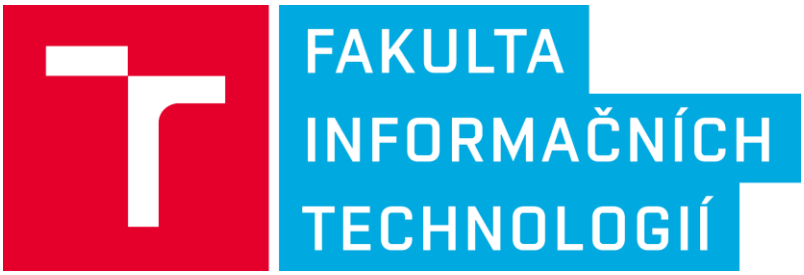


Dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ18

Tím 023, variant I

Michal Vanka (xvanka00)

Daniel Miloslav Očenáš (xocena06)

Šárka Ščavnická (xscavn01)

Romana Džubarová (xdzuba00)

1.decembra 2018

Obsah

1 Úvod 3
2 Návrh a implementácia 3
2.1 Lexikálna analýza 3
2.2 Tabuľka symbolov 3
2.3 Syntaktická analýza 4
2.3.1 Precedenčná syntaktická analýza 4
2.4 Sémantická analýza 5
2.5 Generovanie kódu 5
3 Práca v tíme 5
3.1 Verzovací systém 5
3.2 Rozdelenie práce v tíme 5
4 Záver 6
5 Prílohy 7

1 Úvod

Cieľom projektu bolo vytvorenie programu v jazyku C, ktorý načíta zdrojový kód zapísaný v jazyku IFJ18, ktorý je zjednodušenou podmnožinou jazyka Ruby 2.0 a preloží ho do cieľového medzikódu v jazyku IFJcode18.

Prekladač načíta riadiaci program v jazyku IFJ18 zo zdrojového súboru zadaného v argumente pri spúšťaní prekladaču a zo štandardného vstupu prijíma argumenty pre preklad zdrojového súboru. Následne prekladač generuje výsledný medzikód na štandardný výstup. V prípade výskytu chyby pri preklade zdrojového súboru, sa všetky chybové hlásenia vykonajú na štandardný chybový výstup.

2 Návrh a implementácia

Táto kapitola obsahuje informácie o jednotlivých častiach nášho prekladača a predávaní informácií medzi jednotlivými časťami. Ďalej v tejto kapitole popisujeme použité dátové štruktúry a tabuľku symbolov.

2.1 Lexikálna analýza

Lexikálnu analýzu sme vypracovali na základe navrhnutého deterministického konečného automatu. Hlavnú časť lexikálnej analýzy predstavuje funkcia **next_token,** v ktorej je implementovaný konečný automat, prijímajúci symboly zo zdrojového kódu. Vzhľadom na aktuálny stav a vstupný symbol rozhoduje o zmene stavov podľa ktorých premieňa zdrojový kód na tokeny. Konečný automat akumuluje identifikátory, čísla, operátory, koncové symboly a vynecháva okomentované časti zdrojového kódu.

V prvej verzii lexikálneho analyzátora sme na načítanie symbolov použili dátovú konštrukciu queue, ktorej sme sa venovali v predmete IAL. Použiť túto konštrukciu sme sa rozhodli kvôli tomu, že sme ihneď mohli začať pracovať na funkčnosti konečného automatu. Avšak od začiatku sme vedeli, že táto konštrukcia nie je vhodná pre náš projekt, práve preto, že môže načítať iba obmedzené množstvo symbolov ktoré musí byť vopred stanovené. Po dokončení implementácie konečného automatu, sme túto štruktúru obmenili tak, aby sa alokovala dynamicky. Jej veľkosť sa vždy pri načítaní vstupného symbolu a pri potrebe pridania do reťazca (načítanie identifikátoru, čísla a reťazca) realokuje na veľkosť väčšiu o jeden bajt.

