Projektová dokumentácia

Implementace překladače imperativního jazyka IFJ21

Tým 040, varianta II

Kičinka Kristian (xkicin02): 30%

Ľupták Andrej (xlupta05): 30%

Majer Michal (xmajer22): 20%

Skopár Lukáš (xskopa16): 20%

7.12.2021

Obsah

Úvod

Cieľom projektu bolo vytvoriť program v jazyku C, ktorý načíta zdrojový kód zapísaný v zdrojovom jazyku IFJ21 a preloží ho do cieľového jazyka IFJcode21 (medzikód). Prekladač načíta riadiaci program v jazyku IFJ21 zo štandardného vstupu a generuje výsledný medzikód v jazyku IFJcode21 na štandardný výstup.

Návrh a implementácia

Lexikálna analýza

Lexikálny analyzátor rozpoznáva súvislú, logicky oddelenú, postupnosť znakov, ktoré tvoria lexém. Analýza spracuváva tokeny, ktoré následne posiela syntaktickej analýze.

Implementácia

Implementovali sme to pomocou konečného stavového automatu, v ktorom sme si vytvorili niekoľko stavov. Tieto stavy sú definované v štruktúre `Fsm_states`. Vytvorili sme si štruktúru `Token`, zasielame ju v rámci parametra do syntaktickej analýzy, ktorá s ním pracuje. V rámci implementačného súboru lexikálnej analýzy máme vytvorených niekoľko funkcií na spracovanie integer-u, number-u, keywords a pod. Hlavnou funkciou je `generate_token`, ktorá príjma `Token`, do ktorého sa vkladá nový token a príjma `Custom_string`. `Custom_string` je namieru vytvorená štruktúra, slúži na ukladanie stringových hodnôt. Je to dynamicky alokovateľný string cez malloc, ktorý rieši problém s dĺžkami string-u.

Funkcia generate_token

Aby syntakticka analýza vedela vypýtať ďalší token z lexikálnej analýzy, tak zavolá a použije funkciu `generate_token`. Následne do nej vloží token, do ktorého sa vkladá informácia zo vstupu. Lexikálna analýza prečíta ďalší znak zo vstupu, spracuje ho, a prepošle ho do tokenu do syntaktickej analýzy.

Syntaktická analýza

Spracuváva výrazy od scanner-u , vyhodnocuje syntaktickú správnosť zapísaného výrazu.

Implementácia

Použili sme rekurzívne spracovanie. Vytvorili sme si LL-gramatiku, na základe ktorej vyhodnocujeme jednotlivé pravidlá. V rámci syntaktickej a semantickej analýzy sme si zadefinovali štuktúru, ktorá je v súbore `enums.h`. Sem ukladáme vyhodnotené pravidlá. `syntactic_data_t` je štruktúra, ktorá slúži na preposielanie dát medzi výrazovou analýzou a hlavnou syntaktickou analýzou. Využili sme pri tom štruktúru `Token`. Pri spracováni analýzy využívame lokálnu a globálnu tabuľku. Lokálna tabuľka slúži na ukladanie premenných a globálna tabuľka slúži na ukladanie funkcií. Priamo z analýzy sa volá generátor. Na vracanie tokenov z analýzy výrazov späť do syntaktickej analýzy používame `token_list`. Obsahuje ukazatele na tokeny.

Tabuľka symbolov

Je hashovacia tabuľka, ktorá obsahuje informácie o premenných, funkciach, teda názvy, dátové typy, návratové hodnoty a pod. Parametre a návratové hodnoty funkcií sú reprezentované štruktúrou `Data_list` (v súbore `function_data_list.h`), ktorá je dynamicky alokovateľné pole tokenových typov.

V rámci súboru recursive_syntactic.c sa overuje jak syntax, tak aj semantika

Analýza výrazov

Je implementovaná v súbore expression_analysis.c a jej rozhranie v súbore expression_analysis.h. Je volaná syntaktickou analýzou, keď očakávaným vstupom je výraz. Precedenčná analýza používa pri spracovaní výrazov precedenčnú tabuľku. Jej prvý index symbolizuje tokeny uložené na stacku a druhý index symbolizuje prichádzajúci token. Indexácia je vykonaná pomocou enum štruktúry PSA_symbol. Vždy pri prijatí tokenu si precedenčná analýza zistí pomocou funkcie symbol_from_token, či s prijatým tokenom dokáže pracovať, alebo nie. Ak áno priradí mu správny PSA_symbol, pomocou ktorého sa bude indexovať tabuľka. Po spracovaní prijatého tokenu sa zavolá funkcia generate_token, ktorá nám vráti nasledujúci token. Výraz sa postupne spracováva, pokiaľ nenarazíme na token, s ktorým nedokáže pracovať (napr. token if, while...), token je poslaný napäť do syntaktickej analýzy a precedenčná analýza skontroluje pomocou precedenčnej tabuľky tokeny, ktoré už prijala a v prípade správneho prijatého výrazu sa úspešne ukončí. Na vracanie tokenov z analýzy výrazov späť do syntaktickej analýzy používame token_list. Obsahuje ukazatele na tokeny.

