Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentácia k projektu do predmetu IFJ a IAL

Implementácia prekladača jazyka IFJ17

Tím *126*, varianta *II*
Rozšírenia BOOLOP, BASE

Vedúci:	Kelemen Erik	xkelem01	25 %
	Lakatoš Attila	xlakat01	25 %
	Šober Patrik	xsober00	25 %
	Zubrik Tomáš	xzubri00	25 %

Obsah

1 Úvod

Dokumentácia k projektu popisuje implementáciu prekladača, ktorý načíta zdrojový kód v jazyku IFJ17, ktorý je zjednodušenou podmnožinou jazyka FreeBasic a prevedie preklad na cieľový kód jazyka IFJcode17, ktorý je ďalej interpretovateľný. Prekladač sa skladá z nasledujúcich 3 fundamentálnych častí:

- Lexikálny analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor

2 Úlohy v tíme a rozdelenie práce

2.1 Rozdelenie úloh na jednotlivých častiach projektu

- Kelemen Erik – syntaktický analyzátor, tabuľka symbolov
- Lakatoš Attila – spracovanie výrazov, generovanie kódu
- Šober Patrik – lexikálny analyzátor, testovanie
- Zubrik Tomáš – sémantický analyzátor, dokumentácia

2.2 Priebeh

Tímový projekt takýchto rozmerov prináša nové problémy, ale zároveň dokáže prispieť k našim skúsenostiam. Ako štvorčlenný tím sme si na začiatku spolupráce museli zvoliť vhodný komunikačný kanál a verzovací systém. Ako komunikačný kanál sme použili sociálnu sieť `facebook`. Ako verzovací systém sa rozhodli využiť `git` prostredníctvom webovej služby `GitHub`, kde vedúci tímu založil privátny repozitár, ktorý sme využili pre naše potreby. Určili sme taktiež základné konvencie pre písanie kódu.

Problémy a pripomienky sme riešili cez sociálnu sieť, v prípade nutnosti sme zorganizovali stretnutie vo vopred dohodnutom čase a na dohodnutom mieste. Jednotlivé časti boli implementované súčasne a každý na svojej časti pracoval s pomocou ostatných. S prípadnými otázkami a nejasnosťami sa člen tímu obrátil na vedúceho tímu. Pre testovanie sme využili automatické testy, ktoré boli implementované ako `shell script`.

3 Implementáca prekladača jazyka IFJ17

3.1 Lexikálna analýza

Lexikálny analyzátor je fundamentálnou a nevyhnutnou časťou prekladača. Je založený na deterministickom konečnom automate a jeho hlavnou úlohou je čítať zdrojový kód a na základe lexikálnych pravidiel daného jazyka rozdeliť jednotlivé postupnosti znakov na príslušné lexikálne jednotky – lexémy.

Lexikálne jednotky sú v programe reprezentované ako tokeny, teda zdrojový kód chápeme ako konečnú postupnosť tokenov. Každý token obsahuje dáta o jeho type, prípadne reťazec alebo číselnú hodnotu. Dáta do tokenov sú zapisované pomocou nami implementovaných funkcií v hlavičkovom súbore `string.h`.

Lexikálna analýza ignoruje všetky biele znaky, vrátane medzier, tabulátorov a komentárov, ktoré nie sú podstatné pre programový tok. Zdrojový kód môže obsahovať 2 typy komentárov: Riadkový komentár, ktorý sa začína znakom `'` a končí na konci riadku. Blokový komentár ktorý je ohraničený sekvenciou znakov `/'` a `'/`. Obidva typy komentárov môžu byť umiestnené kdekoľvek v programe.

Po spustení prekladu vytvorí lexikálny analyzátor dané tokeny s príslušnými dátami a uloží ich do zásobníka tokenov, s ktorým ďalej pracuje syntaktický analyzátor. Princíp fungovania lexikálnej analýzy reprezentuje deterministický konečný automat v prílohe 4.A.

3.2 Syntaktická a sémantická analýza

Syntaktický a sémantický analyzátor predstavuje najdôležitejšiu a najzlozitejšiu časť našej implementácie. Riadenie v programe preberá po lexikálnej analýze a riadi tok programu až k úplnému spracovaniu zdrojového kódu.

