VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu do předmětů IFJ a IAL

Kompilátor jazyka IFJ17

Tým 012, varianta II

Filip Kočica (vedoucí týmu)	xkocic01	– 45%
Matej Kňazík	xknazi00	– 25%
Andrea Ficková	xficko00	– 20%
Jiří Fiala	xfiala54	– 10%

Rozšíření
IFTHEN, BASE, UNARY

29.11.2017

Obsah

1 Úvod

Dokumentace obsahuje popis návrhu a způsobu implementace kompilátoru IFJ17, jenž je zjednodušenou podmnožinou programovacího jazyka FreeBASIC.

Vybrali jsme si druhou variantu, implementace takublky symbolů pomocí tabulky s rozptýlenými položkami, protože nám byla bližší.

Zpracování výrazů, jenž mělo být dle zadání řešeno pouze pomocí precedenční syntaktické analýzy, dále jen PSA, jsme bohužel kvůli nedostatku komunikace řešili pomocí rekurzivního sestupu. Když jsme nato přišli, už nám vše fungovalo dle představ, ale i přesto jsme okamžitě začali pracovat na další verzi řešené přes PSA, kterou jsme již bohužel nestihli dokončit, a proto jsme odevzdali první verzi.

2 Implementace

2.1 Lexikální analýza

Lexikální analyzátor jsme naimplementovali na základě námi sestrojeného deterministického konečného automatu. Lexikální analyzátor je první část překladače, která má za úkol rozdělit zdrojový text na jednotlivé lexémy, jež jsou reprezentovány pomocí tokenů. Token typicky obsahuje informaci o typu lexému a pokud je potřeba, tak i lexém samotný. Tyto tokeny si poté parser jeden po druhém žádá pomocí funkce `get_token` a vzhledem k jednoprůchodovému překladu je po zpracování bez dalšího ukládání uvolňuje.

2.2 Syntaktická analýza

Syntaktický analyzátor, zvaný parser, je hlavní část překladače. Pokud se jedná o syntaxí řízený překlad, jímž jsme při řešení postupovali, má parser na starosti jak akce pro syntaktické a sémantické kontroly, tak generování mezikódu.

Část kódu zabívající se syntaktickou analýzou jsme implementovali pomocí rekurzivního sestupu shora dolů na základě gramatiky, která definuje jazyk IFJ17. Z důvodu nepozorného přečtení zadání jsme stejnou metodu použili i při zpracování výrazů. Nakolik námi použitá metoda oproti předepsanému principu PSA funguje přes poměrně složité rekurzivní voláni funkcí, tak to vedlo k značnému znepřehlednění kódu a rovněž zkomplikovalo implementaci následujících modulů sémantické analýzy a generátoru mezikódu. První jsme chtěli parser řešit pomocí implementace derivačního stromu, čemuž jsme se naštěstí vyhnuli a práci bychom si pravděpodobně značně stížili.

2.3 Sémantická analýza

Vzhledem k již zmíněnému syntaxí řízenému překladu se veškeré sémantické akce, jako jsou kontroly kompatibility datových typů a deklarací / definicí proměnných a funkcí, provádí přímo v parseru při průchodu přakládaným zdrojovým textem. Abychom byli schopni určit, zda je program sémanticky správně, budeme potřebovat tabulku symbolů. Přesněji tabulku symbolů s rozptýlenými položkami.

2.3.1 Tabulka symbolů

Vzhledem k možností více rámců ve zdrojovém textu, je implementován lineární seznam těchto tabulek. Při vstupu do nového rámce je přídána další tabulka, naopak při opuštění rámce je poslední přidaná tabulka odstraněna. Vždy existuje alespoň jedna tabulka symbolů pro globální rámec. Pro hashování řetězců je použita známá hashovací funkce `djb2` jejímž autorem je Dan Bernstein [1] a tabulka je alokována pro 500 000 položek.

2.4 Generování mezikódu

Generování probíha již od začátku syntaktické analýzy, kdy je trojadresný kód ukládán do lineárního seznamu aby instrukce nebyly vypsány i při chybě. Instrukce se vypíší na standradní výstup až v případě úspěšného dokončení syntaktické a sémantciké analýzy. Vzhledem k již zmíněnému řešení výrazů pomocí rekurzivního sestupu jsme si tuto část při generování výrazů poměrně stížili, zejména protože tento postup neměl jasná pravidla jak je tomu u PSA, a proto při složitějších výrazech nastávají chyby, zejména vlivem implicitních konverzí.

