Projektová dokumentace

Překladač jazyka IFJ19

Tým 040, Varianta II

Boris Burkalo	(xburka00)	34%
Jiří Herrgott	(xherrg00)	33%
Jan Klusáček	(xklusa14)	33%

Prosinec 2019

Obsah

1. Úvod

Zadáním projektu bylo vytvořit program v jazyce C, který načte zdrojový kód napsaný ve zdrojovém jazyce IFJ19 a přeloží jej do cílového jazyka IFJcode19 (mezikód). Jazyk IFJ19 je zjednodušenou podmnožinou jazyka Python3, a jedná se o dynamicky typovaný imperativní jazyk s funkcionálními prvky.

2. Tým – rozdělení

2.1. Týmová práce

V polovině října jsme se začali jako tým pravidelně scházet v prostorách školy. Nejprve jsme si společně pročítali zadání a snažili jsme se rozdělit si práci. Zpočátku se jednalo jen o malý úsek, a potom jsme si práci dělili podle aktuálních potřeb. Mimo rozdělených úkolů jsme si navzájem pomáhali s různými problémy.

Náš tým zvolil variantu II, která k implementaci použila tabulku s rozptýlenými položkami. Ke zhotovení projektu jsme využili několik dílčích částí, které budou popsané v následujících kapitolách. K řešení jsme využívali informace získané na přednáškách a stránce projektu, ale také jsme si sami vyhledávali potřebné informace na internetu.

2.2. Vývojové prostředí + verzování

Všichni členové týmu byli zvyklí používat operační systém Linux (PopOS, Ubuntu), a tak nikde nenastával problém s kompatibilitou.

K práci jsme využívali verzovací systém GitHub, kde nám stačilo využít jednu větev, jelikož na daném souboru pracovali vždy maximálně dva lidé, a tak díky vzájemné komunikaci (hlavně pomocí aplikace Discord) nebyl problém s mergováním. Využívali jsme editor Atom, který má skvělé propojení s GitHubem, a také má vlastní nástroj pro mergování. Nicméně občas bylo nutné osobní setkání, např. kvůli propojení některých modulů.

Funkčnost jsme testovali vzdáleně na CentOS Merlin, a paměťové úniky pomocí programu Valgrind.

2.3. Tabulka rozdělení práce

Boris Burkalo	Vedení týmu, parser
Jiří Herrgott	Lexikální analyzátor, generování kódu
Jan Klusáček	Parser, dokumentace

3. Implementace

3.1. Lexikální analýza

Hlavním modulem pro lexikální analýzu je modul `scanner`. Stavebním kamenem `scanneru` je funkce `int get_next_token(Token *token)`, která čte znak po znaku vstupní soubor, a ve switch case přechází do jednotlivých stavů konečného automatu (viz obr. 1). V případě načtení znaku patřícího dalšímu tokenu se volá funkce `ungetc`, a na vyžádání `parseru` se začíná načítat další token. Funkce `get_next_token` vrací příslušnou `int` hodnotu v případě neúspěchu, jinak vrací 0. Do struktury `Token` předává typ daného lexému, a případně jeho data. V případě, že se jedná o typ identifikátor, kontroluje se, jestli se nejedná o klíčové slovo.

Kromě posílání tokenů do `parseru` má `scanner` za úkol také odstranit z kódu nepotřebné části, jako jsou například komentáře.

Dalším modulem, který se využívá pro `scanner`, je modul `strings`, ve kterém se nachází struktura a funkce pro práci s dynamickým stringem. Modul obsahuje funkce například pro přidání a odebrání znaku, přidání celého řetězce, nebo dokonce vložení řetězce na určitou pozici v dynamickém stringu.

3.2. Syntaktická a sémantická analýza

Tělem syntaktické a sémantické analýzy je modul `parser`, který však využívá další moduly popsané níže. Funguje na základě analýzy shora dolů a řídí se pravidly LL(1) gramatiky. Z modulu `main` je volána funkce `prog`, která se postará o vše potřebné pro zahájení parsování – inicializace `scanneru`, vytvoření tabulky s rozptýlenými položkami atd. Funkce `body` si žádá o tokeny ze vstupního souboru pomocí funkce `get_next_token`, které jsou předány ze `scanneru` a již prošly lexikální analýzou. Následně volá další funkce, pomocí kterých se zpracovávají celé příkazy.

3.2.1. Kontrola definicí funkcí a proměnných

V případě, že se v zdrojovém souboru vyskytne volání funkce, nebo dojde k použití libovolné proměnné, je nutno zkontrolovat, zda funkce, nebo proměnná již byla definována. Tato kontrola je realizována vždy před použitím proměnné, nebo voláním funkce, pomocí funkce nad tabulkou symbolů – `hSymtab_it* symtab_it_position(char *searched_for, hSymtab *sym_tab)`. Tato funkce prohledá celou tabulku symbolů, pokud hledanou položku najde, vrátí ji, v opačném případě vrátí `NULL`. Pro hledání proměnných je zde také pomocná funkce `int is_item_var_defined(char *desired, hSymtab *sym_tab)`, která se podívá, zda u proměnných je nastavená hodnota `defined na true`.

