Dokumentace projektu

Implementace překladače imperativního jazyka IFJ18

Tým 083, varianta I

Jakub Man (xmanja00)	**25%**
Jiří Tykva (xtykva00)	25%
Jan Martinák (xmarti83)	25%
Adam Melichar (xmelic22)	25%

5. listopadu 2018

Obsah

1. Úvod 3
2. Návrh a implementace 3
2.1 Lexikální analýza 3
2.2. Syntaktická analýza 3
2.3. Precedenční syntaktická analýza (PA) 3
2.4. Sémantická analýza 3
2.5. Generátor kódu 3
3. Datové struktury 4
3.1. Buffer 4
3.2. Tabulka symbolů 4
3.3. Zásobník 4
3.4. Abstraktní syntaktický strom 4
3.5. Jednosměrně vázaný seznam 4
4. Práce v týmu 5
4.1. Způsob práce 5
4.2. Rozdělení úkolů 5
5. Závěr 5
5.1. Zdroje 5
6. Přílohy 6
A. LL Gramatika 6
B. LL – Tabulka 7
C. Precedenční tabulka – Pravidla 7
D. Precedenční tabulka 8
E. Schéma konečného automatu 9

1. Úvod

Cílem tohoto dokumentu je popsat námi vytvořený překladač imperativního jazyka IFJ18 založeného na jazyce Ruby. Námi zvolené varianta nám ukládala povinnost využít při implementaci tabulky symbolů binární vyhledávací strom.
Práce překladače spočívá v načtení vstupního kódu na vstupu příkazové řádky, na kterém proběhne lexikální, syntaktická a sémantická analýza. Na základě těchto analýz je následně generován cílový kód IFJcode18, který se vypíše na `stdout`.

2. Návrh a implementace

Překladač se dělí do několika nezávislých modulů, které jsou popsány níže.

2.1 Lexikální analýza

Prvním modulem překladače je lexikální analyzátor (scanner), mající za úkol rozložení vstupního programu na jednotlivé lexikální symboly vstupního programu, které dostává parser ve formě tokenů voláním funkce `scanner`. Lexikální analyzátor musí také umět odfiltrovat komentáře, bílé znaky a rozpoznat lexikální chyby. Scanner jsme realizovali jako determinitsický konečný automat jehož schéma je obsaženo v příloze E. Každý token je definován strukturou, která se skládá z pole znaků a čísla značícího typ.

2.2. Syntaktická analýza

Syntaktický analyzátor (parser) je srdcem překladače. Pomocí tokenů získaných ze scanneru simulujeme vytváření derivačního stromu. Derivační strom se simuluje shora dolů podle vytvořené LL gramatiky (příloha A). Pokud parser narazí na výraz, je nutné volat funkci `prec_table`, která má na starosti analýzu výrazů. Jestliže se podaří sestavit derivační strom, program je napsán syntakticky správně, když se sestavení nezdaří, překladač se ukončí s příznakem syntaktické chyby.
Po úspěšné simulaci derivačního stromu, zpracování všech výrazů a kontrole sémantiky vytváří skrz modul generátoru (`generator.c`) jednotlivé příkazy cílového jazyka.

2.3. Precedenční syntaktická analýza (PA)

Využívá se pro zpracování výrazů například v podmínkách jednotlivých funkcí. Po zavolání precedenčního analyzátoru parserem při nalezení výrazu jej PA zpracuje zdola nahoru podle precedenčních pravidel (příloha C), vytvoří pravý rozbor (postfixovou notaci), z kterého generujeme abstraktní syntaktický strom.

2.4. Sémantická analýza

Sémantický analyzátor má na starosti typovou kontrolu proměnných využitých ve výrazech, v případě nutnosti provede přetypování (typicky INT + FLOAT). Pro tyto kontroly využívá tabulku symbolů, do které se vkládají všechny využité identifikátory, které třídíme na lokální a globální.

