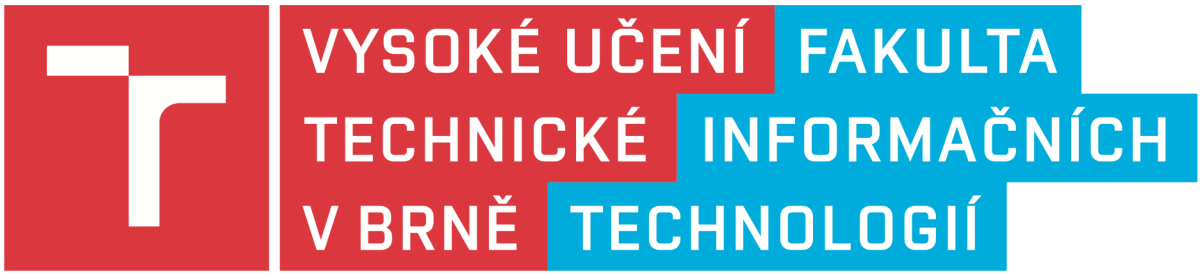


DOKUMENTÁCIA PROJEKTU Z PREDMETOV IFJ A IAL

Implementácia prekladača imperatívneho jazyka IFJ19

Tým 073, varianta I

Vojtěch Ulej	**<xulejv00>**	**25%**
Roman Fulla	<xfulla00>	25%
Pavol Szepsi	<xszeps00>	25%
Daniel Andraško	<xandra05>	25%

29. novembra 2019

Obsah

1 Úvod

Tento projekt je napísaný v jazyku C a jeho cieľom je preložiť zdrojový súbor v jazyku *IFJ19*, ktorý je podmnožinou jazyka Python, do cieľového jazyka *IFJcode19*.

Program vráti 0 ak je zdrojový súbor v poriadku, inak vráti odpovedajúci kód chyby. Zároveň dostane vstupný zdrojový súbor cez štandardný vstup a vráti výsledný súbor na štandardný výstup.

2 Implementácia

Táto kapitola popisuje základné časti programu a ich implementáciu.

2.1 Lexikálna analýza

Ako na jednej z prvých sme začali pracovať na lexikálnej analýze. Lexikálna analýza (angl. scanner) načítava znak po znaku zo zdrojového súboru a zabalí ich do tokenu, podľa prislúchajúceho typu a atribútu.

Analyzátor sa taktiež stará aj o to, aby odstránil a overil nepotrebné lexémy ako sú napríklad komentáre a biele znaky. Načítavanie znakov a následná premena na token je implementovaná pomocou funkcie *gimme_Token*, ktorá vracia už spracované lexémy.

Atribút tokenu môže naberať hodnoty typu *integer*, *double*, *string* alebo *keyword*. Tieto atribúty máme zaznamenané v špeciálnom dátovom type *union*.

Zatiaľ čo typ tokenu máme zaznamenaný vo výčtovom type *enum* a môže naberať hodnoty identifikátorov (*ID, KEY_WORD*), dátových typov (*INT, DOUBLE, STRING, MULTILINE_STR*), symbolov alebo operátorov (*ADD, SUB, LESS, ASSIGN, …*) a nakoniec špeciálnych znakov, ktoré sú : I*NDENT, DEDENT, _EOF, EOL*.

Lexikálna analýza je navrhnutá ako konečný automat (viď príloha 6.1*)* s 26 stavmi.

2.2 Syntaktická analýza

Syntaktická analýza (ďalej len Parser) je spustená volaním funkcie *Analysis*, ktorá inicializuje potrebné dátové štruktúry (*Code_gen, GlobalParserData, GlobalTable*) po úspešnej inicializácií zmienených dátových štruktúr je spustený samotný parser.

Parser je riadený metódou rekurzívneho zostupu, podľa pravidiel uvedených v LL – tabuľke (viď prílohu 6.3). Každé pravidlo *LL – gramatiky* (viď príloha 6.2) je reprezentované jednou funkciou. Pre svoj správny priebeh, parser využíva vyššie zmienené dátové štruktúry *GlobalParserData*, do ktorej ukladá potrebné informácie. Napríklad či sa nachádza vo vnútri

definície funkcie (premenná *defining_function* typu *bool*), premennú *label_gen* slúžiacu ku indexovaniu unikátnych návestí alebo ukazateľ na globálnu tabuľku symbolov (premenná *GlobalTable*).

