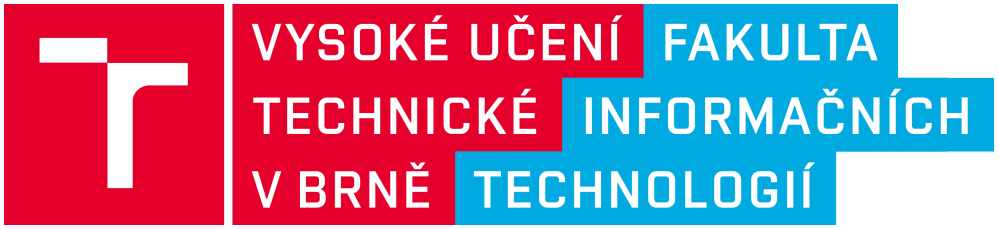


Projektová dokumentace
Implementace překladače imperativního jazyka IFJ21

Tým 17, varianta I

7. prosince 2021

Ladislav Vašina	**(xvasin11)**	25 %
Dominik Vágner	(xvagne10)	25 %
Tomáš Polívka	(xpoliv06)	25 %
Vojtěch Hájek	(xhajek51)	25 %

Obsah

1 Úvod

Tato projektová dokumentace popisuje návrh a implementaci překladače imperativního jazyka IFJ21, který je podmnožinou jazyka Teal. Projekt má za úkol načíst zdrojový kód v jazyce IFJ21 a přeložit jej do tzv. mezikódu IFJcode21. Dle naší varianty zadání (I) implementujeme tabulku symbolů pomocí binárního vyhledávacího stromu.

2 Návrh a implementace

Implementace překladače imperativního jazyka zahrnuje vytvoření několika částí programu, které řeší určité moduly překladače.

2.1 Práce s řetězci

Pro ulehčení vývoje a operací s řetězci jsme si vytvořili knihovnu pro práci s dynamickými řetězci. V této knihovně je implementována struktura `string`, která si uchovává ukazatel na řetězec, informace o jeho délce a množství paměti, které daný řetězec zabírá. Při inicializaci této struktury alokujeme místo v paměti pouze pro nějaké znaky a při dalším vkládání znaků kontrolujeme, zda nám dostačuje alokovaný prostor a popř. ho navýšíme o pevný počet znaků (velikost navýšení je uložena v konstantě). V neposlední řadě jsou v knihovně vytvořeny různé funkce pro editaci řetězců. Knihovna pro práci s řetězci je implementována v souborech `str.c` a `str.h`.

2.2 Lexikální analýza

Jako první část projektu byl naimplementován scanner, který řídí celou lexikální analýzu. Lexikální analyzátor je implementován v souborech `scanner.c` a `scanner.h`. Lexikální analyzátor načítá ze stdin zdrojový soubor, který pomocí předem vytvořeného deterministického konečného automatu, implementovaného nekonečně se opakujícím příkazem switch, generuje dané tokeny, které se skládají z typu tokenu, čísla řádku, čísla sloupce a nakonec samotných dat tokenu. Současně se kontroluje, zda právě generovaný token není klíčové slovo jazyka IFJ21. Pokud lexikální analyzátor nerozpozná aktuální lexému, vyvolá se chyba 1 (ERR_LEX). Vygenerované tokeny jsou získány funkcí `get_token_list()` a následně nahrány do předpřipraveného dvojitě vázaného seznamu, který je implementován v souborech `tokenList.c` a `tokenList.h`, kde je možné je dále procházet a pracovat s nimi.

2.3 Syntaktická analýza

Nejdůležitější částí celého překladače je syntaktická analýza. Implementaci syntaktické analýzy nalezneme v souborech `parser.c` a `parser.h`. Vyjma výrazů se syntaktická analýza řídí předem vytvořenou LL-gramatikou a tzv. metodou rekurzivního sestupu dle pravidel uvedených v LL-tabulce. Syntaktická analýza získává tokeny pomocí dvojitě vázaného seznamu, který obsahuje všechny načtené tokeny. Každé pravidlo má vytvořenu svoji funkci, ve které se implementuje daného pravidla. Hodnotu tokenu získáváme pomocí makra `GET_TOKEN`.

