Vysoké učení technické v Brne

Fakulta informačných technológií

Dokumentácia k projektu z predmetov IFJ a IAL

Implemetácia prekladaču imperatívneho jazyka IFJ22

Tím xhrach06, varianta BVS

Matej Hrachovec, xhrach06

Dominik Truchly, xtruch01

Jakub Brnak, xbrnak01

Michal Ondrejka, xondre15

Obsah

1. Úvod

Táto dokumentácia popisuje implementáciu prekladaču impretívneho jazyka IFJ22 Dokumentácia je rozdelená do kapitol a ich podkapitol a popisuje komunikáciu v tíme, rozdelenie práce a popis jednotlivých častí implementácie.

2. Práca v tíme

a. Komunikácia v tíme

Komunikácia a práca na projekte prebiehala od polovice októbra, osobne sme sa stretávali týždenne každý štvrtok, na osobných stretnutiach sme riešili veci ako kostru projektu, prepájanie jednotlivých častí a rozdelenie práce na ďalší týždeň. Komunikácia zároveň prebiehala cez Discord, kde komunikovali ľudia, ktorí boli priamo zúčastnení pri implementácií aktuálnej časti projektu. Začiatky boli pomalé, ale vďaka tímovej práci a efektívnej výmene informácií sme dokázali začať s implementáciou s dostatočným predstihom.

b. Správa kódu

Kód bol zdieľaný pomocou verzovacieho systému Git na platforme GitHub, každý pridával svoj progres po vykonaní nejakej zmeny aj s krátkym popisom. Pri písaní kódu sme používali Visual Studio Code a jeho rozšírenie Microsoft Live Share ktoré nám umožnilo spoločne upravovať kód v reálnom čase.

c. Priebeh rozdelenia práce

Zo začiatku sme sa rozdelili na dve skupiny, jedna mala na starosti knečný automat a Lexikálnu analýzu (xtruch01, xondre15) a druhá skupina mala na starosti návrh gramatiky, LL tabuľky a syntaktickú analýzu (xhrach06, xbrnak01). Neskôr sa však rozdelenie muselo zmeniť a každý z nás dostal nové úlohy vzhľadom na potreby dokončenia projektu. Xtruch01 a xbrnak01 dostali na starosti generáciu kódu, xondre15 precedenčnú analýzu výrazov a xhrach06 písanie dokumentácie.

3. Implementácia

a. Lexikálna analýza

Lexikálna analýza analýza bola implementovaná pomocou konečného automatu, ktorý spracováva vstup a rozdeľuje ho na lexémy ktoré následne využíva zvyšok programu, každý lexém ma svoj druh a v niektorých prípadoch aj hodnotu. Jej implementácia sa nachádza v súbore Lexical_analysis.c.

b. Syntaktická analýza

Syntaktická analýza funguje na princípe rekurzívneho zostupu. Pravidlá gramatiky sú spracovávané pomocou funkcií ktoré ich popisujú. Pravidlá sa vždy vyberajú na základe terajšieho pravidla a na lexéme ktorý je prijatý funkciou getLexeme() ktorá je popísana v súbore lexikálnej analýzy. Pri narazení na chybu sa vždy zavolá príslušný návratový kód a ukončí program. Postupovali sme pomocou našej LL-gramatiky avšak pri niektorých častiach implementácie sme sa museli od pravidiel ochýliť a gramatiku mierne upraviť. Najväčšou prekážkou bolo dostávanie lexémov v správnom čase, aby sme si nepokazili beh programu.

c. Sémantická analýza

Sémantická analýza prebieha zároveň so syntaktickou, berie si informácie z tabuľky symbolov a využíva operácie nad ňou definované na vyhladávanie premenných, pridávanie funkcií a podobne. Pri narazení na chybu sa vždy zavolá príslušný návratový kód a ukončí program. Prebieha zároveň so syntaktickou analýzou, aby bol zaručený prístup k lexémom. Pri návratových typoch funkcie a pri jej argumentoch bolo ukladanie hodnôt zložitejšie, čo sme vyrišili zavedením dvoch polí priamo v uzle popisujúceho funkciu. Taktiež bolo problematické implementovať, či môže premenná nadobúdať hodnotu null a implementácia návratu z tela funkcie typu void.

d. Precedenčná analýza výrazov

Analýza výrazov prebieha pomocou precedenčnej tabulky spracovávnia výrazov. Zavolá sa vždy, keď program narazí na výraz a validuje ho. Funkcia endExpression vracia typ daného výrazu, ktorý je využívaný na validáciu správnosti výrazu a na prácu s premennými.

e. Tabuľka symbolov

Tabuľka symbolov, jej štruktúra a funkcie nad ňou sú popísané v súbore symtable.c. Skladá sa z binárneho stromu NODE, ktorý uchováva informácie o funkciách a ich rámcoch. Každá funkcia obsahuje svoj binárny strom NODE_LOCAL kde sú uložené informácie o jej premenných.

f. Generácia kódu

Generácia kódu prebieha taktiež zároveň so syntaktickou analýzou. Táto časť bola najproblematickejšia.

