Vysoké Učení Technické v Brně
Fakulta informačních technologií

Dokumentácia k projektu IFJ

Implementácia prekladača imperatívneho jazyka IFJ19

Tím 072, varianta II

Šimon Sedláček	**(xsedla1h)**	**25%**
Marek Žiška	(xziska03)	25%
Martin Osvald	(xosval03)	25%
Marek Sarvaš	(xsarva00)	25%

Brno, 11.12.2019

Contents

1 Úvod

Tento dokument popisuje návrh a implementáciu prekladača imperatívneho jazyka IFJ19, čo je zjednodušená podmnožina jazyka Python 3. Cieľovým jazykom je trojadresný IFJcode19.

Prekladač je implementovaný ako konzolová aplikácia, čiže načíta vstup zo štandardného vstupu, chybové hlášky sa vypisujú na štandardný chybový výstup a vygenerovaný medzikód IFJcode19 na štandardný výstup.

2 Lexikální analýza

Lexikálna analýza ¨je implementovaná v module scanner.c. Úlohou lexikálnej analýzy je rozpoznávanie validných alebo chybných lexémov (tokenov) v prekladanom zdrojovom súbore. Postupne ako syntaktická analýza vyhodnocuje LL pravidlá, tak zároveň volá lexikálnu analýzu o nasledujúce tokeny, teda poskytuje rozhranie pre syntaktickú analýzu formou funkcie get_token().

Lexikálna analýza vždy vracia jeden aktuálny a validný token. Správnosť tokenu sa určuje pomocou konečného automatu, ktorý je popísaný nižšie formou diagramu.

3 Syntaktická analýza

Ústrednou časťou prekladača je syntaktický analyzátor, ktorý má jadro v module parser.c. V tomto module sa nachádza implementácia pravidiel pre rekurzívny zostup, volá sa tu precedenčná syntaktická analýza, funkcie pre generovanie častí kódu, sémantické akcie. Druhou časťou syntaktického analyzátoru je potom modul precedence_analysis.c, ktorý má na starosť syntaktickú analýzu výrazov.

Syntaktický analyzátor využíva funkciu get_token() poskytovanú lexikálnym analyzátorom a implementuje nad ňou abstraktnejšiu operáciu next_token() s ošetrením chybových stavov a prípadným načítáním uloženého tokenu z odkladiska.

Súčásťou syntaktického analyzátoru je globálna štruktúra parser_data, ktorá okrem inštancií lokálnych a globálnych tabuliek symbolov obsahuje premennú curr_token uchovávajúcu aktuálny token, odkladisko tokenov token_stash, ukazatele na symboly, aktuálne definované a aktuálne volané funkcie, indikátor in_function, ktorý obsahuje informaciu o tom, či sa nachádzame vo vnútri definície funkcie a indikátor psa_state. Psa_state obsahuje informáciu pre PSA,ktorá jej umožňuje podľa kontextu spracovávaného výrazu vykonávať sémantické akcie a generovať kód pre príslušné konštrukcie. Ako ďalšie táto štruktúra obsahuje premenu undef_symbol, ktorá umožňuje vykonávať niektoré sémantické kontroly týkajúce sa tieňovania globálnych premenných a obecné definície premenných.

Syntaktickú analýzu hlavného tela programu sme založili na LL(1) gramatike. Pravidlá implementované v module parser.c reflektujú nami vytvorenú LL gramatiku s výnimkou dvoch prípadov:

- Pravidlá pre neterminály <term> a <literal> neboli vo výslednej implementácii syntaktického analyzátoru využité - vo výslednej implementácii sa ukázali ako zbytočné a bolo prehľadnejšie tieto pravidla implementovat priamo v rámci iných pravidiel.
- Pre pravidlá <value> a <rest> bolo nutné nájsť spôsob ako odlíšiť začiatok aritmetického výrazu od volania funkcie. Z tohto dôvodu sme implementovali jednoduchú štruktúru token_stash, ktorá nám umožňuje odložiť na stranu aktuálne zpracovávaný token a pozrieť sa až o dva tokeny dopredu. Vďaka tomuto nahliadnutiu je pak syntaktický analyzátor schopný rozhodnúť, či se bude jednať o výraz (pričom v tomto prípade je predané riadenie PSA) alebo volanie funkcie.

