Vysoké učení technické v Brně
Fakulta informačních technologií

Formální jazyky a překladače
2017/2018

Skupinový projekt

Mitaš Matěj *xmitas02*, vedoucí – 25 %
Hanák Jiří *xhanak33* – 25 %
Holop Patrik *xholop01* – 25 %
Kapoun Petr *xkapou04* – 25 %

UNARY – IFTHEN – BOOLOP – FUNEXP

Tým 027, varianta I

Brno, 6. prosince 2017

1 Úvod

Cílem projektu bylo vytvořit překladač jazyka IFJ17, který je modifikovanou verzí jazyka FreeBASIC do jazyka IFJcode17. Překladač je implementován v jazyce C.

2 Dělba práce

Vzhledem ke komplexnosti projektu byla zvolena metodika práce na menších celcích, které poté byly spojeny ve finální produkt. Toto rozdělení bylo uvedeno v platnost na ustanovující schůzi týmu.

První skupina: Patrik Holop, Petr Kapoun

První skupina měla na starost návrh a implementaci lexikální analýzy. Vzhledem k časové souslednosti odpřednášené látky mohla tato skupina začít pracovat na projektu jako první.

Po uspokojivém dokončení scanneru spolu s funkčními základy syntaktické kontroly, vytvořené druhou skupinou, byly zahájeny práce na sémantické kontrole. Patrik Holop měl na starosti návrh a implementaci zpracování funkcí a rozsahů platnosti, Petr Kapoun sémantickou kontrolu a vyhodnocování výrazů.

Druhá skupina: Jiří Hanák, Matěj Mitaš

Práce druhé skupiny začala návrhem gramatiky, kterou si vzal na starost Jiří Hanák. Matěj Mitaš naimplementoval pomocné datové struktury a navrhl tabulku symbolů.

Po dokončení syntaktické kontroly se druhý tým vrhl na návrh a implementaci generátoru tříadresného kódu. Matěj Mitaš pomohl dokončit prvnímu týmu sémantickou analýzu, primárně implementací kontroly typů. Mezitím Jiří Hanák naimplementoval generátor.

Doplňující informace k týmové spolupráci

Obě skupiny byly v neustálém kontaktu, za účelem zaručení plynulé a rychlé spolupráce.

Každý člen týmu byl zodpovědný za plnou funkčnost své části implementace, nicméně, vzhledem ke komplexnosti projektu, byly vytvořeny obecné testy pro celý program, které měli na starost Patrik Holop a Matěj Mitaš.

Jako komunikační prostředek byl zvolen *Slack*, kde každá část překladače měla svůj kanál. Jako verzovací nástroj byl zvolen *Git* (Bitbucket). Podmínkou schválení každého *Pull requestu* byla kontrola druhým členem podtýmu autora a aspoň jedním členem z druhé skupiny.

Aby se minimalizoval počet změn konečného automatu a gramatiky, hned ze začátku prací bylo jasně určeno, které rozšíření budou navrhnuta a implementována.

Na tvorbě se podíleli všichni členové týmu.

3 Návrh a implementace

3.1 Lexikální analýza

Lexikální analyzátor slouží pro rozdělní vstupního souboru na *tokeny*. Je implementován jako konečný automat (viz sekce 9). Na základě znaků na vstupu automat sestavuje data tokenu reprezentované *řetězcem* a po přijetí znaku, který není očekávaný v daném stavu, vrátí rozpoznaný token nebo vznikne chyba. Jednotlivé vytvořené tokeny vrací postupně, nikoliv jako posloupnost. Token je reprezentovaný strukturou *token*, která obsahuje data tokenu, délku dat a jejich typ.

Jazyk IFJ17 podporuje tzv. *escape sekvence* v řetězci a je úlohou scanneru rozpoznat neplatnou sekvenci. Bylo zapotřebí přidat několik dalších stavů automatu ošetřující číselné escape sekvence `\001` až `\255` a speciální znaky, např. `\t, \n, \", \\`.

Znaky, které musí být pro potřeby interpretování nahrazené příslušnou escape sekvencí,
např. ' ' na `\032` skener přímo nahradí.