2.2 Tabuľka symbolov

Tabuľka symbolov (varianta č.1 – binárny strom) bola navrhnutá s ohľadom na fakt, že jazyk IFJ18 je dynamicky typovaný, a teda v nej neuchovávame hodnotu premenných, iba ich meno, fakt, či ide o premennú alebo funkciu, a či bola daná premenná deklarovaná. Čo sa týka funkcií, ich položka v tabuľke obsahuje zároveň aj ukazateľ na ich vlastný binárny strom, ktorý je ich lokálnou tabuľkou symbolov. Navrhovanie tabuľky zabralo viacej času, než sa očakávalo, a to najmä kvôli dynamickej typovanosti jazyka, kvôli ktorej dochádzalo k opätovných snahám o uchovanie hodnoty premenných.

2.3 Syntaktická analýza

Vypracovanie syntaktickej analýzy sme začali návrhom LL – gramatiky[1] bez epsilon prechodov, ďalej len ε prechodov. Na vytvorenú LL - gramatiku sa snažili oponovať všetci členovia tímu aby sa predišlo chybám implementácie. Rozhodli sme sa zostať pri gramatike bez ε prechodov po celý čas aj napriek tomu, že tím bol niekoľko krát zvádzaný ε prechody do gramatiky zaviesť. Pre overenie LL - gramatiky, sme vytvorili LL – tabuľku[2], ktorá pomohla odhaliť nedostatky. Zachovanie LL - gramatiky bez ε prechodov sa nám nakoniec vyplatilo a zaručilo konečnosť algoritmu. Vzhľadom na blížiaci sa termín odovzdania sme po dokončení lexikálnej analýzy vymysleli štruktúru pre syntaktickú analýzu rekurzívneho zostupu zhora dole na základe stavového automatu.

V projekte sme používali znalosti z predmetu IAL. Pre syntaktickú analýzu sme implementovali obojsmerne viazaný zoznam pre prácu s tokenmi prijatými z lexikálnej analýzy. Rekurzívny zostup zhora dole v našej implementácii pracuje na princípe porovnávania aktívnej položky v liste s očakávanými pravidlami definovanými v LL – gramatike. V prípade nezhody vo vnútri pravidla je potrebné vrátiť sa na token na začiatku pravidla a skúmať zhodu so zvyšnými pravidlami. Keďže obojsmerne viazaný list umožňuje návrat k predošlým tokenom cez jednotlivé položky zoznamu, ktoré uchovávajú informácie o tokene a ukazovatele na predošlú a nasledujúcu položku zoznamu, bolo jeho použitie v našej implementácii najvhodnejšie. Ak sa token zhoduje s očakávaným tokenom, syntaktická analýza vyžaduje ďalší token. Pre volanie tokenu zo zoznamu bola vytvorená funkcia **get_token**, ktorá posúva aktivitu na nasledujúci prvok listu. V prípade, že položka za aktívnou položkou je nedefinovaná, funkcia get_token požiada pomocou funkcie next_token o ďalší token z lexikálnej analýzy.

Počas testovania rekurzívneho zostupu pravidiel sme boli nútení opätovne kontrolovať gramatiku a odhalili sme jej ďalšie nedostatky ale aj mierne nedostatky vo vytvorenom kóde spôsobené ľudskou nepozornosťou. Najväčšie komplikácie pri vypracovaní syntaktickej analýzy sa vytvorili práve pri testovaní navrhnutej LL - gramatiky, pričom sme sa spoliehali na jej správnosť a hľadali chybu v navrhnutom algoritme. Testovaním sa ukázalo, že algoritmus spracovávania syntaktickej analýzy bol navrhnutý správne a ako sa už spomenulo, odhalili a upravili sme chyby v LL – gramatike.

2.3.1 Precedenčná syntaktická analýza

Precedenčná syntaktická analýza nám slúži na spracovanie výrazov, ale aj viet. Implementovaná je v súbore psa.c a jej knižnica je v psa.h.

Precedenčnú syntaktickú analýzu sme začali vytvorením precedenčnej tabuľky. Postupovali sme podľa návodu v ôsmej prezentácii IFJ viz. príloha č.. Precedenčnú tabuľku sme v jazyku C implementovali ako dvojrozmerné pole. Pre relačné (< <= > >=) a porovnávacie operátory (== !=) je v tabuľke rovnaký riadok a aj stĺpec, keďže majú rovnakú prioritu a asociativitu. Pre symboly, ktoré symbolizujú čísla či už číselnou hodnotou alebo identifikátorom, je taktiež vytvorený spoločný riadok a stĺpec. Na transformáciu tokenu na symbol sme použili funkciu **SYMBOL_FROM_TOKEN** a následne funkciu

[1] Príloha č.3
[2] Príloha č. 4

PRECEDENCE_TABLE_INDEX pre vrátenie indexu, ktorý sa nachádza v našej precedenčnej tabuľke.

