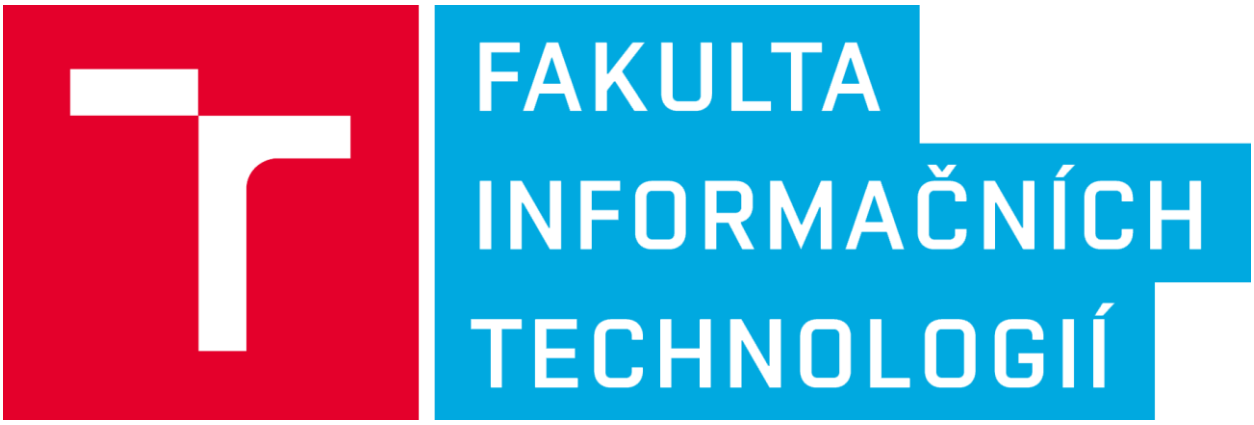


Dokumentace k implementaci překladače imperativního jazyka IFJ17

Tým 76, varianta II

Simeon Borko	xborko02	25 % bodů	Vedoucí týmu
Filip Borčík	xborci01	25 % bodů	Člen týmu
Zdeněk Brhel	xbrhel04	25 % bodů	Člen týmu
Vojtěch Chadima	xchadi05	25 % bodů	Člen týmu

Obsah

1 Úvod

Tato dokumentace popisuje implementaci překladače jazyka IFJ17 (zjednodušená podmnožina jazyka FreeBASIC; jde o imperativní, staticky typovaný jazyk). Tento překladač načte zdrojový kód v jazyce IFJ17 a ten přeloží do cílového jazyka IFJcode17 (mezikód). Proběhne-li překlad bez chyby, vrátí návratovou hodnotu 0, v opačném případě návratovou hodnotu odpovídající povaze chyby.

2 Struktura překladače

Překladač se sestává z těchto hlavních částí:

2.1 Lexikální analyzátor

Lexikální analyzátor načítá ze vstupu programu postupně jednotlivé znaky. Pomocí konečného automatu (jeho diagram vizte v sekci Přílohy) je zpracovává na tokeny, které následně předává syntaktickému analyzátoru k dalšímu zpracování. Bílé znaky a komentáře jsou lexikálním analyzátorem načteny, avšak nejsou dále využívány (jsou tedy de facto přeskakovány).

2.2 Syntakticko-sémantický analyzátor

Syntaktický analyzátor zpracovává jednotlivé tokeny, která získává od lexikálního analyzátoru.

Pro vyhodnocování výrazů jsme zvolili metodu pracující zdola nahoru, konkrétně precedenční syntaktickou analýzu. Pro ostatní programové konstrukce jsme zvolili metodu syntaktické analýzy shora dolů, konkrétně metodu rekurzivního sestupu. Výchozí metodou je tedy rekurzivní sestup, který jakmile narazí na konstrukci typu výraz, předá řízení precedenční syntaktické analýze. Ta výraz zpracuje a rekurzivnímu sestupu vrátí uzel abstraktního syntaktického stromu (o něm více v sekci Implementace) obsahující daný výraz.

