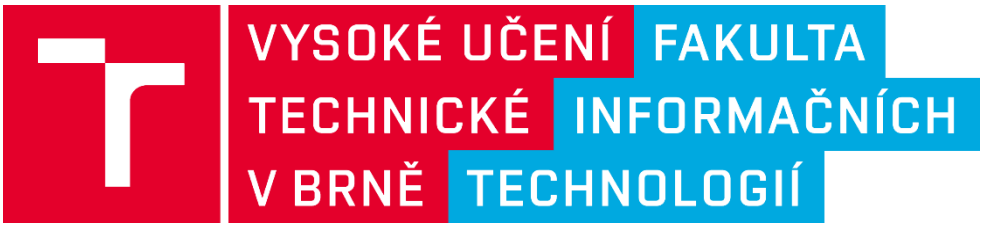


Projektová dokumentácia prekladača jazyka IFJ21

Tím 141, varianta I

Denis Horil - xhoril01 : 24 %

Tomáš Lukáč - xlukac16 : 28 %

Lukáš Macejka - xmacej03 : 24 %

Juraj Mariani - xmaria03 : 24 %

3.12.2021

Obsah

Úvod:

Cieľom projektu bolo vytvoriť program v jazyku C, ktorý načíta zdrojový kód zapísaný v zdrojovom jazyku IFJ21, ktorý je podmnožinou jazyka Teal a preloží ho do cieľového jazyka IFJcode21(medzi kód).

Prekladač funguje ako konzolová aplikácia bez grafického užívateľského rozhrania, ktorá načíta riadiaci program zo štandardného vstupu a generuje výsledný medzi kód na štandardný výstup, alebo v prípade chybného hlásenia, varovania a ladiacich výpisov prevedie na štandardnom chybnom výstupe.

Návrh a rozdelenie práce:

Projekt sme si rozdelili na niekoľko menších častí a každá bola pridelená podľa skúseností člena v daných častiach:

Člen	**Práca – rozdelené časti projektu**
Denis Horil Tomáš Lukáč Lukáš Macejka **Juraj Mariani**	*-Lexikálna analýza, prezentácia* *- Syntaktická analýza, Sémantická analýza* *- Generovanie kódu, testy, dokumentácia* *-Generovanie kódu, Syntaktická analýza*

Lexikálna analýza:

Hlavnou funkciou lexikálnej analýzy je funkcia *GetNextToken*, ktorá má ako parameter štruktúru *token*, do ktorej sa uloží získaný token. Funkcia GetNextToken číta zo štandardného vstupu znak po znaku a znaky ukladá do pomocnej premennej, ktorá tieto znaky na konci analýzy uloží do štruktúry token. Štruktúra *token* je zložená z 2 častí - typ tokenu a atribút.

Typ tokenu môže byť identifikátor, kľúčové slovo, konštanta (celé či desatinné číslo alebo reťazový literál), operátory (aritmetické a porovnávacie) alebo zvyšné znaky, ktoré sa môžu vyskytnúť v jazyku IFJ21 (napr. zátvorky).

Atribút je štruktúra tokenData obsahujúca konkrétne dáta, ale taktiež aj typ dát, pre lepšie spracovanie v ostatných častiach prekladača. Pre lepšiu komunikáciu medzi lexikálnou a syntaktickou analýzou je štruktúra tokenData zložená z viacerých enumerátorov.

