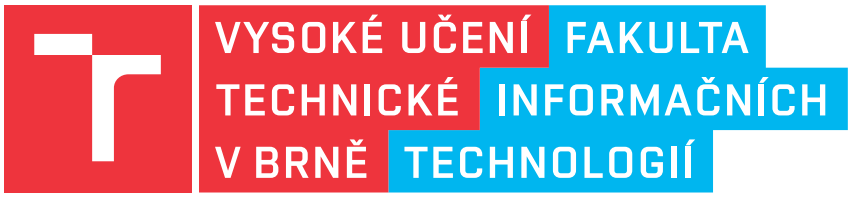
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ18

Tým 78, varianta II

18. listopadu 2019

Igor Mjasojedov	**(xmjaso00)**	25 %
Richard Borbély	(xborbe00)	25 %
Alex Sporni	(xsporn01)	25 %
Daniel Weis	(xweisd00)	25 %

Obsah

1 Úvod

Cieľom projektu, bolo vytvoriť program v jazyku C, ktorý má za úlohu načítať zdrojový kód, v zdrojovom jazyku IFJ18, ktorý je zjednodušenou podmnožinou jazyka Ruby 2.0 a preložiť ho do cieľového jazyka IFJcode18 (medzikód).

Zvolili sme si zadanie II, v ktorom bolo za úlohu implementovať tabuľku symbolov pomocou tabuľky s rozptýlenými položkami. Výsledný program funguje ako konzolová aplikácia, ktorá načíta riadiaci program v jazyku IFJ18 zo štandardného vstupu a bude generovať výsledný medzikód v IFJcode18 na štandardný výstup.

2 Implementácia

2.1 Lexikálna analýza

Pri tvorbe prekladača sme začali implementáciou lexikálnej analýzy. Lexikálna analýza je zostrojená implementáciou dopredu vytvoreného deterministického konečného automatu **??**, ktorý pozostáva z dočasných (pomocných) a koncových stavov. Na základe v ktorom stave skončí, určí typ tokenu.

Hlavnou časťou lexikálnej analýzy z ktorej vychádzame je funkcia `get_next_token`, ktorá spracováva vstup znak po znaku zo zdrojového súboru a prevádza lexémy na tokeny, ktoré sú ďalej spracovávané syntaktickou analýzou. Token sa skladá z typu a atribútu. Srdcom funkcie `get_next_token` je `switch`, kde každý jeden prípad `case` je reprezentovaný práve jedným stavom KA, ktorý sa nachádza v nekonečnom cykle. Komentáre a biele znaky sú lexikálnym analyzátorom ignorované. Pokiaľ sa načítaný znak nezhoduje so žiadnym znakom, ktorý jazyk IFJ18 povoluje, program sa ukončí a vráti prislúchajúcu návratovú hodnotu.

2.2 Syntaktická analýza

Implementácia prebiehala na základe predom vytvorenej LL–gramatiky s využitím dopočučenej metódy rekurzívneho zostupu. Ku každej sade pravidiel určitého neterminálu prislúchala určitá funkcia. Pre správne priebežné vyhodnocovanie jednotlivých pravidiel sa žiada o ďalší token modul SCANNER pomocou makra `GET_NEXT_TOKEN()`, ktoré nám v prípade chyby v lexikálnej analýze vráti príslušnú chybovú hodnotu. Počas analýzy sa nám priebežne volajú funkcie modulu GENERATOR podľa aktuálne spracovávanej konštrukcie jazyka.

2.2.1 Syntaktická analýza výrazov

Výrazy sme spracovávali s využití precedenčnej syntaktickej analýzy. Pred samotnou implementáciou sme potrebovali vytvoriť precedenčnú tabuľku, na základe ktorej sa analýza riadila. Taktiež aj pri tejto analýze sme volali funkcie generátoru, ktorých vykonanie nám zabezpečilo korektné vytvorenie jednotlivých inštrukcií pre prácu nad zásobníkom.

2.3 Sémantická analýza

Všetky akcie sémantickej analýzy sa priebežne vykonávali so syntaktickou analýzou a využívali informácie o jednotlivých funkciách a premenných, ktoré sa priebežne uchovávali a modifikovali v tabuľke symbolov.Na korektné vykonanie sémantickej analýzy sme si potrebovali vytvoriť dve tabuľky symbolov, globálnu a lokálnu.