3.1 Precedenčná syntaktická analýza

Precedenčná syntaktická analýza je implementovaná v súbore precedence_analysis.c. Túto analýzu volá syntaktický analyzátor rovnakým spôsobom ako iné pravidlá z LL gramatiky, ak narazí na výraz.

Spracovávaný výraz sa vyhodnocuje pomocou precedenčnej tabuľky a pravidiel. V tabuľke je skrátený zápis relačných operátorov(riadok a stĺpec označený “r”), pretože majú rovnakú

asociativitu a prioritu. Takýmto spôsobom sú relačné operátory spracované aj vo výslednom programe, kde sú navyše takto skrátene aj operátory "+","-" a "*","/","//". Na vyhodnocovanie výrazu je použitý zásobník, na ktorého začiatok sa vloží "¨$". Pomocou prvého terminálu na zásobníku a novo načítaného tokenu sa získa operácia z tabuľky(vloženie na zásobník, redukcia alebo koniec analýzy). Pri redukcii sa aplikuje pravidlo, ktoré odstráni dané terminály zo zásobníku a pridá neterminál na jeho vrchol.

Analýza končí pri absencii pravidla počas operácie redukcie. Každý novo načítaný token sa ukladá do poľa v infixovej notácii kvôli nasledujúcej sémantickej analýze daného výrazu, ktorá sa volá až po úspešnom dokončení syntaktickej precedenčnej analýzy.

4 Sémantická analýza

Sémantické akcie využívané v rámci rekurzívneho zostupu a z časti aj PSA sú implementované v module semantic.c. Okrem štandardných sémantických kontrôl týkajúcich sa štandartných prípadov definícií premenných a funkcií sme pri implementácií narazili na dva zložitejšie prípady sémantických akcií.

4.1 Referencia lokálnej promennej pred jej definíciou

Jedným z týchto prípadov je referencia globálnej premennej vo funkcii a následná definícia lokálnej premennej s rovnakým menom. Pre odhalenie tohto prípadu sa pri využití globálnej premennej vo funkcii uloží do lokálnej tabuľky symbolov nová nedefinovaná lokálna premenná. Ak by sa užívateľ pokúsil následne definovať túto lokálnu premennú, sémantická analýza ukončí preklad s chybou 3.

4.2 Kontrola dependencií funkcií a lineárny zoznam symbolov

Kvôli kontrole dependencií funkcií pri vzájomných volaniach, je pri volaní funkcie z hlavného tela programu nutné prejsť rekurzívne zoznamy dependencií všetkých dependencií volanej funkcie. Ak sú všetky dependencie definované, potom je volanie funkcie v poriadku. Tu však nastal problém s možnosťou nekonečnej rekurzie, ak by se dve alebo viacero funkcií volalo kruhovo navzájom. Tento problém sme riešili pomocou lineárneho zoznamu už skontrolovaných funkcií - ak sa symbol kontrolovanej funkcie už nachádza v tomto zozname, nie je nutné kontrolovať ďalej jeho dependencie.

4.3 Precedenčná sémantická analýza

Precedenčná sémantická analýza je volaná po úspešnom skončení precedenčnej syntaktickej analýzy, ktorá jej predáva pole token-ov v infixovej notácii. Ako prvá sa prevedie kontrola deklarácie všetkých premenných vo výraze, ak sa nájde nedefinovaná premenná sémantická analýza končí chybou. Po kontrole definícií premenných sa prevedie konverzia infixovej notácie na postfixovú. Následne sa postfixový zápis vyhodnocuje. Všetky operátory sú binárneho typu preto sa vždy vyhodnocujú dva operandy a operátor. Táto dvojica operandov je sémanticky kontrolovaná na správne typy vzhľadom na daný operátor prípadne delenie nulou pri celočíselnom delení.

Po úspešnom ukončení sémantickej analýzy sa podľa hodnoty psa_state volá generovanie kódu, ktorému sa ako parametre predáva aktuálna dvojica operandov a operátor. Úspešné vygenerovanie trojadresného kódu vráti názov premennej medzi-výsledku, ktorá sa ako token uloží na zásobník vyhodnocovania postfixu pre ďalšie spracovanie.

