DOKUMENTÁCIA K PROJEKTU IFJ/IAL

Implementácia prekladača imperatívneho jazyka IFJ18

Tím 048, varianta II

Radoslav Grenčík (xgrenc00) – vedúci týmu	25%
René Bolf (xbolfr00)	25%
Barbora Nemčeková (xnemce06)	25%
Šimon Šesták (xsesta06)	25%

Obsah

1. Úvod 2

2. Návrh a implementácia 2

2.1. Lexikálna analýza 2

2.2. Syntaktická analýza 2

2.2.1. Syntaktická analýza výrazov 2

2.3. Sémantická analýza 3

2.3.1. Sémantická analýza výrazov 3

2.4. Generovanie kódu 3

3. Špeciálne algoritmy 4

3.1. Tabuľka s rozptýlenými položkami 4

3.2. Zásobník symbolov pre precedenčnú syntaktickú analýzu 4

4. Práca v týme 5

4.1. Spôsob práce 5

4.2. Komunikácia a verzovací systém 5

4.3. Rozdelenie práce 5

5. Záver 5

6. Prílohy 6

6.1. Príloha 1 – deterministický konečný automat 6

6.2. Príloha 2 – LL - Gramatika 7

6.3. Príloha 3 – precedenčná tabuľka a pravidlá syntaktickej analýzy 9

1. Úvod

Tento dokument popisuje návrh a implementáciu prekladača imperatívneho jazyka IFJ18, ktorý je zjednodušenou podmnožinou Ruby 2.0. Cieľom projektu bolo vytvoriť program v jazyku C, ktorý načíta zdrojový súbor v jazyku IFJ18 a preloží ho do cieľového jazyka IFJcode18.

2. Návrh a implementácia

2.1. Lexikálna analýza

Lexikálna analýza bola v našom projekte zostavená ako prvá.
Implementovali sme ju na základe deterministického konečného automatu, ktorý sme si sami zostavili. Pozostáva z konečných stavov.
Jej hlavnou funkciou je funkcia get_next_token(), ktorá postupne načítava znaky zo štandardného vstupu (stdin) a prevádza ich na štruktúru Token. Token sa skladá z typu a atribútu. Typy sme si zadefinovali ako makrá, napríklad typ EOL, EOF, identifikátor, klúčové slovo, typy pre binárne, relačné a aritmetické operátori a mnoho ďalších znakov ktoré môžu byť v jazyku IFJ18 použité. Atribút tokenu je implicitne nastavený na NULL, no keď sa narazí na token typu identifikátor, string alebo číslo, jeho názov alebo hodnota bude uložená do atribútu tokenu. Analyzátor taktiež odstraňuje komentáre a biele znaky.
Pre blokový komentár sme si vytvorili samostatnú funkciu. Bolo to pre nás výhodnejšie, pretože sa musí kontrolovať vždy implicitne po tom, čo nastane koniec riadku a taktiež na začiatku súboru.

2.2. Syntaktická analýza

Syntaktická analýza, alebo aj parser, je srdce prekladača a je založená na prediktívnej analýze s metódou rekurzívneho zostupu. Implementovali sme ju na základe LL-gramatiky, ktorú sme si najprv sami navrhli. Parser sa skladá z viacerých funkcií, ktoré kontrolujú konkrétne pravidlá gramatiky. Spolupracuje s lexikálnym analyzátorom, od ktorého žiada tokeny pomocou funkcie get_next_token(). Taktiež sa tu vykonávaj sémantické kontroly a volajú funkcie na generovanie kódu.

2.2.1. Syntaktická analýza výrazov

Syntaktická analýza výrazov je implementovaná na základe precedenčnej tabuľky v samostatnom súbore prec_table.c a jej rozhranie v súbore prec_table.h. Taktiež využíva zásobník, ktorý je implementovaný v súboroch symstack.c a symstack.h.
Po spustení analýzy sa vkladajú prichádzajúce znaky na zásobník, na základe zisteného znamienka v precedenčnej tabuľke implementovanej v dvojrozmernom poli char prec_table[TSIZE][TSIZE]. V prípade, že znak v tabuľke bol >, zavolá sa funkcia rules() a po zistení pravidla sa skontroluje sémantika výrazu.

2.3. Sémantická analýza

Sémantická analýza sa vykonáva v súbore parser.c na základe príznakov, ktoré sa nastavia rôzne v každej funkcií gramatiky, a podľa ich nastavenia sa vykonajú príslušné kontroly. Za pomoci symtable, v ktorej sú uložené identifikátory, sa kontroluje či sú definované identifikátory, ich typy a podobne.