Syntaktické a sémantické kontroly jsou spolu natolik provázané, že jsme se rozhodli nevytvářet separátní modul pro sémantickou analýzu, ale spojit tyto dvě analýzy do společného modulu.

2.3 Generátor kódu

Generátor kódu očekává na vstupu abstraktní syntaktický strom (o něm více v sekci Implementace), kterým prochází od kořene k listům a během tohoto průchodu postupně generuje cílový kód IFJcode17. Protože jazyk IFJ17 obsahuje 4 vestavěné funkce, vygeneruje generátor kódu jako první kód IFJcode17 pro tyto vestavěné funkce a až posléze zahájí průchod samotným AST.

3 Implementační detaily

3.1 Abstraktní syntaktický strom

Abstraktní syntaktický strom (AST) je datová struktura, kterou náš překladač vytváří při syntakticko-sémantické analýze. Když je tato analýza dokončena, je celý tento AST (respektive ukazatel na kořen AST) předán generátoru kódu, který na základě postupného procházení generuje cílový kód IFJcode17.

AST se skládá z elementů typu *uzel* a *list. Uzel* obsahuje informaci o svém typu, pole ukazatelů na další elementy (syny tohoto uzlu) a informaci o maximální velikosti tohoto pole ukazatelů. *List* obsahuje informaci o svém typu a ukazatel do paměti, na jehož adrese je uložena hodnota listu.

3.2 Tabulka symbolů

Tabulka symbolů slouží k ukládání informací o jednotlivých proměnných a funkcích. Dle naší varianty zadání je naše tabulka symbolů implementována v podobě tabulky s rozptýlenými položkami (hashovací tabulka).

Překladač pracuje s právě jednou tabulkou symbolů pro funkce, do které jsou postupně zanášeny záznamy o jednotlivých uživatelských funkcích ze zdrojového programu (vestavěné funkce jsou do tabulky zanesené implicitně okamžitě po jejím vytvoření). Každá funkce má potom přidělenou svoji lokální tabulku symbolů pro proměnné. Hlavní tělo programu (mezi klíčovými slovy Scope a End Scope) má také svoji lokální tabulku symbolů pro proměnné.

3.3 Vyhodnocování výrazů v generátoru

Vyhodnocováni výrazů v generátoru jsme nedělali přes zásobník, ale pomocí pomocných proměnných pro mezivýpočty. Problém nastával v tělech cyklů, a proto jsme se rozhodli zpracovávat každou funkci následovně:

Během vytváření kódu pro funkci jsme kromě deklarací všech pomocných proměnných žádný kód nevypisovali, ale veškerý kód jsme si odkládali do fronty. Teprve až po zpracování funkce jsme tento kód vypsali. Tak jsme obešli chybu interpreta, který nemůže dvakrát narazit na deklaraci jedné proměnné.

3.4 Zpracování parametrů funkce

V LL-gramatice (vizte sekci Přílohy) při zpracování parametrů funkce jsou zanesené dvě pravidla s <*real_params*> na levé straně. V naší implementaci těchto dvou pravidel ovšem využíváme booleovský parametr *first*, která indikuje, zda je právě zpracovávaný parametr první nebo nikoliv. Díky tomu nemusíme implementovat dvě různé funkce pro tyto dvě různá pravidla, ale pouze jednu; podle parametru *first* jsme schopni určit, zda máme očekávat čárku. To stejné platí pro <*params*> (formální parametry).

3.5 Zpracování podmínkového výrazu

Nejprve jsme podmínkový výraz rozdělovali v rekurzivním sestupu do podoby *levý výraz – relační operátor – pravý výraz*. Při následném testování se ale tento přístup ukázal být problematický. Byl-li například celý podmínkový výraz uzavřen v závorkách, rekurzivní sestup si s tímto poradit nedokázal. Přistoupili jsme tedy k tomu, že se celý podmínkový výraz zpracovává metodou precedenční analýzy. Toto řešení se ukázalo být funkční.