5 Generovanie cieľového kódu

V priebehu syntaktickej a sémantickej analýzy sú v rámci rekurzívneho zostupu a PSA volané funkcie pre generovanie výsledného kódu. Cieľový kód je uchovávaný v štruktúre s názvom generate_strings_output_t. Táto štruktúra obsahuje päť dynamických stringov, ktoré sa volajú : main, function_definitions, errors, stash, print. Do týchto stringov sa ukladá cieľový kód jazyka. Na začiatku generovania sa alokujú dynamické stringy. Do stringu function_definitions sa pridávajú vstavané funkcie jazyka IFJ19. Do stringu errors sa pridajú chybové náveští s návratovými kódmi.

5.1 Generovanie definície premenných

Keď sme riešili problém definícií premenných v cykloch while a if-else, tak sme prišli s riešením s pomocí stringu stash. Definície premenných se vždy generujú do stringu main alebo function_definitions v závislosti na tom, či sa aktuálne nachádzame v globálnom alebo v lokálnom kontexte. Ostatné príkazy a konštrukcie sa generujú do stringu stash. Pri opustení aktuálneho kontextu (lokálneho či globálneho) sa string stash pripojí na koniec stringu main alebo function_definitions.

5.2 Generovanie unikátnych názvov premenných, náveští

Na generovanie názvov náveští a premenných využívame štruktúru unique_id_t. Táto štruktúra má deväť int-ových atribútov, ktoré slúžia ako počítadlá na vytvorenie identifikátorov. Tieto atribúty potom používame ako parametre do funkcie create_unique_variable, ktorá nám vytvorí unikátny názov. Podľa potreby potom jednotlivé atribúty z unique_id_t iterujeme, aby sa nám zachovala unikátnosť.

5.3 Generovanie výrazu

Každý výraz je behom syntaktickej analýzy ukladaný do dátového zásobníku . V rámci postfixu je tento výraz spracovávaný a popritom je volaná funkcia generate_expression. Táto funkcia prijíma vždy dva operandy a jeden operátor, a na základe daného operátoru prepína medzi tým aký kód sa vygeneruje pre daný výraz. Pri generovaní voláme pomocné funkcie ako check_if_op_type_eq, generate_jumpeq_string_string, convert_ints_to_floats, append_token_variable_to_assembly, append_token_operands_to_assembly, ktoré nám pomáhajú uľahčiť kód a jeho sprehľadnenie.

6 Dátové štruktúry a použité algoritmy

6.1 Tabuľka symbolov

Súčasťou zadania zo strany predmetu IAL bol požiadavok na implementáciu tabulky symbolov. V našom prípade sa jednalo o druhú variantu zadania - tabulka symbolov mala být implementována ako tabulka s rozptýlenými položkami.

Tabuľku symbolov sme teda implementovali jako TRP s explicitním zreťazením. Ako rozptyľovaciu funkciu sme použili variantu funkce PJW, GNU Hash ELF. Veľkosť pole bucketov v tabuľke sme zvolili na 27457 položek. Pre takto vysokú hodnotu bolo nepravdepodobné, že by naplnenie tabulky presiahlo 75% a začalo by dochádzať ku kolíziam a to ani v prípade väčších vstupných programov.

Položkou v tabuľke symbolov je potom štruktúra symbol, ktorá má okrem ukazovateľa na ďalší prvok v zozname a svojho identifikátoru ešte atribúty type (určuje typ symbolu - funkcia, premenná) a attributes (únia uchovávajúca atribúty špecifické pre daný typ symbolu). Pre premennú má únia attributes len položku defined, ktorá je využívaná pri sémantických kontrolách. Pre funkciu má únia attributes ešte položky param_count (počet parametrov potrebných pre volanie funkcie).

Ďalej dvojicu špeciálnych atribútov, ktoré sa využívajú pri sémantických kontrolách dependencií funkcií pri vzájomných volaniach. Jedná sa o atribút depends - dynamické pole ukazovateľov na symboly iných funkcií; a atribút dep_len, ktorý drží informáciu o aktuálnej veľkosti poľa dependencií.

