bracket
C23	nil coalescing operator
C24	string(multiline string)
N1	integer
N2	double
N3	float
U1	possible double
U2	possible float
U3	possible float with shift
U4	start of string or its content
U5	start of escape sequence
U6-U9	unicode escape sequence
U10	comment
U11	start of multiline comment
U12	possible end of multiline comment
U13-U15	embedded comment
U16	possible nil coalescing operator
E	end

5.2 LL-gramatika

1. <prog> -> <scope>
2. <prog> -> eol <prog>
3. <scope> -> eps
4. <scope> -> func id(<params>) -> <type> {<body>} <scope>
5. <scope> -> func id(<params>) {<body>} <scope>
6. <func_call> -> id(<params_call>)
7. <statement> -> if expression {<body>} <else>
8. <statement> -> if let id {<body>} <else>
9. <statement> -> while expression {<body>}
10. <statement> -> write expression
11. <statement> -> expression
12. <statement> -> return <return_follow>
13. <return_follow> -> eps
14. <return_follow> -> expression
15. <else> -> eps
16. <else> -> else <else_next>
17. <else_next> {<body>}
18. <else_next> if expression {<body>} <else>
19. <body> -> eps
20. <body> -> <statement> <body>
21. <type> -> integer
22. <type> -> double
23. <type> -> string
24. <type> -> integer?
25. <type> -> double?
26. <type> -> string?
27. <value> -> int_value
28. <value> -> double_value
29. <value> -> string_value
30. <value> -> nil
31. <write> -> eps
32. <write> -> expression <write>
33. <params> -> eps
34. <params> -> name id : <type> <params_n>
35. <params_next> -> eps
36. <params_next> -> , <params>
37. <params_call> -> eps
38. <params_call> -> name id <params_call_n>
39. <params_call_next> eps
40. <params_call_next> , <params_call>
41. <end> -> eol <end>
42. <end> -> eof

5.3 LL-tabulka

	scope	eol	func	eof	id	name	if	else	expression	while	write	return)	{	,	}	integer	double	string	integer?	double?	string?	int_value	double_value	string_value	nil	\$
<prog>	1	2																									
<scope>			4,5																								3
<func_call>					6																						
<statement>					7		8		11	9	10	12															
<return_follow>									14								13										
<else>								16									15										
<else_next>							18							17													
<body>											20						19										
<type>																	21	22	23	24	25	26					
<value>																							27	28	29	30	
<write>										32																	31
<params>						34								33													
<params_next>														35	36												
<params_call>						38								37													
<params_call_next>														39	40												
<end>		41		42																							

5.4 Precedenční tabulka

	+ -	* /	id	\$	<=	()	!	??
+ -	>	<	<	>	>	<	>	<	>
* /	>	>	<	>	>	<	>	<	>
id	>	>	F	>	>	F	>	>	>
\$	<	<	<	O	<	<	F	<	<
<=	<	<	<	>	F	<	>	<	>
(<	<	<	F	<	<	E	F	<
)	>	>	F	>	>	F	>	>	>
!	>	>	F	>	>	F	>	F	>
??	<	<	<	>	<	<	>	<	<