2.5 Rozšíření

Vzhledem k tomu, že již při prvním pokusném odevzdání jsme dosáhli 80-ti procent, rozhodli jsme se pro implementaci některých jednoduchých rozšíření.

2.5.1 BASE

Při implementaci rozšíření BASE jsme nejprve do návrhu konečného automatu přidali pravidla pro rozlišení čísel v dvojkové, osmičkové a šestnáctkové soustavě začínající znakem '`&`'. Poté jsme je implementovali ve scanneru a ještě tam se čísla převádějí do desítkové soustavy pomocí funkce `strtol`[2] a až poté jsou předána do parseru.

2.5.2 UNARY

U tohoto rozšíření stačilo do gramatických pravidel přidat unání operátory a při implementaci zkontrolovat typ výrazu a vygenerovat mezikód. U unárních '+' a '−' stačilo rekurzivně zavolat funkci `E`[1] s informací o znaménku.

2.5.3 IFTHEN

Při implementaci tohoto rozšíření jsme využili zkušeností z kurzu ISU[2]. Jako první jsme přidali nová pravidla. Pomocí labelů a podmíněných skoků bylo rozšíření za chvilku hotové. Pomocí globálního čítače, který se pomocí funkce `sprintf` přidává za každý label a inkrementuje se po každé if-else konstrukci nikdy nedojde ke kolizi jmen.

2.6 Metriky zdrojového textu překladače

Počet řádků kódu (LOC):	6919
Počet zdrojových souborů:	23

[1] Funkce `E` slouží k rozparsovaní složených výrazů

[2] Programování na strojové úrovni

3 Práce v týmu

Práce v týmu na rozsáhlejším projektu byla nová zkušenost pro některé členy týmu, avšak někteří měli pár nabytých zkušeností díky předmětu IVS, na základě kterého jsme se snažili předejít chybám vzniklých během vypracování projektu pro výše zmíněný předmět. Přesto, jsme se novým věcem přiučili všichni, ne jen co se týče týmové práce, ale nabyly jsme potřebné poznatky o překladači, čímž jsme si opět rozšířili naše znalosti v oblasti IT, o programování v jazyce C a využívání GIT-u. Důležité v tomto projektu bylo zhodnotit každého zkušenosti s programováním a základě toho adekvátně přidělit úlohy. Na pravidelných setkáních jsme konzultovali a řešily problémy při implementaci překladače.

Práci jsme měli rozdělenou následovně:

Filip Kočica	lexikální analýza, syntaktická analýza, sémantická analýza, generátor vnitřního kódu, zpracování výrazů, tabulka symbolů, vestavěné funkce, makefile, rozšíření, testování, dokumentace
Matej Kňazík	zpracování výrazů, LL-tabulka, LL-gramatika, dokumentace
Andrea Ficková	tabulka PSA, LL-gramatika, dokumentace
Jiří Fiala	tabulka PSA, LL-gramatika, práce na AST než byl zrušen

Objasnění nerovnoměrného rozdělení procent v týmu

Procenta jsme si rozdělili nerovnoměrně, na základě množství odvedené práce. Vedoucí týmu obstál s nejvyšším počtem procent, protože naimplementoval nejvíce částí překladače s rozšířeními, a tedy mu právem patří nejvyšší podíl. V tomto případě by bylo opravdu nespravedlivé rovnoměrné rozdělení procent mezi všechny členy. Za jeho velmi dobře odvedenou práci jsou mu jeho kolegové moc vděční.

4 Závěr

Projekt jsme vypracovali včas a jsme s ním spokojeni. Pouze nás mrzí naše nepozornost u implementace výrazů. V příštích projektech bude komunikace a pozornost naší hlavní prioritou. Na druhou stranu jsme se naučili a dozvěděli množství nám dříve nejasných věcí ohledně překladačů a projekt považujeme za jeden z nejužitečnějších.