2.3.1. Sémantická analýza výrazov

Sémantická analýza výrazov je taktiež implementovaná v súbore prec_table.c, konkrétne funkcia expression_sematics(). Táto funkcia je volaná po priradení správneho pravidla a kontroluje správnosť operandov. V prípade, že pri aritmetických operáciách +,-,*,/ a relačných operáciách dôjde ku kolízií typov int a float v jednom výraze, int je pretypovaný na float. V prípade, že jeden alebo oba operandy (okrem operácií +, ==, !=) sú typu string, nastáva chyba 4.

2.4. Generátor kódu

Generátor kódu funguje na základe volaní zo súboru parser.c a prec_table.c, kde podľa úspešne vykonanej funkcie konkrétneho pravidla zavolá potrebnú funkciu z generator.c. Následne sa vytvorí medzikód IFJcode18, ktorý sa skladá zo zásobníkových a trojadresných inštrukcií.

Vytvorili sme si funkciu AppendString(), kde prvý parameter je štruktúra tList, ktorej definícia je v scanner.h, a druhý parameter je string, ktorý sa pridáva do jednotlivých prvkov listu. Do stringu sa priraďuje premenná časť príkazu, vstavané funkcie sa priradia celé. Makro APPEND kontroluje, či funkcia AppendString() prebehla správne a ak nie, vráti príslušný chybový kód.

3. Špeciálne algoritmy

3.1. Tabuľka s rozptýlenými položkami

Implementácia tabuľky symbolov bola časť, ktorá vyplývala zo zadania, ako súčasť predmetu IAL. Preto na jej implementáciu sme využili vedomosti získané v tomto predmete. Hlavnou funkciou je hash_fun(), hashovacia funkcia, okrem nej sme použili napríklad funkciu na inicializáciu, vyhľadávanie položky, pridanie položky, odstránenie položky, uvoľnenie celej tabuľky ale aj naplnenie tabuľky.

Okrem klasickej tabuľky symbolov, do ktorej sa ukladajú identifikátory a údaje o nich, sme si vytvorili ešte jednu tabuľku, v ktorej sú uložené tabuľky jednotlivých funkcií. Pre túto symtable máme vytvorené samostatné funkcie, a to pre inicializáciu, na uvoľnenie celkovej tabuľky a všetkých jej tabuliek, a taktiež funkciu, ktorá vráti symtable konkrétnej funkcie, prípadne, ak neexistuje, vytvorí ju.

3.2. Zásobník symbolov pre precedenčnú syntaktickú analýzu

V súbore symstack.c a symstack.h je implementácia zásobníka symbolov. Tento zásobník sme využili v precedenčnej syntaktickej analýze. Na jeho implementáciu sme taktiež využili vedomosti nadobudnuté v predmete IAL.

Štruktúra zásobníkovej položky sa skladá zo symbolu, ktorý sme získali z tokenu behom precedenčnej analýzy, typu, ktorý sa nastavuje v prípade, že sme dostali z tokenu identifikátor, číslo alebo string, inak je nastavený na NULL a z ukazateľa na ďalšiu položku.

Implementovali sme základné zásobníkové oparácie ako je symstack_init(), symstack_push() a symstack_pop(). Pre účely syntaktickej analýzy sme pridali ďalšie funkcie ako symstack_pop_many(), popnutie viacerých položiek naraz, symstack_top_term(), ktorá vráti vrchol zásobníka, symstack_count(), ktorá nám vráti počet položiek od nájdenej až po vrchol, ale aj symstack_post_insert(), ktoré vloží prvok za nájdený.

4. Práca v tíme

4.1. Spôsob práce

Na projekte sme začali pracovať asi v polovici októbra. Prácu sme si delili priebežne, na základe preberanej látky v predmetoch IFJ a IAL a na základe našich vedomostí. Na jednotlivých častiach sme pracovali jednotlivo, vo dvojiciach, ale aj spoločne.

4.2. Komunikácia a verzovací systém

V priebehu riešenia projektu sme sa viac krát stretli, kedykoľvek člen tímu potreboval poradiť alebo sme potrebovali vyriešiť komplikovanejšie časti projektu. Na organizáciu stretnutí sme používali aplikáciu Doodle. Mimo stretnutí sme sa komunikáciu používali aplikáciu Discord, buď jednotliví členovia medzi sebou alebo na nami vytvorených kanáloch, týkajúcich sa konkrétnych častí projektu.

Na zdieľanie kódu sme používali verzovací systém Git a ako vzdialený repozitár sme používali GitHub.

