Dokumentácia projektu IFJ

Implementace překladače imperativního jazyka IFJ18

Tím 105, varianta I

Glós Kristián	**xglosk00**
Abikenova Zhamilya	xabike00
Vagala Dominik	xvagal00
Vinš Jakub	xvinsj00

5.12.2018

Obsah

1 Úvod

Dokumentácia slúži na popis návrhu a implementácie interpretu imperatívného jazyka IFJ18 v jazyku C. Program načítava kód v tomto jazyku a preloží ho do cieľového jazyka IFJcode18.

2 Návrh a implementácia

2.1 Lexikálna analýza (Scanner)

Scanner bol vytvorený na základe konečného automatu (Príloha 1) a to pomocou konštrukcie Switch case v C. Funguje na základe načítania jednotlivých znakov v danom kóde, uložených v globálnej premennej "c" ktorá rozhodne o priechode do nového stavu v automate, až pokiaľ nesplníme podmienku v stave konečnom. V takomto prípade vrátime token v podobe štruktúry, ktorá ukladá jeho typ, poprípade k nemu priradenú hodnotu (ak existuje), čo je zaobstarané pomocou funkcie bufferToken, ktorá pri potrebných stavoch ukladá načítaváne znaky pre prípadný návrat. Do premennej c vždy načítame o znak navyše ako v danom tokene, aby pri ďalšom volaní funkcie getToken bola znovu pripravená na porovnávanie.

2.2 Syntaktický analyzátor (Parser)

Je jadrom prekladaču a riadi všetky ostatné časti. Je implementovaný pomocou rekurzívneho zostupu, vytvoreného na základe LL tabuľky (Príloha 3), okrem vyhodnocovania výrazov. Pre každý non-termínal z LL gramatiky (Príloha 2), je zostavená funkcia, ktorá postupne volá ďalšie funkcie na základe prichádzajúceho tokenu z lexikálneho analyzátoru. Tieto tokeny prichádzajú pomocou funkcie nextToken().

Parser nie je implementovaný striktne na základe LL gramatiky. V určitých prípadoch sa nedá rozhodnúť, či aktuálny token je, alebo nie je začiatkom výrazu. Preto si parser vždy uchováva predošlý token a niekedy sa musí aj "pozrieť" na

nasledujúci pomocou funkcie next_token_lookahead(). Tieto tokeny sú uložené v globálnych premenných previousToken a prípadne aheadToken. Tento mechanizmus funguje vďaka tomu, že v priebehu syntaktickej analýzy sa nevolá priamo next_token(), ale enhanced_next_token(). Pomocou tejto funkcie sa tak isto aj ignorujú tokeny so znakom EOL, ak sú viaceré po seba, alebo je nejaký na začiatku súboru. Čiže za každým príkazom bude práve jeden EOL.

V momente keď sa zistí, že aktuálny token, je začiatkom výrazu, tak sa token->type zmení na umelo vytvorený terminál expr. Vďaka tomuto môžeme normálne pokračovať v rekurzívnom zostupe. Nakoniec sa token->type prepíše na svoju pôvodnú hodnotu pomocou original_token_type_backup a riadenie sa predá analyzátoru výrazov pomocou funkcie analyze_expression. Po vyhodnotení celého výrazu vráti prečítaný token, ktorý už do výrazu nepatrí a ďalej pokračuje hlavný syntaktický analyzátor.

2.2.1 Precedenčná syntaktická analýza

Jadrom PSA je funkcia analyze_expression ktorú si volá parser pokiaľ narazí na výraz, ktorý je potreba vyhodnotiť. Analýza pozostáva z dvoch hlavných častí: kontrolovanie pravidiel analýzou zdola nahor pomocou zarážky a súčasné vytváranie postfixu (pomocou shunting-yard algoritmu). Obe využívajú precedenčnú tabuľku (Príloha 4). Výraz sa postupne spracováva volaním funkcie nextToken() až kým nenarazíme na jeden z konečných tokenov(EOF,EOL,then,do), ktorým sa dokončí syntaktická analýza a prejde sa ku generovaniu kódu, tento konečný token je parseru vrátený.