2.4 Zpracování výrazů pomocí precedenční syntaktické analýzy

Pro zpracování výrazů je využita metoda zdola nahoru pomocí precedenční tabulky. Tato metoda je implementována v souborech `prec_table.c` a `prec_table.h`. Ve funkci `expression()` se postupně na zásobník načítají tokeny a určuje se typ daných tokenů a pomocí precedenční tabulky nadále určujeme vztahy mezi nezpracovaným tokenem a tokenem na vstupu. Nadále si vyhledáme dle vstupu v tabulce akci, kterou s výrazem máme provést. Pokud se provádí akce `reduce`, aplikujeme jedno z pravidel. Pokud žádné pravidlo nedokážeme použít, syntaktická analýza skončí s chybou.

2.5 Sémantická analýza

Implementaci sémantické analýzy nalezneme v souborech `parser.c` a `parser.h`, kde se využívá struktura binárního vyhledávacího stromu pro kontrolu jednotlivých funkcí. Pokud zjistíme, že se pracuje s funkcí, tak se jednotlivá data funkce ukládají do struktury `function_data`, která se nachází v souboru `symtable.h`. Nadále je tato struktura uložena do globálního vyhledávacího stromu. Tento globální vyhledávací strom je použit při kontrole vstupních parametrů a návratových hodnot podle určených pravidel. Pro těla funkcí se používá lokální binární vyhledávací strom (také definován v `symtable.h`), kde se jednotlivé stromy berou jako lokální rámce a s postupným zanořováním se ukládají další rámce na zásobník.

2.6 Generování cílového kódu

3 Souhrn použitých struktur

Při vývoji překladače jsme využívali různé datové struktury pro ulehčení celkové implementace programu.

3.1 Dvojitě vázaný seznam

- Tuto strukturu používáme k:
  - načtení jednotlivých tokenů v scanneru
  - průchodu jednotlivých tokenů pro kontrolu pravidel v parseru
  - naplnění nového listu, který se posílá společně s typem hodnoty do funkce `expression`, která se nachází v souboru `prec_table.c`, kde se provádí sémantická a syntaktická analýza výrazů

3.2 Zásobník

Využíváme 2 druhy zásobníku, jeden pro tokeny a druhý pro vyhledávací stromy.

1) Zásobník tokenů se využívá v:
   - parseru, kde se podle něho kontroluje vícenásobné přiřazení
   - v souboru `prec_table.c`, kde se využívá ke kontrole příchozího výrazu a zpracování výrazů do postfixového formátu
2) Zásobník binárního vyhledávacího stromu se využívá pro zanořování v těle funkcí.

4 Práce v týmu

S projektem jsme bohužel začali v celku pozdě, a proto se nám nevyhl počáteční chaos s tím, kdo má co dělat. Nakonec si ale myslím, že jsme se s problémy úspěšně vypořádali.

4.1 Verzovací systém

Při společné práci jsme využívali verzovací systém Git. Hlavním důvodem, proč jsme si v našem týmu zvolili používat Git, byla naše předešlá zkušenost s tímto verzovacím systémem z jiných projektů, které jsme na fakultě vytvářeli. Jako vzdálený repozitář jsme používali GitHub.

4.2 Komunikace

Hlavním textovým informačním kanálem našeho týmu byla platforma Facebook Messenger. Na této platformě jsme společně řešili časy schůzek našeho týmu a drobné problémy, které se v průběhu vývoje vyskytly. Pro video a hlasovou komunikaci jsme využívali platformu Discord. Na platformě Discord jsme také využívali sdílení obrazovky např. k párovému programování.

4.3 Rozdělení práce mezi členy týmu

V průběhu implementace projektu jsme pravidelně všichni vzájemně konzultovali a také jsme si napomáhali s nově vzniklými problémy.

