FAKULTA INFORMAČNÝCH TECHNOLÓGIÍ
VYSOKE UČENÍ TECHNICKÉ V BRNĚ

Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ19

Tím 033, varianta I

Maroš Geffert	<**xgeffe00**>	25 %
Patrik Tomov	<xtomov02>	25 %
Martin Valach	<xvalac12>	25 %
Andrej Pavlovič	<xpavlo14>	25 %

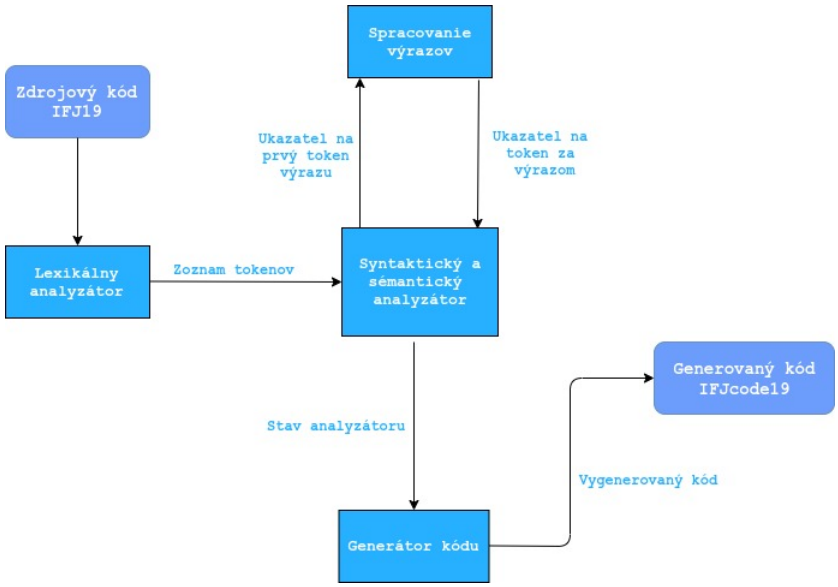
11. prosince 2019

Obsah

1 Úvod

Cieľom nášho projektu bolo vytvoriť prekladač v jazyku C, kotrý načíta zdrojový súbor zapísaný v zdrojovom jazyku IFJ19, ktorý reprezentuje podmnožinu jazyka Python 3, a preloží ho do cieľového jazyka IFJcode19. V prípade chyby vracia odpovedajúci chybový kód.

2 Implementácia



2.1 Lexikálny analyzátor

Pri programovaní prekladača, sme začínali s tvorbou lexikálneho analyzátoru. Základnou funkciou tejto časti je funkcia `get_token`, vďaka ktorej je možne načítávať lexémy[1] zo zdrojového súboru a prevádzajú sa do podoby takzvanej token. Token je štruktúra ktorá obsahuje (atribúty, typy, v prípade identifikátoru si uchováva aj jednotlivý string atď.). Typy tokenov môžu byť klúčove slová, identifikator, rôzne operátory, EOL, EOF a všetko čo jazyk IFJ19 podporuje. Jednotlivé atribúty sa k tokenom priradzujú podľa situácie (napr. pre identifikátor alebo čísla (float, integer).

Lexikálny analyzátor funguje ako deterministický konečný automat. Tento konečný automat sme implementovali ako opakujúci sa switch, kde každý case odpovedá nejakému stavu z automatu. Ak sa načítaný lexem nezhoduje so žiadným odpovedajúcim stavom určeným jazykom IFJ19 tak vraciame chybu 1. Lexikálna analýza končí načítaním posledného znaku zo zdrojového súboru, ktorým je `EOF` (Prípadne skončí v chybovom stave pri internej chybe).

[1] Lexéma je základná jednotka lexikálnej (čiže slovnej) zásoby jazyka, t.j. základná jazyková jednotka, ktorá je nositeľom vecného významu.

2.2 Syntaktická a sémantická analýza

Najpodstatnejšou časťou prekladača je syntaktická a sémantická analýza tzv. Parser. Parser je založený na rekurzívnom zostupe z hora dolu. Jeho úlohou je kontrola, či reťazec tokenov reprezentuje syntaktický správne napísaný program v jazyku IFJ19, pričom zároveň prebieha aj sémantika programu, kde sa kontroluje či je daná premenná definovaná, správny počet argmentov pri volaní funkcie, kompatibilita dátových typov a pod.

V rekurzívnom zostupe si žiadame postupne tokeny od scanneru pomocou funkcie `get_token` a na základe LL pravidiel postupujeme. Každé pravidlo má svoju vlastnú funkciu. Každá takáto funkcia má parameter ukazateľ na `Parser_data`, čo je štruktúra, ktorá obsahuje potrebné premenné na vykonanie syntaktickej a sémantickej analýzy. V prípade, že dojde k chybe, program končí s návratovou hodnotou chyby, ktorá sa vypíše. Táto časť taktiež úzko súvisí s tabuľkou symbolov, ktorá slúži na ukladanie názvov premenných a funkcií(sémantika).

