FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Implementace překladače imperativního jazyka IFJ20

Tým 092, varianta II

Golec Oliver	(xgolec00)	25 %
Hůlek Matěj	(xhulek02)	25 %
Marhefka Adam	**(xmarhe01)**	25 %
Straková Tereza	(xstrak38)	25 %

9. decembra 2020

Obsah

1 Úvod

Cieľom projektu bolo implementovať program v jazyku C, ktorý načíta zdrojový kód zapísaný v zdrojovom jazyku IFJ20 a preloží ho do cieľového jazyka IFJcode20. Náš tím zvolil variantu II.

Táto dokumentácia predstaví algoritmy, metódy a postupy pri riešení projektu do predmetov *Algoritmy* a *Formálne jazyky a prekladače*. Na záver sa pozrieme na prácu v tíme.

2 Implementácia

Projekt sa skladá zo štyroch hlavných celkov, ktorých opis je podaný v jednotlivých podkapitolách tejto sekcie.

2.1 Lexikálna analýza

Prvou časťou prekladača je lexikálny analyzátor, implementovaný v module `scanner.c`. Pomocou funkcie `get_token()` číta zo vstupu znaky, ktoré následne spracuváva pomocou diagramu 1. Ak je postupnosť znakov vyhodnotená ako validný token, je mu v štruktúre `Token` priradený typ a atribút. Typy sú uložené pomocou enumeračného typu `Token_type` a atribúty pomocou štruktúry `Token_attribute`. Typom tokenu môžu byť identifikátor, kľučové slovo, logické alebo aritmetické operátory, čiarka, bodkočiarka, operátor priradenia, operátor definície, reťazec, celé alebo desatinné číslo, EOL a EOF. To, či ide o kľúčové slovo alebo identifikátor, sa vyhodnotí, keď automat po načítaní lexému skončí v stave `S29`
`STATE_IDENTIFIER_OR_KEYWORD` (viď obr. 1). Pomocná funkcia `proc_id()` porovná načítaný lexém s poľom reťazcov, ktoré reprezentujú kľúčové slová jazyka *IFJ20*. Keď je token vyhodnotený ako identifikátor, alebo reťazec, jeho obsah je spracovaný pomocou modulu `string.c`.

Modul `string.c` obsahuje špeciálnu štruktúru na prácu s reťazcami, ktorá okrem užitočných dát obsahuje aj informácie o dĺžke reťazca a veľkosti využitej pamäte. Obsahuje špeciálne funkcie pre prácu s reťazcami prisôsobené pre prácu s touto špeciálnou štruktúrou.

Keď pri čítaní lexému KA skončí v stave, ktorý nie je koncový, vráti odpovedajúci chybový kód. Následne parser uvoľní zdroje a vráti chybový kód `ERR_SCANNER`.

Tokeny sú po spracovaní predané parsru.

2.2 Syntaktická analýza a sémantická analýza

Počas celého behu parsra sa generuje abstraktný syntaktický strom. Jedna položka stromu je reprezentovaná strukturou `Body_item`. `Body_item` obsahuje odkaz na svojich potomkov. Pri generovaní sa v parsru využíva špeciálny zásobník.

Syntaktická analýza je realizovaná pomocou konečného automatu, v ktorom sa opakovane volá funkcia modulu `scanner.c` `get_token()`. Kontrola návratovej hodnoty tejto funkcie a aj ostatných funkcií modulu `parser.c` je realizovaná pomocou sady makier, ktoré okrem tejto kontroly v niektorých prípadoch zabezpečujú aj správne uvoľňovanie priradenej pamäte.

Vo funkcii `Program()` sa inicializuje globálna tabuľka funkcií. Na začiatku skontroluje prítomnosť prologu
(`package main EOL`) Potom je v cykle volaná funkcia `Func()`, ktorá kontroluje syntax definicií užívateľských funkcií a zároveň `func main()`. Pri narazení na `EOF` cyklus končí.