4.3. Rozdelenie práce

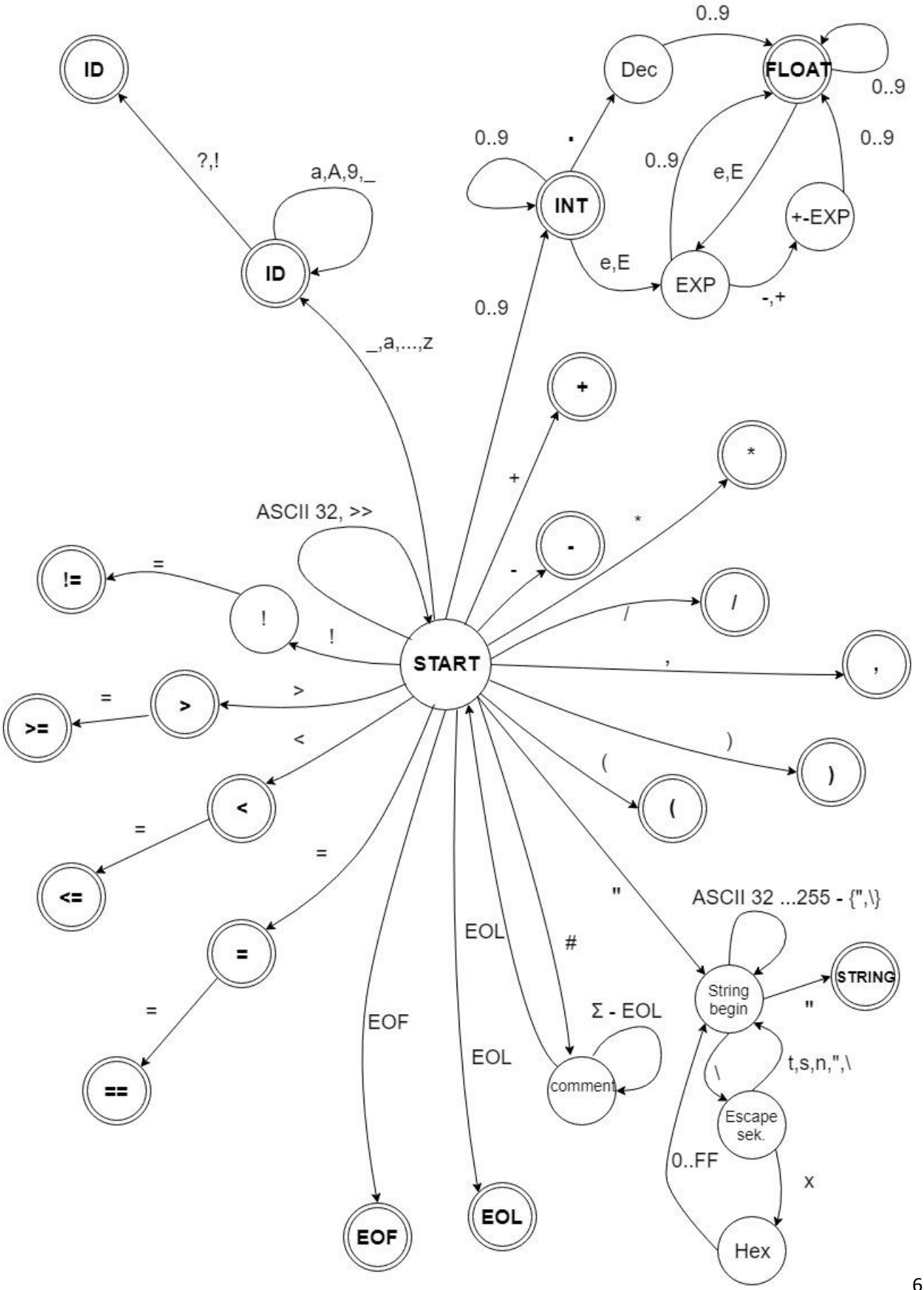
Radoslav Grenčík	Vedenie tímu, Lexikálna analýza, makefile
René Bolf	Syntaktická analýza, sémantická analýza
Barbora Nemčeková	Syntaktická a sémantická analýza výrazov, dokumentácia
Šimon Šesták	Tabuľka symbolov, generovanie kódu

5. Záver

Náš tím precenil svoje sily a čas, ktorý sme na projekt mali, čo sa odrazilo na konečnej funkcionalite prekladača. Po dlhej tímovej práci, náš prekladač nefunguje úplne, chýbajú mu niektoré funkcie.

Napriek tomu sme si z tohto projektu odniesli mnoho nových vedomostí, vyskúšali sme si prácu v menšom tíme a lepšie sme pochopili látku preberanú v predmetoch IAL a IFJ.