3.2.1 Spracovanie jazykových konštrukcií

Syntaktická analýza je implementovaná rekurzívnym zostupom a je riadená na základe vytvorenej LL-gramatiky v prílohe 4.B. Neterminály predstavujú v našej implementácii funkcie, ktoré sú volané v závislosti od nasledujúceho tokenu. Syntakticky správne napísaný zdrojový kód musí podliehať LL-gramatike, len v tom prípade je syntaktická analýza úspešná. Spolu so syntaktickou analýzou sú súčasne vykonávané sémantické kontroly. Pri deklarácii a definícii funkcie alebo premennej je daný identifikátor uložený do príslušnej tabuľky symbolov. Funkcie sú uložené do globálnej tabuľky symbolov, premenné zas do lokálnych tabuliek symbolov jednotlivých funkcií a hlavného tela programu.

Pri spracovaní tela funkcií a hlavného tela programu sa zároveň generujú príslušné inštrukcie, ktoré sú pripojené do výstupného reťazca, reprezentujúceho inštrukčnú pásku.

Ak sa behom syntaktickej analýzy narazí na výraz, je riadenie programu predané precedenčnej syntaktickej analýze pre spracovanie výrazov. Po spracovaní výrazu a vygenerovaní príslušných inštrukcií sa riadenie programu znovu vráti syntaktickej analýze.

Po spracovaní celého zdrojového súboru sa prevádzajú záverečné sémantické kontroly. Kontroluje sa definovanosť všetkých funkcií a prebehne kontrola či sa hlavné telo programu začinajúce kľúčovým slovom `SCOPE` vyskytuje v zdrojovom kóde len raz apod.

Po dokončení záverečných sémantických kontrol je syntaktická a sémantická analýza ukončená a na štandardný výstup sa výpiše celá inštrukčná páska, ktorú spracuje interpret.

3.2.2 Spracovanie výrazov

Spracovanie výrazu sa prevedie vtedy, ak podľa LL-gramatiky nasleduje neterminál `E`, ktorý reprezentuje výraz. Spracovanie výrazov je riadené precedenčnou tabuľkou uvedenou v prílohe 4.D. Spracovanie výrazu prebieha v 3 krokoch. Najprv sa kontroluje sémantika jednotlivých premenných a prevedie sa syntaktická kontrola, či ide o prípustný zápis výrazu. V druhom kroku sa prevedie výraz z infixového zápisu na postfixový pomocou zásobníka. V treťom kroku je vyhodnotený a vyčíslený postfixový výraz, prípadne sú uskutočnené implicitné konverzie a následne sú vygenerované príslušné inštrukcie, ktoré sú pripojené do výstupného reťazca, reprezentujúceho inštrukčnú pásku.

Pri neúspešnom spracovaní výrazu je preklad ukončený a program je ukončený s návratovou hodnotou podľa konvencie reprezentujúci príslušnú chybu.

3.2.3 Volanie funkcií

Volanie funkcií spracúva syntaktická analýza. Kontrolujú sa návratové typy funkcií a premenných do ktorých je funkcia priradená a taktiež typy a počet parametrov, s prípadnými implicitnými konverziami. Pred zavolaním funkcie sú jej parametre uložené na dočasný rámec, ktorý sa v ďalšom kroku pridá na lokálny zásobník rámcov. Následne sa prevádzajú inštrukcie v tele funkcie. Pri návrate z funkcie sa jej návratová hodnota uloží do vyhradenej premennej `%returnval` a je pridaná na vrchol dátového zásobníka interpretu pre následné priradenie do premennej. Inštrukcie sú ďalej vykonávané od miesta, kde bola funkcia zavolaná.

3.2.4 Vstavané funkcie

Dôležitou súčasťou sú vstavané funkcie. Na začiatku prekladu sa do výstupného reťazca, reprezentujúceho inštrukčnú pásku pripoja inštrukcie pre všetky vstavané funkcie a taktiež sa jednotlivé funkcie ako položky uložia do globálnej tabuľky symbolov. Pri následnom volaní vstavanej funkcie ide o klasické volanie funkcie.

3.3 Použitie interpretu

K dispozícii sme mali inštrukčnú sadu. Našou úlohou nebolo interpret implementovať, ale správne skonštruovať prípustné inštrukcie z inštrukčnej sady, ktoré budú interpretovateľné a po ich prevedení sa dostaneme k správnemu výsledku.