Spracovanie výrazov sa taktiež vykonáva v parsri. Spracovanie spočíva v skontrolovaní priorít výrazov (pomocou syntaktickej analýzy PSA popísanej nižšie) a skontrolovaní premenných, u ktorých sme schopný skontrolovať dátové typy pri preklade, resp. sme ich schopný pretypovať (jediné povolené pretypovanie je z integeru na float) V prípade zlej kombinácie dátových typov vraciame chybu 4. Výrazy, ktoré niesme schopný zkontrolovať za prekladu (napr. premenné, ktoré môžu byť definované v hlavnom tele; parametre funkcií; atď.) kontroluje generátor kódu.

2.3 Generovanie kódu

Generovanie výstupného kódu IFJcode19 na štandardný výstup prebieha až po úspešnom ukončení prekladača. Dovtedy sa kód ukladá na jeho internú pamäť. Na generovanie kódu sa využívajú prevažne makrá. Funkcie, kotré generujú kód sú samostatne oddelené v `code_generator.c`. Pri štarte prekladača sa generuje hlavička ktorá obsahuje vstavané funkcie jazyka a preddefinované pomocné globálne premenné využívané pri rôznych operáciách. Funkcie na generovanie kódu sú rozdelené na jednoduché operácie (priradenie, hlavičky a pätičky pre podmienené príkazy, atď.), ktoré sú volané syntaktickou analýzou podľa potreby. Priamo do generovaného kódu väčšina funkcií hneď na začiatku generuje aj jednoriadkový komentár ktorý stručne popisuje čo táto funkcia vykonáva.

2.4 Interpret

Interpret vytvára interpretáciu trojadresného kódu, generovaného syntaktickým analyzátorom. Trojadresný kód je uloženy v instrukčnom liste, ktorý je implementovaný pomocou jednosmerného viazaného lineárneho zoznamu. Pre každú inštrukciu je tento kód reprezentovaný typom inštrukcie, adresou výsledku a adresami prvého a druhého operandu. Pri spracovávaní aritmeticko-relačných operácií robí interpret sémantickú kontrolu.

3 Algoritmy

3.1 Dynamický string

Prvý algoritmus, ktorý sme použili je `Lexem_string` pre operácie s reťazmi rôznej dĺžky. Štruktura má v sebe uložene ukazatel na reťazec, dĺžku reťazca a aj koľko pamäte je vyhradené pre reťazec. Implementované operácie : inicializácia (Pri inicializácii sa vyhradí pamäť práve pre určitý počet znakov a ak je potrebné tak sa realokuje), uvoľnenie dát, pridanie znaku alebo stringu do reťazca atd.

3.2 Precedenčná tabuľka

Na vyhodnocovanie priorít vykonávaných operácií pri výrazoch sme implementovali precedenčnú syntaktickú analýzu (PSA). PSA vo svojom algoritme využíva precedenčnú tabuľku, ktorú sme implementovali ako dvojrozmerné pole, v ktorej sú zapísané všetky možné operácie ktoré môže nastať a sú v nej zároveň porovnané. Pre porovnávanie využívame zásobník, na ktorom je vždy terminal (operátor, ktorý bol spracovaný), ktorý porovnávame so znakom, ktorý sme dostali zo vstupu (pomocou funckie `get_token` a následného spracovania pre index tabuľky, pomocou funkcií `get_symbol` a `get_index)` a v podľa výsledku porovania vykoná pravidlo PSA. PSA pokračuje pokým nie je celý výraz zkontrolovaný, prípadne pokým v ňom nenájde chybu.

3.3 Tabuľka symbolov

Tabuľku symbolov sme implementovali ako tabuľku s rozptýlenými položkami(hash function). Keďže rozlišujeme, či je premenná globálna alebo lokálna tak rozlišujeme aj globálnu a lokálnu tabuľku. V tabuľke symbolov sa nachádzaju položky, ktoré reprezentuje štruktúra `Sym_table_item`, ktorá obsahuje: char* key(názov), ktorý slúži taktiež na pridelenie indexu v tabuľke na základe hashovacej funkcie. IData data je štruktúra, ktorá obsahuje informácie o položke. Medzi informácie ktoré obsahuje patrí či bola premenna definovaná, či je globálna alebo lokálna a štruktúru `Sym_table_item *next`, čo je ukazateľ na ďalší item.

4 Práca v tíme

Náš tím sa stretol po zadaní projektu. Vedúcim tímu boli hneď aj rozdelené úlohy. Tím väčšinou medzi sebou komunikoval osobne alebo cez discord.