Príloha 1 – konečný deterministický automat



Príloha 2 – LL - Gramatika

1) <prog> -> <statement><func><prog>EOF

2) <func> -> DEF ID (<param>) EOL <statement>END EOL
3) <func> -> Ɛ

4) <statement> -> IF <expression> THEN EOL <statement> ELSE EOL <statement> END EOL<statement>
5) <statement> -> WHILE <expression> DO EOL <statement> END EOL<statement>
6) <statement> -> ID <starts_w_id> EOL <statement>
7) <statement> -> <expression> EOL <statement>
8) <statement> -> <builtin_func> EOL <statement>
9) <statement> -> Ɛ

10) <starts_w_id> -> <function_call>
11) <starts_w_id> -> = <inicialization>

12) <inicialization> -> <builtin_func>
13) <inicialization> -> <function>
14) <inicialization> -> <value>
15) <inicialization> -> NIL
16) <inicialization> -> <expression>

17) <builtin_func> -> PRINT <builtin_call>
18) <builtin_func> -> CHR <builtin_call>
19) <builtin_func> -> ORD <builtin_call>
20) <builtin_func> -> SUBSTR <builtin_call>
21) <builtin_func> -> LENGTH <builtin_call>
22) <builtin_func> -> INPUTF <input_call>
23) <builtin_func> -> INPUTS <input_call>
24) <builtin_func> -> INPUTI <input_call>

25) <function> -> ID <function_call> EOL

26) <function_call> -><term_user>
27) <function_call> -> (<term_user>)

28) <builtin_call> -> <term>
29) <builtin_call> -> (<term>)

30) <input_call> -> ()
31) <input_call -> Ɛ

32) <param> -> <value><param_next>
33) <param> -> Ɛ

34) <param_next> -> , <value><param_next>
35) <param_next> -> Ɛ

36) <term_user>-> <term>
37) <term_user> -> Ɛ

38) <term> -> <value> <term_next>
39) <term> -> NIL <term_next>
40) <term> -> ID <term_next>

41) <term_next> -> , <term>
42) <term_next> -> Ɛ

43) <value>->INTEGER
44) <value> -> FLOAT
45) <value> -> STRING

	DEF	ID	IF	WHILE	PRINT	CHR	ORD	SUBSTR	LENGTH	INPUTF	INPUTS	INPUTI	INTEGER	FLOAT	STRING	NIL	,	(	)	=	$
<prog>	1	1	1	1	1	1	1	1	1	1	1	1									1
<func>	2																				3
<statement>		6	4	5	8	8	8	8	8	8	8	8									9
<starts_w_id>		10											10	10	10	10		10		11	10
<inicialization>		13			12	12	12	12	12	12	12	12	14	14	14	15					
<builtin_func>					17	18	19	20	21	22	23	24									
<function>		25																			
<function_call>		26											26	26	26	26		27	26		26
<builtin_call>		28											28	28	28	28		29			
<input_call>																		30			31
<param>													32	32	32				33		
<param_next>																	34		35		
<term_user>		36											36	36	36	36			37		37
<term>		40											38	38	38	39					
<term_next>																	41		42		42
<value>													43	44	45						

Príloha 3 – precedenčná tabuľka a pravidlá pre precedenčnú analýzu

	+	-	*	/	<	<=	>	>=	==	!=	(	)	i	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	X	X	X	X	>	>	<	>	<	>
<=	<	<	<	<	X	X	X	X	>	>	<	>	<	>
>	<	<	<	<	X	X	X	X	>	>	<	>	<	>
>=	<	<	<	<	X	X	X	X	>	>	<	>	<	>
==	<	<	<	<	<	<	<	<	X	X	<	>	<	>
!=	<	<	<	<	<	<	<	<	X	X	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	X
)	>	>	>	>	>	>	>	>	>	>	X	>	X	>
i	>	>	>	>	>	>	>	>	>	>	X	>	X	>
$	<	<	<	<	<	<	<	<	<	<	<	X	<	☺

Pravidlá pre precedenčnú analýzu :

1. EXPRESSION -> EXPRESSION * EXPRESSION
2. EXPRESSION -> EXPRESSION / EXPRESSION
3. EXPRESSION -> EXPRESSION + EXPRESSION
4. EXPRESSION -> EXPRESSION – EXPRESSION

5. EXPRESSION -> EXPRESSION < EXPRESSION
6. EXPRESSION -> EXPRESSION <= EXPRESSION
7. EXPRESSION -> EXPRESSION > EXPRESSION
8. EXPRESSION -> EXPRESSION >= EXPRESSION
9. EXPRESSION -> EXPRESSION == EXPRESSION
10. EXPRESSION -> EXPRESSION != EXPRESSION

11. EXPRESSION -> (EXPRESSION)
12. EXPRESSION -> i

Vysvetlenie pre EXPRESSION -> i i = INTEGER, FLOAT, STRING