FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Překladač jazyka IFJ21 - Projektová dokumentace

Tým 004 - varianta II.

Vojtěch Eichler	**xeichl01**	**25 %**
Adam Zvara	xzvara01	25 %
Václav Korvas	xkorva03	25 %
Tomáš Matuš	xmatus37	25 %

2021

1 Úvod

Cílem tohoto projektu bylo vytvořit překladač, který přeloží zadaný jazyk IFJ21 do cílového mezijazyka IFJCode2021. Jazyk IFJ21 je staticky typovaný a imperativní jazyk. Jedná se o podmožinu jazyka Teal. Překladač je napsaný v jazyce C.

2 Implementace

Překladač je sestaven ze 3 hlavních částí a to scanner, parser a generátor kódu. Scanner provádí lexikální analýzu a načítání vstupního kódu, parser pak provádí syntaktickou a sémantickou analýzu.

2.1 Lexikální analýza

Lexikální analýza je implementována v souborech `scanner.c` a `scanner.h`. Hlavní funkce lexikálního analyzátoru je funkce `get_token`, která postupně čte znak po znaku ze standardního vstupu. Tyto znaky poté převádní na strukturu `token_t`. Tato struktura obsahuje atribut a typ tokenu. Jako typ tokenu může být identifikátor, řetězec, celé nebo desetinné číslo, EOF, aritmetické a porovnávací operátory nebo jiné znaky povolené v jazyce IFJ21. Atribut mají pouze tokeny typu řetězec, identifikátor, klíčové slovo nebo čísla. Zda-li se jedná o klíčové slovo nebo identifikátor se stará pomocná funkce `check_keyword`, která porovná daný řetězec se všemi klíčovými slovami, pokud se s žádným z nich neshoduje nastaví jako typ tokenu identifikátor. Funkce `get_token` pak také využívá řadu dalších pomocných funkcí například pro převod řetězce na číslo.

Lexikální analyzátor byl implementován na základě námi vytvořeného deterministického konečného automatu (DKA). Tento DKA je implementován jako jeden nekonečně se opakující switch, kde jednotlivé případy case reprezentují jednotlivé stavy tohoto automatu. Pokud se automat dostane do jednoho z koncových stavů, tak se funkce ukončí a vrátí odpovídající token. Pokud ovšem automat skončí v nekoncovém stavu nebo je načten znak, který není povolen v jazyce IFJ21, tak nastává lexikální chyba a funkce je ukončena s návratovým kódem 1.

Escape sekvence jsou řešeny za pomoci pole, do kterého se postupně ukládají znaky, které mohou být v rozsahu 0-9. Toto decimalní číslo jehož maximální hodnota je 255, je poté převedeno na korespondující ASCII hodnotu.

2.2 Syntaktická analýza

Syntaktická analýza je implementována metodou rekurzivního sestupu. Překlad zdrojového kódu začíná zavolaním funkce `parse` z funkce main ve zdrojovém souboru `parser.c`. Následuje inicializace tokenu, do kterého lexikální analyzátor zapisuje hodnoty, globální tabulky symbolů, paměti pro postupné ukládání generovaných instrukcí a také se nainicializuje pomocná struktura.

Během implementace jsme postupně zjišťovali, že bychom pro generování a sémentické kontroly potřebovali držet v paměti různé hodnoty jako například počty parametrů u volání funkce, ukazatel na funkci do globální tabulky symbolů a několik dalších hodnot. Z tohoto důvodu jsme vytvořili pomocnou strukturu `parser_helper` a funkce nad ní, implementovanou v souboru `parser_helper.c`.

Z funkce `parse` se syntaktický analyzátor postupně rekurzivně zanořuje podle sekvence tokenů, které získavá od lexikálního analyzátoru. Po celou dobu běhu se v globální proměnné `ret` udržuje aktuální stav a pokud analyzátor objeví chybu, nebo nastane interní chyba, tato proměnná se nastaví na nenulovou hodnotu a začne vynořování z rekurze.