3.4 Použité datové štruktúry

3.4.1 Tabuľka s rozptýlenými položkami

Pre implementáciu tabuľky symbolov sme zvolili variantu *II.* - tabuľku s rozptýlenými položkami. Dôležitý faktor tvorí rýchlosť vyhľadávania jednotlivých položiek. Položka obsahuje kľúč, z ktorého sa vypočíta príslušný index v tabuľke, dôležité dáta o položke a ukazateľ na ďalšiu položku. Položky na tom istom indexe v tabuľke sú prepojené ukazateľmi a tvoria jednosmerný explicitne viazaný lineárny zoznam. Ako rozptyľovaciu funkciu sme zvolili vhodnú variantu pre reťazce.

3.4.2 Zásobník

Zásobník je dynamická datová štruktúra, ktorá umožňuje pracovať s položkou na vrchole zásobníka. Zásobník sme použili pri spracovaní výrazov, konkrétne pri prevode z infixového zápisu na postfixový a pri ich vyčíslovaní.

3.4.3 Reťazec

Reťazec je v podstate dynamické pole znakov alebo vektor. V našej implementácii táto štruktúra obsahuje konkrétny reťazec, jeho dĺžku a kapacitu nutnú pre jeho alokáciu v pamäti. V prípade potreby sa zvýši kapacita alokovaného reťazca.

3.4.4 Dynamické pole

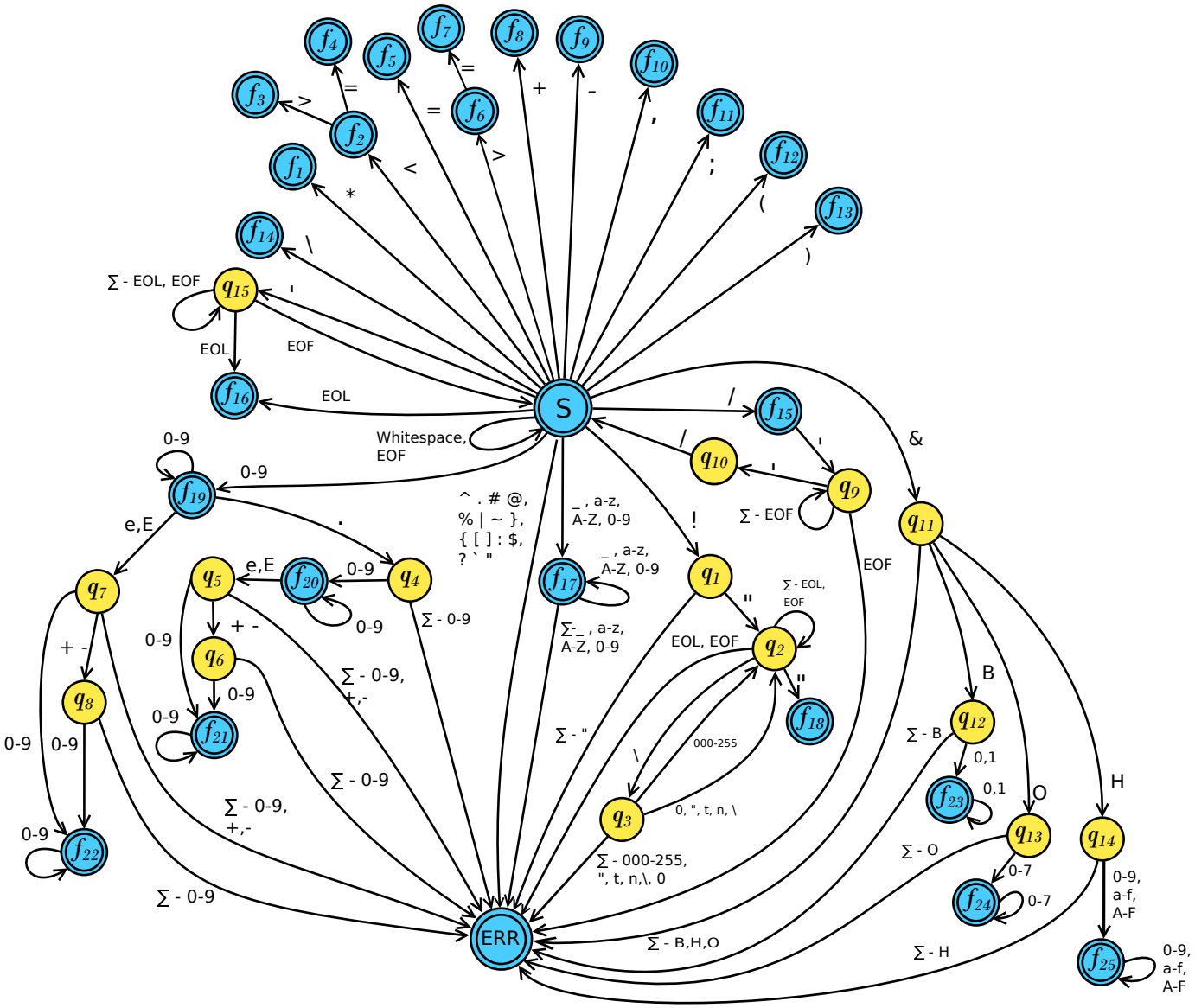
Dynamické pole je dátová štruktúra, ktorá sa využíva ak vopred nie je známy presný počet položiek. Dynamické pole sme využili pre implementáciu zásobníka tokenov, ktorý sa po spustení programu naplní ukazateľmi na tokeny a je možné s nimi ďalej pracovať.