Pre získanie tokenov (alebo terminálov) analýza volá funkciu *gimme_Token*, ktorá predstavuje lexikálny analyzátor.

2.2.1 Precedenčná syntaktická analýza

Precedenčná syntaktická analýza postupuje zospodu nahor, čím sa líši od tej klasickej. Je implementovaná v súbore *expression_parser.c* a jej rozhranie sa nachádza v súbore *expression_parser.h*. Používa sa na spracovanie výrazov.

Na zásobník sa ako prvý prvok vloží dolár (dno zásobníku) a, ak prejde všetkými testami správnosti, aj prvý token. Potom nasleduje cyklické spracovanie nasledujúcich tokenov, kým nenastane chyba, alebo výraz nie je spracovaný. Tento proces je riadený precedenčnou tabuľkou (viď príloha 6.4).

Pri spracovaní sa buď token vloží na zásobník, alebo sa vrch zásobníka spracuje podľa jedného z predom daných pravidiel. O to sa stará funkcia *process_rule()*. Tá sa zároveň pokúša overiť správnosť typov pre rôzne operácie (keďže *IFJ19* je dynamický jazyk, nie vždy sa to dá overiť už pri preklade).

V našej implementácii PSA očakáva, že jej boli pri zavolaní poskytnuté dva, jeden alebo žiadny token. Ak jej nejaký z nich chýba, požiada si oň sama.

V spracovaní výrazov používame zjednodušenú implementáciu popísanú v demonštračnom cvičení z roku 2008, kde na zásobník nevkladáme zarážky a správne pravidlo zisťujeme podľa toho čo sa už na zásobníku nachádza.

Pri úspešnom spracovaní výrazu, funkcia vráti parseru čím bol výraz ukončený, dvojbodkou alebo koncom riadku/súboru. Ak pri spracovaní nastane nejaká chyba, chybový výstup sa na ňu nastaví a pri kontrole jeho hodnoty sa analýza ukončí a chyba sa vráti syntaktickej analýze, ktorá si analýzu výrazov zavolala, na spracovanie.

2.3 Sémantická analýza

Sémantická analýza prebieha súbežne so syntaktickou analýzou.

2.4 Generovanie kódu

Generovanie kódu prebieha pri syntaktickej analýze. Medzikód je ukladaný do pomocnej dátovej štruktúry *code_gen*, ktorá v sebe uchováva dva dynamicky alokované reťazce. Prvý reťazec uchováva hlavné telo programu (*global_body*), druhý reťazec obsahuje definície užívateľských funkcií (*functions*).

Pre vkladanie inštrukcií nám slúži makro *ADD_INS*, ktoré ako parameter dostane reťazec a ten vloží do štruktúry *code_gen*. Makro *ADD_INS* sa podľa premennej *defining_function* (uloženej

v štruktúre *GlobalParserData*, viď syntaktická analýza) rozhodne, či má byť inštrukcia vložená do reťazca *functions* alebo *global_body*.

Pokiaľ neboli zistené žiadne chyby vo vstupnom programe, tak kód zapísaný v štruktúre *code_gen* sa vypíše na štandardný výstup. Najskôr sa vypíše úvodný riadok *.IFJcode19*, následne skoková inštrukcia na návestí hlavného tela programu a ďalej sú vypísané vstavané funkcie, užívateľské funkcie, návestie pre hlavné telo programu a následne telo programu.

2.5 Preklad projektu

Preklad projektu je možný pomocou nástroja *GNU Make*. Zdrojové súbory projektu obsahujú makefile. Po spustení príkazu make prebehne automatizovaný preklad projektu. Výstupom je binárny súbor *IFJ19Compiler*.