Jedna z největších výzev byla absence ukončovacího znaku za příkazem, nebo pevně stanovené pravidla odsazování, což zkomplikovalo práci s načtenými tokeny. Proto analyzátor pracuje s proměnnou `backup_token`, která kromě role dočasného držení tokenu, plní roli při rozhodování, zda je nutné načítat další token, nebo je potřeba pracovat s tokenem který už má načtený. Tímto způsobem jsme řešili situaci, kdy je při vyhodnocování výrazů nevyhnutelně načten jeden token za koncem výrazu, jinak totiž nejsme schopní určit konec výrazu.

2.2.1 Syntaktická analýza výrazů

Analýza výzazů se provadí metodou zdola nahoru. K tomuto bylo zapotřebí sestavit precedenční tabulku. Pro její zjednodušení jsme sloučili operátory se stejnou asociativitou a prioritou. Celá analýza výrazů je implementována v souboru `expression.c` a využívá stack implementovaný ve `stack.c`.

Použitý algoritmus vychází z algoritmu představeného na přednášce IFJ. Po zavolání funkce `expression` se inicializuje stack a vloží se na něj pomocný znak `DOLLAR`, který značí začátek výrazu. Načtený token se funkcí `token_to_symbol` převeda na vnitřní symbol, se kterým se dále pracuje na stacku. Pomocí funkce `symbol_to_index` se zjistí z vrcholu stacku a nového symbolu souřadnice precedenčního znaku tabulce.

- Pro znak $=$ se uloží na zásobník načtený symbol a provede se načtení nového tokenu.
- Pro znak $<$ se uloží na zásobník pomocný znak `HANDLE`, symbol načtený ze vstupu a provede se načtení dalšího tokenu.
- Pro znak $>$ se provede funkce `reduce`, která podle daných pravidel redukuje symboly uložené na zásobníku. Funkce zjistí kolik symbolů se nachází před `HANDLE`a podle jejich typu deterministicky volí redukční pravidlo. Pravidlo se provede odebráním potřebného počty symbolů ze zásobníku a nahradíme je symbolem `NON_TERM`. Pokud neexistuje pravidlo pro redukci, tak funkce hlásí syntaktickou chybu.
- Znak $-$ značí konec analýzy.

Algoritmus se provádí dokud se nenarazí na ukončení. Nakonec se zkontroluje stav stacku a pokud se liší od $E, funkce hlásí syntaktickou chybu.

Dále se zde řeší přiřazení funkce, tzn. pokud narazí na identifikátor, který se nachází v globální tabulce, funkce končí a předává řízení zpět do parseru.

2.2.2 Stack pro syntaktickou analýzu

Pro precedenční analýzu se využívá stack, na který se ukládají vstupní symboly a provádí se na něm redukování výrazů podle pravidel.

Implementace se nachází ve `stack.c`. Umožňuje všechny klasické operace nad zásobníkem a navíc umožňuje několik funkcí navíc pro zjednodušení kontroly pravidel. Funkce `push_above_term` slouží pro vložení symbolu za první terminál na stacku. Funkce `items_to_handle` vrací počet symbolů, které se nachází před `HANDLE` a využívá se pro vyhodnocení typu pravidla. Funkce `find_len_op` vyhledá na stacku operátor # a funkce `get_top_operator` vrací operátor na vrcholu zásobníku. Obě funkce se používájí pro sémantickou kontrolu.

2.3 Sémantická analýza

Sémantická analýza pracuje se dvěma typy tabulek symbolů. První z nich je globální tabulka symbolů, do které jsou na začátku syntaktické analýzy uloženy vestavěné funkce a postupně během překladu i uživatelem definované funkce. Druhý typ tabulky je lokální tabulka symbolů, na začátku funkce se vytvoří první lokální tabulka a při dalším zanoření do cyklu, nebo v těle podmínky se vytvoří další, která se pomocí ukazatele napojí na ostatní lokální tabulky. Vznikne tak jednostranně vázaný seznam, na jehož začátku se nachází nejvíce zanořená tabulka a končí tabulkou vytvořenou při vstupu do těla funkce. Průchodem seznamem lokálních tabulek jsme tak schopni kontrolovat zda a případně v jakém bloku platnosti jsou proměnné definované. Obě tabulky jsou naimplementované v souboru `symtable.c`.