3.5 Rozšírenia

Implementované rozšírenia v našom projekte sú BOOLOP a BASE. Rozšírenie BASE spočívalo v pridaní nových stavov príslušiacich pre dvojkové, osmičkové a šestnástkové čísla. Následne sme implementovali funkciu na konverziu daného typu čísla na integer. Rozšírenie BOOLOP spočívalo v priadaní nových elementov - operátorov do precedenčnej tabuľky a následnej implementácii, ktorá sa prejavila hlavne v precedenčnej syntaktickej analýze.

4 Prílohy

4.A Diagram konečného automatu lexikálnej analýzy [1]



Označení	Název stavu	Označení	Název stavu
q_1	EXCLAMATION_MARK	q_{10}	BLOCK_COMMENT_END
q_2	STRING_LITERAL_BEGINS	q_{11}	BASE
q_3	ESCAPE_SEQUENCE	q_{12}	BINARY_START
q_4	DOUBLE_1	q_{13}	OCTAL_START
q_5	DOUBLE_2	q_{14}	HEX_START
q_6	DOUBLE_3	q_{15}	LINE_COMMENT
q_7	INT_EXP_1		
q_8	INT_EXP_2		
q_9	BLOCK_COMMENT_START		

Tabulka 1: Konečný automat lexikálneho analyzátora - Stavy

Označení	Název stavu	Označení	Název stavu
S	START, EOF	f_{14}	DIV2
f_1	MUL	f_{15}	DIV
f_2	LESS_THAN	f_{16}	NEWLINE
f_3	NOT_EQUAL	f_{17}	IDENTIFICATOR
f_4	LESS_OR_EQUAL	f_{18}	STRING_LITERAL
f_5	EQUAL	f_{19}	INTEGER
f_6	GREATER_THAN	f_{20}	DOUBLE
f_7	GREATER_OR_EQUALS	f_{21}	DOUBLE_WITH_EXP
f_8	ADD	f_{22}	INT_WITH_EXP
f_9	SUB	f_{23}	BIN_NUM
f_{10}	COMA	f_{24}	OCTAL_NUM
f_{11}	SEMICOLON		
f_{12}	LEFT_PARANTHESIS		
f_{13}	RIGHT_PARANTHESIS		

