Dokumentácia k projektu IFJ/IAL
Implementácia prekladača jazyka IFJ19

Tím 010, varianta II:

Dominik Boboš (xbobos00) – vedúci tímu
Peter Hudeček (xhudec34)
Timotej Kováčik (xkovac49)
Dávid Oravec (xorave05)

9. Prosince 2019

Obsah

1 Úvod

Cieľom projektu bolo vytvoriť program v jazyku C, ktorý načíta zdrojový kód zapísaný v jazyku IFJ19 a preloží ho do cieľového jazyka IFJcode19. Pokiaľ preklad prebehne bez chýb, program vracia návratovú hodnotu 0. Ak dôjde k chybe, program vracia kód chyby.

2 Rozdelenie

2.1 Dominik Boboš (xbobos00)

Tvorba gramatických pravidiel a LL tabuľky

Tvorba syntaktického analyzátoru

Tvorba generátoru kódu (spoločne)

2.2 Peter Hudeček (xhudec34)

Tvorba precedenčnej tabuľky

Tvorba sémantického analyzátoru (spoločne)

Tvorba generátoru kódu

2.3 Timotej Kováčik (xkovac49)

Tvorba gramatických pravidiel a LL tabuľky

Tvorba sémantického analyzátoru (spoločne)

Tvorba generátoru kódu (spoločne)

Zásobník pre syntaktickú analýzu – pomocný súbor

2.4 Dávid Oravec (xorave05)

Tvorba konečného automatu k lexikálnej analýze

Tvorba lexikálneho analyzátoru

Tvorba sémantického analyzátoru (spoločne)

Tvorba generátoru kódu (spoločne)

Tabuľka symbolov

Zásobník pre scanner – pomocný súbor

Implementácia dynamického reťazca – pomocný súbor

2.5 Spoločná práca

Testovanie kódu

2.6 Zdôvodnenie odchýlky od rovnomerného počtu bodov

Odchýlka od rovnomerného počtu bodov bola spôsobená nerovnomerným rozdelením práce a spoločnou prácou na istých častiach prekladača.

3 Časti

3.1 Lexikálna analýza

Pri tvorbe projektu IFJ19 sme začali najprv s lexikálnou analýzou. Lexikálna analýza bola implementovaná na základe deterministického konečného automatu (viz. Príloha 7.1), ktorý sme vytvorili ako prvý. Úlohou lexikálneho analyzátoru (scanner.c) je rozpoznať jednotlivé lexémy, transformovať ich na tokeny a správne ich posielať ďalej syntaktickej analýze.

Naša štruktúra ***token*** je zložená z atribútu a typu. Typ tokenu je určený na základe spracovania znakov zo zdrojového súboru a konečného automatu. Môže nadobúdať hodnoty dátových typov (int,string,float), znaky matematických či logických operácii, znaky odsadenia (indent,dedent), identifikátory, kľúčové slová, EOF a EOL, komentár a ďalšie znaky, ktoré sú definované jazykom IFJ19. Atribút tokenu, je štruktúra, ktorá v sebe uchováva informácie o danom tokene. Uchováva hodnoty daného tokenu, to znamená, že pokiaľ je token identifikátor alebo string, atribút uchováva jeho hodnotu, ak je číslo, uchováva dané číslo, ak je kľúčové slovo, priradí sa mu dané kľúčové slovo. Inak nie je pridelený žiadny atribút.

Celý lexikálny analyzátor je implementovaný v nekonečnom while cykle. Na začiatku sa zo zdrojového súboru načíta znak a na základe konečného automatu sa dostáva do určitých stavov, kde sa daný znak spracuje a pokračuje sa v cykle s ďalším načítaným znakom, až pokiaľ nie je načítaným znakom EOF.

