FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Dokumentace k projektu předmětů IFJ a IAL

Tým 97, varianta I

Gáfrik Patrik	**xgafri00**	**23 %**
Horváth Adrián	xhorva14	23 %
Kipila Maxim	xkipil00	24 %
Dohnal Ondřej	xdohna45	30 %

Obsah

1 Úvod

Cílem projektu je vytvořit program, který načte kód v jazyce IFJ21 (zjednodušená verze jazyka Teal), který přeloží do mezikódu IFJcode21. Projekt je součástí předmětů IFJ a IAL a jeho implementace probíhala v jazyce C. Hlavní náplní práce byla implementace syntaktického analyzátoru, parseru (tj. syntaktická a sémantická analýza) a generátoru instrukcí.

2 Lexikální analýza

První částí překladače je lexikální analyzátor. Je implementován v souborech `scanner.c` a `scanner.h` jako konečný deterministický automat.

Nejdůležitější činností scanneru je rozpoznání a navrácení tokenu ve funkci `getNextToken` na základě vstupního kódu v jazyce IFJ21. Ffunkce postupně načítá data ze vstupu a anylyzuje je dle navrženého konečného automatu. Funkce prochází stavy a na konci vrací o jaký token se jedná. To je zajištěné přepínačem switch umístěného v nekonečném `while` cyklu. Jednotlivé `case` reprezentují právě jeden stav automatu a při každém opakování syklu je načten právě jeden znak an vstupu.

Pro práci s tokeny využíváme strukturu `Token`. Tato struktura se skládá z typu tokenu a dat tokenu. Typy tokenů a stavy konečného automatu jsou uloženy ve výčtu (enum), aby se s nimi dalo jednoduše pracovat. U typů tokenů jsou také vypsané neterminály, které dále využíváme v syntaktické analýze a klíčová slova.

Tokeny typu `id` porovnáváme s klíčovými slovy. Pokud se jedná o klíčové slovo, změníme typ tokenu. Ve funkci `strAddChar` si alokujeme paměť pro data identifikátorů, čísel a řetězců a průběžně je vkládáme do pole. Pole musíme v průběhu zvětšovat, aby nedošlo k přetečení.

Uvolňování alokované paměti využívané pro data tokenů probíhá ve fuknci `freeTokenData`. Pokud dojde k nějaké lexikální chybě, scanner vrátí chybový kód 1.

3 Syntaktická analýza

Naše varianta projektu je založená na implementaci tabulek symbolů jako binární vyhledávací stromy. Každý uzel stromu obsahuje identifikátor, podle kterého se poté hledá, ukazatel na lokální tabulku symbolů funkce a počet parametrů každé funkce. Všechny funkce jsou implementovány v souboru `symtable.c` a jejich definice v hlavičkovém souboru `symtable.h`.

Syntaktická analýza je implementována v souborech `parser.c` a `parser.h`. K Syntaktické analýze využíváme princip rekurzivního sestupu. Na zásobník dáme nejprve vždy neterminál `prolog`, který postupně rozkládáme podle gramatických pravidel v LL tabulce na řetězec terminálů. v případě, kdy je na zásobníku neterminál, rozložíme ho ve funkci `expand` na základě gramatických pravidel. V případě, že je na vrcholu zásobníku terminál, porovná se ve funkci `parse` vstupní terminál s terminálem na vrcholu zásobníku. Pokud se liší, jedná se o syntaktickou chybu. Pokud jsou stejné, terminál na vrcholu zásobníku ze zásobníku vyjmeme. Poté pokračujeme s dalšími kontrolami.

Práci s termy a výrazy jsme vyřešili tak, že v případě, kdy syntaktická analýza očekává term nebo výraz, zavolá si příslušnou funkci (`isTerm`, `isExpr`). Ta vrací jestli je token správný. LL tabulku jsme vytvářeli podle postupu z přednášek na základě gramatických pravidel. Obtížné pro nás bylo zapsání pravidel tak aby se jednalo o LL a zároveň podchycení všech možných způsobů zapsání programu.