Algoritmus finálnej PSA sme vytvorili na základe algoritmu, ktorý je popísaný v ôsmej prezentácii predmetu IFJ. Syntaktická analýza uloží tokeny do zoznamu, pričom z tohoto zoznamu si následne berie precedenčná syntaktická analýza tokeny a spracováva ich na základe spomínanej precedenčnej tabuľky. Precedenčná analýza končí v prípade, ak za výrazom nasleduje token, ktorý symbolizuje koniec výrazu (then, do, EOL ...) a na stacku, v ktorom sme si redukovali výraz na pravidlá ostane iba symbol $. Následne sa korektne uvoľní pamäť a precedenčná analýza vráti syntaktickej analýze počet tokenov, ktoré spracovala.

2.4 Sémantická analýza

Túto časť projektu sme plánovali uskutočniť spolu s generovaním kódu. Typová kontrola mala byť vykonávaná priamo v IFJcode18, kde by sa pred vykonaním každého príkazu vyžadujúceho 2 operandy s rovnakým dátovým typom vykonala kontrola ich typov pomocou príkazu **TYPE** a následného porovnania výsledkov. Z dôvodu zamerania sa na správne fungovanie syntaktickej analýzy však nebola implementovaná.

2.5 Generovanie kódu

Pri generovaní kódu sme zvolili možnosť okamžitého vypisovania vygenerovaného medzikódu IFJcode18 na štandardný výstup. Do generácie sme implementovali aj čiastočnú sémantickú analýzu a to kontrolu typov. Generáciu výsledného kódu, sa nám ale nepodarilo implementovať do celkového projektu z viacerých dôvodov.

3 Práca v tíme

Prácu na projekte sme zahájili spoločným stretnutím po zverejnení zadania projektu. Rozanalyzovali sme jednotlivé súčasti projektu a spoločne začali s návrhom deterministického konečného automatu pre lexikálnu analýzu. Vedúci tímu navrhol pre účely vypracovania projektu využiť verzovací systém GitHub, viac v odseku 4.1. Členom tímu navrhol oboznámiť sa so zadaním projektu a zopakovať si znalosti získané v prednáškach. Následne sa jednotlivé súčasti projektu sme rozdelili medzi členov tímu a začali sme s vypracovaním. Pravidelne sme medzi sebou diskutovali aby projekt čo najviac vyhovoval zadanej špecifikácii. Ukázalo sa, že vedúci tímu sa stal značnou oporou pre ostatných členov tímu. Napriek včasnému začatiu prác sme sa ocitli v časovom sklze. Časový sklz sme sa snažili dobehnúť dlhými stretnutiami, na ktorých sme spoločne navrhovali rozšírenie funkčnosti projektu a následnú implementáciu.

3.1 Verzovací systém

Pri práci na tomto projekte sme využili verzovací systém GitHub a užívateľský klient GitKraken. Tento systém nám uľahčil zdieľanie súborov medzi jednotlivými členmi tímu. Zároveň zabezpečil pre členov prístup k aktuálnym verziám súborov ich tímových kolegov.

Veľmi nám pomohla utilita Diff View klientu GitKraken, vďaka ktorej sme po zmenení súboru mohli nahliadnuť iba na zmeny vykonané voči predošlej verzii súboru.