Pri tvorbe lexikálneho analyzátora sme potrebovali implementovať aj pomocný zásobník pre generovanie odsadenia INDENT a DEDENT. Tento pomocný zásobník sme si implementovali v domácej úlohe v predmete IAL a tak sme ho využili aj pri IFJ19. Princíp je taký, že na vrchole zásobníku je uložená vždy pomocná 0, ktorá odtiaľ nikdy nie je vymazaná a to nám umožňuje efektívne generovať odsadenie. Pri zvyšovaní úrovne odsadenia si na zásobník vkladáme počet medzier (bielych znakov) vtedy, ak je ich počet vyšší ako číslo na vrchole zásobníku a generujeme token INDENT. Naopak pri znižovaní úrovne sme si museli dať pozor na to, či je momentálny počet medzier (bielych znakov) rovný niektorému číslu na zásobníku. Vtedy generujeme token DEDENT, v opačnom prípade je to chyba.

Lexikálny analyzátor je využívaný najmä v syntaktickej analýze, ktorá si ho volá pomocou funkcie *getNextToken*.

3.2 Syntaktická analýza

Syntaktická analýza je implementovaná pomocou metódy rekurzívneho zostupu a LL-gramatiky , ktorú sme si na začiatok navrhli. Implementácia syntaktickej analýzy sa nachádza v súboroch **parser.c/h**. Vďaka LL-gramatike sme potom implementovali jednotlivé pravidlá. Kontroluje sa správna postupnosť tokenov, ktoré si parser pýta z lexikálneho analyzátoru pomocou funkcie *getNextToken*. V prípade správnosti vracia parser návratovú hodnotu 0, v prípade syntaktickej chyby kód 2. V našej implementácií využívame štruktúru **ParserData** (parser.h), ktorá uchováva dôležité dáta ako napríklad momentálne spracovávaný token (*Token*) , globálnu a lokálnu tabuľku symbolov (*globalT, localT*). Aktuálne spracovávané identifikátory (*currentID, leftID, rightID*). Pomocné premenné pre tvorbu labels v generátore (uniqLabel). Pri analýze parametrov vo funkcií využívame *paramIndex* . Na záver pomocné premenné pre správne fungovanie súčastí *(in_declaration, was_return, in_function*). V parser.c odkazujeme aj na ostatné súčasti ako generovanie kódu IFJcode19, spracovanie výrazov *precedenčnou analýzou* v *expr.c/h* .

3.3 Sémantická analýza

Sémantická analýza (najmä v súbore *expr.c,* ale sčasti aj v *parser.c* a *generator.c*) prebieha na základe kontroly operácií dátových typov a ich správnosti zápisu. Analyzátor kontroluje možné kombinácie zápisu dátových typov a ich návratových hodnôt vo výrazoch a rieši ich prípadné pretypovanie. Pri nemožnosti overenia dátového typu staticky, sa správna sémantika analyzuje dynamicky za behu.

Na určovanie priority operátorov je použitá precedenčná tabuľka (príloha 7.4) podľa ktorej sú následne riadené operácie na zásobníku. Operátory s rovnakou prioritou (+ , - alebo / , *) sú kvôli zjednodušeniu implementácie zapísané v jednom riadku/stĺpci. Vrchný terminál je označený riadkami, a stĺpec označuje aktuálne prijatý token. Podľa kombinácie týchto operandov a precedenčnej tabuľky analyzátor určuje operáciu (znak <,>,=) ktorú je potrebné vykonať pre správne analyzovanie výrazu.

Pri znaku = vložíme aktuálne prijatý symbol na vrchol zásobníku a pokračujeme získaním ďalšieho tokenu. Pri znaku < vložíme na vrch zásobníku pomocou funkcie *sym_insert_stop_NT()* znak STOP a pokračujeme získaním ďalšieho tokenu. Pri znaku > redukujeme znaky na zásobníku podľa pravidiel až po najbližší znak STOP, kde sa otestuje správnosť sémantiky výrazov. Tento proces prebieha pokiaľ sa v zásobníku nenachádza žiaden ďalší znak.

3.4 Generátor kódu

Úlohou generátoru je generovať medzikód IFJcode19 po dokončení kontroly správnosti vstupného kódu. Funkcie na generovanie kódu sú uložené v súbore *generator.c*. Vygenerovaný kód a inštrukcie, sú zapisované do dynamického reťazca, ktorý je na záver vypísaný na štandardný výstup.