Pro práci se zásobníkem využíváme funkce:

- `stInit` - inicializace zásobníku
- `stPush` - vložení hodnoty na vrchol zásobníku
- `stPop` - odstranění z vrcholu zásobníku
- `stTop` - vrací vrchol zásobníku
- Zásobník je uložen ve struktuře `Stack`.

3.1 Precedenční analýza výrazů

Precedenční analýza je implementovaná v souboru `expressions.c` a `expressions.h` Pro zpracování výrazů je vytvořena precedenční tabulka.

Analýza výrazů skončí právě tehdy, dokud na zásobníku nezůstanou neterminály či samotný terminál `$`. Poté se funkce ukončí a jestliže proběhla v pořádku, pokračuje parser v analýze. v případě chyby se vypíše chybová hláška a program skončí chybou.

4 Sémantická analýza

Sémantická analýza je implementována v souborech `parser.c` a `parser.h`. v rámci sémantické analýzy se nově definované proměnné a funkce vkládají do tabulky symbolů za pomocí funkcí `addFn` a `addVar`. Platnost definice proměnných v rámci bloku je řešena tak, že každý blok má vlastní tabulku symbolů. Na začátku nového bloku je aktuální tabulka symbolů uložena na zásobník a je vytvořena nová tabulka. Na konci bloku je aktuální tabulka zrušena a nahrazena tabulkou z vrcholu zásobníku. Při zjišťování jestli je použitá proměnná definovaná se prohledává aktuální tabulka symbolů, a pak postupně všechny tabulky symbolů na zásobníku ve směru od vrcholu zásobníku dolů. K tomu slouží funkce `symTableSearchAll`. Pro práci se zásobníkem využíváme funkce:

- `symStInit` - inicializace zásobníku
- `symStPush` - vložení hodnoty na vrchol zásobníku
- `symStPop` - odstranění z vrcholu zásobníku

Zásobník je uložen ve struktuře `SymStack`. Sémantická analýza provádí následující kontroly:

- Jestli existuje funkce main
- Jestli po package následuje main
- Jestli je použitá proměnná definovaná
- Duplicitní deklarace proměnných a funkcí

5 Generování instrukcí

Generování výsledného mezikódu probíhá ve funkci `printHeaderCode`, která je implementovaná v souboru `codegenerator.c` a `codegenerator.h`. Před samotným voláním funkce je na začátku `parseru` zavolána funkce `baseCode`, která vygeneruje hlavičku souboru v mezikódu, která je nutná pro další funkčnost celkového programu a jeho následnou interpetaci. v této části se přeskočí na funkci `main`.

Generování instrukcí jsme nestihli dokončit, proto máme implementovaný jen tento základ.

6 Práce v týmu

Komunikace probíhala především prostřednictvím Discord serveru, který jsme vytvořili přímo pro tento projekt. Se spolužáky jsme se viděli na prvotních schůzkách osobně, poté jsme (a také díky aktuální situaci) komunikovali online. Nadále jsme průběžně komunikovali o nastálých změnách a problémech, na které se v průběhu tvorby projektu naráželo.

6.1 Rozdělení práce

Práci na projektu jsme si rozdělili následujícím způsobem:

Gáfrik Patrik: Tvorba tabulky symbolů, implementace parseru, generování kódu

Horváth Adrián: Návrh konečného automatu, implementace precedenční analýzy výrazů

Maxim Kipila: Implementace parseru, tvorba dokumentace

Dohnal Ondřej: Precedenční analýza, lexikální analýza, LL tabulka, LL gramatika, implementace parseru

6.2 Verzování

Pro větší přehled ve změnách v projektu a lepší organizaci společné práce jsme se rozhodli použít verzovací systém Git, konkrétně na platformě Github.com. Díky tomuto nástroji byla možná práce více lidí na více částech projektu současně. Snažili jsme se využívat také větvení (branch), které nám umožňovalo lépe dokumentovat opravy a nové funkce přidané do projektu vždy s možností "vrátit se zpět"a souběžnou zároveň práci více lidí na jednom souboru. Většina práce se ale nakonec odehrála v hlavní větvi, jelikož na konkrétním souboru většinou pracoval pouze jeden člen týmu.