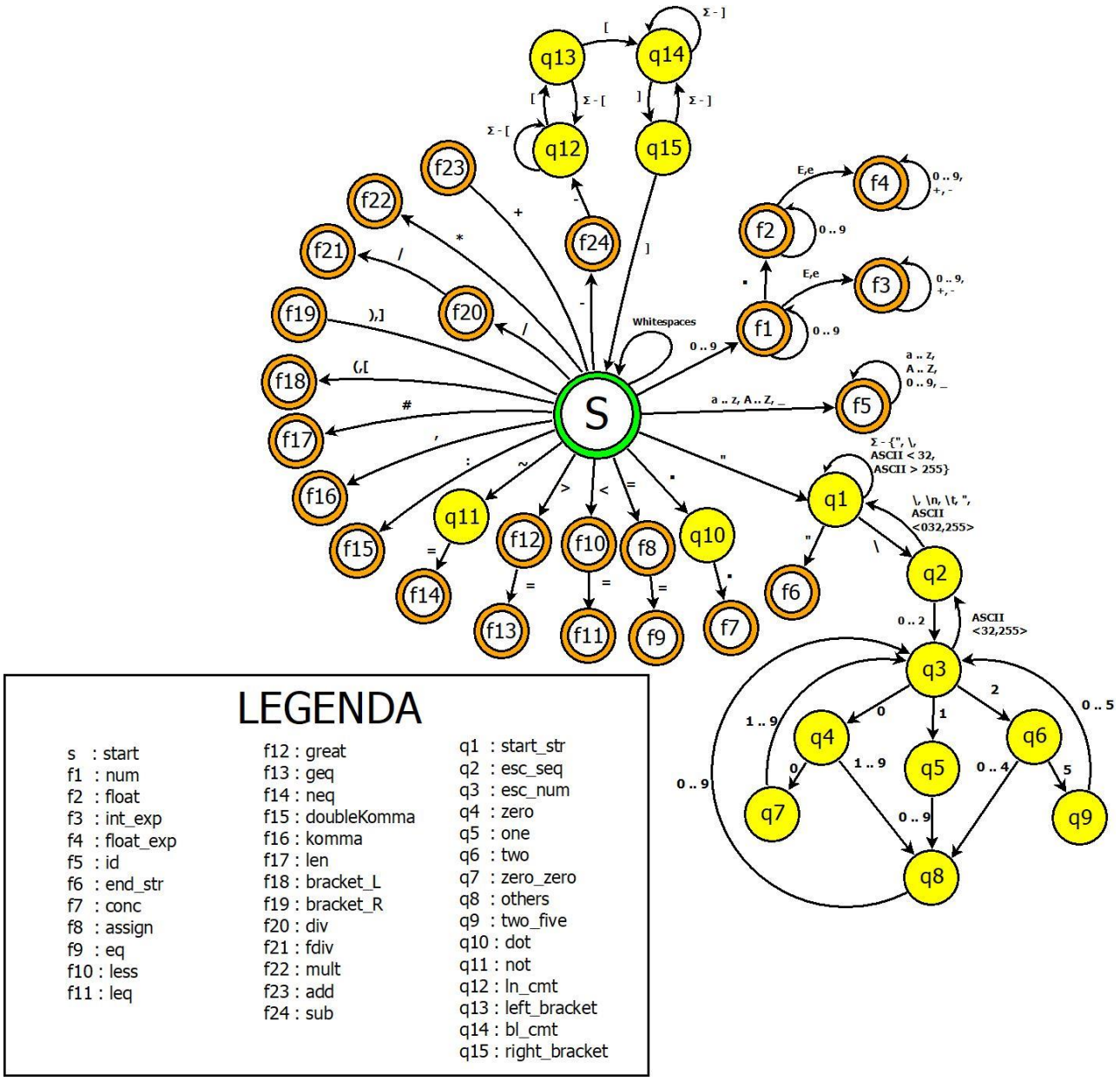
V prípade, že bude typ tokenu identifikátor alebo konštanta, tak typ dát bude string (integer v prípade celého čísla alebo number(float) v prípade des. čísla) a atribút daný reťazec.
Ak typ tokenu bude kľúčové slovo, operátor alebo iný znak, potom do atribútu bude priradené číslo z konkrétneho enumerátora.

Identifikátory sa načítavajú do dynamického poľa a akonáhle sa ukončuje stav identifikátoru, vyvolá sa funkcia Is_Keyword, ktorá porovná daný identifikátor s poľom kľúčových slov. Ak sa nachádza identifikátor v tomto poli, funkcia vracia číslo indexu kľúčového slova v danom poli, čo je zároveň aj číslo v enumeratóre, inak vracia -2.

Escape sekvencie v tvare "\ddd" sa spracovávajú vo vlastných funkciách podľa číslic na jednotlivých pozíciách. Pre zápis týchto číslic je vytvorené pole znakov veľkosti 3, ktoré sa po načítaní 3. číslice prekonvertuje na znak s odpovedajúcim ASCII kódom a zapíše sa do reťazového literálu. Ostatné escape sekvencie prepíšu znak '\' v reťazovom literáli na odpovedajúci znak danej escape sekvencie(napr. namiesto "\n" sa zapíše '\n'). Takto uložený token potom pokračuje do ďalších častí prekladača.

Lexikálny analyzátor je implementovaný ako deterministický konečný automat podľa vopred vypracovaného diagramu. Tento konečný automat je rozdelený na niekoľko okruhov (perimetrov) podľa stavov automatu - v hlavnej funkcii sa podľa vstupného znaku určí stav, do ktorého sa automat prepne a presunie sa do prvého okruhu. Tu sa automat ukončí a vráti výsledný token alebo načítava ďalšie znaky zo vstupu a podľa toho sa rozhoduje, či sa automat prepne do ďalšieho stavu (a teda aj do ďalšieho okruhu), alebo zostáva v danom stave. Ak je na vstupe znak, ktorý nepatrí do abecedy daného stavu, znak sa vráti späť na vstup, automat sa ukončí a vráti sa token. V niektorých prípadoch sa program ukončí a vráti sa nula(napr. keď je v reťazovom literáli ENTER a v číslicovom literáli iný znak ako číslo (okrem ε)).

