Dokumentácia projektu

Implementácia prekladača imperatívneho jazyka IFJ17

Tím 048, variant II

3. decembra 2017

Vedúci tímu:	**Crkoň Jakub (xcrkon00)**	**25%**
	Blašková Barbora (xblask04)	25 %
	Bárteček Bronislav (xbarte14)	25 %
	Kalusová Katarína (xkalus05)	25 %

1. Úvod

V tejto dokumentácii sú vysvetlené postupy, základné princípy a algoritmy, ktoré sme používali pri riešení. Taktiež je tu popísané rozdelenie práce v tíme a princíp našej spolupráce.

Cieľom tohto projektu bolo vytvoriť program v jazyku C (prekladač), ktorý načíta zdrojový kód v jazyku IFJ17 a preloží ho do medzikódu IFJcode17.

2. Tímová práca

Práca na projekte začala vytvorením konečného automatu pre lexikálnu analýzu. Počas riešenia projektu bolo viacero stretnutí a video-hovorov, kde sa riešila problematika jednotlivých častí projektu. Prácu sme si na začiatku rozdelili, avšak princíp postupu riešenia jednotlivých častí sme vždy medzi sebou prekonzultovali, aby sa predišlo zbytočnej nekompatibilite medzi modulmi projektu.

Rozdelenie práce v tíme:

Jakub Crkoň: precedenčná a sémantická analýza, tabuľka symbolov, generácia kódu
Barbora Blašková: syntaktická analýza, správa tokenov, testovanie, dokumentácia
Bronislav Bárteček: lexikálna analýza, generovanie kódu
Katarína Kalusová: tabuľka symbolov, testy, dokumentácia

3. Riešenie projektu

3.1 Lexikálna analýza

(Diagram konečného automatu pre lexikálnu analýzu viď Prílohy 6.1.)

Lexikálna analýza (ďalej LA) prevádza text (kód) zo štandardného vstupu na tokeny. Je implementovaná ako konečný automat (príloha 6.1). Funguje tak, že berie znak po znaku a rozhoduje o prípadnom postupe ďalej. Pre spracovávanie určitých tokenov (ako reťazcový literál, indentifikátor, celé číslo, desatinné číslo..) bolo potrebné naimplementovať *buffer*, ktorý sa na konci pretypuje na požadovaný typ. Napríklad pri type celého čísla vracia ukazateľ na integer, pri indentifikátore vracia ukazateľ na char a pod. Pri číselných hodnotách a reťazcovom literáli sa kontroluje ich správnosť. Pre odhalenie rezervovaných slov bola naimplementovaná samostatná funkcia. V prípade nedovoleného znaku či chyby sa korektne uvoľní pamäť a program sa ukončí s chybou 1. LA vracia token v štruktúre, ktorá obsahuje typ tokenu a dátovú časť, pričom typ tokenu je koncový stav v KA v prílohe. Taktiež bola implementovaná možnosť vrátenia tokenu pre potreby syntaktickej analýzy.

3.2 Syntaktická analýza

(Pravidlá gramatiky a LL-tabuľku viď Prílohy 6.2 a 6.3)

Syntaktická analýza (pre jednoduchosť ďalej len SA) bola prevedená na základe rekurzívneho zostupu. Pre gramatiku bola vytvorená tabuľka pravidiel na základe ktorej bola SA implementovaná. SA si vždy žiadala od lexikálnej analýzy ďalší token, pričom tokeny bolo možné aj vrátiť naspäť lexikálnej analýze, čo bolo napríklad výhodné pri kontrole ε-

pravidiel, alebo pri kontrole konca výrazu v precedenčnej analýze. Pre každý neterminál gramatiky jazyka bola vytvorená zvlášť funkcia a tieto funkcie sa medzi sebou volali čím bol odsimulovaný derivačný strom.

3.3 Precedenčná analýza

(*Pravidlá a tabuľka precedenčnej analýzy viď Prílohy* 6.4*)*

Precedenčná analýza (ďalej len PA) bola implementovaná na základe troch operácií a to :

1. Redukovanie - jedná sa o operáciu pri ktorej sa *popuje* zo zásobníku až kým sa nenarazí na zarážku ktorú v našom prípade predstavuje člen zásobníku s typom S_TOP
2. Shift - operácia ktorá vloží na zásobník zarážku pred prvý nájdený terminál a potom prvok na vstupe
3. Handle - vloženie prvku na vstupe na zásobník

Za účelom rozoznávania ktorá z operácií má byť vykonaná bola vytvorená precedenčná tabuľka. Ak nie je možné previesť ani jednu z týchto operácií nastáva syntaktická chyba.