7 Testování

Díky povaze projektu (předem definované chybové kódy, které nastávají v předem specifikovaných situacích) jsme se od začátku rozhodli používat automatické testy, které nám řeknou, jestli nová změna provedená v kódu neovlivní funkčnost jiné (již funkční) části projektu. Taktéž měly testy za úkol projekt "prověřit" v situacích, se kterými se při psaní kódu nemuselo počítat (tzv "edge cases").

A LL-gramatika

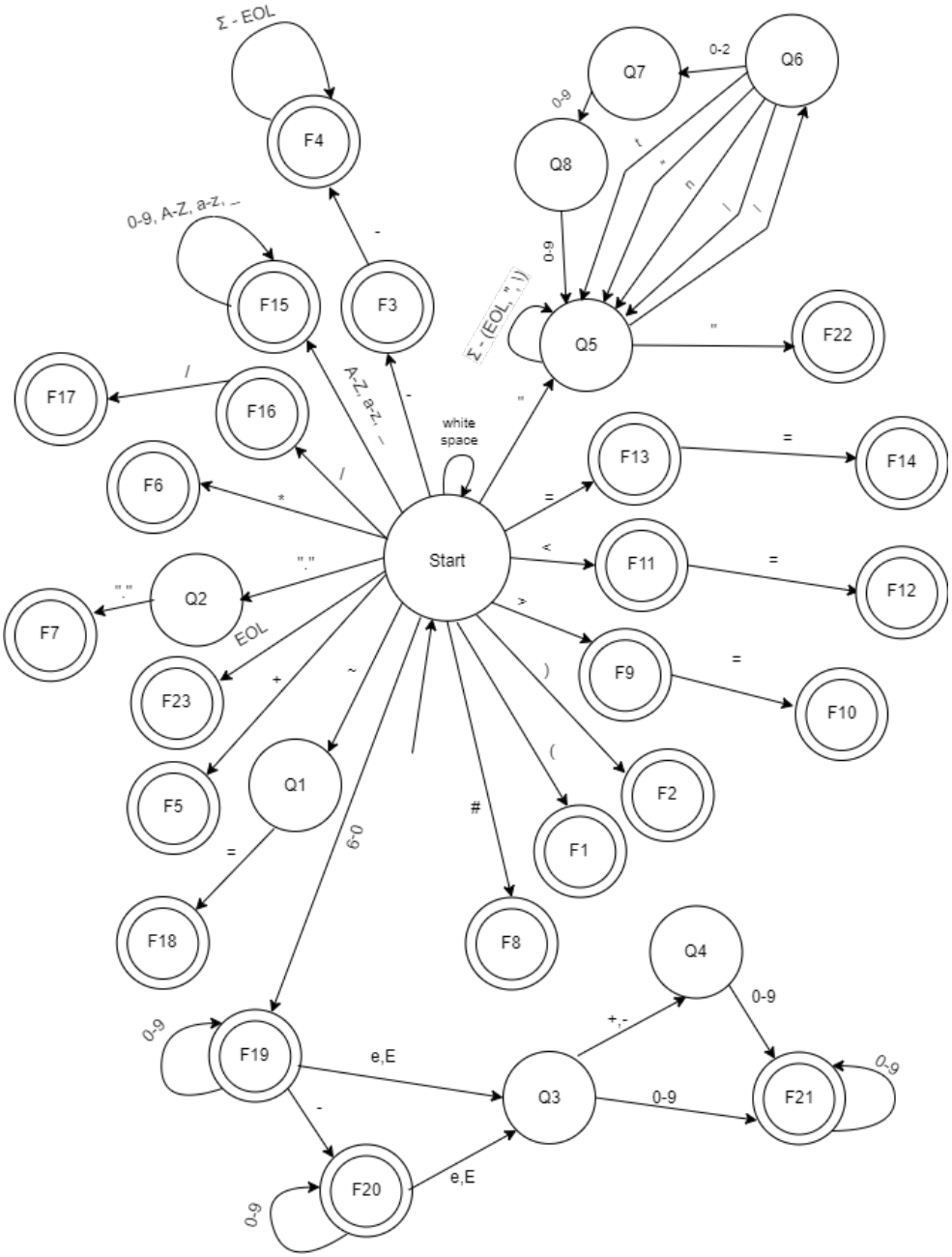
1. <prolog> → require ”ifj21” <prog> EOF
2. <prog> → global ID : function (<params_type>) <ret_type> <prog>
3. <prog> → function ID (<params>) <ret_type> <statement> end <prog>
4. <prog> → ID (<arg>)
5. <params_type> → <type> <params_type_n>
6. <params_type> → ε
7. <params_type_n> → , <type> <params_type_n>
8. <params_type_n> → ε
9. <ret_type> → : <type>
10. <ret_type> → ε
11. <params> → ID : <type> <params_n>
12. <params> → ε
13. <params_n> → , ID : <type> <params_n>
14. <params_n> → ε
15. <statement> → local ID : <type> <eq> <statement>
16. <statement> → ID <id_list> = <def_value> <statement>
17. <statement> → if <expression> then <statement> else <statement> end <statement>
18. <statement> → while <expression> do <statement> end <statement>
19. <statement> → return <ret_exp> <statement>
20. <statement> → write (<arg>) <statement>
21. <statement> → ε
22. <ret_exp> → <expression> <expression_list>
23. <ret_exp> → ε
24. <def_value> → <builtin>
25. <def_value> → ID <func>
26. <def_value> → <expression> <expression_list>
27. <def_value> → <value>
28. <func> → (<arg>)