Funkcia `Func()` pri kontrolovaní správnej syntaxe funkcií používa pomocné funkcie na syntaktickú kontrolu parametrov a návratových hodnôt. Všetky informácie o funkciách sú ukladané do globálnej hashovacej tabuľky funkcií, ktorá je implementovaná v module `symtable.c`. Po kontrole parametrov a návratových hodnôt volá funkciu `Body()`, ktorá kontroluje telo funkcie. Tá funguje rekurzívne. Funkcia `Body()` je volaná s ukazateľom na tabuľku premenných, ktorá sa nanovo skopíruje pri každom volaní funkcie `Body()` kvôli zanoreniu, aby lokálne telá cyklov a podmienok mali vždy osobitnú tabuľku a nemenili premenné v pôvodnej.

Odkaz na predchádzajúcu tabuľku symbolov je uložený do zásobníku pre tabuľky symbolov, aby bolo na záver prekladu možné uvoľniť pamäť, ktorá im bola priradená.

Pri volaní funkcie, ktorá doposiaľ nebola definovaná (a teda nie je ani v tabuľke funkcií), je volaná funkcia preemtívne vložená do tabuľky s príznakom `DEFINED`, ktorý je typu `bool` nastaveným na hodnotu `FALSE`. Na záver prekladu sa volá funkcia `symtable_definde()`, ktorá skontroluje, či sú všetky funkcie v tabuľke definované.

Keď očakávame výraz, funkcia `Body()` volá funkciu `Expression()`. Tá potom kontroluje sytnaktickú správnosť výrazu a ukladá tokeny do zásobníku tokenov a zároveň kontroluje aj sémantickú správnosť pomocou výstupu funkcie modulu `psa.c` `prec_parse()`, ktorá používa metódu precedenčnej syntaktickej analýzy.

2.2.1 Modul psa.c

Modul `psa.c` preberá zásobník tokenov z funkcie `Expression()`. Funkcia `prec_parse()` kontroluje asociativitu a prioritu operátorov a zároveň vytvára z výrazu syntaktický strom, na ktorý vracia odkaz. Funkcia `prec_parse()` na začiatku inicializuje zásobník, do ktorého na základe pravidiel z precedenčnej tabuľky ukladá tokeny zo statického poľa ukazateľov na tokeny. Vrchol zásobníka sa porovná indexom poľa a vyhodnotí nasledujúcu akciu podľa funkcie `return_index()`. V prípade znaku > vytvorí list stromu, alebo strom. V prípade znaku < uloží prvok poľa na zásobník a inkrementuje index poľa. Ak funkcia vráti znak =, pravú zátvorku preskočíme a ľavú odstránime zo zásobníka. Iné prípady sú označené ako chybové stavy. Modul `psa.c` tiež zabezpečuje kontrolu dátových typov.

Na začiatku parsingu sú vstavané funkcie vložené do globálnej tabuľky s ich očakávanými parametrami a návratovými hodnotami. Precedenčný parser pomocou zásobníku tokenov skontroluje, či sú správne dátové typy a syntax výrazu. Tiež vytvára abstraktný syntaktický strom a to pomocou pravidiel z precedenčnej tauľky, viď tabuľka 5. Parser rieši vačšinu sémantickej analýzy.

2.3 Generovanie cieľového kódu

Generovanie kódu zaisťuje modul `interpret.c`, ktorý využíva podporné funkcie z modulu `code_gen.c`. Rekurzívnym vykonávaním vytvára cieľový kód na základe AST. Pre jednoduchší chod využíva zásobník. Nevykonávajú sa už žiadne optimalizácie. Pred generovaním jednotlivých funkcií sa skontroluje strom danej funkcie, kde interpret kontrolueje veci, ktoré sa nedajú skontrolovat v parsru, napríklad delenie nulou, korektný počet argumentov pri volaní funkcie a priradenie pri volaní funkcie. Pred začiatkom generovania jednotlivých funkcií sa inicializujú potrebné prostriedky a vygeneruje sa základná štruktúra cieľového kódu a vstavané funkcie, ktoré sú predpísané. Funkcie sa generujú v náhodnom poradí, pričom funkcia `main()` je vygenerovaná ako posledná. Interpret musí zaisťovať prevod určitých ASCII na korektné escape sekvencie podporované jazykom IFJ20. Výsledný kód je navrhnutý tak, aby nadradená funkcia volala funkciu `main()` a po jej skonění skočila na koniec programu. Výsledný kód je ukladaný v štruktúre `String` a až potom čo je vytvorená aj funkcia `main()` a nedošlo k žiadnej chybe ani nebola odhalená chyba v strome je vygenerovaný kód vypísaný na štandardný výstup a uvoľňuje všetky priradené zdroje.