3.6 Rozdílné názvy parametrů při deklaraci a definici funkcí

Předpokládali jsme, že názvy parametrů při definici funkce se musí shodovat s názvy parametrů při její deklaraci (proběhla-li). Tím pádem jsme tabulku symbolů pro proměnné vytvářeli již při deklaraci (a vkládali jsme do ní parametry funkce). Později jsme ale na základě reálného chování jazyka FreeBASIC zjistili, že názvy parametrů při definici funkce se nemusí shodovat s těmi uvedenými při deklaraci. Přepracovali jsme tedy modul syntakticko-sémantický analyzátor tak, že při deklaraci funkce se parametry do tabulky symbolů pro proměnné neukládají; děje se tak až při definici funkce.

3.7 Posloupnost znaků EOL

V případě, že se ve zdrojovém souboru nachází dva a více znaky End of Line (EOL), syntakticko-sémantický analyzátor by to považoval za chybu. Proto jsme přistoupili k řešení, kdy lexikální analyzátor při posloupnosti dvou a více znaků EOL vrátí pouze jeden token typu EOL.

Další problém představovala situace, kdy zdrojový soubor obsahuje posloupnost *EOL-komentář-EOL*. Museli jsme zajistit, aby i v této situaci, kdy se znaky EOL nevyskytují hned za sebou, ale jsou odděleny komentářem, vrátil pouze jeden token typu EOL. Toho jsme dosáhli použitím statické proměnné indikující typ posledního vráceného tokenu.

3.8 Neočekávaný konec souboru

V syntakticko-sémantickém analyzátoru jsme pro zjednodušení předpokládali, že lexikální analyzátor vždy načte a předá další token. V případě, že syntakticko-sémantický analyzátor narazil na neočekávaný konec souboru, nastala chyba přístupu do paměti (segmentation fault). Řešení tohoto problému spočívalo ve vytvoření makra, které kontrolovalo, zda byl další token načten. Pokud tomu tak nebylo, detekovali jsme tímto konec souboru a program mohl skončit s návratovou hodnotou odpovídající syntaktické chybě.

3.9 Ladící makro

Pro významné usnadnění ladění jsme implementovali ladící makro, které v případě chyby v jakékoliv funkci syntakticko-sémantického analyzátoru vypíše na standardní chybový výstup nejenom jméno zdrojového souboru a číslo řádku, kde daná chyba nastala, ale také i zásobník volání funkcí (takzvaný *stack trace).*

4 Způsob práce v týmu

Jádrem týmové práce byly pravidelné týdenní schůzky, na kterých jednotliví členové týmu prezentovali ostatním to, co za uplynulý týden vytvořili. Následovala diskuze o jednotlivých řešeních a implementačních detailech. Na konci každé schůzky vedoucí týmu rozvrhl a rozdělil práci na další týden.

Pro komunikaci mimo osobní schůzky jsme využívali instant messaging (Facebook Messenger) a webová úložiště (Dropbox). Pro sdílení doposud vytvořeného kódu jsme využívali systém správy verzí (GIT).

U složitějších a rozsáhlejších částí překladače (například implementace syntakticko-sémantického analyzátoru) byla hojně využívána technika párového programování, které se ukázalo být efektivnější než práce jednotlivců na jednotlivých částech překladače.

Testování jednotlivých částí překladače měli na starost převážně jejich autoři, na závěrečném testování překladače jako celku se podíleli všichni členové.

5 Rozdělení práce mezi členy týmu

Simeon Borko je vedoucím projektu. Je autorem lexikálního analyzátoru a spoluautorem syntakticko-sémantického analyzátoru, generátoru kódu, precedenčního syntaktického analyzátoru, tabulky symbolů, abstraktního syntaktického stromu a LL-tabulky.

Filip Borčík je autorem precedenčního syntaktického analyzátoru a s ním spojené precedenční tabulky. Dále se podílel na vytváření generátoru kódu.

Zdeněk Brhel je autorem implementace abstraktního syntaktického stromu, podílel se na tvorbě lexikálního analyzátoru, vytvořil LL-tabulku.

Vojtěch Chadima je autorem tabulky symbolů, podílel se na implementaci syntakticko-sémantické analýzy. Je spoluautorem LL-tabulky. Vytvořil dokumentaci.