6.2 Nekonečný string

Pretože bolo nutné mať v rámci lexikálnej analýzy možnosť načítať ľubovoľne dlhé lexémy, implementovali sme si vlastnú dynamickú štruktúru string, ktorá nám prácu s potenciálne nekonečnými reťazcami umožňuje. Táto štruktúra taktiež našla využitie v rámci generovania kódu, kde slúži ako vyrovnávacia pamäť a umožňuje flexibilnejšiu manipuláciu s vygenerovaným kódom.

6.3 Obecný stack

Pri implementaci sme v rámci lexikálnej analýzy, PSA i generovania kódu pre niektoré operácie potrebovali využít strukturu stack. Prípady využitia stacku sa však líšily natoľko, že sme ako najlepšie riešenie vyhodnotili implementaciu obecného stacku založeného na jednosmernom viazanom zozname, ktorý môže uchovávat data obecného typu.

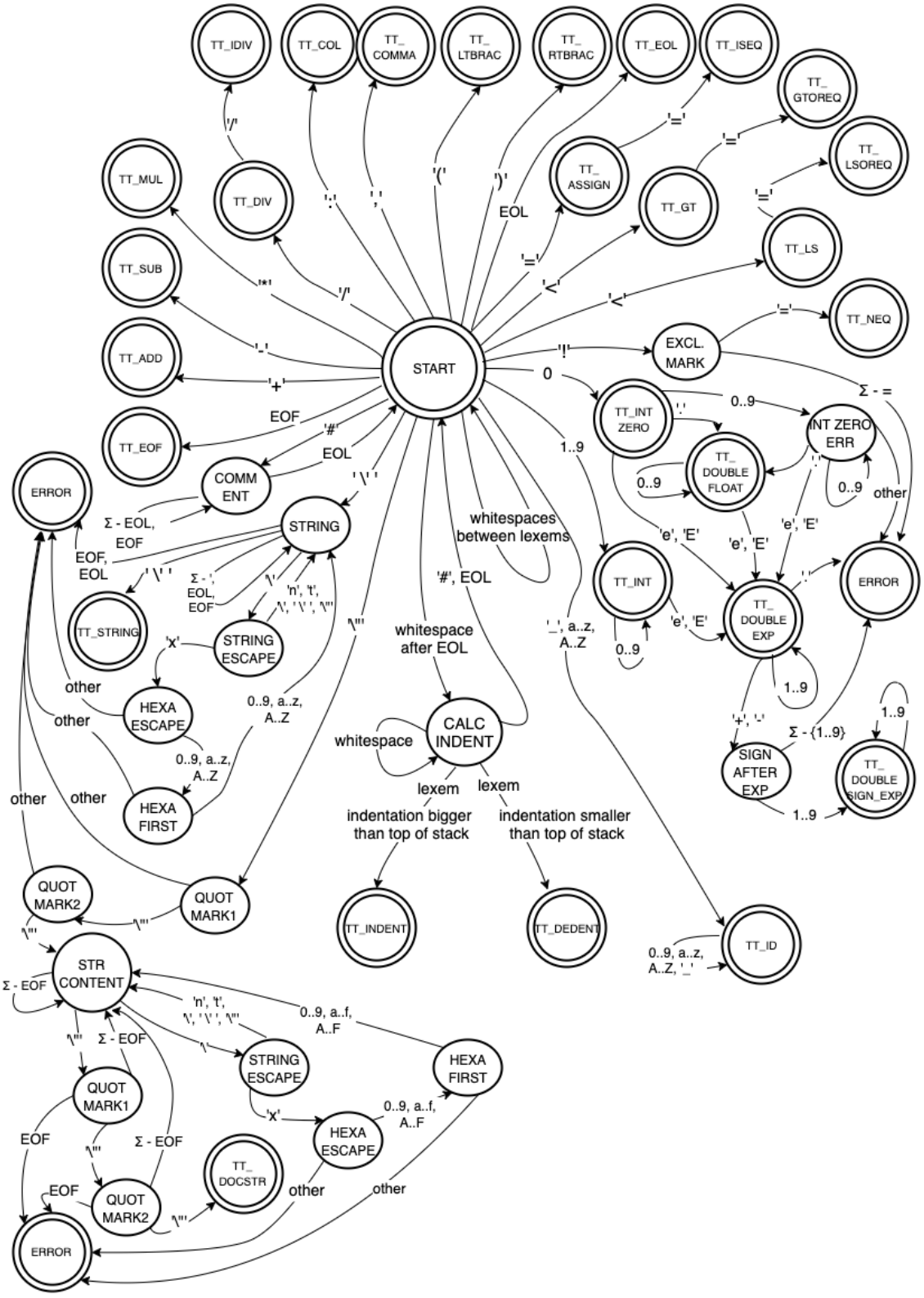
7 Tímová práca

V rámci tímovej spolupráce sme využívali verzovací systém Git. Pre komunikáciu sme zvolili službu Discord. Zo začiatku sme mali problém projekt obsiahnuť - dlho nám trvalo, kým sme sa ako tím získali jasný prehľad o tom, akým smerom sa má projekt uberať. Napriek tomu sa nám nakoniec projekt podaril včas a dúfame v štastný koniec.

Prácu na projekte sme medzi členov tímu rozdelili nasledovne:

Šimon Sedláček (xsedla1h)	tabuľka symbolov, syntaktická analýza, sémantická analýza, dynamický string, testovanie
Marek Sarvaš (xsarva00)	precedenčná syntaktická a sémantická analýza, testovanie, dokumentácia
Marek Žiška (xziska03)	lexikálna analýza, generovanie kódu, testovanie, dokumentácia
Martin Osvald (xosval03)	generovanie kódu, obecný stack, dynamický string , testovanie, dokumentácia

8 Diagram konečného automatu použitého v lexikálnom analyzátore



9 LL gramatika

1 <program> → ε
2 <program> → <statement> <program>
3 <program> → <function_def> <program>
4 <statement_list> → ε
5 <statement_list> → <statement> <statement_list>
6 <function_def> → def id (<param_list_def>) : eol indent <statement> <statement_list> dedent