3 Práce v týme

Vzhľadom na nepriaznivú situáciu sa konzultácie nemohli uskutočňovať osobne. Komunikácia prebiehala prevažne na platforme *Discord* a menej na *Messenger* od *Facebook*. Diskutovanie o projekte prebiehalo buď medzi jednotlivcami alebo ako skupinový hovor. Veľmi často bola využívaná funkcia zdieľania obrazovky, ktorá takmer nahradila osobné stretnutia.

4 Rozdelenie práce

Adam Marhefka	syntaktická analýza, sémantická analýza, testovanie
Oliver Golec	precedenčná analýza, tabuľka symbolov, generovanie AST pre výrazy, parser, testovanie
Matěj Hůlek	lexikálna analýza, generovanie kódu, vedenie súčastí tímu, generovanie AST, sémantická analýza, testovanie
Tereza Straková	dokumentácia, testovanie

Tabuľka 1: Rozdelenie práce v tíme

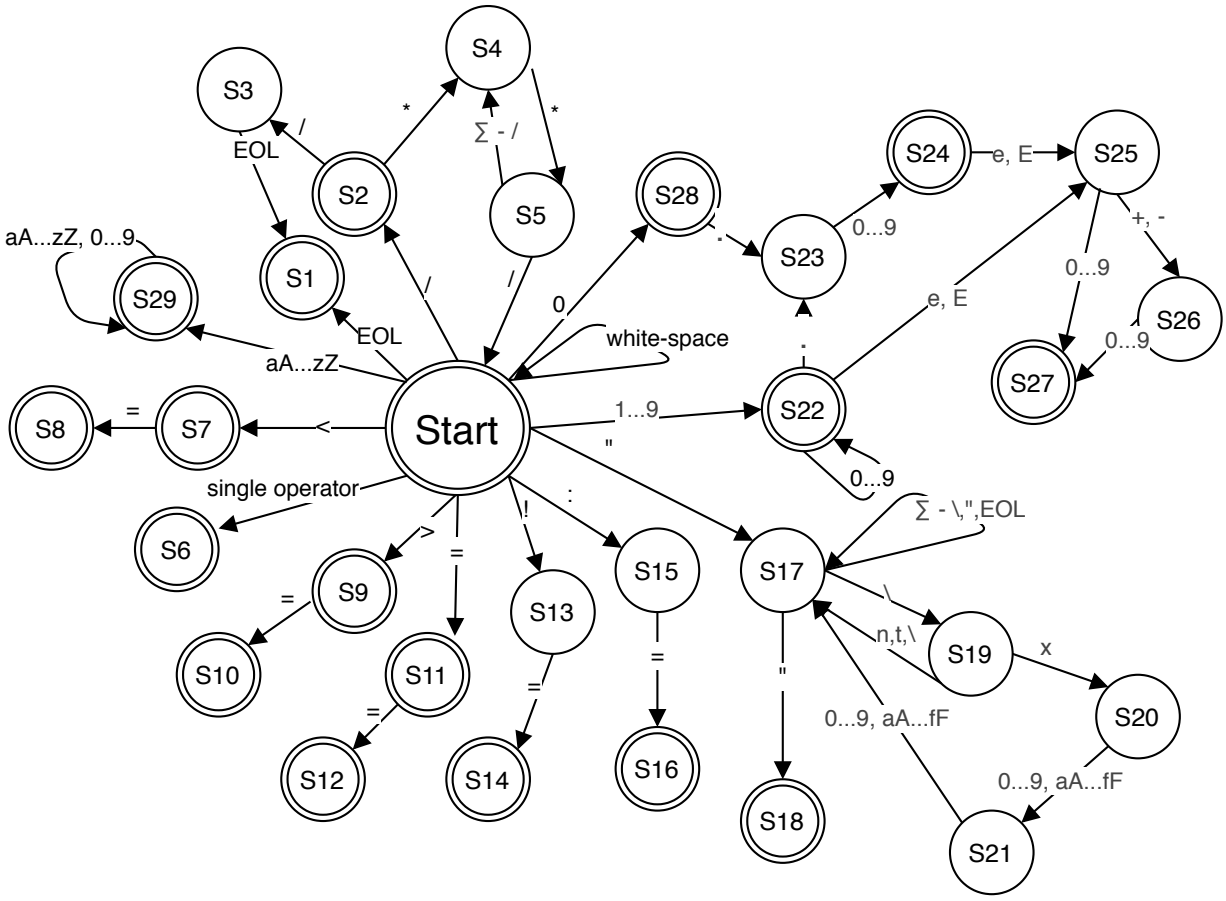
5 Záver

Projekt bol pre nás výzvou. Na začiatku nás projekt zaskočil svojim rozsahom. Chvíľu trvalo, kým sme sa dostali ku tomuto projektu aj kvôli iným zadaniam z ostatných predmetov. Prácu sme začali až začiatkom novembra. Postupne sme si naštudovali všteky komponenty prekladača. Informácie sme čerpali najmä z prednášok. Projekt sa ukázal ako jeden z najťažších v našom doterajšom štúdiu. K najzávažnejším komplikáciám