V našom riešení projektu IFJ19 generátor bohužiaľ nie je kompletný, kvôli problémom v komunikácii a následnom časovom sklze, ktorý sme už nedokázali dobehnúť. Našou základnou myšlienkou bolo na určitých miestach v syntaktickej a sémantickej analýze generovať potrebné inštrukcie IFJcode19 do dynamicky alokovaného reťazca.

Generovanie zabezpečujú rôzne funkcie, ktoré slúžia na generovanie výsledného kódu. Funkcia *generateHead()* generuje hlavičku .*IFJcode19* a pomocné globálne premenné, tá je generovaná ako prvá,

hneď po inicializácii dynamického reťazca pre zápis inštrukcii. Pokračujeme funkciou *generateBuiltIn(),* ktorá vygeneruje vstavané funkcie IFJ19 ešte pred hlavným telom programu. Následne je vo výslednom kóde generované náveštie **MAIN**, ktoré je vygenerované ešte pred samotným začiatkom parsovania programu funkciou generateMain(). Samotné funkcie vstupného kódu (if, while, deklarácia premenných, priradenie hodnoty, deklarácia funkcii atď.) sú volané v potrebný moment počas syntaktickej analýzy alebo sémantickej analýzy. Na záver je generované správne ukončenie hlavného tela programu a výsledok vypísaný na štandardný výstup.

4. Pomocné súbory

4.1 Zásobník pre syntaktickú analýzu

Zásobník obsahuje základné funkcie pre prácu so zásobníkom. Pridali sme funkciu *sym_insert_stop_NT()* ktorá vkladá znak STOP na prvé miesto za TERMINAL. Pridaná bola tak isto funkcia *symbol_top_term()* ktorej úlohou je vrátiť najvrchnejší terminál zo zásobníku.

4.2 Tabuľka symbolov

Tabuľka symbolov je v našom projekte vypracovaná ako tabuľka s rozptýlenými položkami (hash tabuľku). Pri implementácii sme využili znalosti z predmetu IAL, ale taktiež z predmetu IJC, kde bolo úlohou implementovať práve hash tabuľku. Preto sme už vytvorenú hash tabuľku modifikovali na potreby k vypracovaniu projektu.

Tabuľka obsahuje položky, kde každá položka uchováva informácie ako sú *key, Data, next* pričom next je ukazateľ na nasledovníka v tabuľke, Data je štruktúra, ktorá v sebe uchováva informácie, ktoré využívame predovšetkým v syntaktickej analýze a key je kľúč k danej položke.

Veľkosť hash tabuľky musí byť prvočíslo a tak sme si spolu s kolegami vybrali hodnotu **32771**. Táto hodnota bola vybraná preto, aby sme zabezpečili dostatočne veľkú kapacitu k účelom projektu a taktiež pri porovnávaní časovej výkonnosti pomocou príkazu *time* nám práve táto hodnota vyšla ako najlepšia možnosť.

4.3 Dynamický reťazec

Pri implementácii lexikálneho analyzátoru sme si vytvorili aj pomocný súbor *str.c* v ktorom sa nachádza implementácia dynamického reťazca. Keďže dopredu nevieme aký bude reťazec veľký, potrebovali sme si vytvoriť funkcie na dynamické alokovanie pamäti pre reťazec, pridanie znaku do reťazca, pridanie konštantnej sekvencie znakov do reťazca a uvoľnenie pamäti.

5 Práca v tíme

V tíme bola práca na jednotlivých častiach projektu rozdelená dopredu počas stretnutí pred zadaním projektu. Počas práce na projekte sme v prípade problému zahájili stretnutie na ktorom sme problém detailne prediskutovali a poradili sa o možných riešeniach. Na vzdialenú komunikáciu sme používali aplikáciu Discord. Na zdieľanie a verzovanie kódu sme používali repozitár GIT.

5.1 Problémy pri práci v tíme

V tíme sme sa stretli aj s malým množstvom problémov. Miestami zlyhávala komunikácia medzi jednotlivými členmi tímu, ktorú sme ale po osobnom stretnutí a diskusii zlepšili.