3.2 Rozdelenie práce v tíme

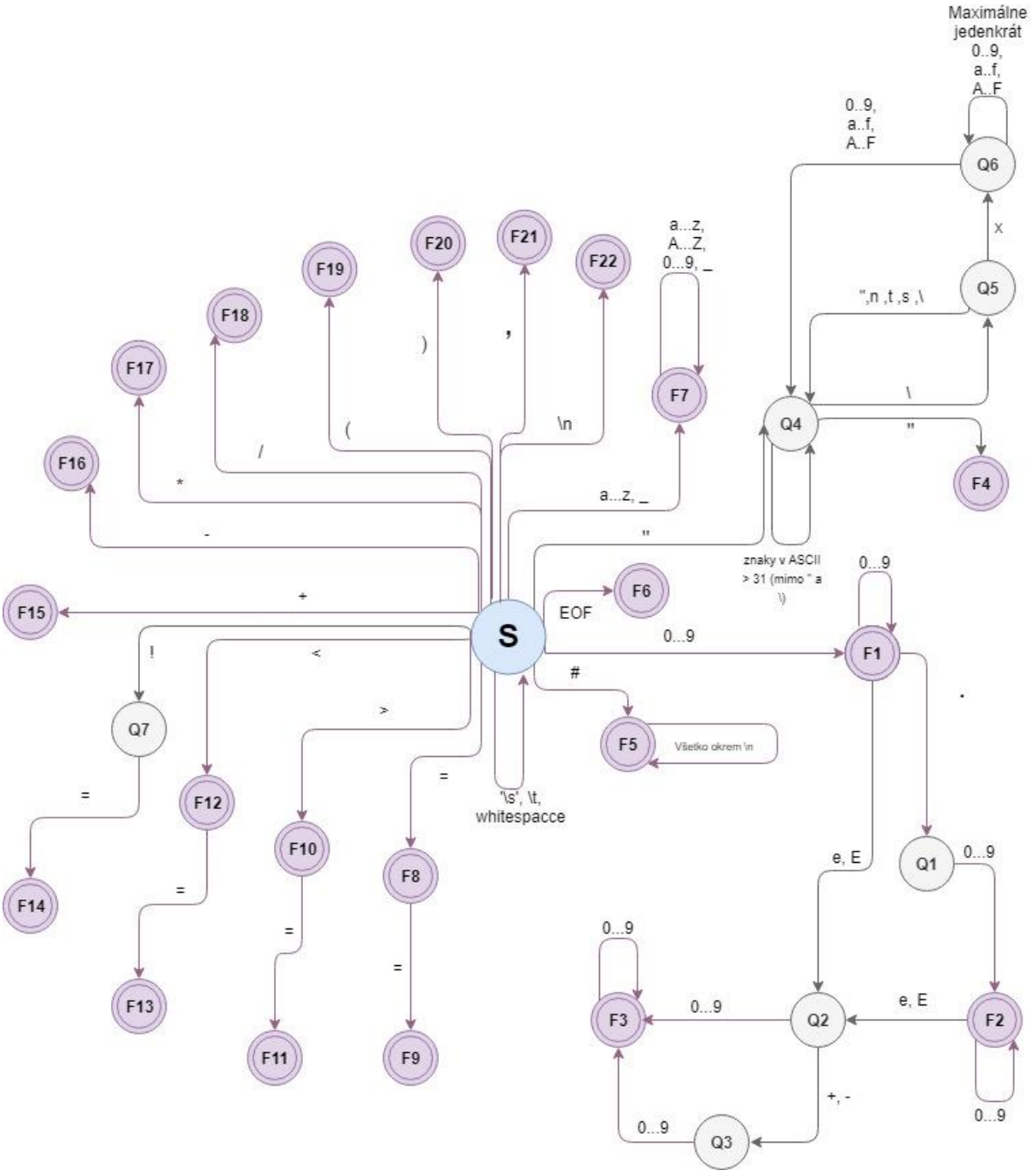
Prácu sme sa snažili rozdeliť jednotlivým ľuďom v rámci ich možností. Na ťažších úrovniach sme preto pracovali v dvojiciach niekedy aj v trojiciach.

Meno	Práca
Michal Vanka	Tabuľka symbolov, LL – gramatika, precedenčná syntaktická analýza, generácia strojového kódu, všeobecná pomoc.
Daniel Miloslav Očenáš	Konečný automat, lexikálna analýza, LL – gramatika, syntaktická analýza.
Šárka Ščavnická	Konečný automat, generácia strojového kódu.
Romana Džubarová	LL – tabuľka, precedenčná syntaktická analýza

4 Záver

Napriek tomu, že sa nám nepodarilo dokončiť všetky časti projektu, považujeme dokončenú časť za úspech. Náš prekladač dokáže správne deliť kód na lexémy, podľa navrhnutej gramatiky skontrolovať syntaktickú správnosť kódu, a zároveň vyhodnocovať výrazy pomocou precedenčnej tabuľky. Vďaka tomuto projektu sme sa naučili skutočne veľa o tímovej práci, pričom nás obohatil aj o znalosť fungovania kompilátorov.