patrilo pochopenie všetkých aspektov zadania, návrh parseru a vyriešenie problematiky blokov platnosti jazyka IFJ20. Veľkým prínosom boli znalosti a schopnosti získané v predmetoch *Algoritmy* (IAL) a *Jazyk C* (IJC). Aj napriek všetkým komplikáciám sme projekt dokončili pár dní pred odovzdaním. Odniesli sme si veľa skúseností do nášho budúceho profesiónálneho života a prehĺbili sme naše skúsenosti a vedomosti v jazyku C.

Literatúra

http://www.cse.yorku.ca/ oz/hash.html

A Diagram konečného automatu



Obr. 1: Diagram konečného automatu

```
S1 STATE_EOL
S2 STATE_SLASH
S3 STATE_COMMENT
S4 STATE_COMMENT_BLOCK_START
S5 STATE_COMMENT_BLOCK_END
S6 is_single_operator() +-*/() ,;
S7 STATE_LESS <
S8 STATE_LESS_OR_EQ
S9 STATE_MORE <
S10 STATE_MORE_OR_EQ
S11 STATE_ASSIGN
S12 STATE_EQL
S13 STATE_NOT_EQ
S14 STATE_NOT_EQ_END
S15 STATE_INIT
S16 STATE_INIT_END
S17 STATE_STRING_START
S18 STATE_STRING_START_END
S19 STATE_STRING_ESCAPE
S20 STATE_STRING_ESCAPE_HEX
S20 STATE_STRING_ESCAPE_HEX_END
S22 STATE_NUMBER
S23 STATE_NUMBER_POINT
S24 STATE_NUMBER_FLOAT
S25 STATE_NUMBER_EXPONENT
S24 STATE_NUMBER_EXPONENT_SIGN
S27 STATE_NUMBER_EXPONENT_FINAL
S28 STATE_NUMBER_ZERO
S29 STATE_IDENTIFIER_OR_KEYWORD
```