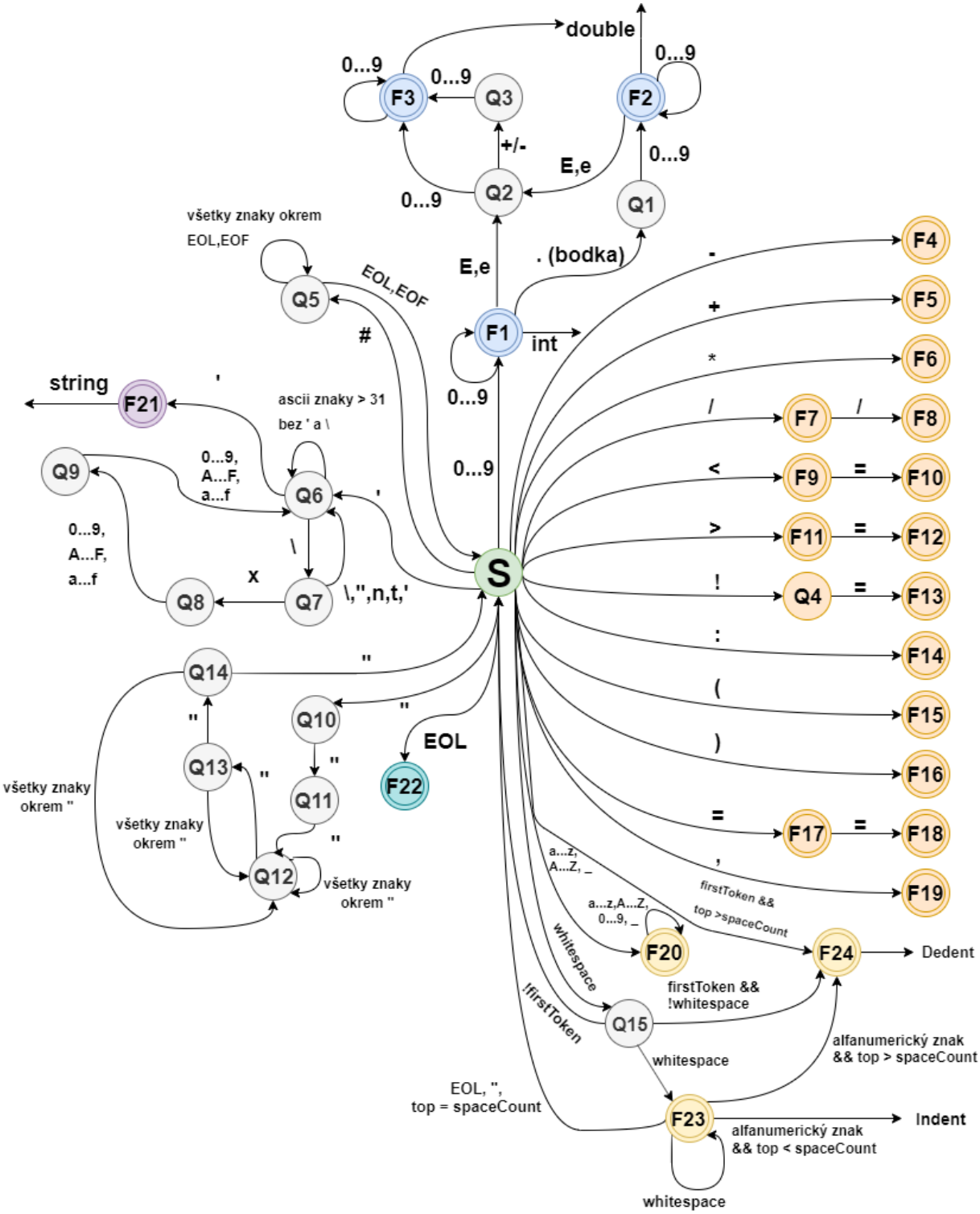
6 Záver

Projekt by sme zhodnotili ako pomerne zložitý, avšak prednášky a demo cvičenia nám značne pomohli s jeho úspešným vypracovaním. Taktiež sme využili vedomosti z ostatných predmetov, hlavne z predmetu IAL. Na predmete sme sa oboznámili s mnohými novými informáciami z oblasti prekladačov a vyskúšali sme si prácu v tíme.