Diagram konečného automatu pre lexikálny analyzátor:



Syntaktická analýza:

Syntaktická analýza sa riadi podľa pravidiel špecifikovaných priloženou LL gramatikou. Od lexikálnej analýzy preberá naplnený token funkciou GetNextToken a následne postupuje zostupne podľa pravidiel.
V prípade že nájde začiatok výrazu, analýza zavolá funkciu *F_Expression*, prepne sa do režimu prediktívnej analýzy a čaká na návrat z funkcie.
Okrem toho analýza priamo volá funkcie na generovanie kódu.
V prípade, že analýza nájde hlavičku funkcie alebo definíciu premennej, zavolá funkciu tabuľky symbolov, skontroluje či nejde o redefiníciu a zapíše.
V prípade neočakávaného tokenu sa volá makro *Rerror* s kódom chyby.
Na zápis parametrov a výstupov funkcie je použitý array, kde enumerator_ender symbolizuje koniec poľa.
Na predávanie hodnôt medzi analýzami, premennými a funkciami je použitá štruktúra *expression_block*. Táto štruktúra nesie informáciu o dátovom type v nej, informácie na vygenerovanie identifikátoru jej unikátnej, a tiež ďalšie informácie použité v prediktívnej analýze.
Ďalej je použitá štruktúra *BubbleStack*, čo je zásobník odkazov na štruktúru expression_block. Je použitá na vyhodnocovanie funkcii a priradení s viacerými premennými.
Na tento účel sa používa aj štruktúra VL, ktorá je array odkazov na elementy v tabuľke symbolov (TreeElement) končiaca odkazom NULL.

LL tabuľka:

	global	function	fID	ID	,	:	Integer	Number	String	local	=	(	)	Const	#	while	if	else	nil	return	end	EOF
<prog>	1	2	3																			4
<params>				6			5	5	5				5						5			
<second_param>					8								9									
<param_type>				10									11									
<second_param_type>					12								13									
<params_r>	15	15	15	15		14										15	15			15	15	
<second_param_r>	17	17	17	17	16											17	17			17	17	
<type>							20	19	18										21			
<st-list>			22	22						22						22	22			22	23	
<statement>			28	25						24						26	27			29		
<else>																		31			30	
<expr_afterloc>			35	35						35	36					35	35	35		35	35	
<exprb>			43	44								44		44	44				44			
<sent_par>			37	37								37	38	37	37							
<s_par>					39								40									
<s_expr>			42	42	41					42						42	42	42		42	42	
<s_var>					45						46											
<ret>			32	33								33		33	33				33		34	

LL gramatika:

(1) PROG → global fID ':' function '(' PARAMS ')' PARAMS_R PROG
(2) PROG → function fID '(' PARAM_TYPE ')' PARAMS_R ST_LIST 'end' PROG
(3) PROG → fID '(' ')' PROG
(4) PROG → EOF
(5) PARAMS → TYPE SECOND_PARAM 0
(6) PARAMS → ID ':' TYPE SECOND_PARAM
(7) PARAMS → ε
(8) SECOND_PARAM → ',' PARAMS
(9) SECOND_PARAM → ε
(10) PARAM_TYPE → ID ':' TYPE SECOND_PARAM_TYPE
(11) PARAM_TYPE → ε
(12) SECOND_PARAM_TYPE → ',' ID ':' TYPE SECOND_PARAM_TYPE
(13) SECOND_PARAM_TYPE → ε
(14) PARAMS_R → ':' TYPE SECOND_PARAM_R
(15) PARAMS_R → ε
(16) SECOND_PARAM_R → ',' TYPE SECOND_PARAM_R
(17) SECOND_PARAM_R → ε
(18) TYPE → string
(19) TYPE → number
(20) TYPE → integer
(21) TYPE → nil
(22) ST_LIST → STATEMENT
(23) ST_LIST → ε
(24) STATEMENT → local ID ':' TYPE EXPR_AFTERLOC ST_LIST
(25) STATEMENT → ID S_VAR '=' EXPR_B S_EXPR ST_LIST
(26) STATEMENT → while EXPR_B do ST_LIST end ST_LIST
(27) STATEMENT → if EXPR_B then ST_LIST ELSE end ST_LIST
(28) STATEMENT → fID '(' SENT_PAR ')' STATEMENT
(29) STATEMENT → return RET
(30) ELSE → ε
(31) ELSE → else ST_LIST
(32) RET → fID '(' SENT_PAR ')' S_EXPR
(33) RET → expression S_EXPR
(34) RET → ε
(35) EXPR_AFTERLOC → ε
(36) EXPR_AFTERLOC → '=' EXPR_B
(37) SENT_PAR → EXPR_B S_PAR
(38) SENT_PAR → ε
(39) S_PAR → ',' EXPR_B S_PAR
(40) S_PAR → ε
(41) S_EXPR → ',' EXPR_B S_EXPR
(42) S_EXPR → ε
(43) EXPR_B → fID '(' SENT_PAR ')'
(44) EXPR_B → expression
(45) S_VAR → ',' ID S_VAR
(46) S_VAR → ε