Najprv sme si stanovili štruktúru celého projektu. V priebehu prvých šiestich týždňov sa toho veľmi neurobilo, implementoval sa lexikálny analyzátor, členovia študovali a zbierali informácie ohľadom svojej časti programu, ktorá im bola zadaná vedúcim tímu. V priebehu ďalších týždňov sme implementovali celý program. Testovali sme jednotlivé časti osobitne, ale neskôr sme to už testovali spolu ako celok. K zdieľaniu kódu sa používal repozitár git.

Meno	**Login**	**Rozdelenie práce**
Maroš Geffert	xgeffe00	Lexikálna analýza, dokumentácia, testy, testovací skript
Patrik Tomov	xtomov02	Syntaktická analýza, testy
Martin Valach	xvalac12	Sémantická analýza
Andrej Pavlovič	xpavlo14	Generátor kódu

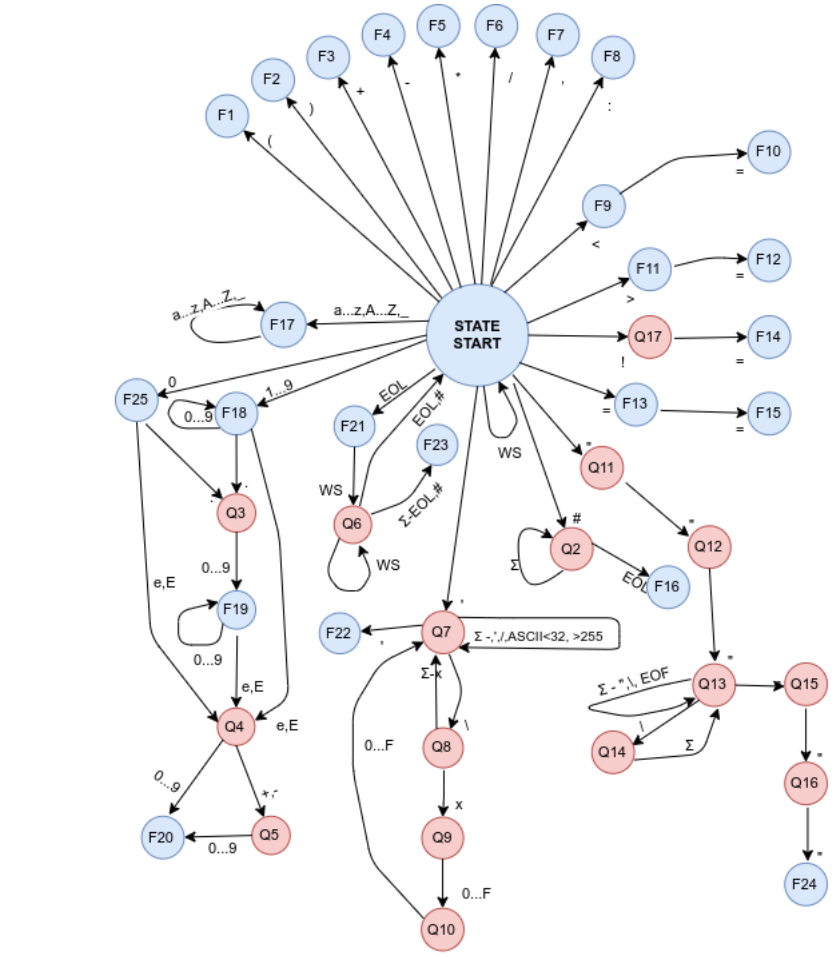
5 Záver

Projekt nás až tak nezarazil, vzhľadom nato, že sme sa naňho začínali pripravovať v celku dosť skoro. Tím sme si zostavili veľmi rýchlo a tým, že sme na rovnakom internáte tak nebol problém s akoukoľvek komunikáciou. Jednotlivé časti programu sme riešili z väčšej časti individuálne. Správnosť projektu sme si overili automatickými testami a pokusným odovzdaním, vďaka ktorému sme boli schopní projekt viac doladiť.

Na tomto projekte sme si vyskúšali implementáciu niektorých zaujímavých dátových štruktúr, algoritmov, teóriu formálnych jazykov v praxi a spoluprácu v tíme.