6 Závěr

Podařilo se nám implementovat funkční překladač. Za největší úskalí projektu považujeme jeho celkovou časovou náročnost a z ní vyplývající nutnost dobrého rozvržení práce na jednotlivých částech překladače mezi jednotlivé členy týmu a specifikaci vstupů a výstupu jednotlivých modulů, což nebylo úplně snadné.

Práce na tomto projektu měla také za výsledek zlepšení schopnosti spolupráce v týmu, řešení z našeho pohledu netriviálních programátorských úkolů, práce s verzovacím systémem GIT a v neposlední řadě jsme porozuměli mnoha aspektům teorie formálních jazyků a překladačů a nejrůznějších algoritmických struktur.

Celkově je překladač tvořen přibližně 6000 řádky kódu rozdělenými do 20 zdrojových souborů.

7 Reference

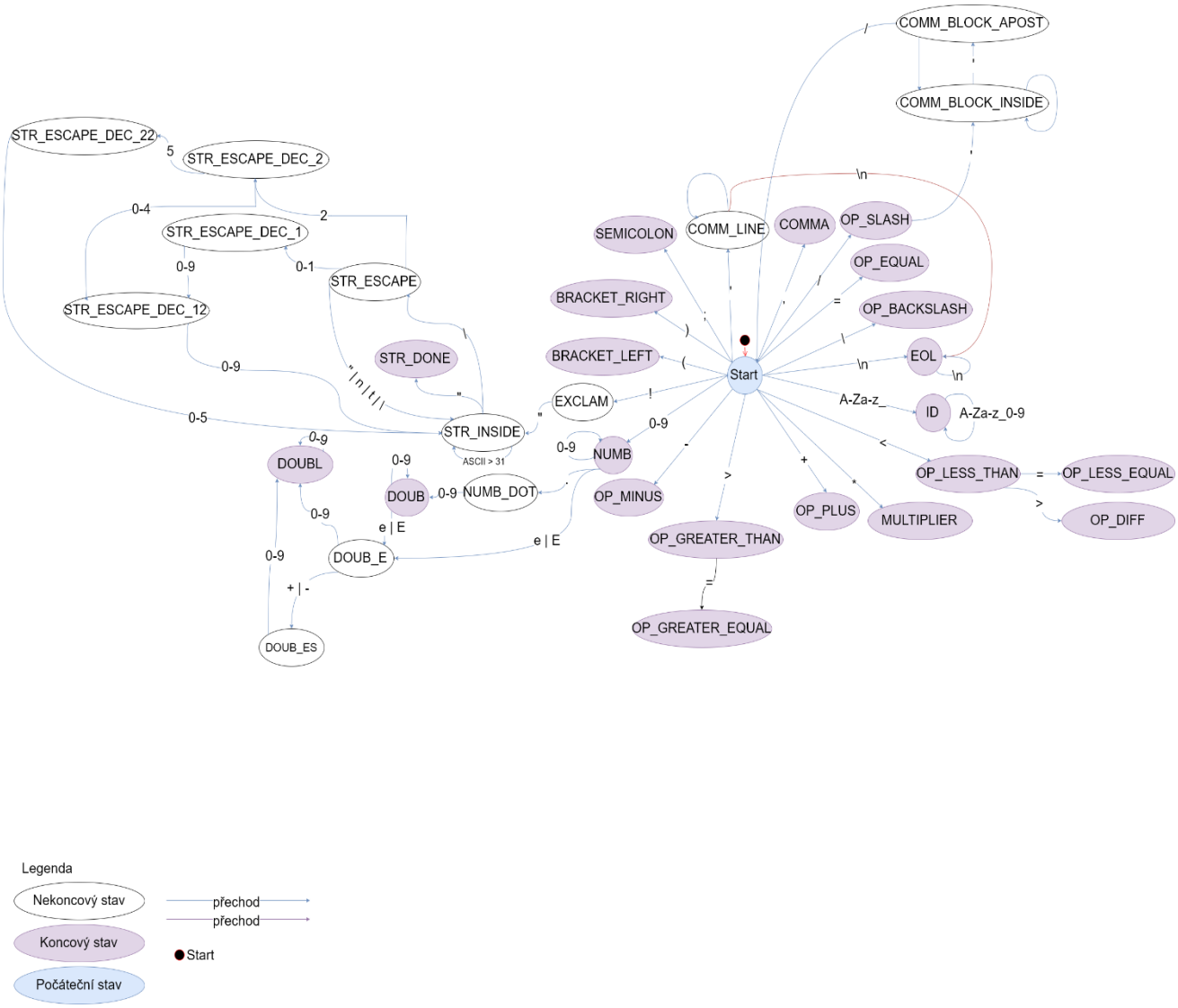
Materiály z přednášek a cvičení předmětů IFJ a IAL.