Zhodli sme sa na tom, že projekt bol veľmi zaujímavý a zábavný. Každý z nás si vyskúšal robiť takýto veľký projekt prvý krát a preto nám dal mnoho užitočných skúseností do budúceho štúdia, ale aj profesijnej kariéry.

7 Prílohy

7.1 Model konečného automatu



7.2 LL-gramatika

1. <prog> -> KEYWORD_DEF TYPE_IDENTIFIER(<params>)TYPE_COLON TYPE_EOL TYPE_INDENT <statement> TYPE_DEDENT <prog>
2. <prog> -> TYPE_EOL <prog>
3. <prog> -> <statement> TYPE_EOL <prog>
4. <prog> -> TYPE_EOF <end>
5. <statement> -> KEYWORD_IF <expression> TYPE_COLON TYPE_EOL TYPE_INDENT <statement> TYPE_DEDENT KEYWORD_ELSE TYPE_COLON TYPE_EOL TYPE_INDENT <statement> TYPE_DEDENT <statement_next>
6. <statement> -> KEYWORD_WHILE <expression> TYPE_COLON TYPE_EOL TYPE_INDENT <statement> TYPE_EOL TYPE_DEDENT <statement_next>
7. <statement> -> KEYWORD_RETURN <expression>
8. <statement> -> TYPE_IDENTIFIER TYPE_ASSIGN_VALUE <expression> <statement_next>
9. <statement> -> TYPE_IDENTIFIER(<params>) <statement_next>
10. <statement_next> -> TYPE_EOL <statement>
11. <statement> -> KEYWORD_PRINT(<expression>) <statement_next>
12. <statement> -> KEYWORD_PASS <statement_next>
13. <statement> -> KEYWORD_INPUTS() <statement_next>
14. <statement> -> KEYWORD_INPUTI() <statement_next>
15. <statement> -> KEYWORD_INPUTF() <statement_next>
16. <statement> -> KEYWORD_LEN(TYPE_STRING) <statement_next>
17. <statement> -> KEYWORD_SUBSTR(TYPE_STRING TYPE_COMMA TYPE_INT TYPE_COMMA TYPE_INT) <statement_next>
18. <statement> -> KEYWORD_CHR(TYPE_STRING TYPE_COMMA TYPE_INT) <statement_next>
19. <statement> -> KEYWORD_ORD(TYPE_INT) <statement_next>
20. <params> -> TYPE_IDENTIFIER <param_next>
21. <params> -> ε
22. <param_next> -> TYPE_COMMA TYPE_IDENTIFIER <param_next>
23. <param_next> -> ε
24. <end> -> ε
25. <statement_next> -> ε

7.3 LL-tabuľka

	KEYWORD_DEF	TYPE_IDENTIFIER	TYPE_EOL	TYPE_EOF	TYPE_INDENT	TYPE_DEDENT	KEYWORD_IF	KEYWORD_WHILE	KEYWORD_RETURN	KEYWORD_PRINT	KEYWORD_PASS	KEYWORD_INPUTS	KEYWORD_INPUTI	KEYWORD_INPUTF	KEYWORD_LEN	KEYWORD_SUBSTR	TYPE_COMMA	KEYWORD_ORD	KEYWORD_CHR	$
<prog>	1	3	2	4			3		3	3	3	3	3	3	3	3		3	3	
<statement>		8					5	6	7	11	12	13	14	15	16	17		19	18	
<statement_next>			10			24														25
<params>		20																		21
<param_next>																	22			23
<end>																				24

7.4 Precedenčná tabuľka

	+ -	* / //	Compare	(	)	i	$
+ -	>	<	>	<	>	<	>
* / //	>	>	>	<	>	<	>
Compare	<	<		<	>	<	>
(	<	<	<	<	=	<	
)	>	>	>		<		>
i	>	>	>		>		>
$	<	<	<	<		<	