2.5. Generátor kódu

Generátor cílového kódu (v našem případě IFJcode18) vytváří pomocí tříadresného kódu získaného ze syntaktického analyzátoru.

3. Datové struktury

3.1. Buffer

Buffer využíváme pro ukládání znaků jednotlivých lexémů při jejich zpracovávání. Lexikální analyzátor vkládá znaky lexému do bufferu voláním funkce `add_to_buffer()`, jakmile je je analyzátorem vyhodnocen jako kompletní, tak jej zavoláním funkce `send_buffer()` vloží do struktury výstupního tokenu, který je následně předán do parseru.
Velikost bufferu je defaultně nastavena na 4, pokud hrozí při dalším znaku dosažení maximální velikosti bufferu, je velikost zdvojnásobena.

3.2. Tabulka symbolů

Náš cílový jazyk využívá tři paměťové rámce (globální, lokální a dočasné), pro jejich realizaci si však vystačíme s jednou tabulkou symbolů pro globální proměnné a následně generujeme tabulku symbolů pro každou definovanou funkci. Do tabulky symbolů se ukládají identifikátory o všech využitých proměnných, funkcích a jejich parametrech. Tyto informace jsou nezbytné pro generování kódu, říkají nám, jestli už byla daná proměnná či funkce využita, jakého je typu, nebo jestli je deklarována.
Tabulky symbolů máme vytvořenou jako binární vyhledávací strom, a to konkrétně rekurzivní metodou.

3.3. Zásobník

Zásobník neboli stack je potřebný při zpracování syntaktické analýzy, kde se do něj nejprve vkládají očekávaná pravidla LL gramatiky (příloha A) a následně se porovnávají se vstupem, když vstupní lexém odpovídá, tak se ze zásobníku odstraní vrchní lexém a pokračujeme s kontrolou, dokud nejsme na dnu zásobníku. Když ale porovnávaný lexém neodpovídá, tak jsme zjistili syntaktickou chybu.
Dále využíváme zásobník při zpracovávání výrazů, kde ukládáme identifikátory a konstanty užité ve výrazu. Do dalšího zásobníku ukládáme pravidla pravého rozboru a pomocí těchto dvou zásobníků vytváříme abstraktní syntaktický strom.

3.4. Abstraktní syntaktický strom

Po provedení všech analýz a zjištění všech informací o struktuře vstupního kódu je zapotřebí vygenerovat vnitřní formu, která je v našem případě reprezentována abstraktním syntaktickým stromem.
Vnitřní uzly tohoto syntaktického stromu tvoří operátory a jejich listy jsou operandy.
Využíváme průchodu `PostOrder()`. Po vytvoření stromu provedeme konverzi datových typů a prověříme jejich platnost.

3.5. Jednosměrně vázaný seznam

Do tohoto seznamu ukládáme parametry funkcí a tabulku symbolů.

4. Práce v týmu

4.1. Způsob práce

Po několika úvodních schůzkách týkajících se rozvržení práce a způsobu programování jsme začali nejprve samostatně, později však kolektivně studovat látku potřebnou k správnému pochopení funkce překladače. V aplikaci Trello jsme si sepsali všechny potřebné úkoly a pomocí této aplikace jsme sledovali průběh práce. Komunikace v teamu probíhala především osobně nebo pomocí Facebooku.
Samotné programování překladače, jsme začali řešit koncem října a to téměř vždy při hromadném setkání v prostorech fakulty. Při práci jsme využívali metod agilního programování, především párové nebo extrémní programování.
Pro tvorbu programu jsme využívali textový editor Atom s rozšířením Teletype, které umožňovalo pracovat zároveň více lidem na jednom souboru. Testování probíhalo buď s pomocí aplikace Cygwin nebo přímo na Ubuntu. Verzování souborů probíhalo pomocí softwaru Git a repozitáře GitHub.
Pro ověření správné funkce jednotlivých dílčích částí překladače jsme si vytvořili komplexní sadu testů.