Pri volaní PA jej syntaktická analýza predáva prvý token výrazu z dôvodu rozpoznávania prípadu kedy nasleduje výraz a kedy nasleduje volanie funkcie. Následne si PA pýta tokeny od lexikálnej analýzy, kontroluje tabuľku a podľa nájdeného symbolu na mieste v tabuľke vykonáva danú operáciu. Tokeny sú pri vkladaní na zásobník premenené na prvky zásobníku PA.

Počas operácií PA sú prevádzané sémantické funkcie pre výrazy. Kontrolujú sa typy operandov podľa ich záznamu v tabuľke symbolov a aj to, či je operácia medzi nimi sémanticky možná.

Výsledkom PA je výsledný sémantický typ spracovaného výrazu. V prípade chyby ukončí program s návratovým kódom danej chyby. Pri precedenčnej analýze taktiež vraciame lexikálnej analýze naspäť posledný token (EOL, EOF, ...), aby mohol potom byť opäť predaný parseru.

3.4 Sémantická analýza

Sémantická analýza bola implementovaná na základe typových pravidiel jazyka IFJ17. Sémantická analýza využíva funkcie tabuľky symbolov na vyhľadanie premennej alebo funkcie, ktorej sa má sémantická kontrola týkať. Kontroluje všetky sémantické akcie ako napríklad existencie premennej/funkcie, návratové hodnoty funkcií, správne použitie dátových typov a správne použitie operátorov. Taktiež zaisťuje premenu sémantického typu výrazu a kontroly možného implicitného pretypovania.

Sémantická kontrola taktiež pre jednoduchosť využíva funkcie generátoru kódu a generuje niektoré dôležité časti výstupného kódu IFJcode17.

3.5 Generovanie kódu

Funkcie generovania kódu boli implementované v samostatnom súbore. Funkcie generátoru následne volá sémantická a syntaktická analýza. Sémantická analýza generuje veci ňou kontrolované a syntaktická analýza generuje veci ako cykly a podmienky. Pre správne fungovanie vnorených cyklov a podmienok bol v samostatnom súbore implementovaný zásobník generátoru ktorý zaisťuje správne poradie generovania *náveští*

.

4. Algoritmy a dátové štruktúry

4.1 Tabuľka s rozptýlenými položkami

Vo variante II bolo našou úlohou implementovať tabuľku symbolov ako tabuľku s rozptýlenými položkami. Základom tabuľky sú jednosmerne zreťazené zoznamy, kde každá položka obsahuje ukazovateľ na nasledujúcu.

Pri implementácii tejto tabuľky bolo nutné využiť aj *hashovaciu* funkciu, ktorá nám na základe vstupného reťazca určila jeho index v tabuľke. Pomocou indexu získame ukazovateľ na prvú položku zoznamu a môžeme ním prechádzať až dokým položku nenájdeme.

Na zostavenie tabuľky symbolov sme využili metódy a algoritmy, ktoré boli vyučované v predmete IAL.

Implementované boli dve tabuľky pričom jedna slúži na uchovávanie všetkých potrebných informácií o premenných a druhá slúži na uchovávanie informácií o funkciách.

Tabuľka okrem vkladania obsahuje aj funkcie postupného inicializovania prvku tabuľky a vloženia až po prevedení sémantickej kontroly.

4.2 Zásobník

Implementované boli dva zásobníky. Jeden pre precedenčnú analýzu a druhý pre generátor kódu. Oba boli implementované za pomoci zreťazených zoznamov kvôli neurčitému počtu prvkov ktoré bolo potrebné na tieto zásobníky vkladať.