Rozšíření *UNARY* přidalo několik nových typů tokenů, např. `+=`.

3.2 Syntaktická analýza

Překladač je navrhnutý pro syntaxí řízený překlad implementovaný pomocí prediktivní syntaktické analýzy, proto byla tato část projektu kritická (viz sekce 10). Pomocí funkce *get_next_token* určíme, zda-li se má načítat znak souboru (pouze pro účely testování) nebo ze *stdin*, dále získává tokeny ze skeneru. Na základě tokenu a terminálu uloženého nejblíže vrcholu na zásobníku, se určí pravidlo, které se má aplikovat a na zásobník se vloží potřebné terminály a neterminály.

Pravidlo je reprezentováno strukturou *Rule*. Terminály a neterminály sdílejí strukturu *Terminal*. Ta obsahuje data, jejich délku, typ terminálu a pravdivostní hodnotu, jedná-li se o terminál, či neterminál. Na základě tokenů probíhá postupné nahrazovaní neterminálů na zásobníku terminály a při shodě terminálu na vrcholu zásobníku s očekávaným tokenem se z vrcholu zásobníku odstraní.

Na základě pravidla syntaktická analýza rozhodne, jestli se např. má vytvořit nová proměnná v tabulce symbolů a určí příslušnou sémantickou akci. Dále rozlišuje speciální případ pro práci s *výrazy*. V tomto případě se nevykonává kontrola podle pravidel, ale volá se funkce *solve_expr*, která pracuje na základě precedenční tabulky reprezentované dvourozměrným polem, jež převádí výraz na postfixový tvar.

3.3 Sémantická kontrola

V naší implementaci byla použita metoda přímého generování, což znamená, že se na základě pravidel určených gramatikou vykoná příslušná sémantická akce, která může přímo generovat kód.

3.3.1 Kontrola funkcí a rozsahů platnosti

Tato část programu se zabývá sémantickou kontrolou, která nevyžaduje vyhodnocení výrazů. Do tohoto módu spadá opakovaná deklarace funkcí, rozdílný typ deklarace a definice funkce, kotrola typu definovaných parametrů a podobně. Na základě příchozích pravidel se rozeznávají různé *oblasti kódu*[1]. Tato informace je dále využita pro kontrolu shodnosti typů deklarace a pozdější definice funkce.

Sémantická kontrola vždy pracuje s *aktivní* tabukou symbolů (viz sekce 4). Při deklaraci nebo definici nové funkce, případně vstupu do hlavního těla programu určí, jestli se tato funkce má stát aktivní a poté se jednotlivé sémantické akce vztahují na ni. V zájmu zjednodušení implementace se hlavní tělo programu ukláda stejně jako lokální funkce.

3.4 Vyhodnocování výrazů

Výrazy se vyskytují téměř v každém programu, ať už při inicializaci proměnných nebo vyhodnocení argumentů při volání funkcí. Vyhodnocení výrazů pracuje s precedenční tabulkou (viz sekce 7), na základě které je určena priorita operátorů. Při vyhodnocení výrazů se výraz převádí na postfixový tvar a pracuje se přímo s datovým zásobníkem interpretu tak, aby po vyhodnocení zůstal na vrcholu zásobníku výsledek. Při logických výrazech *true* nebo *false*, při aritmetických výrazech číslo. Kvůli rozšíření *BOOLOP* jsme při návrhu museli do precedenční tabulky zahrnout logické operátory *and*, *or* a podobně.

3.4.1 Typová kontrola

Kontrola typů na svoji činnost využívá speciální typový zásobník, do kterého se při vyhodnocení výrazu vkládají typy a rozhodne se, jako se má dál postupovat, případně zda-li vrátit chybu. Pokud při aritmetickém výrazů si nejsou dva typy na vrcholu zásobníků rovny, zahlásíme chybu, jinak vrátíme beze změny, či vhodně přetypujeme.

3.5 Volaní funkcí

Kvůli rozšíření *FUNEXP* se jako argument při volání funkce může vyskytnout i výraz, který může vyžadovat přetypování. Proto se před voláním funkce uloží všechny parametry na zásobník, vyhodnotí se výrazy a určí se co je třeba přetypovat (jestli je vůbec potřeba).