V globálnej tabuľke sme uchovávali informácie o všetkých vytvorených funkciách vrátane preddefinovaných funkií a informácie o premenných vytvorených v hlavnom tele programu.

V lokálnej tabuľke symbolov sme uchovávali informácie o premenných vytvorených v tele definície danej funkcie a taktiež o parametroch danej funkcie. Z definície funkcie bola možnosť prístupu do globálnej tabuľky jedine k položkám, ktoré nám reprezentovali už vytvorené funkcie a v prípade snahy využitia premennej, ktorej informácie boli uchované jedine v globálnej tabuľke došlo k sémantickej chybe, nakoľko náš jazyk nepodporoval možnosť globálnych premenných.

Najväčšou obtiažnosťou sémantických resp. typových kontrôl sa nám zdala byť kontrola typov operandov v definícii funkcie pri situácii kedy daný operand resp. operandy boli parametrami funkcie. Obtiažnosť týchto kontrôl vyplývala zo samotnej konštrukcie jazyka, kde pri definícii funkcie sme nepoznali typy daných parametrov. Kvôli tejto vlastnosti jazyka sme boli nútený sémantické kontroly nad parametrami funkcie riešiť až na úrovni generovania cieľového kódu, a to spôsobom korektného generovania jednotlivých inštrukcií.

2.4 Generovanie cieľového kódu

Finálnou časťou celého projektu bola implementácia modulu GENERATOR, ktorý mal za úlohu generovať finálny trojadresný kód `.IFJcode18`. Funkcionalita modulu GENERATOR spočívala vo volaní jeho funkcií z modulov PARSER a PARSER_EXPRESSIONS v priebehu syntaktickej a sémantickej analýzy.

Generovaný kód sa priebežne zapisoval do dvoch dynamických reťazcov, ktoré nám slúžili na oddelený zápis definícii funkcii a kódu hlavného tela programu. K rozhodnutiu o výbere cieľového dynamického reťazca k zápisu aktuálne generovaného kódu nám slúžil indikátor aktuálnej pozície a to buď tela definície funkcie alebo tela hlavného programu.

Jednotlivé funkcie tohto modulu nám zabezpečovali generovanie správnych konštrukcií kódu pri typovej kontrole výrazu, podmienenom príkaze, príkaze cyklu, pri potrebe pretypovania číselného typu, pri volaní funkcie a taktiež správnu konštrukciu definície funkcie atď...

3 Algoritmy a dátové štruktúry

3.1 Tabuľka s rozpýlenými položkami

Tabuľku symbol sme vytvorili pomocou tabuľky s rozptýlenými položkami, keďže sme si zvolili práve túto variantu zadania. Ako veľkosť tabuľky sme si zvolili prvočíslo 12289 z dôvodu percentuálne nízkej šance zhlukovania jednotlivých položiek tabuľky.

Pri implementácii sme položku tabuľky reprezentovali štruktúrou `tHTItem`, ktorá obsahovala ukazateľ na ďalšiu položku, unikátny kľúč, ktorý nám taktiež reprezentoval názov funkcie alebo premennej. Ako poslednú súčasť štruktúry `tHTItem` sme si implementovali štruktúru `tData`, kde sme uchovávali nasledovné informácie: typ premennej/návratový typ funkcie, indikátor typu položky (funckia/premenná), indikátor overujúci predošlé definovanie funkcie, typy parametrov funkcie, reprezentované pomocou pola znakov, v ktorom každý jeden znak označuje typ jedného parametru.

Výber hashovacej funkcie sme uskutočnili na základe nasledujúcich nami uprednostňovaných faktorov: frekvencia kolízii, efektivita daného algoritmu a obtiažnosť implementácie. Po porovnaní rôznych typov hash funkcií sme rozhodli pre GNU ELF Hash.

3.2 Zásobník

Počas syntaktickej/sémantickej analýzy výrazov v module PARSER_EXPRESSIONS sme bezpodmienečne potrebovali funkcionalitu zásobníku, a preto sme museli vytvoriť aj túto dátovú štruktúru.