Sémantická analýza ke své činnosti dále využívá řetězce na ukládání typů různých hodnot a strukturu `parser_helper`, popsanou v kapitole o syntaktické analýze. Do řetězců se ukládají například typy proměnných při inicializaci, které se pak porovnávají s hlavičkou funkce, nebo s typy hodnot, kterými dané proměnné inicializujeme. Tímto způsobem jsme schopni kontrolovat nejen typy hodnot, ale i jejich počet, jelikož se do řetězce ukládají jen jeden znak pro každou hodnotu. Obdobně jsou řešené i parametry u volání funkce, nebo například návratové hodnoty u funkcí.

2.4 Generování kódu

Pro generování kódu jsme zvolili metodu přímého generování zásobníkových instrukcí během rekurzivního sestupu v syntaktické analýze.

Instrukce se generují na základě aktuálně vykonávané funkce v syntaktické analýze a také na základě kontextu (informace uložené v struktuře `parser_helper`). Vygenerované instrukce se ukládají do pomocné struktury `ibuffer` - struktura definována v souboru `ibuffer.c` obsahující "flexible array member", do kterého se ukládají samotné instrukce, které se v průběhu parsování tisknou na standardní výstup.

Jeden z problémů je generovaní unikátních jmen identifikátorů mezi funkcemi nebo při vytvoření nového lokálního rámce ve funkci (podmínka / cyklus). Tento problém jsme vyřešili pomocí generovaní unikátního jména identifikátoru: `jméno_funkce$zanoření$jméno_identifikátoru` pro proměnné, které jsou v hlavním těle funkce. Jména proměnných v nových lokálních rámcích jsou navíc doplněna o `if_counter` a `while_counter`, které představují počet aktuálních zanoření, aby se dala vytvářet stejná jména proměnných uvnitř více lokálních rámců. Tedy výsledný formát je:

`jméno_funkce$zanoření$if_counter$while_counter$jméno_identifikátoru`

Další problém, na který jsme při generovaní narazili, je deklarace identifikátorů uvnitř cyklů. To jsme vyřešili použitím pomocné struktury `defvar_buffer` a stavového řetězce v struktuře `parser_helper`. Pokud stavový řetězec obsahuje informaci, že jsme v těle cyklu, všechny deklarace identifikátorů se generují do `defvar_bufferu` (ostatní instrukce se generují do `ibufferu`). Když zjistíme, že se nenacházíme v cyklu, vytiskne se obsah `defvar_bufferu` a následně obsah `ibufferu`, čímž zabráníme vícenásobné deklaraci proměnné.

Vestavěné funkce se generují na konci celého programu. Aby jsme negenerovali zbytečně všechny vestavěné funkce, používáme strukturu `builtin_used`, do které si ukládáme informace o použitých funkcích, které se pak generují. Speciálním případem je funkce `write`, která se generuje přímo na místě volání této funkce.

3 Práce v týmu

3.1 Rozdělení práce

Člen týmu	Přidělená práce
Vojtěch Eichler	Vedoucí týmu, syntaktická analýza, sémantická analýza, testování
Adam Zvara	Sémantická analýza, generování kódu, testování
Václav Korvas	Lexikální analýza, testování
Tomáš Matuš	Precedenční analýza výrazů, sémantická analýza, dokumentace