2.3 Sémantická analýza

V globálnej premennej global_symtable, sa ukladajú všetky názvy funkcií, ich parametre a premenné definované v hlavnom tele programu. Actual_symtable je buď totožná s global_symtable, alebo reprezentuje lokálnu tabuľku symbolov, ak sa analyzuje práve vnútro funkcie. Tam sa ukladajú lokálne premenné definované

vo funkcií a jej argumenty. Sémantické kontroly sa vykonávajú priebežne so syntaktickou analýzou.

2.4 Generovanie kódu

Výsledný kód IFJcode18 sa generuje do pomocnej štruktúry Tstring, ktorá je implementovaná ako obojsmerne viazaný list. Sú vytvorené 3 premenné s touto štruktúrou. Main_code_list predstavuje hlavné telo programu, príkazy medzi definíciami funkcií a functions_code_list obsahuje definície funkcií. Active_code_list, je vždy jedna z nich a určuje, do ktorého listu sa má v danom čase generovať kód.

Jeden uzol listu predstavuje ľubovoľnú časť z kódu. Do textu v danom uzle sa dá pripojiť nový text, alebo za uzol pridať nový uzol s textom. Navyše ak text v danom uzle predstavuje začiatok while cyklu (nie vnoreného), tak before_me_is_good_place_for_defvar je nastavené na true. Vďaka tomuto sa definície premenných generujú vždy pred toto miesto a nie vnútri cyklu. Funkcia find_nearest_good_place_for_defvar(), postupuje od konca listu a ak nájde uzol s takým označením, tak ho vráti.

Automaticky generované pomocné premenné a náveštia vždy obsahujú $, alebo % a pridanú globálnu premennú generic_label_count. Tá sa vždy inkrementuje po vygenerovaní takého názvu, čiže nikdy nemôže nastať kolízia.

Kód sa generuje priebežne so syntaktickou analýzou a ak preklad prebehol bez chýb generovaný kód sa vypíše na stdout pomocou print_code(). Vždy sa najprv vypíše functions_code_list a až potom main_code_list. Vďaka tomu, že niektoré uzly majú hodnotu is_start_of_new_line nastavenú na true, tak sa kód vypíše so správnym formátovaním.

2.4.1 Generovanie výrazov

Generovanie začína vo funkcii queue_evaluation, postupným vyhodnocovaním postfixu (postfix evaluation algorithm) uloženého v output_queue, na ktoré je potreba stack, pre uloženie operandov. Queue sa vyhodnocuje od jej začiatku, každý operand, na ktorý sa narazí je pushnutý na postfix_stack, keď narazíme na operátor tak sa zoberú dva vrchné operandy zo stacku, vykoná sa potrebná operácia a výsledok sa pushne naspäť na stack.

Ku generovaniu výrazov sme pristupovali tromi spôsobmi, záležiac na tom či sme poznali hodnotu operandu alebo nie. Funkcia semantics, pracuje s operáciou pri ktorej vieme hodnoty operandov, pokiaľ je jedna neznáma, tak sa rozhodne zavolať one_is_undefined_semantics a pokiaľ nevieme hodnotu ani jedného volá sa funkcia both_are_undefined. Pokiaľ hodnoty vieme, tak sa vykoná sémantická analýza, prípadne pretypovanie operandov, zvyšok sa rieši až vo výslednom troj adresnom kóde behovou kontrolou.

Pri väčšine operácií využívame datový zasobník interpretu, výnimkou je napríklad zlúčenie dvoch stringov alebo kontrolovanie či delitel nie je nula. Pomenovanie náveštia súvisí s jeho významom (zlúčenie názvov pri porovnávaní typov), obsahujú $ a jedinečné číslo generic_label_count.