B LL – gramatika

1. <program> -> <opt_eol> package main EOL <Func> EOF
2. <opt_eol> -> ε
3. <opt_eol> -> EOL <opt_eol>
4. <Func> -> ε
5. <Func> -> func ID (<params>) <ReturnVals> { EOL <body> } <Func>
6. <params> -> ε
7. <params> -> ID <Ptype> <params_n>
8. <params_n> -> ε
9. <params_n> -> , ID <Ptype> <params_n>
10. <ReturnVals> -> ε
11. <ReturnVals> -> (<ReturnTypes>)
12. <ReturnTypes> -> ε
13. <ReturnTypes> -> <Type> <ReturnTypes_n>
14. <ReturnTypes_n> -> ε
15. <ReturnTypes_n> -> , <Type> <ReturnTypes_n>
16. <Ptype> -> int
17. <Ptype> -> float64
18. <Ptype> -> string
19. <Type> -> int
20. <Type> -> float64
21. <Type> -> string
22. <body> -> ε
23. <body> -> ID <ass/def/func> EOL <body>
24. <body> -> if <expr> { EOL <body> } else { EOL <body>
25. <body> -> for <def> ; <expr> ; <ass> { EOL <body> } EOL <body>
26. <body> -> return <expr> <expr_n> EOL
27. <ass/def/func> -> := <expr>
28. <ass/def/func> -> <id_n> = <expr/func>
29. <ass/def/func> -> (<params_call>)
30. <def> -> ε
31. <def> -> ID := <expr>
32. <ass> -> ε
33. <ass> -> ID <id_n> = <expr/func>
34. <expr/func> -> <expr> <expr_n>
35. <expr/func> -> "ID(" <params_call>)
36. <expr> -> <value> <op>
37. <expr> -> (<opt_eol> <expr>) <op>
38. <op> -> ε
39. <op> -> <operators> <opt_eol> <expr>
40. <value> -> ID
41. <value> -> intVal
42. <value> -> float64Val
43. <value> -> stringVal

Tabuľka 2: LL – gramatika

44. <expr_n> -> ε
45. <expr_n> -> , <expr> <expr_n>
46. <id_n> -> ε
47. <id_n> -> , ID <id_n>
48. <params_call> -> ε
49. <params_call> -> <value> <params_n_call>
50. <params_n_call> -> ε
51. <params_n_call> -> , <value> <params_n_call>
52. <operators> -> +
53. <operators> -> -
54. <operators> -> *
55. <operators> -> /
56. <operators> -> <
57. <operators> -> <=
58. <operators> -> >
59. <operators> -> >=
60. <operators> -> ==
61. <operators> -> !=

Tabuľka 3: LL – gramatika

C LL – tabuľka

	package	main	func	(	)	{	}	;	,	ID(	ID	int	float64	string	if	else	for	return	:=	=	intVal	float64Val	stringVal	+	-	*	/	<	<=	>	>=	==	!=	eol	$
<program>	1																																	1	
<opt_eol>	2			2							2										2	2	2											3	
<Func>			5																																4
<params>					6						7																								
<params_n>					8				9																										
<ReturnVals>				11		10																													
<ReturnTypes>					12							13	13	13																					
<ReturnTypes_n>					14				15																										
<Ptype>												16	17	18																					
<Type>												19	20	21																					
<body>							22				23				24		25	26																	
<ass/def/func>				29					28										27	28															
<def>								30			31																								
<ass>						32					33																								
<expr/func>				34						35	34										34	34	34												
<expr>				37							36										36	36	36												
<op>					38	38		38	38															39	39	39	39	39	39	39	39	39	38	
<value>											40										41	42	43												
<expr_n>						44			45																									44	
<id_n>									47											46															
<params_call>					48						49										49	49	49												
<params_n_call>					50				51																										
<operators>																								52	53	54	55	56	57	58	59	60	61		

Tabuľka 4: LL – tabuľka použitá pri syntaktickej analýze

D Precedenčná tabuľka

	ID	+-	*/	(	)	L_OP	$
ID		>	>		>	>	>
+-	<	>	<	<	>	>	>
*/	<	>	>	<	>	>	>
(	<	<	<	<	=	<	
)		>	>		>	>	>
L_OP	<	<	<	<	>		>
$	<	<	<	<		<	

Tabuľka 5: Precedenčná tabuľka