Při řešení projektu nám ohromně pomohly a ušetřily množství času testy vytvořené jednou z aktivních skupin a tímto bychom jim rádi poděkovali.

5 Literatura

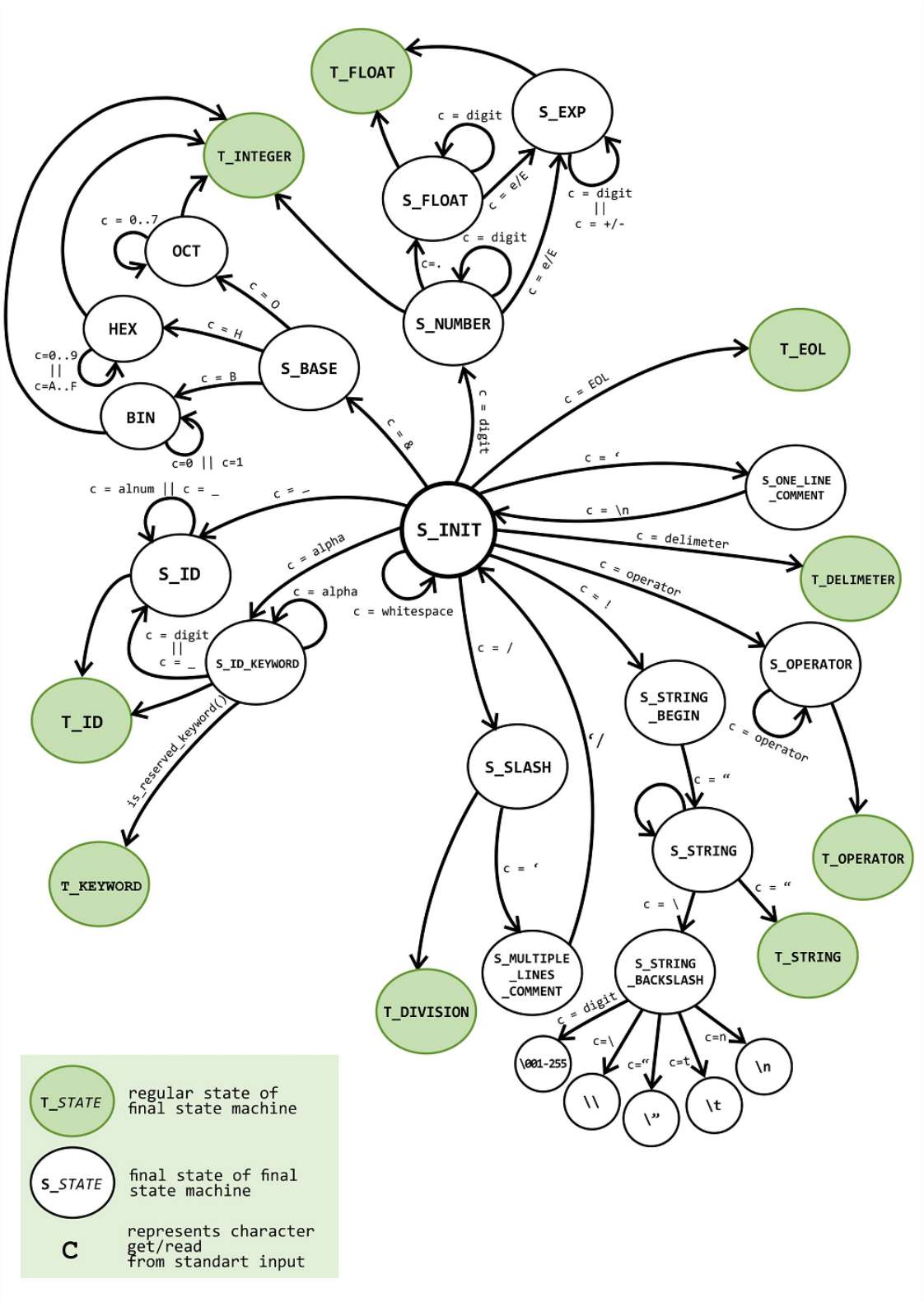
Reference

[1] *Hash function* [online]. [cit. 2017-12-06]. Dostupné z: http://www.cse.yorku.ca/ oz/hash.html