4. Dátové Štruktúry

a. Lexém

Struct Lexeme obsahuje enumerátor kind, ktorý určuje druh lexému, reťazec string, ktorý uchováva jeho skutočnú hodnotu, hodnoty value a fvalue predstavujú hodnoty číselných literálov a lineNumber uchováva riadok na ktorom sa nachádza.

b. AutomatState

Tento enumerátor predstavuje stavy konečného automatu.

c. Token

Dátová štruktúra token popisuje jednotlivé premenné v tabuľke symbolov, obsahujú jeho typ, názov a hodnotu.

d. Tabuľka symbolov

Tabuľka symbolov sa skladá z dvoch hlavných častí, štruktúra NODE uchováva infomácie o funkciách: návratový typ, meno, argumenty a ich typy, a taktiež či môže funkcia vraciať hodnotu null. Funguje ako binárny vyhľadávací strom kde kľúč predstavuje jej názov. Každá NODE obsahuje NODE_LOCAL, v ktorej sa uchovávaju jednotlivé tokeny, kľúčom tejto tabuľky je taktiež názov premennej uchováva informácie o tom, či je premenná zároveň aj parametrom, jej typ a hodnotu.

e. Zásobník precedenčnej analýzy

Obsahuje prvky štruktúry StackItem ktorá zapúzdruje lexém, naviac obsahuje dátový typ výrazu a typ operandu

5. LL-gramatika

1: p_start -> prologue p_body p_end

2: prologue -> <?php declare (strict_types = 1) ;

3: p_end -> EOF

4: p_end -> ?>

5: p_body -> ε

6: p_body -> p_prikaz p_body

7: params_in_start -> ε

8: params_in_start -> params_in params_in_next

9: params_in -> expression

10: params_in -> VAR

11: params_in_next -> , params_in params_in_next

12: params_in_next -> ε

13: types -> int

14: types -> float

15: types -> string

16: func_typesq -> ? func_types

17: func_typesq -> func_types

18: func_types -> types

19: func_types -> void

20: var_typesq -> ? types

21: var_typesq -> types

22: func_def -> function ID (params) : func_types { f_body }

23: params -> ε

24: params -> var_types VAR params_next

25: params_next -> , var_types VAR params_next

26: params_next -> ε

27: f_body -> f_prikaz f_body

28: f_body -> ε

29: f_prikaz -> VAR = assignment ;

30: f_prikaz -> func_call ;

31: f_prikaz -> if (expression) { f_body } else { f_body }

32: f_prikaz -> while (expression) { f_body }

33: f_prikaz -> return ret_value ;

34: ret_value -> expression

36: ret_value -> ε

37: ret_value-> VAR

38: p_prikaz -> function ID (params) : func_types { f_body }

39: p_prikaz -> VAR = assignment ;

40: p_prikaz -> func_call ;

41: func_call -> ID (params_in_start)

42: p_prikaz -> if (expression) { in_body } else { in_body }

43: p_prikaz -> while (expression) { in_body }

44: in_body -> in_prikaz in_body

45: in_body -> ε

46: in_prikaz -> VAR = assignment ;

47: in_prikaz -> func_call ;