Sémantická analýza

Sémantika sa overuje v súbore recursive_syntactic.c, na základe načítaného tokenu a záznamu v v tabuľke symbolov. Keď narazí na syntaktický alebo semantický error, tak sa vypíše validnu error správu a ukončí program.

Generovanie cieľového kódu

Generátor kódu, ktorého rozhranie je uložené v súbore generator.h a implementácie v súbore generator.c. Obsahuje funkcie, ktoré sú volané v rámci syntaktickej analýzy, po overení syntaxe a semantiky.

Chybové stavy

Zachytenie chybových stavov máme poriešené v rozhraní error.h, kde sa nachádza enum štruktúra Error_type, na základe definovaných stavov, ktorá má vlastné pridelené číslo. Následne sme vytvorili implementačný súbor error.c, ktorý obsahuje funkciu process_error, do ktorej pridelíme enum hodnotu s typom erroru. Táto funkcia vyhodnotí error, vypíše chybovú hlášku a skončí program s korektnou návratovou hodnotou.

Práca v tíme

Rozloženie práce

Prvé stretnutie ohľadom projektu a rozdelenia práce sme mali v polovici októbra, kedy sme si rozvrhli jednotlivé časti zadania. Postupne sme pracovali na častiach, zároveň s prednášanými študijnými oporami.

Práca

Všetci členovia programovali vo visual studio code . Súbory s kódom posielali a aktualizovali cez GitHub. Takto sme pracovali na viacerých častiach zároveň a udržiavali tempo práce.

Komunikácia

Stretávali sme sa osobne v priestoroch fakulty aspoň raz týždenne. Postupom času pribúdali tímové meetingy online cez aplikáciu Discord, kde sme riešili aktuálne problémi s kódom. Pomocou tejto aplikácie sme aj jednotlivo komunikovali medzi sebou.

Rozdelenie práce v tíme

Kristian Kičinka - Vedúci tímu, lexikálna analýza, tabuľka symbolov, pomocné štruktúry

Andrej Ľupták - syntaktická analýza, sémantická analýza, LL tabuľka

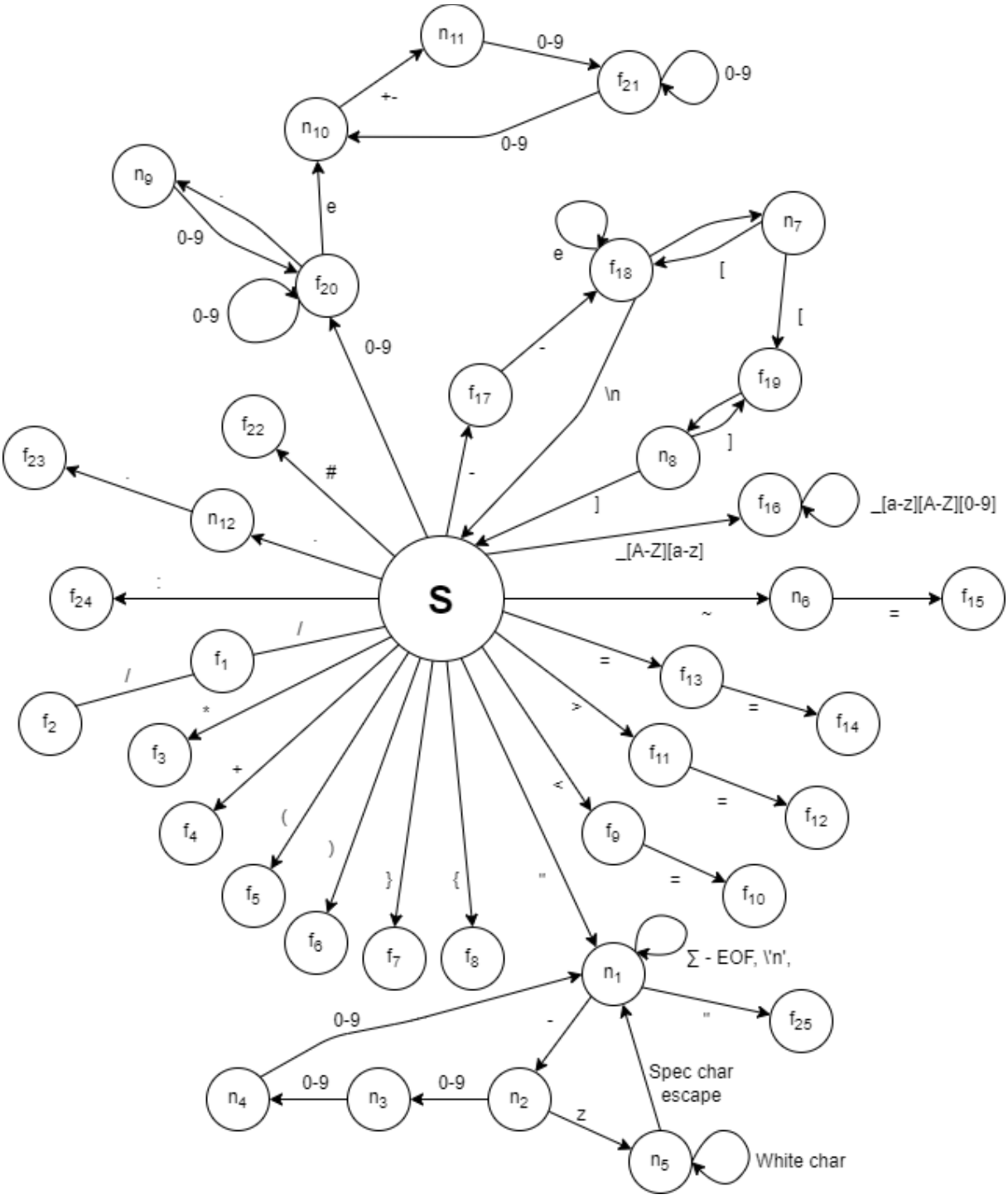
Michal Majer - sémantická analíza, tabuľka výrazov