4.3 Zreťazený zoznam

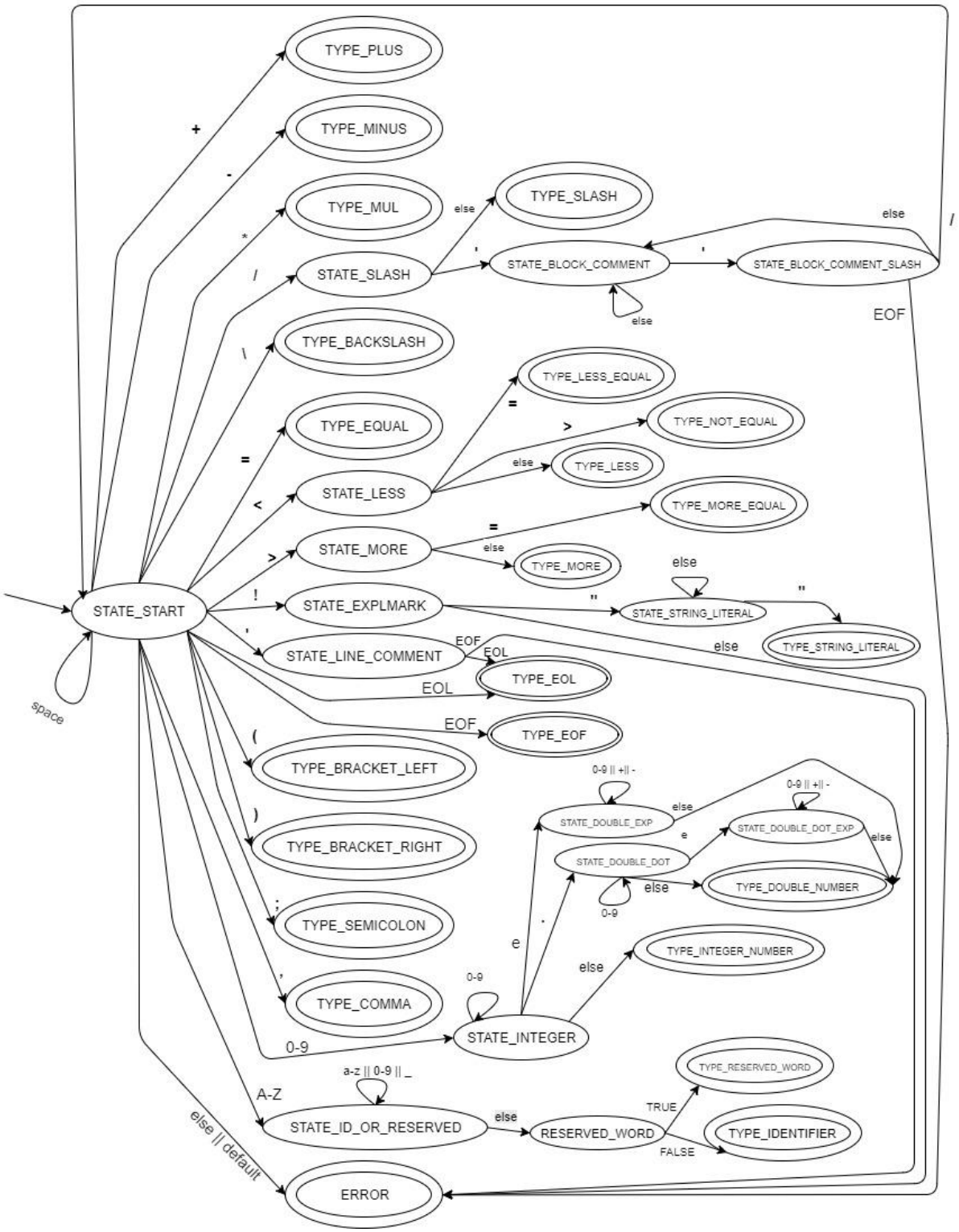
Jednosmerne viazaný zreťazený zoznam bol vytvorený za kvôli jednoduchosti uvoľňovania alokovaného miesta. Sú doň ukladané pointery na všetky vytvorené tokeny ktoré sú následne na konci behu programu uvoľnené.

5. Záver

Tento projekt bol pre nás skvelou skúsenosťou či už v programovaní alebo tímovej práci. Myslíme si, že zadanie sa nám podarilo splniť.