Tabulka 1: Rozdělení práce v týmu

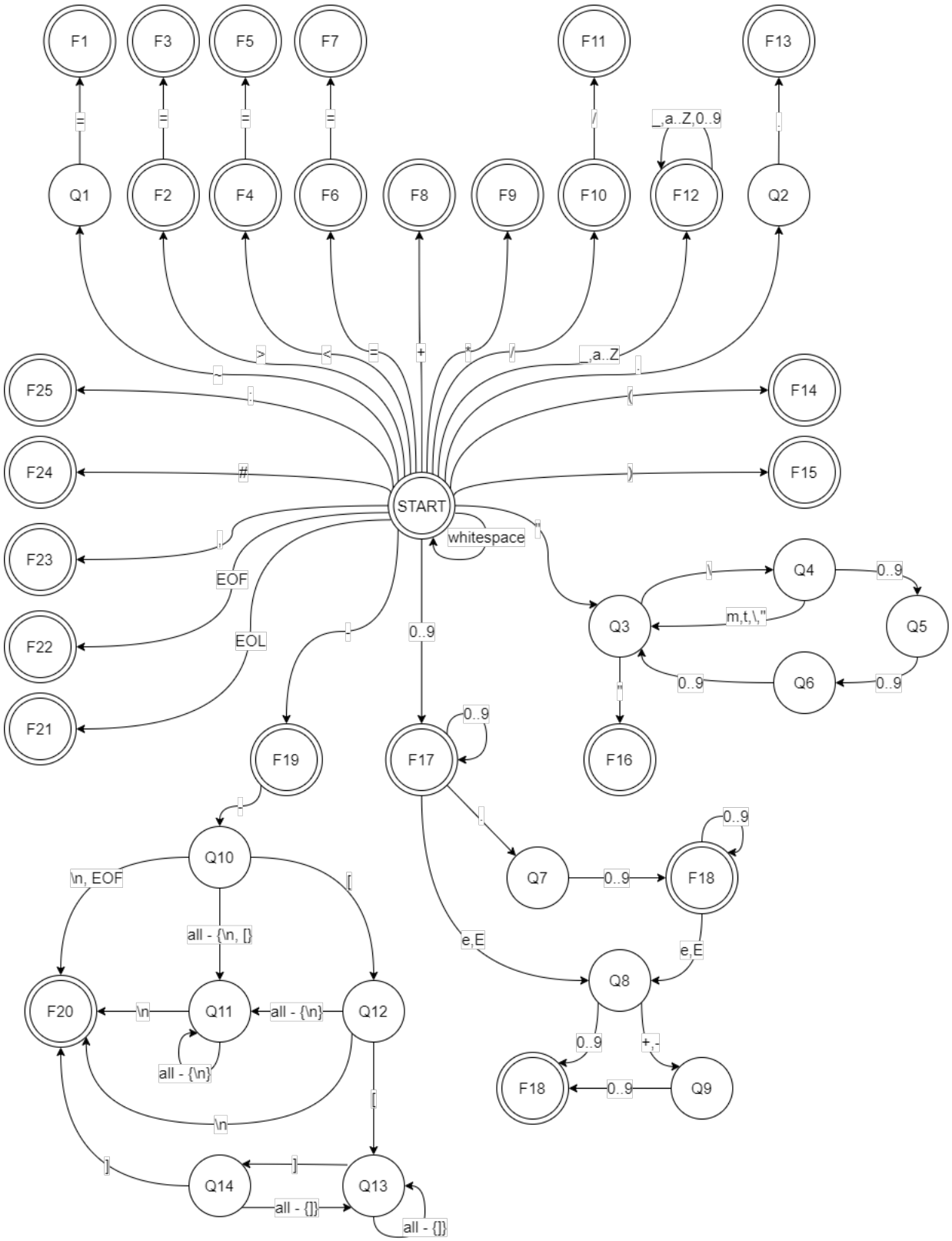
3.2 Komunikace

Našim hlavním komunikačním nástrojem byl Discord. Dále jsme se osobně setkávali na předem dohodnutých schůzkách ve studovnách a nebo se bavili během přestávek mezi přednáškami.

3.3 Verzovací systém

Pro verzování jsme používali nástroj `git` a pro vzdálené sdílení repozitáře jsme využili `GitHub`.

4 Diagram konečného automatu



Obrázek 1: Diagram konečného automatu specifikující lexikální analyzátor

F1	NOT_EQUAL	F14	LEFT_BRACKET	Q2	CONCAT_START
F2	GREATER_THAN	F15	RIGHT_BRACKET	Q3	STRING_START
F3	GREATER_THAN_EQUAL	F16	STRING_LITERAL	Q4	STRING_ESCAPE
F4	LESSER_THAN	F17	INTEGER	Q5	STRING_ESCAPE_HEXADEC_1
F5	LESSER_THAN_EQUAL	F18	NUMBER	Q6	STRING_ESCAPE_HEXADEC_2
F6	ASSIGN	F19	MINUS	Q7	NUMBER_START_DOT
F7	IS_EQUAL	F20	COMMENT	Q8	NUMBER_E
F8	PLUS	F21	EOL	Q9	NUMBER_E_PLUS_MINUS
F9	MUL	F22	EOF	Q10	COMMENT_START
F10	DIV	F23	COMMA	Q11	COMMENT_SKIP
F11	INT_DIV	F24	STRING_LENGTH	Q12	COMMENT_BLOCK_START
F12	ID	F25	COMMA	Q13	COMMENT_BLOCK
F13	CONCAT	Q1	NOT_EQUAL_START	Q14	COMMENT_BLOCK_STOP