Základné funkcie nad štruktúrou `Item_Stack` sme implementovali na základe nadobudnutých vedomostí z prednášok predmetu IAL. Avšak pre potreby precedenčnej syntaktickej analýzy sme museli dodatočne implementovať špecializované funkcie vhodné na túto činnosť.

3.3 Dynamický reťazec

Pri implementácii sme narazili na komplikáciu práce s reťazcami v jazyku C, a preto sme si museli vytvoriť štruktúru `String_DYNAMIC` a prislúchajúce funkcie pre prácu s ňou.

V danej štruktúre sme uchovávali ukazateľ na reťazec, veľkosť aktuálneho reťazca a počet bajtov pridelenej pamäte pre aktuálny reťazec. V prípade rozširovania reťazca sme vlastnosť dynamickosti zabezpečili porovnávaním aktuálnej veľkosti reťazca a počtu bajtov už pridelenej pamäte, kde sme v prípade nedostatočného množstva voľnej pridelenej pamäte pristúpili k jej rozšíreniu.

4 Práca v týme

4.1 Príprava, plán a spôsob práce

Keďže sme si uvedomovali náročnosť a rozsiahlosť daného projektu, rozhodli sme sa začať čo najskôr. Prvotná príprava a plány započali na úvodných prednáškach predmetu IFJ a IAL. Prácu sme si delili postupne a rovnomerne, keďže sme zo začiatku nemali jasnú predstavu finálneho konceptu a implementácie, neskôr sme si jednotlivé časti rozdelili podľa schopností jednotlivca.

4.2 Komunikácia

Komunikácia prebiehala najmä osobne, zo začiatku sme sa dohodli na pravidelných stretnutiach vo fakultnej knižnici, menšie pripomienky a otázky sme diskutovali pomocou sociálnych sietí.

4.3 Verzovací systém

Pre správu verzií súborov projektu sme používali verzovací systém Git. Hlavným dôvodom bola kompatibilita s vývojovým prostredím CLion. Ako vzdialený repozitár nám poslúžil GitHub, ktorý nám ponúkol okrem

privátneho repozitára možnosť sledovať prácu jednotlivca a graf priebežného vývoja. Väčšinu práce na jednotlivých častiach projektu sme riešili na oddelených vetvách aby sa zamedzidlo prípadným konfliktom pri správe aktuálnych verzií. Po schválení vedúcim týmu sme jednotlivé vetvy zlúčili do hlavnej vývojovej vetvy.

4.4 Prekladový systém

Preklad nášho projektu sme zabezpečovali pomocou nástroja CMake alebo GNU Make.

4.4.1 GNU Make

V zadaní sme mali špecifikovanú požiadavku na pridanie súboru `Makefile` do odovzdávaného archívu, na základe ktorej sme museli zabezpečiť funkcionalitu prekladu aj cez tento nástroj.

Nami vytvorený súbor `Makefile` disponoval nielen základnou funkcionalitou prekladu zdrojových súborov – `make`, ale aj doplňujúcimi príkazmy na vytvorenie `.zip` archívu – `make zip`, vytvorenie `tar.gz` archívu – `make pack` a taktiež na vymazanie prekladom vytvorených objektových a binárnych súborov – `make clean`.

4.4.2 CMake

Nakoľko sme celý projekt vyvájali hlavne vo vývojovom prostredí CLion, ktorý má už v sebe štandardne zabudovaný nástroj CMake, sa nám zjednodušila práca s prekladom celého projektu nakoľko sa všetky novo pridané súbory automaticky pridali do súboru `CMakeLists.txt`, kde boli taktiež nastavené všetky pravidlá pre preklad.

4.5 Rozdelenie práce

Prácu na projekte sme si rozdelili rovným dielom, Náplň práce jednotlivých členov je popísana v tabuľke **??**

Člen týmu	Pridelená práca
Igor Mjasojedov	Implementácia syntaktickej analýzy a sémantickej analýzy (modul parser), revízia kódu, vedenie týmu, organizácia práce, štruktúa projektu
Alex Sporni	Implementácia lexikálnej analýzy (modul scanner), vytvorenie dokumentácie, vytvorenie prezentácie
Richard Borbély	Generovanie cieľového kódu, testovanie
Daniel Weis	Implementácia dátových štruktúr (tabuľka s rozptýlenými položkami), zásobník, testovanie

Tabulka 1: Rozdelenie práce v týme

5 Záver

K riešeniu projektu sme pristupovali zodpovedne a systematicky, samotný rozsah projektu a náročnosť boli pre nás výzvou, s akou sme sa na iných predmetoch doposial nestretli. Projekt nám okrem mnohých praktických znalostí, pomohol zlepšiť komunikáciu v rámci skupiny a lepšie formulovať myšlienky pri pripomienkach.