48: in_prikaz -> if (expression) { in_body } else { in_body }

49: in_prikaz -> while (expression) { in_body }

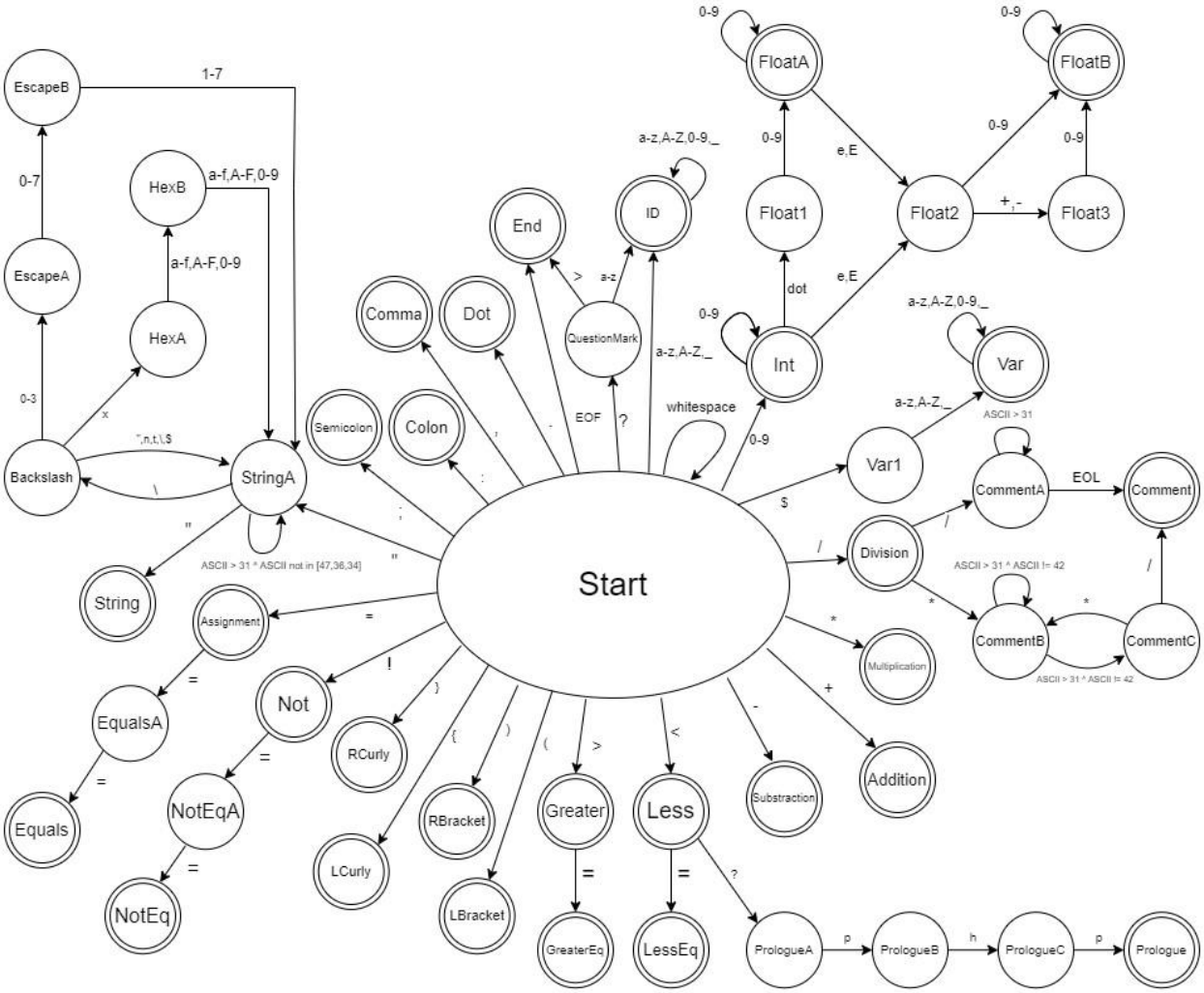
50: assignment -> expression

51: assignment -> func_call

6. LL-tabuľka

Nonterminal	declare	(	strict_types	=	1	)	;	EOF	?>	ε	expression	VAR	,	int	float	string	?	void	function	ID	:	{	}	var_types	if	else	while	return	$
p_start	1																												
prologue	2	2	2	2	2	2	2																						
p_end								3	4																				
p_body										5		6							6	6					6			6	
params_in_start										7	8	8																	
params_in											9	10																	
params_in_next										12			11																
types														13	14	15													
func_typeseq														17	17	17	16	17											
func_types														18	18	18		19											
var_typeseq														21	21	21	20												
func_def																			22										
params										23													24						
params_next										26			25																
f_body										28		27								27				27		27		27	
f_prikaz												29								30				31		32		33	
ret_value										36	34	37							38	40				42		43			
p_prikaz												39								41									
func_call																				44				44		44			
in_body										45		44								47				48		49			
in_prikaz												46								51									
assignment											50																		

7. Konečný automat



8. Precenčná tabuľka na spracovanie výrazov

*	/	+	-	.	(	)	i	rel	$	Input/Stack
>	>	>	>	>	<	>	<	>	>	*
>	>	>	>	>	<	>	<	>	>	/
<	<	>	>	>	<	>	<	>	>	+
<	<	>	>	>	<	>	<	>	>	-
<	<	>	>	>	<	>	<	>	>	.
<	<	<	<	<	<	=	<	<	-	(
>	>	>	>	>	-	>	-	>	>	)
>	>	>	>	>	-	>	-	>	>	i
<	<	<	<	<	<	>	<	>	>	rel
<	<	<	<	<	<	-	<	<	-	$

9. Členenie súborov

Lexical_analysis.c: implementácia lexikálnej analýzy

Syntax_analysis.c: implementácia syntaktickej a sémantickej analýzy, generácia kódu

Syntax_analysis.h: importovanie potrebných súčastí na beh analýzy

Precedent_analysis.c: implementácia precedenčnej analýzy

Makefile: preklad pomocou gcc symtable.c:

implementácia tabuľky symbolov

10. Záver

Tento projekt každého z nás veľa naučil a ukázal nám princíp fungovania prekladačov a tímovej práce. Verím, že naša implementácia je úspešná, aj keď bol projekt náročný.