Tabulka 2: Konečný automat lexikálneho analyzátora - Konečné stavy

4.B LL-gramatika

```
START-> FUNBLOCK SCOPEBLOCK
SCOPEBLOCK-> scope STATEMENTBLOCK end scope
FUNBLOCK-> declare function id left FUNDECPARAMS right as TYPE eol FUNBLOCK
FUNBLOCK-> function id left FUNDECPARAMS right as TYPE eol STATEMENTBLOCK
           end function eol FUNBLOCK
FUNBLOCK-> eol FUNBLOCK
FUNBLOCK-> epsilon
FUNDECPARAMS-> id as TYPE FUNDECPARAMSNEXT
FUNDECPARAMS-> epsilon
FUNDECPARAMSNEXT-> comma FUNDECPARAMS
FUNDECPARAMSNEXT-> epsilon
TYPE-> integer
TYPE-> double
TYPE-> string
STATEMENTBLOCK-> DECORASSIGN STATEMENTBLOCK
DECORASSIGN-> dim id as TYPE DECASSIGN eol
DECASSIGN-> equal E
DECASSIGN-> epsilon
STATEMENTBLOCK-> FUNCALLORASSIGN STATEMENTBLOCK
FUNCALLORASSIGN-> id equal FUNCALLORASSIGN2
FUNCALLORASSIGN2-> id left FUNCALLPARAMS right eol
FUNCALLORASSIGN2-> E eol
FUNCALLPARAMS-> FUNCALLPARAM FUNCALLPARAMSNEXT
FUNCALLPARAMS-> epsilon
FUNCALLPARAM-> id
FUNCALLPARAM-> CONSTVALUE
FUNCALLPARAMSNEXT-> comma  FUNCALLPARAMS
FUNCALLPARAMSNEXT-> epsilon
CONSTVALUE-> integer_value
CONSTVALUE-> double_value
CONSTVALUE-> string_value
STATEMENTBLOCK-> if E semicolon then eol STATEMENTBLOCK ELSESTATEMENT
                 end if STATEMENTBLOCK
ELSESTATEMENT-> else eol STATEMENTBLOCK
ELSESTATEMENT-> epsilon
STATEMENTBLOCK-> print E semicolon EXPRNEXT eol STATEMENTBLOCK
EXPRNEXT-> E semicolon EXPRNEXT
EXPRNEXT-> epsilon
STATEMENTBLOCK-> input id eol STATEMENTBLOCK
STATEMENTBLOCK-> do while E eol STATEMENTBLOCK loop STATEMENTBLOCK
STATEMENTBLOCK-> return E
STATEMENTBLOCK-> epsilon
STATEMENTBLOCK-> eol STATEMENTBLOCK
```

4.C LL-tabuľka

	scope	end	declare	function	id	left	right	as	eol	comma	integer	double	string	dim	equal	int_val	double_val	string_val	if	semicolon	then	else	print	input	do	while	loop	return	$
START	1		1	1					1																				
FUNBLOCK	6		3	4					5																				
SCOPEBLOCK	2																												
STATEMENTBLOCK		40			18				41					14					31			40	34	37	38		40	39	
FUNDECPARAMS					7		8																						
TYPE										11	12	13																	
FUNDECPARAMSNEXT							10			9																			
DECORASSIGN														15															
DECASSIGN									17						16														
EXPRESSION																													
FUNCALLORASSIGN					19																								
FUNCALLORASSIGN2					20																								
FUNCALLPARAMS					22		23				22	22	22																
FUNCALLPARAM					24						25	25	25																
FUNCALLPARAMSNEXT							27			26																			
CONSTVALUE											28	29	30																
ELSESTATEMENT		33																				32							
EXPRNEXT									36																				

4.D Precedenčná tabuľka

INPUT-> STACK	+	-	*	/	\	<	>	<=	>=	<>	=	(	)	ID	LITERAL	AND	OR	NOT	$
+	>	>	<	<	<	>	>	>	>	>	>	<	>	<	<	>	>	<	>
-	>	>	<	<	<	>	>	>	>	>	>	<	>	<	<	>	>	<	>
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	>	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	>	>	<	>
\	>	>	<	<	>	>	>	>	>	>	>	<	>	<	<	>	>	<	>
<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	>	>	<	>
>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	>	>	<	>
<=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	>	>	<	>
>=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	>	>	<	>
<>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	>	>	<	>
=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	>	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	<	>
)	>	>	>	>	>	>	>	>	>	>	>		>			>	>	>	
ID	>	>	>	>	>	>	>	>	>	>	>		>			>	>		>
LITERAL	>	>	>	>	>	>	>	>	>	>	>		>			>	>		>
AND	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	>	>	<	>
OR	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	>	>	<	>
NOT	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	>	>	<	>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	<	<	<	<	

5 Referencie

[1] WALLACE, E. *Finite State Machine Designer* [online]. 2010 [cit. 2015-12-13]. Dostupné na: http://madebyevan.com/fsm/

[2] Prof. Ing. Jan Maxmilián Honzík, CSc. *Algoritmy IAL: Sudijní opora* [online]. Verze 17R. 2017-12-3 [cit. 2017-12-5]. Dostupné na: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IAL-IT/texts/Opora-IAL-2017-verze-17-B.pdf