Člen týmu	Přidělená práce
Ladislav Vašina	knihovna pro práci s řetězci, precedenční tabulka, syntaktická analýza výrazů, dokumentace, prezentace, makefile
Dominik Vágner	generování kódu, lexikální analýza, scanner
Tomáš Polívka	sémantická analýza, parser, LL gramatika, dokumentace
Vojtěch Hájek	parser, LL gramatika, LL tabulka, syntaktická a sémantická analýza, syntaktická analýza výrazů

Tabulka 1: Rozdělení práce mezi jednotlivé členy týmu

5 Metriky kódu

Počet zdrojových souborů: 19
Počet řádků kódu: **TO-DO!!!**
Velikost spustitelného souboru: **TO-DO!!!**

6 Závěr

Tvorba překladače imperativního jazyka umožnila každému členovi týmu si vyzkoušet využití teoretických poznatků získaných z přednášek předmětů *Formální jazyky a překladače* a *Algoritmy*. Bylo zajímavé sledovat postup implementace programu a následovně naše hlubší pochopení problémů implementace překladače. Tento projekt byl pro každého z nás zatím největším projektem, co jsme v naší programátorské kariéře tvořili a s přesvědčením můžeme říci, že nás každého obohatil, jak ve zkušenostech programovat takto rozsáhlý projekt, tak i ve společné týmové komunikaci a vzdálené spolupráci při programování. Řešení našeho překladače bylo primárně testováno na platformě Ubuntu či PopOs.

A LL – gramatika

1) START -> require "ifj21" MAIN_LIST eof
2) MAIN_LIST -> function function_id (LIST_OF_PARAMS) RETURN_LIST_OF_TYPES STATEMENT end MAIN_NEXT
3) MAIN_LIST -> function_id(ENTRY_LIST_PARAMS) MAIN_NEXT
4) MAIN_LIST -> global function_id : function (LIST_OF_TYPES) RETURN_LIST_OF_TYPES MAIN_NEXT
5) MAIN_NEXT -> MAIN_LIST
6) MAIN_NEXT -> ε
7) LIST_OF_PARAMS -> ε
8) LIST_OF_PARAMS -> value_id : TYPE_VALUE PARAM_NEXT
9) PARAM_NEXT -> , value_id : TYPE_VALUE PARAM_NEXT
10) PARAM_NEXT -> ε
11) ENTRY_LIST_PARAMS -> ENTRY_PARAM ENTRY_PARAM_NEXT
12) ENTRY_LIST_PARAMS -> ε
13) ENTRY_PARAM -> expression
14) ENTRY_PARAM -> value_id
15) ENTRY_PARAM_NEXT -> , ENTRY_PARAM ENTRY_PARAM_NEXT
16) ENTRY_PARAM_NEXT -> ε
17) LIST_OF_TYPES -> ε
18) LIST_OF_TYPES -> TYPE_VALUE TYPE_NEXT
19) RETURN_LIST_OF_TYPES -> ε
20) RETURN_LIST_OF_TYPES -> : TYPE_VALUE TYPE_NEXT
21) TYPE_NEXT -> , TYPE_VALUE TYPE_NEXT
22) TYPE_NEXT -> ε
23) STATEMENT -> value_id VALUE_ID_NEXT = INIT_VALUE INIT_VALUE_NEXT STATEMENT
24) STATEMENT -> local value_id : TYPE_VALUE INIT_LOCAL_VALUE STATEMENT
25) STATEMENT -> function_id(ENTRY_LIST_PARAMS) STATEMENT
26) STATEMENT -> if expression then STATEMENT STATE_ELSE end STATEMENT
27) STATEMENT -> return RETURN_LIST STATEMENT
28) STATEMENT -> while expression do STATEMENT end STATEMENT
29) STATEMENT -> ε
30) RETURN_LIST -> ε
31) RETURN_LIST -> function_id(ENTRY_LIST_PARAMS) RETURN_VALUE_NEXT
32) RETURN_LIST -> value_id RETURN_VALUE_NEXT
33) RETURN_LIST -> expression RETURN_VALUE_NEXT
34) RETURN_VALUE_NEXT -> , value_id RETURN_VALUE_NEXT
35) RETURN_VALUE_NEXT -> , expression RETURN_VALUE_NEXT
36) RETURN_VALUE_NEXT -> , function_id(ENTRY_LIST_PARAMS) RETURN_VALUE_NEXT
37) RETURN_VALUE_NEXT -> ε
38) STATE_ELSE -> else STATEMENT
39) STATE_ELSE -> ε
40) INIT_VALUE -> expression
41) INIT_VALUE -> function_id(ENTRY_LIST_PARAMS)
42) INIT_VALUE -> value_id