3 Dátové štruktúry a špeciálne algoritmy

3.1 Binárny vyhľadávací strom

Ako spôsob implementovania symtable.c sme si zvolili binárny vyhľadávací strom. Premenné a funkcie sú ukladané do stromu na základe ich názvu, keďže vychádzame z faktu, že názvy musia byť jedinečné. Každý uzol teda obsahuje svoj jedinečný “key” a data ktoré vypovedajú o tom, či uzol reprezentuje funkciu a pokiaľ áno tak aj zoznam argumentov v podobe listu zadefinovaného v

param_list.c. Štruktúra obsahuje základné funkcie na vkladanie, hľadanie a vymazanie celého stromu a veľa ďalších pre uľahčenie práce s ním. Is_variable_defined, is_func_defined vracajú true, pokiaľ nájdu premennú alebo funkciu s daným názvom v strome. Add_variable_to_func_params slúži na vkladanie argumentov funkcie do listu a Is_variable_already_in_func_params vracia true pokiaľ je premenná už v liste vložená. Taktiež je tu funkcia get_name_of_defined_param_at_position vracajúca názov parametru na n-tej pozícii.

3.2 Pomocné štruktúry

3.2.1 Param_list

Param_list je naprogramovaný formou jednosmerného listu a slúži na ukladanie argumentov funkcie. Okrem klasických funkcií na obsluhu listu obsahuje aj get_nth_element, ktorá vracia n-tý element v liste.

3.2.2 Sym_stack

Stack potrebný pri precedenčnej analýze zdola nahor. Slúži na ukladanie adekvátnych tokenov a ich redukciu pomocou pravidiel. Okrem základných funkcií potrebných pre prácu s ním bolo vytvorených aj pár, ktoré umožňujú analýzu. Get_count_after_stop vracia počet symbolov od najvrchnejšej zarážky. Insert_after_top_terminal vloží symbol za najvrchnejší terminál.

3.3 Postfix_data_managment

3.3.1 Operator_stack

Stack slúžiaci na ukladanie operátorov pri prevode na postfix a ich následné premiestenie do output_queue pomocou funkcie pop_to_output_queue.

3.3.2 Output_queue

Štruktúra queue, ktorá je vlastne jednosmerne viazaný list (ne)priamym dočinením jednoho z naších programátorov, slúži na uloženie postfixu. Element

v tomto liste je struct s názvom P_item, ktorý obsahuje typ elementu, jeho hodnotu, taktiež bool is_operator, ktorý vypovedá o tom či je element operátorom. Okrem klasických funkcií obsahuje: determine_type_and_insert, ktorá zistí typ vkladaného elementu a patrične nastaví jeho hodnotu. Taktiež je tu funcia delete_first, ktorá vymaže prvý element v "queue".

3.3.3 Postfix_stack

Stack potrebný pri vyhodnotení operator_queue na uloženie operandov. Prvok v stacku je tiež typu P_item. Okrem klasických funkcií obsahuje: determine_type_and_push, ktora zistí typ vkladaného elementu a patrične nastaví jeho hodnoty. Obsahuje tiež funkciu, first_from_queue_to_stack, ktorá zoberie prvý element z queue a pushne ho na stack.

4 Práca v tíme

4.1 Verziovací systém a komunikácie

Pri tvorbe projektu rozdeľovanie práce prebiehalo počas tímových stretnutí. Projekt sme sa snažili rozdeliť čo najviac rovnomerne na podproblémy, pričom sme neustále medzi sebou komunikovali a konzultovali pomocou skupinových chatov. Na zdieľanie kódu sme využívali repozitár Git a na verziovanie GitHub.