[1] oblast kódu je definovaná jako samostatný funkční blok, např. definice funkce.

3.6 Generátor

Generátor jako takový je sada funkcí, které generují konkrétní části programu a jsou volané při syntaktické a sémantické kontrole. Aby se zajistila jedinečnost vygenerovaných identifikátorů, má generátor vlastní interní strukturu, která při vygenerovaní nové proměnné, funkce nebo parametru určí jeho jedinečné id a to vrátí syntaktické a sémantické kontrole, které ho vloží do tabulky symbolů k danému prvku a při volání generátorových funkcí už dále pracují s tímto číslem.

Aby se minimalizoval počet pomocných proměnných, vygeneruje generátor při svojí činnosti čtyři globální proměnné fungující jako registry.

4 Tabulka symbolů

Každá funkce potřebuje pro svoji činnost vlastní tabulku symbolů, což je řešené jako lineární seznam tabulek *SYMT_LIST*. Samotná tabulka je implementovaná jako struktura *Symt_table*, která obsahuje jméno funkce, id, s kterým pracuje generátor, informaci o počtu parametru a jejich typu, návratový typ samotné funkce a ukazatel na *BST* (vyhledávací binární strom), do kterého se *symboly*, lokální proměnné a parametry, ukládají. Parametry i lokální proměnné jsou reprezentovány pomocnou strukturou *Symbol*, která obsahuje jejich název, tak i specifické id určené generátorem, jejich typ a případně hodnotu. Pro každou funkci existuje strom pro parametry a lokální proměnné. Sémantická část určí, která tabulka je v tomto seznamu aktivní, tedy v rámci které funkce se program pohybuje.

5 Doplňující informace k projektu

Na testovaní projektu byly vytvořené dvě sady testů. Na testování návratových kódů sloužil skript *test.sh*, který spouštěl program na předem definované sadě testů.

Na testovaní výstupu programu sloužila druhá sada testů a program *test.c*, který efektivně porovnával očekáváné výstupy se skutečným výstupem překladače.

Využili jsme obě pokusná odevzdaní. Celkové hodnocení prvního odevzdání bylo 80 %, druhého 93 %.

5.1 Garbage Collector

V projektu byl použit vlastnoručně naimplementovaný *Garbage Collector*, za účelem usnadnění práce s pamětí. Rozsah projektu byl poměrně značný, proto dávalo smysl investovat čas do tvorby podprogramu, který bezchybný kód nepotřebuje, avšak v praxi je to velice užitečná věc. Byla vytvořena obalovací funkce na systémové funkce *malloc* a *free* s pomocným lineárním seznamem, který si uchovává adresy naalokovaného místa. V případě uvolnění uživatelem je uvolní, avšak podporuje i uvolnění paměti na konci programu, což napomáhá programátorské přívětivosti programu.

5.2 Problémy

Při implementaci se vyskytly situace, kdy na základě nevhodně pochopeného zadání, či nedostatečné komunikace v podtýmech nebo mezi týmy, docházelo ke zbytečným implementacím či dokonce k nevhodným úpravám již vytvořených věcí. Nicméně tyto záležitosti tvořily pouze minortní část projektu, proto bych se jim neměla přikládat velká váha, ale musí být brány na zřetel.

6 Závěr

Celkově hodnotíme projekt kladně, podařilo se nám vytvořit, až na několik drobných chyb, plně funkční překladač. Zvládli jsme práci v týmu a každý člen týmu si plně zasloužil svých 25 % bodů.