43) INIT_VALUE_NEXT -> ε
44) INIT_VALUE_NEXT -> , INIT_VALUE INIT_VALUE_NEXT
45) INIT_LOCAL_VALUE -> ε
46) INIT_LOCAL_VALUE -> = INIT_VALUE
47) VALUE_ID_NEXT -> ε
48) VALUE_ID_NEXT -> ,value_id VALUE_ID_NEXT
49) TYPE_VALUE -> string
50) TYPE_VALUE -> number
51) TYPE_VALUE -> integer
52) TYPE_VALUE -> nill

B LL – tabulka

	require	string	function	function_id	global	(	)	return	,	:	=	value_id	expression	local	if	then	while	do	else	string	number	integer	nill	end	eof
START	1																								
MAIN_LIST			2	3	4																				
MAIN_NEXT			5	5	5																				6
LIST_OF_PARAMS							7					8													
PARAM_NEXT							10		9																
ENTRY_LIST_PARAMS							12					11	11												
ENTRY_PARAM												14	13												
ENTRY_PARAM_NEXT							16		15																
LIST_OF_TYPES																				18	18	18	18		
RETURN_LIST_OF_TYPES			19	19	19			19		20		19		19	19		19							19	
TYPE_NEXT									21, 22																
STATEMENT				25				27				23		24	26		28							29	
RETURN_LIST			30	30, 31								30, 32	33	30	30		30							30	
RETURN_VALUE_NEXT			37	37					34, 35, 36			37		37	37		37							37	
STATE_ELSE																			38					39	
INIT_VALUE				41								42	40												
INIT_VALUE_NEXT			43	43					44			43		43	43		43							43	
INIT_LOCAL_VALUE			45	45							46	45		45	45		45							45	
VALUE_ID_NEXT									48		47														
TYPE_VALUE																				49	50	51	52		