Pri implementácii sme sa stretli s komplikáciami ohladne typovej kontroly pri behu programu. Nejasnosti, ktoré vyplývali zo zadania sme riešili na diskusnom fóre určenom pre projekt, kde sa daná problematika väčšinou už rozoberala. Správnosť našich riešení sme si pravidelne overovali pomocou automatických testov, ktoré sme si vytvorili a taktiež pokusnými odovzdaniami, ktoré nám pomohli odhaliť nedostatky jednotlivých modulov projektu.

6 Pouzitá literatúra

7 Príloha

A LL Gramatika

<program>	→	EOF
	→	EOL <program>
	→	def <def> EOL <program>
	→	<statement> EOL <program>
<def>	→	id (<param_list>) EOL <state_list> end
<state_list>	→	<statement> EOL <state_list>
	→	ε
	→	EOL <state_list>
<statement>	→	if <expression>then EOL <state_list> else EOL <state_list> end
	→	while <expression> do EOL <state_list> end
	→	id = <id_assign>
	→	<fnc_call>
	→	<expression>
<id_assign>	→	<fnc_call>
	→	<expression>
<param_list>	→	id <param_next>
	→	ε
<param_next>	→	, id <param_next>
	→	ε
<fnc_call>	→	id <argument_list>
	→	inputs <argument_list>
	→	inputf <argument_list>
	→	inputi <argument_list>
	→	print <argument_list>
	→	length <argument_list>
	→	substr <argument_list>
	→	ord <argument_list>
	→	chr <argument_list>
<argument_list>	→	(<arguments>)
	→	<arguments>
<arguments>	→	<term> <argument_next>
	→	ε
<argument_next>	→	, <term> <argument_next>
	→	ε
<term>	→	id
	→	INTEGER_VALUE
	→	FLOAT_VALUE
	→	STRING_VALUE

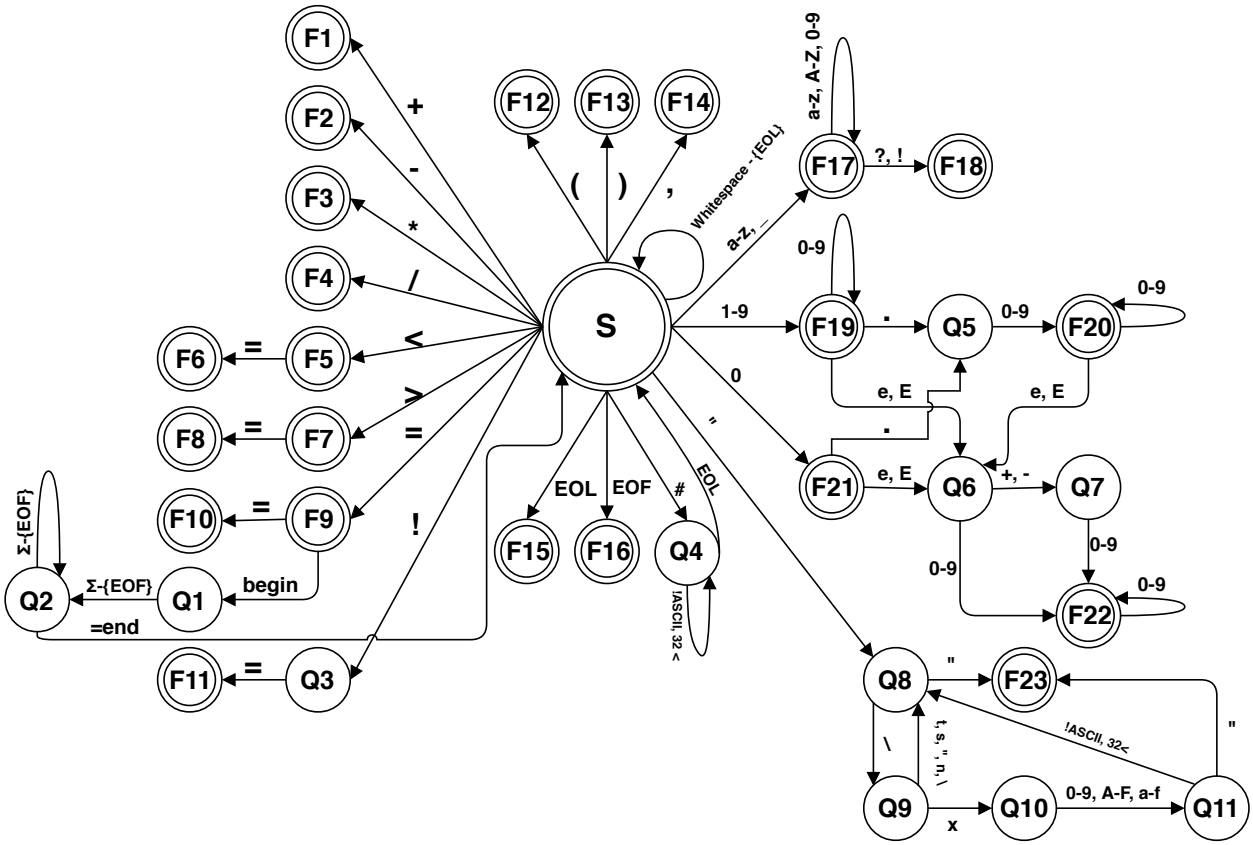
Tabulka 2: LL – gramatika riadiaca syntaktickú analýzu