4.2. Rozdělení úkolů

- Jakub Man – Vedoucí týmu, testování, generátor
- Adam Melichar – Scanner, parser, sémantický analyzátor
- Jiří Tykva – Scanner, parser, sémantický analyzátor
- Jan Martinák – Scanner, dokumentace, prezentace

5. Závěr

Projekt jsme bohužel začali zpracovávat později, než jsme měli naplánováno, což nám zkomplikovalo pozdější práci a poslední dva týdny vypracovávání projektu byly vcelku hektické. Při programování jsme narazili na řadu problémů v nejasnosti zadání, postupně jsme je ale vyřešili díky informacím z fóra předmětu, nebo pečlivějším přečtením zadání. Celá naše skupina se shodla na tom, že největší přínos tohoto projektu byl v poznání skupinové práce na složitějším projektu.

5.1. Zdroje

- ČEŠKA, Milan, Tomáš HRUŠKA a Miroslav BENEŠ. Překladače. Brno: Vysoké učení technické, 1993. ISBN 80-214-0491-4.
- ČEŠKA, Milan a Zdena RÁBOVÁ. Gramatiky a jazyky. 3. vyd. Brno: Vysoké učení technické, 1988.
- ČEŠKA, Milan a Tomáš HRUŠKA. Gramatiky a jazyky: cvičení. 2. vyd. Brno: Vysoké učení technické, 1988.

6. Přílohy

A. LL Gramatika

<PROG> → def id (<PARAMS>) eol <STATEMENT_N> end eol <PROG>
<PROG> → STATEMENT eol <PROG>
<PROG> → ε
<PARAMS> → id <PARAMS_N>
<PARAMS> → ε
<PARAMS_N> → , id <PARAMS_N>
<PARAMS_N> → ε
<STATEMENT> → if expression then eol <STATEMENT_N> else eol <STATEMENT_N> end
<STATEMENT> → while expression do eol <STATEMENT_N> end
<STATEMENT> → id <DEF_ARGS>
<STATEMENT> → ε
<STATEMENT_N> → <STATEMENT> eol <PROG>
<STATEMENT_N> → ε
<DEF_ARGS> → = <DEFINE>
<DEF_ARGS> → <ARGS>
<DEFINE> → id <ARGS>
<DEFINE> → expression
<DEFINE> → ε
<ARGS> → ε
<ARGS> → (<ARGS>)
<ARGS> → <VALUE> <ARGS_N>
<ARGS_N> → , <VALUE> <ARGS_N>
<ARGS_N> → ε
<VALUE> → int
<VALUE> → float
<VALUE> → id
<VALUE> → string
<VALUE> → nil

B. LL – Tabulka

	def	id	(	)	eol	end	,	if	expression	then	else	while	do	=	int	float	string	nil	$
PROG	1	2			2	3		2			3	2							3
PARAMS		4		5															
STATEMENT_N		12			12	13		12			13	12							
STATEMENT		10			11			8				9							
PARAMS_N				7			6												
DEF_ARGS		15	15		15									14	15	15	15	15	
DEFINE		*			18				17										
ARGS		21	20	19	19										21	21	21	21	
VALUE		26													24	25	27		
ARGS_N				23	23		22												

*Poznámka: Pravidlo 16 platí pro uživatelem definované funkce, pravidlo 17 pro proměnné.

C. Precedenční tabulka – Pravidla

E → E + E
E → E - E
E → E * E
E → E / E
E → (E)
E → i
E → int
E → float
E → string
E → E < E
E → E <= E
E → E > E
E → E >= E
E → E == E
E → E != E

D. Precedenční tabulka

	+	-	*	/	<	>	≤	≥	==	!=	(	)	i	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<					>	>	<	>	<	>
>	<	<	<	<					>	>	<	>	<	>
≤	<	<	<	<					>	>	<	>	<	>
≥	<	<	<	<					>	>	<	>	<	>
==	<	<	<	<	<	<	<	<			<	>	<	>
!=	<	<	<	<	<	<	<	<			<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
i	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	

E. Schéma konečného automatu