4.2 Rozdelenie práce

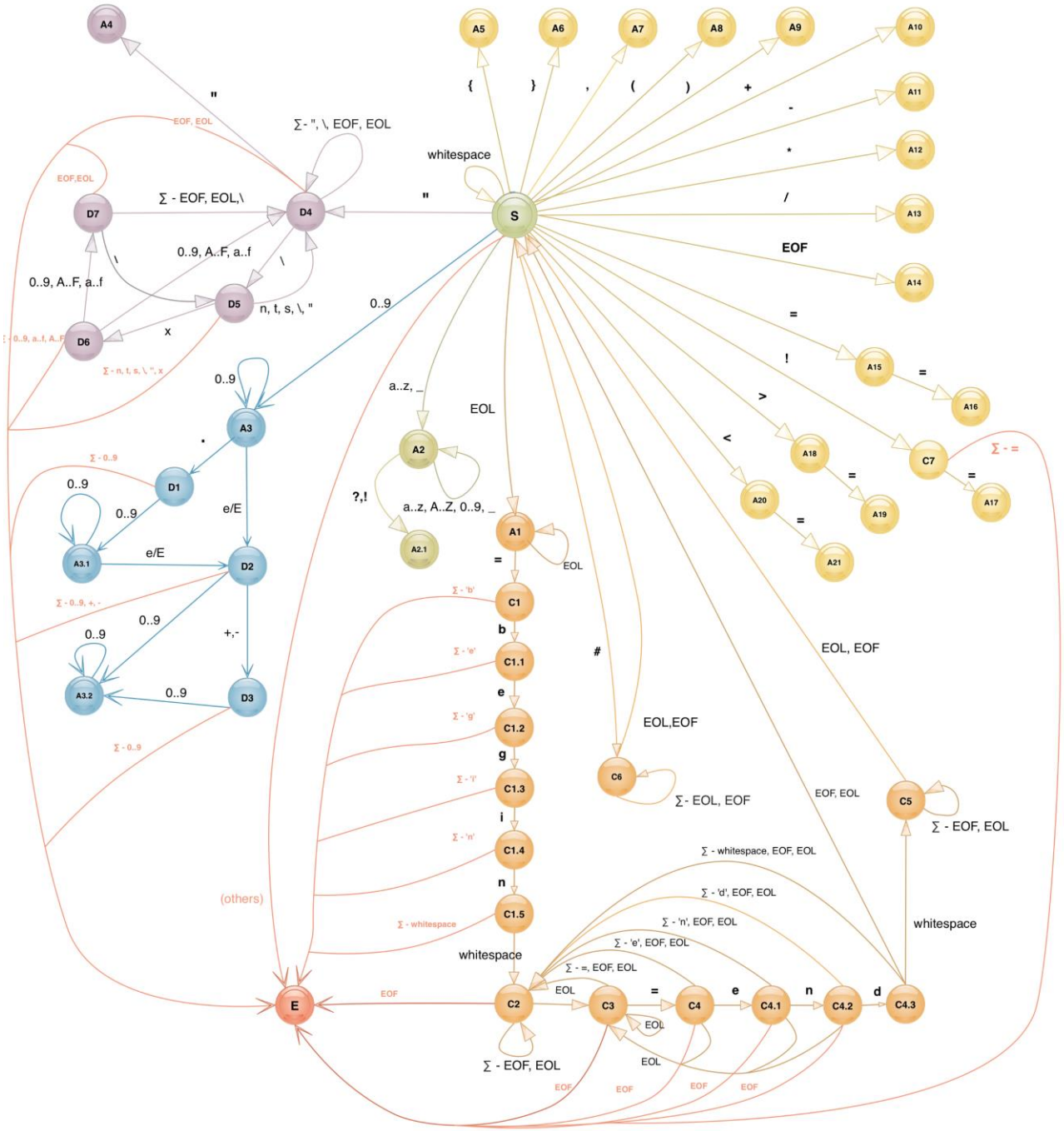
- 25% Glós Kristián: Vedenie tímu, organizácia práce, lexikálna analýza, dokumentácia
- 25% Abikenova Zhamilya: Lexikálna analýza, dokumentácia, testovanie
- 25% Vagala Dominik: Syntaktická analýza, generovanie kódu, testovanie, dokumentácia

- 25% Jakub Vinš: Syntaktická analýza, tabuľka symbolov, generovanie kódu.