Lukáš Skopár - generovanie cieľového kódu, dokumentácia

Záver

Najviac času na projekte nám zabralo pochopenie a porozumenie zadania. Počas práce na projekte sme museli niekoľko krát prepisovať kód, pretože sme sa dozvedeli, že to daným spôsobom nemôže byť alebo nebude fungovať. Keďže zadanie bolo zložité, mali sme problém v komunikácií pri tvorbe a prepájaní jednotlivých súborov. Často sme uviazli v bode, kde sme museli premyslieť následovný postup. Na projekte sme pracovali do poslednej chvíle, počas vývoja sme si prešli viacerými obtiažnými situáciami a vynaložili sme všetkú snahu tento projekt vyriešiť a zvládnuť.

Diagram konečného automatu, ktorý špecifikuje lexikálny analizátor



Legenda

S	Start	n_5	Escape seq. white	f_{19}	Start block com. f
f_1	Backslash	f_9	Less than	n_8	End block com. n
f_2	Backslash	f_{10}	Less than or equals	f_{20}	Number
f_3	Multiplicate	f_{11}	More than	n_9	Number dot
f_4	Plus	f_{12}	More than or equals	n_{10}	Number exponent
f_5	Left round bracket	f_{13}	Equals	n_{11}	Number expo. sign
f_6	Right round bracket	f_{14}	Type ekv	f_{21}	Numb. expo. number
f_7	Right curly bracket	n_6	Not equal	f_{22}	Hashtag
f_8	Left curly bracket	f_{15}	Type ekv	n_{12}	Concat
n_1	Escape sequence	f_{16}	Identi. or keyword	f_{23}	String concat
n_2	First number o ES	f_{17}	Minus	f_{24}	Colon
n_3	Second number o ES	f_{18}	Line commentary	f_{25}	Escape seq. final
n_4	Third number o ES	n_7	Start block com. n		

LL - Gramatika

0 - <prolog_nt> -> kw_require string_const <start>

1 - <start_nt> -> <func_call_nt> <start_nt>

2 - <start_nt> -> <func_dec_nt> <start_nt>

3 - <start_nt> -> <func_nt> <start_nt>

4 - <start_nt> -> end

5 - <code_nt> -> <while_nt> <code_nt>

6 - <code_nt> -> <if_nt> <code_nt>

7 - <code_nt> -> <func_call_nt> <code_nt>

8 - <code_nt> -> <return_nt> <code_nt>

9 - <code_nt> -> <assign_existing_nt> <code_nt>

10 - <code_nt> -> <assign_new_nt> <code_nt>

11 - <code_if_nt> -> <code_nt> <code_if_nt>

code_nt -> eps

12 - <while_nt> -> kw_while <expression_nt> kw_do <code_nt> kw_end

13 - <if_nt> -> kw_if <expression_nt> kw_then <code_if_nt> kw_else <code_nt> kw_end