5 Prílohy



Príloha č.1: Konečný automat

S	DA_START
Q1	DA_NUMBER Float Point
Q2	DA_NUMBER Exponential Sign
Q3	DA_NUMBER Exponent
Q4	DA_STRING Loop

Q5	DA_STRING
Q6	DA_STRING Start of Hexadecimal
F1	DA_NUMBER
F2	DA_NUMBER Float Final
F3	DA_NUMBER Exponential Final
F4	DA_STRING String Final
F5	DA_LINECOMM
F6	EOF
F7	DA_IDENTIFICATOR
F8	DA_EQUAL
F9	DA_OPERATOR Assignment
F10	DA_OPERATOR More
F11	DA_OPERATOR More or Equal
F12	DA_OPERATOR Less
F13	DA_OPERATOR Less or Equal
Q7	Exclamation
F14	DA_OPERATOR Not Equal
F15	DA_OPERATOR Plus
F16	DA_OPERATOR Minus
F17	DA_OPERATOR Times
F18	DA_OPERATOR Division
F19	DA_OPERATOR Left Bracket
F20	DA_OPERATOR Right Bracket
F21	DA_OPERATOR Comma
F22	DA_NEWLINE

Príloha č. 2: Popis stavov konečného automatu

1. <prog> -> <stat> <prog>
2. <prog> -> EOF
3. <stat> -> EOL
4. <stat> -> def id (<it-list1> EOL <do>
5. <stat> -> if <expression> then EOL <then>
6. <stat> -> while <expression> do EOL <do>
7. <stat> -> print <op-term1>
8. <stat> -> id=<asg>
9. <stat> -> <asg>
10. <then> -> <stat> <then>
11. <then> -> else EOL <else>
12. <else> -> <stat> <else>
13. <else> -> end EOL
14. <do> -> <stat> <do>
15. <do> -> end EOL
16. <func> -> length (<it-list1>
17. <func> -> substr (<it-list1>
18. <func> -> inputs ( )
19. <func> -> inputi ( )
20. <func> -> inputf ( )
21. <func> -> ord (<it-list1>
22. <func> -> chr (<it-list1>
23. <op-term1> -> (<it-list1> EOL
24. <op-term1> -> <term> <op-term2>
25. <op-term2> -> , <term> <op-term2>
26. <op-term2> -> EOL
27. <it-list1> -> <term> <it-list2>
28. <it-list1> ->)
29. <it-list2> -> ,<term> <it-list2>
30. <it-list2> ->)
31. <asg> -> <expression> EOL
32. <asg> -> <func> EOL
33 <asg> -> id <op-term1>
34. <term> -> id
35. <term> -> int
36. <term> -> double
37. <term> -> string
38. <term> -> nil

Príloha č.3: LL – gramatika

	EOF	EOL	def	if	while	print	id =	id	else	end	length	substr	inputs	inputi	inputf	ord	chr	(	,	)	int	double	string	nil
<prog>	2	1	1	1	1	1	1				1	1	1	1	1	1	1							
<stat>		3	4	5	6	7	8	9			9	9	9	9	9	9	9							
<then>		10	10	10	10	10	10	10	11		10	10	10	10	10	10	10							
<else>		12	12	12	12	12	12	12		13	12	12	12	12	12	12	12							
<do>		14	14	14	14	14	14	14		15	14	14	14	14	14	14	14							
<func>											16	17	18	19	20	21	22							
<op-term1>								24										23			24	24	24	24
<op-term2>		26																	25					
<it-list1>								27												28	27	27	27	27
<it-list2>																			29	30				
<asg>								33			32	32	32	32	32	32	32							
<term>								34													35	36	37	38

Príloha č.4: LL – tabuľka

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

Príloha č.5: Precedenčná tabuľka