Makefile ďalej obsahuje targety pre zjednodušenie vývoja nášho projektu. Napríklad target *pack*, ktorý zabalí všetky potrebné súbory do archívu odpovedajúcemu formátu pre odovzdanie. Ďalej *is_it_ok*, ktorý spustí target *pack* a následne skontroluje správnosť formátu archívu pomocou skriptu *is_it_ok* (cestu ku skriptu je potrebné nastaviť vo vnútri Makefile). Ďalej targety *test* a *test_parser*, ktoré preložia a následne spustia naše testy (pre tieto targety sú potrebné zdrojové súbory testov, ktoré nie sú súčasťou odovzdávaného archívu).

Predvolený prekladač pre náš projekt je nastavený na *gcc* (premenná CC). Argumenty pre prekladač sú *-std=c11* (pre nastavenie štandardu jazyka C) *-Wall -Wextra* (pre viac podrobný výpis chyb alebo varovaní), kde argumenty sú uložené v premennej *CFLAGS*.

3 Doplňujúce dátové štruktúry

3.1 Dynamický reťazec

Pre prácu s reťazcami premenlivej dĺžky, ktorú vykonáva napríklad lexikálny analyzátor alebo generátor trojadresného kódu, sme potrebovali vhodnú štruktúru. V našej implementácii je to štruktúra *dm_str* (abreviácia anglického „*dynamic string*“), ktorá uchováva okrem samotného ukazovateľa na pole znakov aj dĺžku daného reťazca a veľkosť jemu alokovaného miesta na disku.

Táto štruktúra je spolu s funkciami, ktoré s ňou pracujú popísaná v súbore *dynamic_string.h*. Spomínané funkcie sú implementované v súbore *dynamic_string.c*. Medzi tieto funkcie patrí inicializácia, rozšírenie reťazca o jeden alebo viac znakov (*reťazec*), vrátenie prvého alebo posledného znaku, porovnanie dvoch reťazcov, kópia a samozrejme uvoľnenie reťazca.

Pri inicializácii sa vytvorí prázdny reťazec. Aby pri uvoľnení reťazca neostával jeden neuvoľnený bajt, reťazec sa nevracia do stavu po inicializácii, ale každá funkcia si v prípade NULLového reťazca inicializáciu volá sama.

Pamäť sa alokuje len v prípade potreby a vždy sa alokuje o niečo viac (podľa parametru v hlavičkovom súbore), aby sa procesy o niečo urýchlili.

3.2 Tabuľka symbolov

Podľa zadania máme tabuľku symbolov implementovanú v súbore *symtable.c*, ako binárny strom v ktorej ukladáme potrebné dáta. Súbor *symtable.h* obsahuje užitočné štruktúry pre správne fungovanie binárneho stromu. *BTNode* je základná štruktúra, kde *BTNodePtr* je jej ukazateľ. Obsahuje identifikátor, štruktúru dát (*BTData*) a ukazateľov na ľavý a pravý poduzol.

Po prebratí látky a vypracovaniu 2. domácej úlohy z predmetu IAL sme vedeli ako tabuľku symbolov implementovať. Tabuľku symbolov využívame v syntaktickej analýze, pre ukladanie potrebných dát. Pre správne fungovanie niektorých funkcií (napr. *BTNode_delete_all*), sme potrebovali použiť aj zásobník, ktorý je implementovaný v súbore *stack.c*.

3.3 Zásobník

Ako jeden z pomocných dátových štruktúr je aj zásobník, ktorý sa využíva pri uchovávaní odsadení v precedenčnej syntaktickej analýze, a zároveň pri uvoľňovaní pamäte v tabuľke symbolov.

Zásobník sme implementovali v súbore *stack.c*, pričom súbor *stack.h* obsahuje deklarácie dátových štruktúr a dokumentáciu použitých funkcií.

4 Práca v tíme

Po zadaní projektu netrvalo dlho, kým sme mali prvé stretnutie, kde sme sa dohodli na prerozdelení práce, komunikačných kanáloch a verzovacom systéme.

Počas vypracovávania projektu sme komunikovali pomocou Discordu a Facebooku, ktoré slúžili aj na dohadovanie osobných stretnutí. Na osobných stretnutiach sme riešili problémy, jednotlivé časti projektu a ďalšie kroky k jeho úspešnému vypracovaniu.