Pokud dojde k volání funkce v definici jiné funkce, před samotnou definicí volané funkce, volaná funkce a počet parametrů, se kterými byla zavolána, se uloží na zásobník, který se na konci parsování celý vyprázdní a zkontroluje (ve funkci `int sym_stack_pop_all(hSym_fct_node* f_top, hSymtab* act_table)`), zda všechny volané funkce byly také definovány.

3.2.2. Výrazy

Na zpracování výrazů se používá precedenční tabulka, která je implementována ve zvláštním modulu `expression`.

	* /	//	+ -	R	(	)	ID	$
* /	>	>	>	>	<	>	<	>
//	>	>	>	>	<	>	<	>
+ -	<	<	>	>	<	>	<	>
R	<	<	<	>	<	>	<	>
(	<	<	<	<	<	E	<	N
)	>	>	>	>	<	>	N	>
ID	>	>	>	>	<	>	N	>
$	<	<	<	<	<	>	<	N

Legenda

< push vstupního symbolu na zásobník

> pop ze zásobníku (v případě ID, hned následuje push na zásobník ID)

E pravá závorka – pop ze zásobníku až po levou závorku (včetně)

N nedefinovaný stav (chyba)

3.3. Generování kódu

V modulu `generator` se nacházejí funkce, které jsou volány za běhu parsování a postupně vytvářejí dynamický string, obsahující výsledný kód. String je poslán na standartní výstup v případě, že nenastala žádná chyba. Pomocný modul `generator_functions` slouží pro generování vestavěných funkcí.

Při generování kódu jsme narazili na menší problém, a to, že definice funkce může být v jazyce IFJ19 kdekoli v těle programu. V našem výsledném kódu se tedy nacházela na stejném místě jako ve zdrojovém kódu, a interpret do ní vešel při procházení programem.

Vyřešili jsme to tím, že jsme vytvořili dva dynamické stringy, jeden pro funkce a druhý pro zbytek programu. Tyto stringy se na závěr konkatenují, a tak jsou funkce vždy nad tělem programu.

4. Závěr

Před odevzdáním nám velmi pomohlo zkušební odevzdání. Vyděsilo nás, jak málo procent jsme dostali, a tak jsme si domluvili konzultace s Ing. Zbyňkem Křivkou Ph.D. Pomohl nám a navedl nás na chybu, kvůli které většina testů neprošla. Mimo to jsme ale objevili i další chyby, které jsme poté měli možnost opravit. Celkově nás projekt překvapil svým rozsahem, ale mnoho nás naučil.