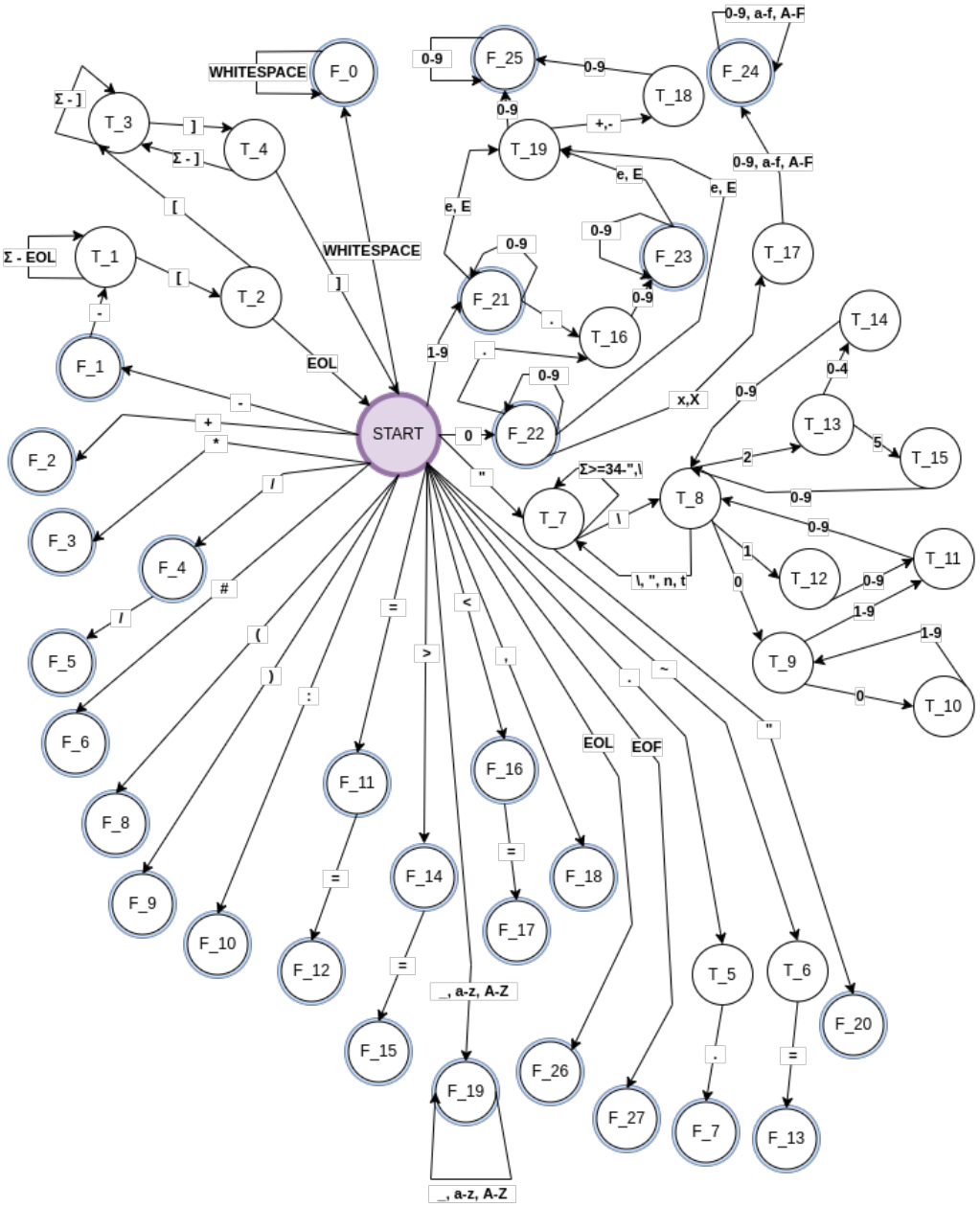
Tabulka 2: LL – tabulka použitá při syntaktické analýze

C Precedenční tabulka

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

Tabulka 3: Precedenční tabulka využívaná při precedenční syntaktické analýze výrazů

D Diagram konečného automatu popisující lexikální analyzátor



E Legenda diagramu konečného automatu popisující lexikální analyzátor

START	S_START	**T_1**	S_COMMENT0
F_0	S_SPACE	**T_2**	S_COMMENT1
F_1	S_SUB	**T_3**	S_COMMENT2
F_2	S_ADD	**T_4**	S_COMMENT3
F_3	S_MUL	**T_5**	S_DOT0
F_4	S_DIV	**T_6**	S_TILDA
F_5	S_IDIV	**T_7**	S_STR_START
F_6	S_STRLEN	**T_8**	S_STR_T1
F_7	S_DOT1	**T_9**	S_STR_T2
F_8	S_LEFT_PAR	**T_10**	S_STR_T3
F_9	S_RIGHT_PAR	**T_11**	S_STR_T4
F_10	S_DOUBLE_DOT	**T_12**	S_STR_T5
F_11	S_ASSIGN	**T_13**	S_STR_T6
F_12	S_EQL	**T_14**	S_STR_T7
F_13	S_NEQL	**T_15**	S_STR_T8
F_14	S_GT	**T_16**	S_FP_DOT
F_15	S_GTE	**T_17**	S_HEX0
F_16	S_LT	**T_18**	S_EXP_1
F_17	S_LTE	**T_19**	S_EXP0
F_18	S_COMMA		
F_19	S_ID		
F_20	S_STR_FIN		
F_21	S_INT		
F_22	S_INT0		
F_23	S_NUMBER		
F_24	S_HEX1		
F_25	S_EXP2		
F_26	S_EOL		
F_27	S_EOF		