Tabulka 2: Legenda konečného automatu pro lexikální analýzu

5 LL - gramatika

1. <require> -> REQUIRE STRING_LIT <prog>
2. <prog> -> GLOBAL ID : FUNCTION (<params>) <ret_params> <prog>
3. <prog> -> FUNCTION ID (<params_2>) <ret_params> <body> <prog>
4. <prog> -> ID (<args> <prog>
5. <prog> -> EOF
6. <params> -> <types-keyword> <params_n>
7. <params> -> eps
8. <params_n> -> , <types-keyword> <params_n>
9. <params_n> -> eps
10. <params_2> -> ID : <types-keyword> <params2_n>
11. <params_2> -> eps
12. <params_2_n> -> , ID : <types-keyword> <params2_n>
13. <params_2_n> -> eps
14. <ret_params> -> : <types-keyword> <ret_params_n>
15. <ret_params> -> eps
16. <ret_params_n> -> , <types-keyword> <ret_params_n>
17. <ret_params_n> -> eps
18. <body> -> LOCAL ID : <types-keyword> <init> <body>
19. <body> -> IF <expr> THEN <body> ELSE <body> END <body>
20. <body> -> WHILE <expr> DO <body> END <body>
21. <body> -> ID <body_n> <body>
22. <body> -> END
23. <body> -> RETURN <R_side> <body>
24. <body_n> -> (<args>
25. <body_n> -> = <assign_single>
26. <body_n> -> , ID <assign_multi> <R_side>

Tabulka 3: Gramatika řídící syntaktickou analýzu

27. <assign_single> -> <expr>
28. <assign_multi> -> , ID <assign_multi>
29. <assign_multi> -> =
30. <R_side> -> <expr> <R_side_n>
31. <R_side_n> -> , <R_side>
32. <R_side_n> -> eps
33. <init> -> = <init_n>
34. <init> -> eps
35. <init_n> -> <expr>
36. <args> ->)
37. <args> -> <term> <args_n>
38. <args_n> -> , <args>
39. <args_n> ->)
40. <term> -> ID
41. <term> -> INT_LIT
42. <term> -> NUM_LIT
43. <term> -> STRING_LIT
44. <types-keyword> -> STRING
45. <types-keyword> -> NUMBER
46. <types-keyword> -> INTEGER
47. <expr> -> ID (<args>

Tabulka 4: Gramatika řídící syntaktickou analýzu – pokračování

6 LL - tabulka

	REQUIRE	GLOBAL	FUNCTION	ID	EOF	STRING	NUMBER	INTEGER	,	ID	:	LOCAL	IF	WHILE	END	RETURN	(	=	INT_LIT	NUM_LIT	STRING_LIT
reqire	1																				
prog		2	3	4	5																
params						6	7	8													
params_n									9												
params_2										10											
params_2_n									11												
ret_params											12										
ret_params_n									13												
body				14								15	16	17	18	19					
body_n									20								21	22			
assign_single				23																	
assign_multi									24									25			
r_side				26					27												
r_side_n				28					29												
init				30																	
init_n				31																	
args				32															33	34	35
args_n									36												
term				37															38	39	40
types_keyword						41	42	43													

Tabulka 5: LL – tabulka

7 Precedenční taulka

Načtený token										
Vrchol stacku		#	* / //	+ -	..	r	(	)	id	$
	#	–	>	>	–	>	<	–	<	>
	* / //	<	>	>	–	>	<	>	<	>
	+ -	<	<	>	–	>	<	>	<	>
	..	–	–	–	<	>	<	>	<	>
	r	<	<	<	<	–	<	>	<	>
	(	<	<	<	<	<	<	=	<	–
	)	>	>	>	>	>	–	>	>	>
	id	–	>	>	>	>	–	>	>	>
	$	<	<	<	<	<	<	–	<	–

Tabulka 6: Precedenční tabulka