Ako verzovací systém sme použili Git. GitHub nám všetkým umožňoval pracovať na projekte zároveň. Na ktorom sme sťahovali, upravovali a nahrávali najnovšie verzie repozitára nášho projektu.

Nasledujúca tabuľka ukazuje členov tímu, vedúceho tímu, hodnotenie členov a každého náplň práce.

Meno	Hodnotenie	Náplň práce
Vojta Ulej	25 %	vedúci týmu, dohľad a kontrola, syntaktická analýza, generátor kódu, správa repozitára, makefile
Roman Fulla	25 %	precedenčná syntaktická analýza, dynamický reťazec, generátor kódu
Pavol Szepsi	25 %	lexikálna analýza, vstavané funkcie, testovanie, prezentácia
Daniel Andraško	25 %	tabuľka symbolov, zásobník, testovanie, dokumentácia

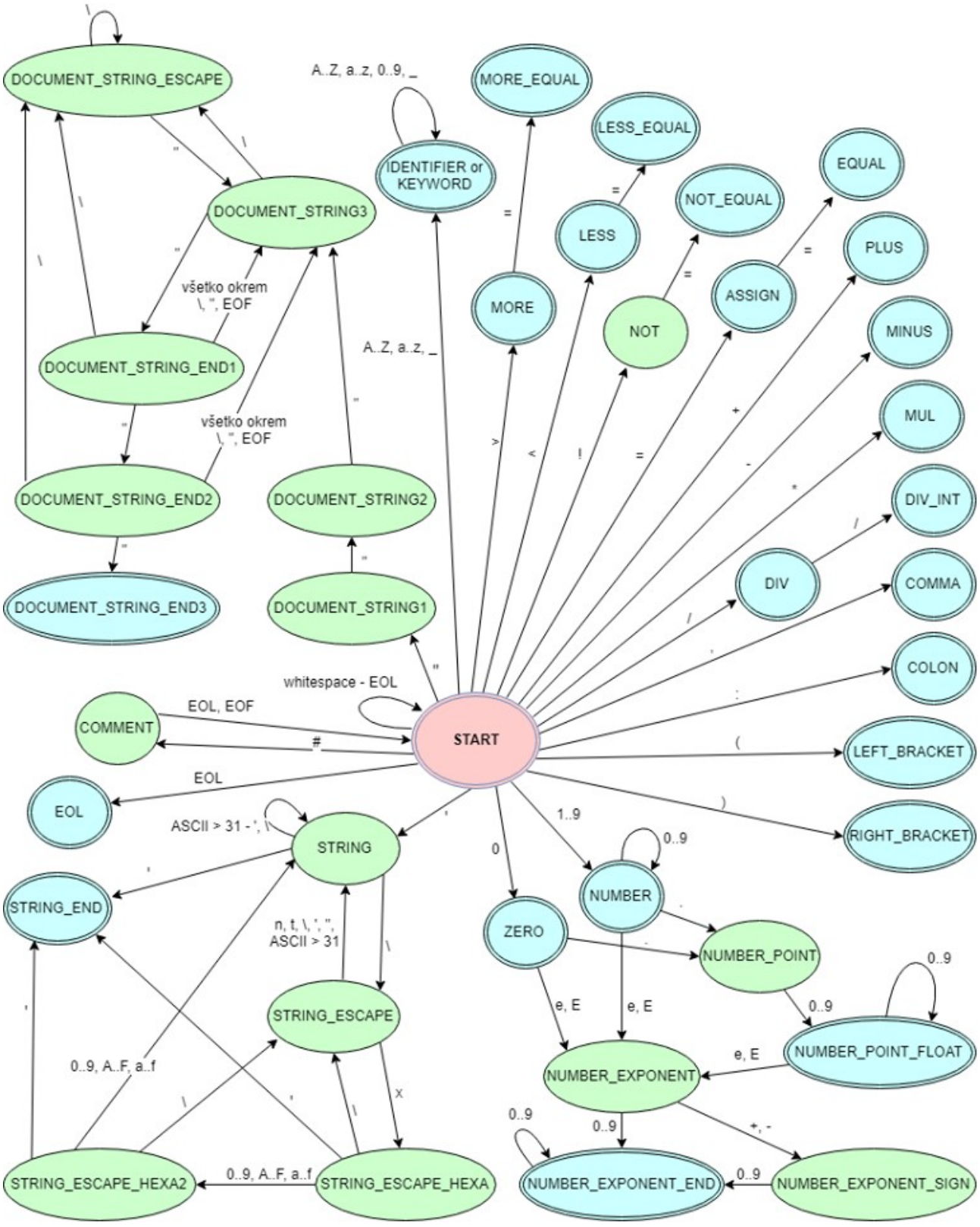
5 Záver

Zadanie a samotné vypracovanie projektu sa nám páčilo. Hoci boli aj ťažšie chvíle kedy sme sa museli potýkať s problémami alebo nejasnosťami, tak na koniec sme sa toho veľa naučili či už ohľadom fungovania samotného prekladača alebo práce a komunikácií na skupinovom projekte.

Z počiatku to bolo náročnejšie, ale vďaka diskusii ku projektu v Kachničce, osobných stretnutiach, a preberaným látkam na hodinách IFJ a IAL sme to mohli zvládnuť.

6 Prílohy

6.1 Konečný automat



6.2 LL - gramatika

1: INPUT → STMTS _eof ,
2: STMTS → ε ,
3: STMTS → STMT STMTS ,
4: STMT → id = ASSIGN ,
5: STMT → if EXPRESION : eol indent S
6: STMT → while EXPRESION : eol inde
7: STMT → FUNC_DEF eol ,
8: STMT → pass eol ,
9: STMT → eol ,
10: STMT → CALL_FUNC eol ,
11: STMT → return RET_VAL eol ,
12: ASSIGN → EXPRESION ,
13: ASSIGN → TERM eol ,
14: ASSIGN → CALL_FUNC eol ,
15: VALUE → int ,
16: VALUE → double ,
17: VALUE → str ,
18: VALUE → multiline_str ,
19: VALUE → none ,
20: TERM → id ,
21: TERM → VALUE ,