14 - <func_dec> -> kw_global func_id : kw_function <(> param_nt <)> <double_dot_nt> <start>

15 - <double_dot_nt> -> : <datatype double_dots_nt>

<double_dot_nt > -> eps

16 - <double_dots_nt> -> , <datatype double_dots_nt>

<double_dots_nt> -> eps

17 - <param_nt> -> <datatype params_nt>

<param_nt> -> eps

18 - <params_nt> -> , <datatype params_nt>

<params_nt> -> eps

19 - <check_eof_nt> -> end

20 - <check_eof_nt> -> <start>

21 - <func_nt> -> kw_function func_id <(> arg_nt <)> <double_dot_nt> <code_nt> kw_end <check_eof_nt>

22 - <arg_nt> -> id : <datatype argument_nt>

<arg_nt> -> eps

23 - <argument_nt> -> , id : datatype <argument_nt>

<argument_nt> -> eps

24 - <return_nt> -> kw_return <check_ret_params_nt>

25 - <check_ret_params_nt> -> <value returns_nt>

<check_ret_params_nt> -> eps

26 - <returns_nt> -> , <value returns_nt>

<returns_nt> -> eps

27 - <func_call_nt> -> func_id <(> call_param_nt <)> <check_eof_nt>

28 - <call_param_nt> -> <value call_params_nt>

<call_param_nt> -> eps

29 - <call_params_nt> -> , <value call_params_nt>

<call_params_nt> -> eps

30 - <assign_existing_nt> -> <to_assign_nt> = <assign_value_nt>

31 - <to_assign_nt> -> id <to_assign2_nt>

32 - <to_assign2_nt> -> , id <to_assign2_nt>

to_assign2_nt -> eps

33 - <assign_value_nt> -> <expression_nt> <assign_values_nt>

34 - <assign_value_nt> -> <func_call_nt> <assign_values_nt>

assign_values_nt -> eps

35 - <assign_values_nt> -> , <assign_from>

36 - <assign_from> -> <value assign_values_nt>

37 - <assign_from> -> <func_call_nt> <assign_values_nt>

38 - <assign_new_nt> -> kw_local id : <datatype> <optional_ekv_nt>

39 - <assign_nt> -> <expression_nt>

40 - <assign_nt> -> <func_call_nt>

41 - <value> -> int_const

42 - <value> -> number_const

43 - <value> -> string_const

44 - <value> -> id

45 - <assval_nt> -> int

46 - <assval_nt> -> number

47 - <assval_nt> -> string

48 - <assval_nt> -> id

49 - <assval_nt> -> nil

50 - <datatype> -> int

51 - <datatype> -> number

52 – <datatype> -> string

53 - end -> <token_eof_t>

55 - <expression> -> <value>

69 - <check_eof_nt> -> <code_nt>

70 - <value> -> nil

71 - <optional_ekv_nt> -> = <assign_nt>

72 - <optional_ekv_nt> -> <code_nt>

LL - tabuľka

	REQUIRE	DO	ELSE	END	FUNCTION	GLOBAL	IF	LOCAL	NIL	NUMBER
<prolog>	0									
<value_nt>										
<assval_nt>									49	46
<datatype_nt>										51
<expression_nt>										
<func_dec>						14				
<param_nt>										17
<params_nt>										
<func_nt>					21					
<arg_nt>										
<argument_nt>										
<code_nt>							6	10		
<return_nt add>										
<check_ret_params_ntE>										
<returns_nt>										
<func_call_nt>										
<call_param_nt>										
<call_params_nt>										
<assign_existing_nt>										
<to_assign_nt>										
<assign_value_nt>										
<assign_values_nt>										
<assign_from_nt>										
<to_assign2_nt>										
<assign_new_nt>								38		
<assign_nt>									70	
<while_nt>										
<if_nt>							13			
<start>				4	3	2				
<double_dot_nt>										
<double_dots_nt>										
<check_eof_nt>			69	19	20	20	69	69		
<code_if_nt>							11	11		
<end_nt>										
<optinal_ekv_nt>							72	72	70	