6 Digramy

6.1 Deterministický konečný automat popisujúci lexikálnu analýzu



F1 - Left bracket
F2 - Right bracket
F3 - Plus
F4 - Minus
F5 - Mul
F6 - Divide
F7 - Comma
F8 - Colon
F9 - Less
F10 - Less or equal
F11 - More
F12 - More or equal
Q17 - State not equal
F14 - Not equal
F13 - Assign
F15 - Equal

Q11 - State doc. string
Q12 - State doc. string second
Q13 - State doc string
Q14 - Escape sec.
Q15 - State doc. string finish
Q16 - State doc. string finish second
F24 - Doc. string
Q2 - State comment
F16 - Comment
Q7 - State string
Q8 - State escape sec.
Q9 - Escape hexadec.
Q10 - Esccape hexadec. one
F22 - String
F21 - EOL
Q6 -Counting WS for indentation

F23 - Indent/Dedent
F18 - Integer
Q3 - State point
F19 - Float
Q4 - State exponent
F20 - Float
Q5 - State plus/minus exponent
F25 - Integer

Obrázek 1: Deterministický automat špecifikujúci lexikálny analyzátor

6.2 Precedenčná tabulka

	==	!=	<=	>=	<	>	+	-	*	/	//	)	(	VAL	$
==							<	<	<	<	<	>	<	<	>
!=							<	<	<	<	<	>	<	<	>
<=							<	<	<	<	<	>	<	<	>
>=							<	<	<	<	<	>	<	<	>
<							<	<	<	<	<	>	<	<	>
>							<	<	<	<	<	>	<	<	>
+	>	>	>	>	>	>	>	>	<	<	<	>	<	<	>
-	>	>	>	>	>	>	>	>	<	<	<	>	<	<	>
*	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>
/	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>
//	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>
)	>	>	>	>	>	>	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	<	<	<	=	<	<		
VAL	>	>	>	>	>	>	>	>	>	>	>	>			>
$	<	<	<	<	<	<	<	<	<	<	<		<	<	

Obrázek 2: Precedenčná tabuľka použitá pri analýze výrazov

6.3 LL

6.3.1 LL gramatika

```
1.  <prog_body> -> indent
2.  <prog_body> -> def id ( <par_list> ) : EOL <statement>  <prog_body>
3.  <prog_body> -> EOL  <prog_body>
4.  <prog_body> -> <main_body>
5.  <main_body> -> EOL <statement>
6.  <main_body> -> EOF <end_main>
7.  <end_main> -> EOL <end_main>
8.  <end_main> -> EOF
9.  <par_list> -> id <par_list2>
10. <par_list> -> ε
11. <par_list2> , <par_list2>
12. <par_list2> ε
13. <statement> -> indent
14. <statement> -> if <expression>: EOL <statement> else: EOL <statement>
     EOL <statement>
15. <statement> -> while <expression>: EOL <statement> EOL <statement>
16. <statement> -> id = <def_value> EOL <statement>
17. <statement> -> print ( <term>, <print_rule> ) EOL <statement>
18. <statement> -> return <expression> EOL <statement>
19. <statement> -> pass EOL <statement>
20. <statement> -> def <prog_body>
21. <statement> -> EOL <statement>
22. <statement> -> <value> <expression> <statement>
23. <statement> -> dedent
24. <statement> -> EOF <main_body>
25. <statement> -> ( <expression> <statement>
26. <def_value> -> None <statement>
27. <def_value> -> id ( <arg_list> )
28. <def_value> -> ord ( <arg_list> )
29. <def_value> -> chr ( <arg_list> )
30. <def_value> -> len ( <arg_list> )
31. <def_value> -> substr ( <arg_list> )
32. <def_value> -> inputs ( <arg_list> )
33. <def_value> -> inputi ( <arg_list> )
34. <def_value> -> inputf ( <arg_list> )
35. <def_value> -> <expression>
36. <arg_list> -> <value> <arg_list2>
37. <arg_list> -> ε
38. <arg_list2> -> , <arg_list2>
39. <arg_list2> -> ε
40. <value> -> id
41. <value> -> value_int
42. <value> -> value_float
43. <value> -> value_string
44. <print_rule> , <print_rule>
45. <print_rule> ) <statement>
```

Obrázek 3: Pravidlá LL gramatiky pre syntaktickú analýzu

6.3.2 LL Tabuľka

	def	EOL	EOF	id	,	if	while	print	return	pass	ord	chr	len
<prog_body>	2	3											
<main_body>		5	6										
<end_main>		7	8										
<par_list>				9									
<par_list2>					11								
<statement>	20	21	24	16		14	15	17	18	19			
<def_value>				27							28	29	30
<arg_list>													
<arg_list2>					38								
<value>				40									
<print_rule>					44								

	substr	inputs	inputi	inputf	int*	float*	string*	None	(	)	indent	dedent	$
<prog_body>											1		4
<main_body>													
<end_main>													
<par_list>													10
<par_list2>													12
<statement>					22	22	22		25		13	23	
<def_value>	31	32	33	34				26					35
<arg_list>					36	36	36						37
<arg_list2>													39
<value>					41	42	43						
<print_rule>										45			

*value of

Obrázek 4: LL tabuľka