6. Prílohy

6.1 Diagram konečného automatu pre lexikálnu analýzu



6.2 LL- Gramatika

Rule num.	Non-terminal										
1	**<program>**	**<functions>**	**<scope>**	EOF							
2	**<scope>**	epsilon									
3	**<scope>**	scope	**<end>**	**<scope_content>**	end	scope					
4	**<scope_content>**		<body>								
5	**<statement>**	dim	id	as	**<type>**	**<assignment>**					
6	**<assignment>**	epsilon									
7	**<assignment>**	='	**<expression>**								
8	**<type>**	integer									
9	**<type>**	double									
10	**<type>**	string									
11	**<end>**	EOL	**<end_n>**								
12	**<end_n>**	epsilon									
13	**<end_n>**	EOL	**<end_n>**								
14	**<body>**	**<statement>**	**<end>**	**<body>**							
15	**<body>**	epsilon									
16	**<statement>**	id	='	**<right_side>**							
17	**<right_side>**	**<expression>**									
18	**<right_side>**	id	(	**<params>**	)						
19	**<params>**	epsilon									
20	**<params>**	**<expression>**	**<expression_params_n>**								
21	**<expression_params_n>**	epsilon									
22	**<expression_params_n>**	,	**<expression>**	**<expression_params_n>**							
23	**<statement>**	input	id								
24	**<statement>**	print	**<expression>**	;	**<expression_n>**						
25	**<expression_n>**	epsilon									
26	**<expression_n>**	**<expression>**	;	**<expression_n>**							
27	**<statement>**	if	**<expression>**	then	**<end>**	**<body>**	else	**<end>**	**<body>**	end	if
28	**<statement>**	do	while	**<expression>**	**<end>**	**<body>**	loop				
29	**<functions>**	epsilon									
30	**<functions>**	**<function>**	**<end>**	**<functions>**							
31	**<function>**	**<func_declaration>**									
32	**<function>**	**<func_definition>**									
33	**<func_declaration>**	declare	function	id	(	**<func_params>**	)	as	**<type>**		
34	**<func_params>**	epsilon									
35	**<func_params>**	id	as	**<type>**	**<func_params_n>**						
36	**<func_params_n>**	,	id	as	**<type>**	**<func_params_n>**					
37	**<func_params_n>**	epsilon									
38	**<func_definition>**	function	id	(	**<func_params>**	)	as	**<type>**	**<end>**	**<func_scope>**	
39	**<func_scope>**	**<func_body>**	end	function							
40	**<func_body>**	**<func_statement>**	**<end>**	**<func_body>**							
41	**<func_body>**	epsilon									
42	**<func_statement>**	**<statement>**									
43	**<func_statement>**	return	**<expression>**								

6.3 LL-tabuľka

	$	id	scope	end	function	declare	print	dim	do	while	if	then	else	input	as	integer	double	string	(	)	,	;	"="	EOL	EOF	loop	expr	return
<program>			1		1	1																						
<scope>			3																						2			
<scope_content>		4					4	4	4		4			4														
<statement>		16					24	5	28		27			23														
<assignment>																							7	6				
<type>																8	9	10										
<end>																								11				
<end_n>		12	12	12	12	12	12	12	12		12		12	12										13		12		
<body>		14		15			14	14	14		14			14														
<right_side>		17																									18	
<params>																				19							20	
<expression_params>																				21	22							
<expression_n>																								25			26	
<functions>			29		30	30																						
<function>					32	31																						
<func_declaration>						33																						
<func_params>		35																		34								
<func_params_n>																				37	36							
<func_definition>					38																							
<func_scope>		39					39	39	39		39			39														39
<func_body>		40		41			40	40	40		40			40														40
<func_statement>.		42					42	42	42		42			42														43

6.4 Precedenčná tabuľka a pravidlá

E -> id

E -> (E)

E -> E + E

E -> E - E

E -> E * E

E -> E / E

E -> E \ E

E -> E < E

E -> E > E

E -> E <= E

E -> E >= E

E -> E <> E

E -> E = E

	+'	-	*	/	\	id	(	)	$	<	>	<>	<=	>=	"="
+'	R	R	S	S	S	S	S	R	R	R	R	R	R	R	R
-	R	R	S	S	S	S	S	R	R	R	R	R	R	R	R
*	R	R	R	R	R	S	S	R	R	R	R	R	R	R	R
/	R	R	R	R	R	S	S	R	R	R	R	R	R	R	R
\	R	R	S	S	S	S	S	R	R	R	R	R	R	R	R
id	R	R	R	R	R	E	E	R	R	R	R	R	R	R	R
(	S	S	S	S	S	S	S	H	E	S	S	S	S	S	S
)	R	R	R	R	R	E	E	R	R	R	R	R	R	R	R
$	S	S	S	S	S	S	S	E	F	S	S	S	S	S	S
<	S	S	S	S	S	S	S	R	R	R	R	R	R	R	E
>	S	S	S	S	S	S	S	R	R	R	R	R	R	R	E
<>	S	S	S	S	S	S	S	R	R	R	R	R	R	R	E
<=	S	S	S	S	S	S	S	R	R	R	R	R	R	R	E
>=	S	S	S	S	S	S	S	R	R	R	R	R	R	R	E
"="	S	S	S	S	S	S	S	R	R	R	R	R	R	R	E

R - reduce, S - shift, H - handle, E - error, F - finish