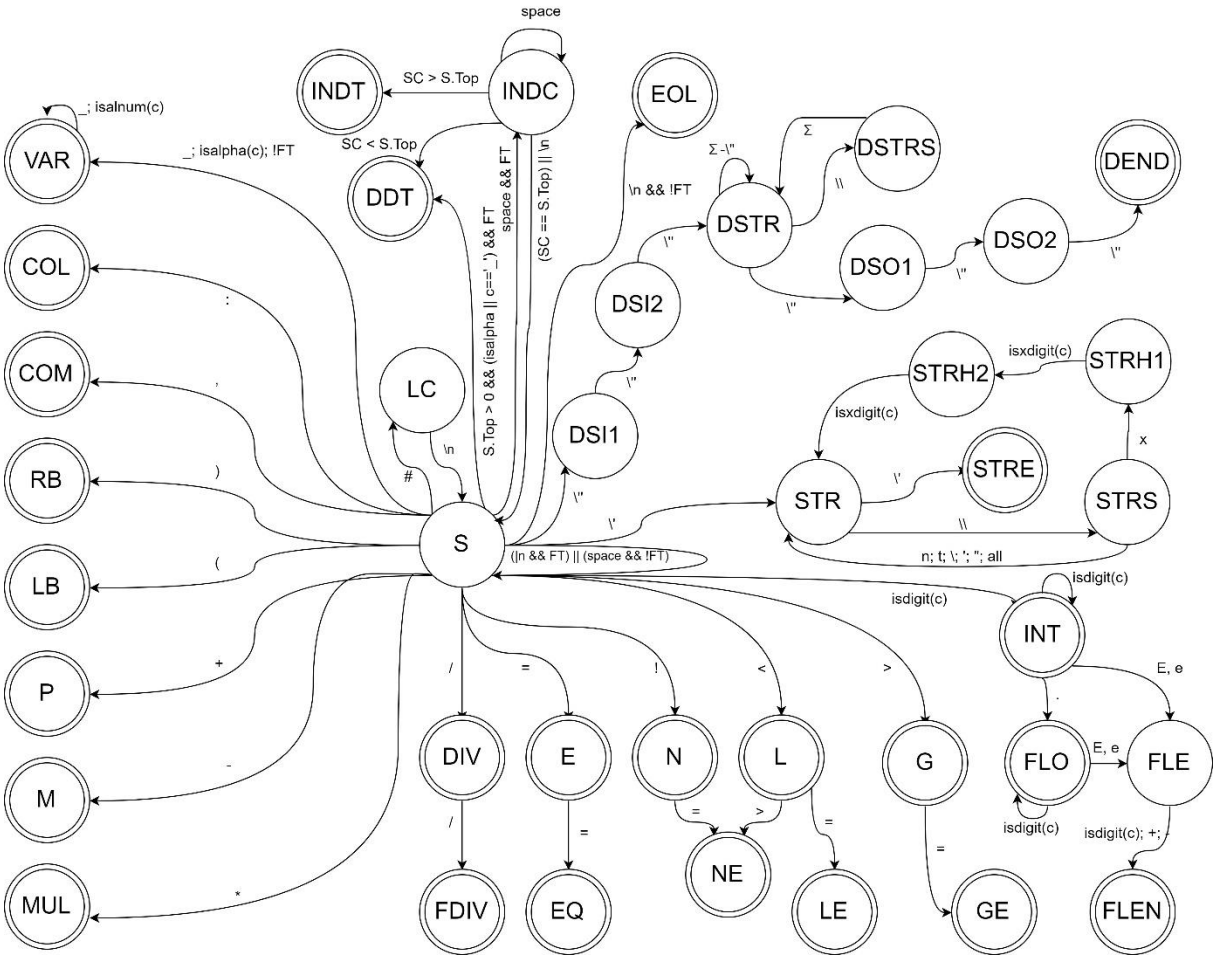
5. Použitá literatura

1) Meduna, A., Lukáš, R.: IFJ, přednášky [online]. Brno, URL:

http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/index.php

6. Diagramy a tabulky

Diagram konečného automatu pro lexikální analýzu (obr. 1)



S	start	G	greater	STRH1	string hexa 1
P	plus	GE	greater or equal	STRH2	string hexa 2
M	minus	VAR	variable	STRS	string special
MUL	multiply	INT	int	STRE	string end
DIV	divide	FLO	float	INDC	indent counter
FDIV	floor division	FLE	float_e	INDT	indent
LB	left bracket	FLEN	float_en	DDT	dedent
RB	right bracket	BC1	block comment in 1	LC	line comment
COM	comma	BC2	block comment in 2	DSI1	documentation string
COL	colon	BC	block comment	DSI2	documentation string
TAB	tab	BCO1	block comment out 1	DSTR	documentation string
E	equals/assignment	BCO2	block comment out 2	DSTRS	documentation string
EQ	equality	LC	line comment	DSO1	documentation string
N	neagtion	FT	first token	DSO2	documentation string
NE	not equal	SC	space count	DEND	documentation string
L	lesser	S.Top	space stack top	EOL	end of line
LE	less or equal	STR	string		

LL gramatika a příslušná LL tabulka

[1] <prog> -> <body>
[2] <body> -> ε
[3] <body> -> <command> EOL <body>
[4] <command> -> ID = <assignment>
[5] <command> -> FUNCTION (<fct_call>
[6] <command> -> <predef_fct>
[7] <command> -> def <fction_start>
[8] <command> -> expression
[9] <command> -> if <if_statement>
[10] <command> -> while <while_statement>
[11] <command> -> ε
[12] <assignment> -> FUNCTION (<fct_call>
[13] <assignment> -> expression
[14] <assignment> -> <predef_fct>
[15] <if_statement> -> expression : EOL INDENT <statement_body> DEDENT <else_statement>
[16] <else_statement> -> else : EOL INDENT <statement_body>
[17] <else_statement> -> ε
[18] <while_statement> -> expression : EOL INDENT <statement_body> DEDENT
[19] <statement_body> -> <command> EOL <statement_body>
[20] <statement_body> -> DEDENT
[21] <fct_call> -> <type> <arg_n>)
[22] <arg_n> -> , <type> <arg_n>
[23] <arg_n> -> ε
[24] <fction_start> -> FUNCTION (ID <arg_n>): EOL <fction_body> INDENT
[25] <fction_body> -> <command> EOL <fction_body>
[26] <fction_body> -> ε
[27] <predef_fct> -> inputs ( )
[28] <predef_fct> -> inputf ( )
[29] <predef_fct> -> inputi ( )
[30] <predef_fct> -> print (<type> <arg_n>)
[31] <predef_fct> -> len (<type>)
[32] <predef_fct> -> substr (<type>, <type>, <type>)
[33] <predef_fct> -> ord (<type>, <type>)
[34] <predef_fct> -> chr (<type>)
[35] <type> -> STR
[36] <type> -> ID
[37] <type> -> INT
[38] <type> -> FLOAT

	ID	FUNCTION	DEF	EXPRESSION	IF	WHILE	ELSE	DEDENT	INPUTS	INPUTF	INPUTI	PRINT	LEN	SUBSTR	ORD	CHR	STR	ID	INT	FLOAT	ε
<prog>																					
<body>																					2
<command>	4	5	7	8	9	10															11
<assignment>		12		13																	
<if_statement>				15																	
<else_statement>							16														17
<while_statement>				18																	
<statement_body>								20													
<fct_call>																					
<arg_n>																					23
<fction_start>		24																			
<fction_body>																					26
<predef_fct>									27	28	29	30	31	32	33	34					
<type>																	35	36	37	38	