29. <func> → ε
30. <id_list> → , ID <id_list>
31. <id_list> → ε
32. <expression_list> → , <expression> <expression_list>
33. <expression_list> -> ε
34. <eq> → = <def_value>
35. <eq> → ε
36. <builtin> → reads (<arg>)
37. <builtin> → readi (<arg>)
38. <builtin> → readn (<arg>)
39. <builtin> → tointeger (<arg>)
40. <builtin> → substr (<arg>)
41. <builtin> → ord (<arg>)
42. <builtin> → chr (<arg>)
43. <arg> → <value> <arg_n>
44. <arg> → ε
45. <arg_n> → , <value> <arg_n>
46. <arg_n> → ε
47. <value> → INTEGER
48. <value> → NUMBER
49. <value> → STRING
50. <value> → ID
51. <value> → nil
52. <type> → integer
53. <type> → number
54. <type> → string
55. <type> → nil

B Precedenční tabulka

	..	+ -	*/	#	r	(	i	)	$
..	<	<	<	<	>	<	<	>	>
+ -	>	>	<	<	>	<	<	>	>
***/**	>	>	>	<	>	<	<	>	>
#	>	>	>		>	<	<	>	>
r	<	<	<	<	<	<	<	>	>
(	<	<	<	<	<	<	<	=	
i	>	>	>		>			>	>
)	>	>	>		>			>	>
$	<	<	<	<	<	<	<		

C Konečný stavový automat lexikální analýzy



D LL tabulka

	require	global	ID	:	function	)	end	,	local	if	else	while	return	=	write	reads	readi	readn	tointeger	substr	ord	chr	INT_VAL	NUM_VAL	STR_VAL	nil	integer	number	string	EXPRESSION
<prolog>	1																													
<prog>		2	4		3																									
<params_type>								6																		5	5	5	5	
<params_type_n>						8		7																						
<ret_type>		10	10	9	10				10	10		10	10		10															
<params>			11			12																								
<params_n>						14		13																						
<statement>			16				21		15	17	21	18	19		20															
<ret_exp>			23				23		23	23	23	23	23		23															22
<def_value>			25													24	24	24	24	24	24	24								26
<id_list>								27						28																
<expression_list>			30				30	29	30	30	30	30	30		30															
<eq>			32				32		32	32	32	32	32	31	32															
<builtin>																33	34	35	36	37	38	39								
<arg>						41																	40	40	40	40				
<arg_n>						43		42																						
<value>			47																				44	45	46	48				
<type>																										52	49	50	51	