5 Záver

Projekt bol definitívne rozsiahly a vyžadoval mnoho hodín práce a vedomostí, získaných najmä pri predmetoch IFJ a IAL, ale aj samoštúdiom. Riešenie projektu nám dalo veľa skúseností ohľadne práce v tíme a tvorení náročnejších programov.

1. Diagram konečného automatu specifikující lexikální analyzátor



Legenda:

S	START	D7	STRING-HEX_TWO
E	ERROR	A4	STRING
A1	EOL	A5	LEFT_CURLY_BRACKET
C1	BLOCK_COMMENT_SATRT	A6	RIGHT_CURLY_BRACKET
C1.1-C1.5	BEGIN	A7	COMMA
C2	COMMENT_INSIDE	A8	LEFT_BRACKET
C3	COMMENT_INSIDE_NEWLINE	A9	RIGHT_BRACKET
C4	BLOCK_COMMENT_TRY_TO_END	A10	PLUS
C4.1-C4.3	END	A11	MINUS
C5	BLOCK_COMMENT_END	A12	MULT
A2	IDENTIFIER_OR_KEYWORD	A13	DIV
A2.1	IDENTIFIER_FUNCTION	A14	EOF
A3	NUMBER	A15	ASSIGN
D1	NUMBER_POINT	A16	EQUALS
A3.1	NUMBER_DOUBLE	C7	NOT_EQUAL_START
D2	NUMBER_EXPONENT	A17	NOT_EQUAL
D3	NUMBER_EXP_SIGN	A18	GREATER_THAN
A3.2	NUMBER_EXP_FINAL	A19	GREATER_THAN_OR_EQUAL_TO
D4	STRING_START	A20	LESS_THAN
D5	SRTING_ESCAPE	A21	LESS_THAN_OR_EQUAL_TO
D6	STRING_HEX_ONE	C6	SINGLE_COMMENT

2. LL – gramatika

1. Prog -> Main_body
2. Main_body -> State Main_body
3. Main_body -> Def_func Main_body
4. Main_body -> eps
5. State -> id After_id eol
6. After_id -> = Func_or_expr
7. Func_or_expr -> Call_func
8. Call_func -> id Call_func_args
9. Func_or_expr -> Expr
10. After_id -> Call_func_args
11. Call_func_args -> eps
12. Call_func_args -> (Arg_in_brackets)
13. Call_func_args -> Term More-args
14. Arg_in_brackets -> eps
15. Arg_in_brackets -> Term More-args
16. More-args -> , Term More-args
17. More-args -> eps
18. Term -> id
19. Term -> int
20. Term -> double
21. Term -> string
22. Term -> nil
23. Expr -> expr
24. State -> Expr eol
25. State -> if Expr then eol St_list else eol St_list end eol
26. State -> while Expr do eol St_list end eol
27. Def_func -> def id (Param) eol St_list end eol
28. St_list -> State St_list
29. St_list -> eps
30. Param -> eps
31. Param -> id More_params
32. More_params -> , id More_params
33. More_params -> eps

3. LL – tabuľka

	id	eol	=	(	)	,	int	double	string	nil	if	then	else	end	while	do	def	$
Prog	1										1				1		1	1
Main_body	2										2				2		3	4
State	5										25				26			
Def_func																	27	
After_id	10	10	6	10			10	10	10	10								
Func_or_expr	7																	
Call_func	8																	
Call_func_args	13	11		12			13	13	13	13								
Arg_in_brackets	15				14		15	15	15	15								
Term	18						19	20	21	22								
More-args		17			17	16												
St_list	28										28		29	29	28			
Param	31				30													
More_params					33	32												

4. Precedenčná tabuľka

	+ -	/ *	r	(	)	I	$
+ -	>	<	>	<	>	<	>
/ *	>	>	>	<	>	<	>
r	<	<		<	>	<	>
(	<	<	<	<	=	<	
)	>	>	>		>		>
I	>	>	>		>		>
$	<	<	<	<		<	