22: CALL_FUNC → func_id (ARGS ,
23: CALL_FUNC → inputs ( ) ,
24: CALL_FUNC → inputi ( ) ,
25: CALL_FUNC → inputf ( ) ,
26: CALL_FUNC → len (TERM) ,
27: CALL_FUNC → substr (TERM , TE
28: CALL_FUNC → print (ARGS ,
29: CALL_FUNC → ord (TERM) ,
30: CALL_FUNC → chr (TERM) ,
31: FUNC_DEF → def func_id (ARGS :
32: RET_VAL → TERM ,
33: RET_VAL → EXPRESION ,
34: ARGS → TERM NEXT_ARG ,
35: NEXT_ARG → , TERM NEXT_ARG ,
36: NEXT_ARG →)

6.3 LL - tabuľka

	_eof	id	=	if	:	eol	indent	dedent	else	while	pass	return	int	double	str	multiline_str	none	func_id	(	inputs	)	inputi	inputf	len	substr	,	print	ord	chr	def	$
INPUT	1	1		1		1				1	1	1						1		1		1	1	1	1		1	1	1	1	
STMTS	2	3		3		3		2		3	3	3						3		3		3	3	3	3		3	3	3	3	
STMT		4		5		9				6	8	11						10		10		10	10	10	10		10	10	10	7	
ASSIGN		13											13	13	13	13	13	14		14		14	14	14	14		14	14	14		
EXPRESION																															
FUNC_DEF																													31		
CALL_FUNC																		22		23		24	25	26	27		28	29	30		
RET_VAL		32											32	32	32	32	32														
TERM		20											21	21	21	21	21														
VALUE													15	16	17	18	19														
ARGS		34											34	34	34	34	34														
NEXT_ARG																					36					35					

6.4 Precedenčná tabuľka

Stack \ Scan	+	-	*	/	//	<	<=	>	>=	==	!=	(	)	ID	$
+	>	>	<	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
//	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<=	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
>=	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
==	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
!=	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>		>		>
ID	>	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	