B Precedenčná tabuľka

	+	**-**	*****	**/**	$<$	$\leq$	$>$	$\geq$	**==**	**!=**	**(**	**)**	**ID**	**$**
+	$>$	$>$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
-	$>$	$>$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
*****	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
/	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
$<$	$<$	$<$	$<$	$<$	$<$				$>$	$>$	$<$	$>$	$<$	$>$
$\leq$	$<$	$<$	$<$	$<$					$>$	$>$	$<$	$>$	$<$	$>$
$>$	$<$	$<$	$<$	$<$					$>$	$>$	$<$	$>$	$<$	$>$
$\geq$	$<$	$<$	$<$	$<$					$>$	$>$	$<$	$>$	$<$	$>$
==	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$			$<$	$>$	$<$	$>$
!=	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$			$<$	$>$	$<$	$>$
(	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	=	$<$	
)	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$		$>$		$>$
ID	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$		$>$		$>$
$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$		$<$	

Tabulka 3: Precedenčná tabuľka použitá pri precedenčnej syntaktickej analýze výrazov

C Diagram konečného automatu pre lexikálny analyzátor



Obrázek 1: Diagram konečného automatu pre lexikálny analyzátor

Legenda:

S	START	**F12**	LEFT_BRACKET	**Q1**	BLOCK_COMMENTARY
F1	PLUS	**F13**	RIGHT_BRACKET	**Q2**	COMMENT_IGNORE
F2	MINUS	**F14**	COMMA	**Q3**	NOT
F3	MUL	**F15**	EOL	**Q4**	LINE_COMMENTARY
F4	DIV	**F16**	EOF	**Q5**	FLOATING_NUMBER
F5	LESS	**F17**	ID_KEY	**Q6**	E_NUMBER
F6	LESS_EQUAL	**F18**	FUNCTION_ID	**Q7**	PLUS_MINUS
F7	GREATER	**F19**	INTEGER_NUMBER	**Q8**	STRING
F8	GREATER_EQUAL	**F20**	FINAL_FLOAT_NUMBER_1	**Q9**	STRING_ESCAPE
F9	ASSIGN	**F21**	ZERO_NUMBER	**Q10**	STRING_HEXA
F10	EQUAL	**F22**	FINAL_FLOAT_NUMBER_2	**Q11**	STRING_HEXA_ADVANCED
F11	NOT EQUAL	**F23**	STRING_F		

D LL – tabulka

Tabulka 4: LL – tabuľka