Wikipedia contributors. Abstract syntax tree. *Wikipedia, The Free Encyclopedia.* [Online] [Citace: 23. Listopad 2017.] https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=804974356.

Yigit, Ozan. Hash functions. *Ozan Yigit.* [Online] [Citace: 20. Listopad 2017.] http://www.cse.yorku.ca/~oz/hash.html.

8 Přílohy

8.1 Diagram konečného automatu



8.2 LL-gramatika

Vysvětlivky:

- Číslo v prvním sloupci udává číslo daného pravidla.
- Označení před šipkou (→) uzavřené v těchto závorkách <…> značí levou stranu pravidla.
- Označení modrým písmem značí neterminál.
- Označení červeným písmem značí terminál či výraz.

1	<prog> → declare <function-header> <prog>
2	<prog> → <function-header> <st-list> end function EOL <prog>
3	<prog> →scope EOL <st-list> end scope
4	<function-header> → function <idf> (<form_params>) as <type> EOL
5	<st-list> → <state> EOL <st-list>
6	<st-list> → ε
7	<form_params> → ε
8	<form_params> → <id> as <type> <form_list>
9	<type> → Integer
10	<type> → String
11	<type> → Double
12	<form_list> → ε
13	<form_list> → , <id> as <type> <form_list>
14	<state> → if <expression> then EOL <st-list> else EOL <st-list> end if
15	<state> → Dim <id> as <type> <ival>
16	<ival> → ε
17	<ival> → = <expression>
18	<state> → print <expression>; <printlist>
19	<printlist> → <expression>; <printlist>
20	<printlist> → ε
21	<state> → <id> = <right_value>
22	<real_params> → ε
23	<real_params> → <term> <real_list>
24	<real_list> → ε
25	<real_list> → , <term> <real_list>
26	<state> → do while <expr> EOL <st-list> loop
27	<state> → return <expression>
28	<state> → input <id>
29	<right_value> → <expression>
30	<right_value> → <idf>(<real_params>)

8.3 LL-tabulka

Z důvodů úspory místa byly některé vstupy zkráceny, zde je jejich seznam:

- decl. = declare
- scp. = scope
- int. = integer
- str. = string
- dbl. = double
- <expr.> = <expression>
- Ret. = return

	ε	decl.	scp.	function	<idf>	<id>	Int	Str	Dbl	if	<expr.>	dim	=	print	<term>	,	do	ret.	input
<prog>		1	3	2															
<function-header>				4															
<st-list>	6					5				5		5		5			5	5	5
<form_params>	7					8													
<type>							9	10	11										
<form_list>	12															13			
<state>						21				14		15		18			26	27	28
<ival>	16												17						
<printlist>	20										19								
<real_params>	22														23				
<real_list>	24															25			
<right_value>					30						29								

8.4 Precedenční tabulka

	*	/	\	+	-	=	<>	<	<=	>	>=	(	)	id	$
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
\	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	<	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	<	>	>	>	>	>	>	>	>	<	>	<	>
=	<	<	<	<	<							<	>	<	>
<>	<	<	<	<	<							<	>	<	>
<	<	<	<	<	<							<	>	<	>
<=	<	<	<	<	<							<	>	<	>
>	<	<	<	<	<							<	>	<	>
>=	<	<	<	<	<							<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>		>		>
id	>	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	