7 Precedenční tabulka

	+	−	*	/	\	<	>	<=	>=	=	<>	And	Or	Not	(	)	id	,	$	u−	f
+	>	>	<	<	<	>	>	>	>	>	>	>	>	<	<	>	<	>	>	<	<
−	>	>	<	<	<	>	>	>	>	>	>	>	>	<	<	>	<	>	>	<	<
*	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	>	>	<	<
/	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	>	>	<	<
\	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	>	>	<	<
<	<	<	<	<	<							>	>	<	<	>	<	>	>	<	<
>	<	<	<	<	<							>	>	<	<	>	<	>	>	<	<
<=	<	<	<	<	<							>	>	<	<	>	<	>	>	<	<
>=	<	<	<	<	<							>	>	<	<	>	<	>	>	<	<
=	<	<	<	<	<							>	>	<	<	>	<	>	>	<	<
<>	<	<	<	<	<							>	>	<	<	>	<	>	>	<	<
And	<	<	<	<	<	<	<	<	<	<	<	>	>	<	<	>	<	>	>	<	<
Or	<	<	<	<	<	<	<	<	<	<	<	<	>	<	<	>	<	>	>	<	<
Not	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	>	>	<	<
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	=		<	<
)	>	>	>	>	>	>	>	>	>	>	>	>	>	>		>		>	>	>	
id	>	>	>	>	>	>	>	>	>	>	>	>	>	>		>		>	>		
,	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	=		<	<
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<			<	<
u−	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>	>	<	<
f															=						