Sémantická analýza:

Sémantická analýza sa riadi precedenčnou tabuľkou. Využíva zásobník BubbleStack na štruktúry expressionBlock.
Pre sémantickú analýzu je dôležité, že do expressionBlocku sa dajú uložiť aj znaky konca zásobníka, zátvoriek a operátorov. V prípade sémantickej chyby sa vyvolá makro *comError* s exit kódom.

Analýza priamo volá funkcie na generáciu kódu a funkcie tabuľky symbolov, pričom vždy spracuje len jeden výraz a potom odovzdá riadenie syntaktickej analýze. Absencia znaku konca riadka je riešená upravenou precedenčnou tabuľkou a množinou výrazov použitých ako koncová množina.
Výstup zo sémantickej analýzy je jeden expressionBlock, ktorý obsahuje informácie na vygenerovanie identifikátora.
O priradenie do premennej a volanie funkcie, okrem spracovania parametrov, sa stará syntaktická analýza.

Precedenčná tabuľka:

	#	*,/,//	+,-	..	relačné	(	)	i	$
#	>	>	>	>	>	<	>	<	>
*,/,//	<	>	>	>	>	<	>	<	>
+,-	<	<	>	>	>	<	>	<	>
..	<	<	<	>	>	<	>	<	>
relačné	<	<	<	<	>	<	>	<	>
(	<	<	<	<	<	<	=	<	
)	>	>	>	>	>		>		>
i	>	>	>	>	>		>		>
$	<	<	<	<	<	<		<	

Generovanie cieľového kódu:

Generovanie kódu začína vyvolaním funkcie *generate_header*, ktorá vypíše úvodný riadok „.IFJcode21“ a definuje vstavané funkcie. Ďalej je použitá dvojica funkcií *generate_execute_block* a *generate_execute_jump*, ktorých cieľom je preskočenie definícií funkcií, aby sa vykonali len raz. Pre predávanie parametrov z funkcie a naspäť do nej sme použili stack s poradím parametrov podľa štandardu *cdecl*.

Pre zjednodušenie generovania výsledného kódu sú vytvorené aj pomocné funkcie, ktorých cieľom je minimalizácia ich volania v parseri. Sú navrhnuté tak, aby v sebe zahŕňali všetky inštrukcie potrebné na vykonanie daného kľúčového slova.

Výpis kódu je vykonaný pomocou makier.

Záver:

Na projekte sme začali pracovať v druhej polovici októbra. Ku kompletnému pochopeniu zadania nám pomohli prednášky, ktoré nám objasnili problematiku a mohli sme si rozdeliť prácu. Komunikovali sme hlavne pomocou Discordu ale taktiež aj cez osobné schôdze.
Počas vývoja sme narazili na niekoľko komplikácii, no vždy sme mali vyhradený dostatočný časový limit na stanovené odovzdávanie.

Našou najväčšou komplikáciou bola zo začiatku nevyvážene rozdelená práca v tíme, viackrát sme menili úlohy medzi členmi, no ani na záver sa nám nepodarilo o presné rozdelenie práce na štvrtiny a preto sme sa museli odkloniť od rovnocenného rozdelenia bodov v tíme. Menším problémom bolo aj v neporozumení funkčnosti prekladača, kde sme si mysleli až do pokusného odovzdania že má program po nájdení a opravení chyby pokračovať ďalej.

V pláne sme mali zahrnutú aj prácu s rozšíreniami jazyka IFJ21, s ktorými sme počítali od začiatku projektu no na ich kompletné dokončenie sme nemali dostatok času a tak ostali rozšírenia len z časti funkčné (BOOLTHEN, FUNEXP).

Ako tím sme spolupracovali a vzájomne si pomáhali, projekt nám priniesol nové skúsenosti a znalosti pri tvorbe prekladača, či komunikácií v tíme alebo práci na komplexnejšom a náročnejšom projekte.