[2] *Strtol* [online]. [cit. 2017-12-06]. Dostupné z: http://www.cplusplus.com/reference/cstdlib/strtol

6 Přílohy

6.1 Final state diagram



6.2 LL-gramatika

1: <S> -> <declaration-list> scope <statement-list> end scope
2: <S> -> EOF
3: <declaration-list> -> eps
4: <declaration-list> -> T_EOL
5: <declaration-list> -> dim T_ID as <data-type> <assign> T_EOL <declaration-list>
6: <declaration-list> -> declare <function-decl> T_EOL <declaration-list>
7: <declaration-list> -> <function-decl> T_EOL <statement-list> End Function T_EOL <declaration-list>
8: <function-decl> -> Function T_ID (<param-list>) as <data-type>
9: <data-type> -> Integer
10: <data-type> -> Double
11: <data-type> -> String
12: <assign> -> eps
13: <assign> -> = <E>
14: <param-list> -> <param> <next-param>
15: <param> -> T_ID as <data-type>
16: <next-param> ->
17: <next-param> -> , <param> <next-param>
18: <statement-list> -> <statement> T_EOL <statement-list>
19: <statement-list> -> eps
20: <statement> -> eps
21: <statement> -> T_ID <call-assign>
22: <statement> -> print <print-list>
23: <statement> -> input T_ID
24: <statement> -> if <E> then T_EOL <statement-list> else T_EOL <statement-list> end if
25: <statement> -> do while <E> T_EOL <statement-list> loop
26: <statement> -> return <E>
27: <call-assign> -> = <value>
28: <value> -> <E>
29: <value> -> T_ID <call>
30: <call> -> (<argument-list>)
31: <argument-list> -> <E> <next-arg>
32: <next-arg> -> eps
33: <next-arg> -> , <E> <next-arg>
34: <print-list> -> <E> <next-print>
35: <next-print> -> eps
36: <next-print> -> ; <E> <next-print>
37: <E> -> <E> op <E>
38: <E> -> (<E>)

6.3 LL-tabulka

	scope	end	EOF	T_EOL	dim	T_ID	T_STRING	T_INTEGER	T_REAL	as	declare	function	Integer	Double	String	,	print	input	if	then	else	do	while	loop	return	(	)	'=	;	ε
<S>	1		2	1	1						1	1																		1
<declaration-list>				4	5						6	7																		3
<function-decl>												8																		
<data-type>													9	10	11															
<assign>						13	13	13	13																	13				12
<param-list>						14																								
<param>						15																								
<next-param>																17														16
<statement-list>						18											18	18	18			18			18					19
<statement>						21											22	23	24			25			26					20
<call-assign>																												27		
<value>						29	28	28	28																	28				
<call>																										30				
<argument-list>						31	31	31	31																	31				
<next-arg>																33														32
<print-list>						34	34	34	34																	34				
<next-print>																													36	35
<E>																														

6.4 Precedenční tabulka

* TOP OF STACK * \ * INPUT TOKEN *	$	lit	id	+	-	*	/	\	(	)	=	<>	<	>	<=	>=
$		<	<	<	<	<	<	<	<		<	<	<	<	<	<
lit	>			>	>	>	>	>		>	>	>	>	>	>	>
id	>			>	>	>	>	>		>	>	>	>	>	>	>
+	>	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>
-	>	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>
*	>	<	<	<	<	>	>	>	<	>	>	>	>	>	>	>
/	>	<	<	<	<	>	>	>	<	>	>	>	>	>	>	>
\	>	<	<	<	<	<	<	>	<	>	>	>	>	>	>	>
(		<	<	<	<	<	<	<	<	=	<	<	<	<	<	<
)	>			>	>	>	>	>		>	>	>	>	>	>	>
=	>	<	<	<	<	<	<	<	<	>						
<>	>	<	<	<	<	<	<	<	<	>						
<	>	<	<	<	<	<	<	<	<	>						
>	>	<	<	<	<	<	<	<	<	>						
<=	>	<	<	<	<	<	<	<	<	>						
>=	>	<	<	<	<	<	<	<	<	>						

reduction	>
shift	<
error	*free space*
lit	int/float/string