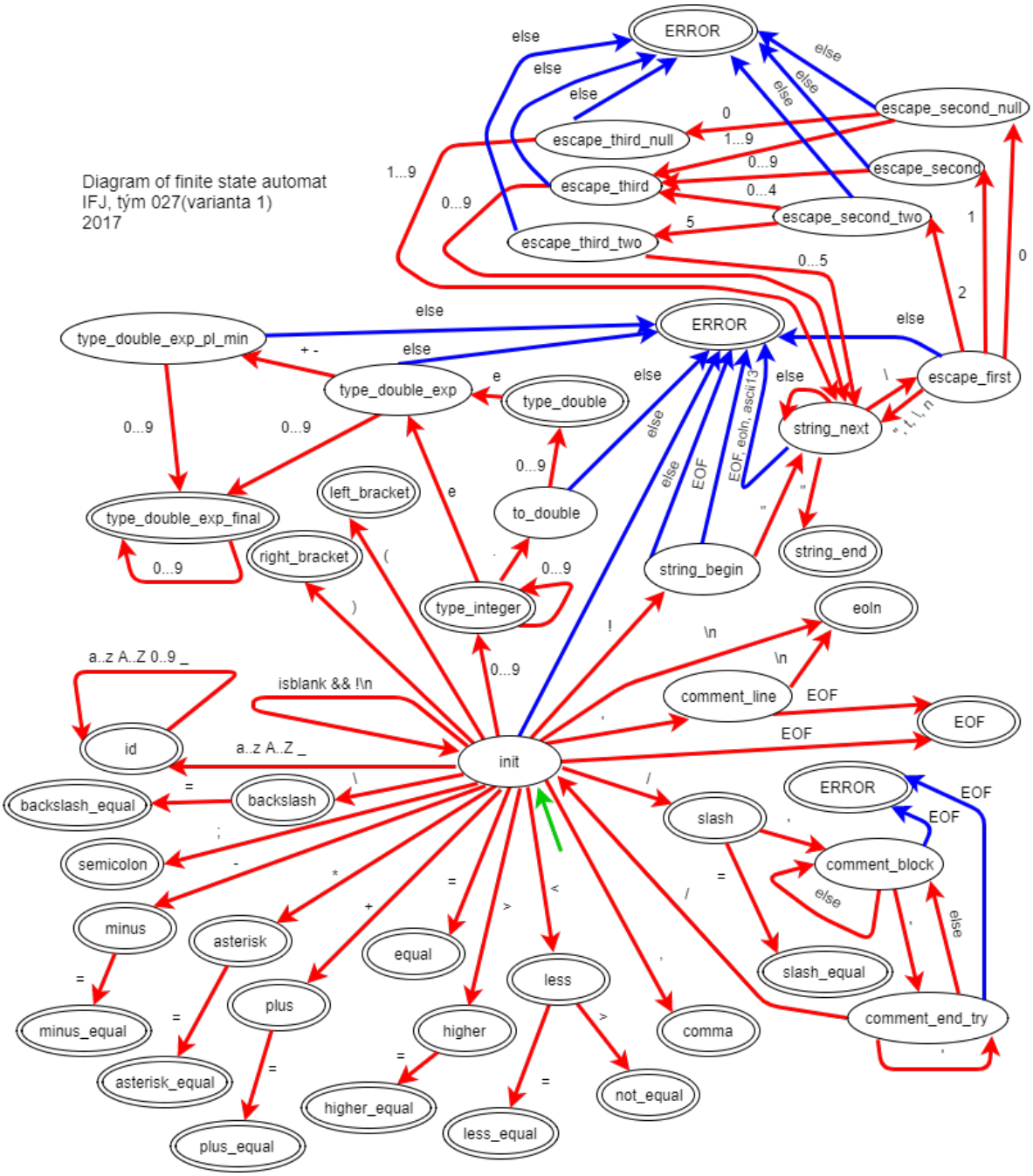
Tabulka 1: Precedenční tabulka

8 LL tabulka

	start_state	par_dec_multi	parameter	function	init	row	eol	statement	order	type	else	elseif	expr	print_list
kw_string										13				
kw_boolean										14				
kw_integer										15				
kw_double										16				
kw_scope	2			62			65							
kw_as														
kw_asc							65							
kw_declare	2			3			65							
kw_function	2			4			65							
kw_dim						12	65	9						
kw_loop						64	65							
kw_do						12	65	17						
kw_if						12	65	23						
kw_elseif						64	65					22		
kw_else						64	65				21	67		
kw_end						64	65				66	67		
kw_chr							65							
kw_input						12	65	18						
kw_print						12	65	19						
kw_length														
kw_return						12	65	8						
kw_true													44	20
kw_false													44	20
integer													44	20
double													44	20
string													44	20
plus_equal									25					
minus_equal									26					
asterisk_equal									27					
slash_equal									28					
backslash_equal									29					
eoln	2				63	64	11							68
minus														20
equal					10				30					
left_backet														20
right_backet		60	61											
id			5			12	65	24						20
comma		7												
eof							65							
unary_minus													37	20
id_fce														20

Tabulka 2: LL gramatika

9 FSM (scanner)



10 LL gramatika

2.	start_state	→	eol function **fkw_scope eoln** eol row eol **kw_end kw_scope** eol
3.	function	→	**kw_declare kw_function id (** parameter par_dec_multi **) kw_as** type **eoln** eol function
4.	function	→	**kw_function id (** parameter par_dec_multi **) kw_as** type **eoln** eol row **kw_end kw_function eoln** eol function
62.	function	→	epsilon
5.	parameter	→	**id kw_as** type
61.	parameter	→	epsilon
7.	par_dec_multi	→	**comma** parameter par_dec_multi
60.	par_dec_multi	→	epsilon
8.	return	→	**kw_return** expr
9.	statement	→	**kw_dim idkw_as** type init
10.	init	→	**equal** expr
63.	init	→	epsilon
11.	eol	→	**eoln** eol
65.	eol	→	epsilon
12.	row	→	statement **eoln** eol row
64.	row	→	epsilon
13.	type	→	**kw_string**
14.	type	→	**kw_boolean**
15.	type	→	**kw_integer**
16.	type	→	**kw_double**
17.	statement	→	**kw_do kw_while** expr **eoln** eol row **kw_loop**
18.	statement	→	**kw_input id**
19.	statement	→	**kw_print** expr **;** print_list
20.	print_list	→	expr **;** print_list
68.	print_list	→	epsilon
21.	else	→	**kw_else eoln** eol row
66.	else	→	epsilon
22.	elseif	→	**kw_elseif** expr **kw_then eoln** eol row elseif
67.	elseif	→	epsilon
23.	statement	→	**kw_if** expr **kw_then eoln** eol row elseif else **kw_end kw_if**
24.	statement	→	**id** order
25.	order	→	+ = expr
26.	order	→	− = expr
27.	order	→	∗ = expr
28.	order	→	/ = expr
29.	order	→	\ = expr
30.	order	→	**equal** expr

11 Použitá literatura

MEDUNA A., LUKÁŠ R.: *Formální jazyky a překladače IFJ, Studijní opora* [online]. 2006, Česká republika. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php?file=%2Fcourse%2FIFJ-IT%2Ftexts%2FOporaIFJ.pdf