	RETURN	THEN	WHILE	SEMICOLN	R_BRACKET	L_BRACKET	COMMA	DOUBLEDOT	ASSIGN	EOF
<prolog>										
<value_nt>										
<assval_nt>										
<datatype_nt>										
<expression_nt>										
<func_dec>										
<param_nt>										
<params_nt>							18			
<func_nt>										
<arg_nt>										
<argument_nt>							23			
<code_nt>	8		5							
<return_nt add>	24									
<check_ret_params_ntE>										
<returns_nt>							26			
<func_call_nt>										
<call_param_nt>										
<call_params_nt>							29			
<assign_existing_nt>										
<to_assign_nt>										
<assign_value_nt>										
<assign_values_nt>							35			
<assign_from_nt>										
<to_assign2_nt>							32			
<assign_new_nt>										
<assign_nt>										
<while_nt>			12							
<if_nt>										
<start>										4
<double_dot_nt>								15		
<double_dots_nt>							16			
<check_eof_nt>	69									53
<code_if_nt>	11		11							
<end_nt>										53
<optinal_ekv_nt>	72		72							

	ID	FUNC_ID	INT	STRING	STRING_CONST	INT_CONST	NUMBER_CONST	PLUS	MINUS	MULTIPLY
<prolog>										
<value_nt>	44				43	41	42			
<assval_nt>	48		45	47						
<datatype_nt>			50	52						
<expression_nt>	55				55	55	55			
<func_dec>										
<param_nt>			17	17						
<params_nt>										
<func_nt>										
<arg_nt>	22									
<argument_nt>										
<code_nt>	9	7								
<return_nt add>										
<check_ret_params_ntE>	25				25	25	25			
<returns_nt>										
<func_call_nt>		27								
<call_param_nt>	28				28	28	28			
<call_params_nt>										
<assign_existing_nt>	30									
<to_assign_nt>	31									
<assign_value_nt>	33	34			33	33	33			
<assign_values_nt>										
<assign_from_nt>	36	37			36	36	36			
<to_assign2_nt>										
<assign_new_nt>										
<assign_nt>	39	40			39	39	39			
<while_nt>										
<if_nt>										
<start>		1								
<double_dot_nt>										
<double_dots_nt>										
<check_eof_nt>	30	20								
<code_if_nt>	11	11								
<end_nt>										
<optinal_ekv_nt>	72	72								

	DIV	DOUBLEDIV	CONCAT	LESS	LE	GREATER	GE	NOTEQUAL	EQUAL
<prolog>									
<value_nt>									
<assval_nt>									
<datatype_nt>									
<expression_nt>									
<func_dec>									
<param_nt>									
<params_nt>									
<func_nt>									
<arg_nt>									
<argument_nt>									
<code_nt>									
<return_nt add>									
<check_ret_params_ntE>									
<returns_nt>									
<func_call_nt>									
<call_param_nt>									
<call_params_nt>									
<assign_existing_nt>									
<to_assign_nt>									
<assign_value_nt>									
<assign_values_nt>									
<assign_from_nt>									
<to_assign2_nt>									
<assign_new_nt>									
<assign_nt>									
<while_nt>									
<if_nt>									
<start>									
<double_dot_nt>									
<double_dots_nt>									
<check_eof_nt>									
<code_if_nt>									
<end_nt>									
<optinal_ekv_nt>									71

Precedenčná tabuľka

stdin → / stack ↓	+	-	*	/	//	..	#	<	<=	>	>=	==	~=	ID	(	)	$$
+	>	>	<	<	<	*	<	>	>	>	>	>	>	<	<	>	>
-	>	>	<	<	<	*	<	>	>	>	>	>	>	<	<	>	>
*	>	>	>	>	>	*	<	>	>	>	>	>	>	<	<	>	>
/	>	>	>	>	>	*	<	>	>	>	>	>	>	<	<	>	>
//	>	>	>	>	>	*	<	>	>	>	>	>	>	<	<	>	>
..	*	*	*	*	*	>	*	*	*	*	*	>	>	<	<	>	>
#	>	>	>	>	>	*	*	>	>	>	>	>	>	<	<	>	>
<	<	<	<	<	<	*	<	*	*	*	*	*	*	<	<	*	>
<=	<	<	<	<	<	*	<	*	*	*	*	*	*	<	<	*	>
>	<	<	<	<	<	*	<	*	*	*	*	*	*	<	<	*	>
>=	<	<	<	<	<	*	<	*	*	*	*	*	*	<	<	*	>
==	<	<	<	<	<	<	<	*	*	*	*	*	*	<	<	*	>
~=	<	<	<	<	<	<	<	*	*	*	*	*	*	<	<	*	>
ID	>	>	>	>	>	>	*	>	>	>	>	>	>	*	*	>	>
(	<	<	<	<	<	<	<	*	*	*	*	*	*	<	<	=	*
)	>	>	>	>	>	>	*	>	>	>	>	>	>	*	*	>	>
$$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	*	until

<	PUSH ADD	PA
>	REDUCE	R
=	PUSH ADD	P
*	ERROR	E