7 <param_list_def> → ε
8 <param_list_def> → id <params_def>
9 <params_def> → ε
10 <params_def> → , id <params_def>
11 <param_list> → ε
12 <param_list> → TERM <params>
13 <params> → ε
14 <params> → , <term> <params>
15 <statement> → pass eol
16 <statement> → <if_else>
17 <statement> → <cycle>
18 <statement> → <function_ret> eol
19 <statement> → <value> eol
20 <value> → id <rest>
21 <value> → expr
22 <rest> → = expr
23 <rest> → expr
24 <rest> → <function_call>
25 <function_call> → (<param_list>)
26 <function_ret> → return <retvalue>
27 <retvalue> → ε
28 <retvalue> → expr
29 <if_else> → if expr : eol indent <statement> <statement_list> dedent else : eol indent <statement> <statement_list> dedent
30 <cycle> → while expr : eol indent <statement> <statement_list> dedent
31 <term> → id
32 <term> → <literal>
33 <literal> → int
34 <literal> → float
35 <literal> → string
36 <literal> → none

10 LL tabuľka

	def	id	(	)	eol	dedent	,	pass	expr	=	return	if	while	int	float	string	none	eof / $
<program>	3	2						2	2		2	2	2					1
<statement>		19						15	19		18	16	17					
<function_def>	6																	
<statement_list>		5				4		5	5		5	5	5					
<param_list_def>		8		7														
<params_def>				9			10											
<param_list>		12		11										12	12	12	12	
<term>		31												32	32	32	32	
<params>				13			14											
<if_else>												29						
<cycle>													30					
<function_ret>											26							
<value>		20							21									
<rest>			24						23	22								
<function_call>			25															
<retvalue>					27				24									
<literal>														33	34	35	36	

11 Tabuľka popisujúca precedenčnú analýzu

	+	-	*	/	//	(	)	i	r	$
+	>	>	<	<	<	<	>	<	>	>
-	>	>	<	<	<	<	>	<	>	>
*	>	>	>	>	>	<	>	<	>	>
/	>	>	>	>	>	<	>	<	>	>
//	>	>	>	>	>	<	>	<	>	>
(	<	<	<	<	<	<	=	<	<	0
)	>	>	>	>	>	0	>	0	>	>
i	>	>	>	>	>	0	>	0	>	>
r	<	<	<	<	<	<	>	<	0	>
$	<	<	<	<	<	<	0	<	<	0