Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ18

Tím 126, varianta II

Pavel Podlužanský (xpodlu01)

Matúš Škúta (xskuta04)

Jaromír Hradil (xhradi15)

Adam Richter (xpodlu01)

30. novembra 2018

Obsah

1. Úvod

Cieľom projektu bolo vytvoriť program v jazyku C, ktorý načíta zdrojový kód jazyka IFJ18, ktorý je podmnožinou jazyka Ruby a preloží ho do cieľového jazyka IFJcode18.

Program funguje ako konzolová aplikácia, ktorá načítava zdrojový program zo štandardného vstupu a generuje výsledný medzikód (IFJcode18) na štandardný výstup, prípadne vracia chybové hlásenie na štandardný chybový výstup.

2. Návrh a implementácia

Projekt je zostavený z častí, ktoré sú popísané v tejto kapitole.

2.1 Lexikálna analýza

Na začiatok našej práce sme začali implementovať lexikálnu analýzu. Najprv sme si spísali, lexikálnu gramatiku, ktorú sme využili pri tvorbe a identifikácii lexémov. Vďaka tejto gramatike sme boli schopný aj vytvoriť nákres deterministického konečného automatu, na základe ktorého sme implementovali lexikálnu analýzu.

Hlavnou funkciou tejto analýzy je **getToken.** Je to funkcia vracajúca identifikátor tokenu a premenná token obsahuje hodnotu tokenu. Funkcia načítava zo štandardného vstupu znak po znaku, a ak je charakter platný, tak ho pripojí (appendne) do tokenu. Pri nájdení lexikálnej chyby vracia **getToken** hodnotu -1.

Celý lexikálny analyzátor je implementovaný ako deterministický konečný automat podľa diagramu. Tento diagram je v prílohách **obr.1**.

2.2 Syntaktická analýza

Na počiatku sme si vytvorili LL- gramatiku, na základe ktorej sme začali vytvárať syntax. Veľmi nápomocná bola ukážka kódu "jednoduchý interpret" v súboroch k predmetu. Pri syntaktickej analýze sme mali najväčší problém pri spracovaní výrazov, kvôli tomu, že bolo pre nás náročné zistiť, ako daný výraz identifikovať.

Vybrali sme si na implementáciu pomocou LL- gramatiky formu rekurzívneho zostupu. LL-gramatika je popísaná v prílohách **obr.2**, okrem výrazov, kde pre výrazy používame precedenčnú syntaktickú analýzu. Syntaktická analýza si vypýta tokeny pomocou funkcie **getToken**, ktoré ukladá na stack, ktorý používame na komunikáciu medzi precedenčnou analýzou a syntaktickou analýzou. Aby ostatné parsovacie funkcie vedeli, že sa nachádzame v definícii funkcie, máme vytvorenú globálnu premennú **currFunction**, ktorá obsahuje názov funkcie, inak je prázdna, pokiaľ nie sme vo funkcii. Volania nedefinovaných funkcií pushujeme do stacku. Po prejdení celého programu syntaktickou analýzou, skontrolujeme či všetky volané funkcie na stacku sú definované.

2.2.1 Spracovanie výrazov pomocou precedenčnej analýzy

Pre spracovanie výrazov sa používa precedenčná tabuľka operátorov, popísaná v prílohách **obr.4.** Kedy každý symbol v tabuľke predstavuje index, podľa ktorého sa v tabuľke vyhľadáva, ktorá operácia sa má previesť v danom kroku. K tomu využívame funkciu

operatorTypeAssignment, ktorá priradí podľa typu tokenu z vrcholu stacku a podľa typu tokenu načítaného zo vstupu indexy v tabuľke. Ako indexy tabuľky sa berú klasické binárne operátory(+,-,*,/), typ ID, ktorý reprezentuje prvky, ktoré sa môžu nachádzať medzi operátormi, zátvorky, správne keywordy, ohraničujúce koniec výrazu, označené ako $ a relačné operátory.

Po spustení precedenčnej syntaktickej analýzy, sa hlavná funkcia **precedenceAnalysisParsing** rozhodne podľa stavu, ktorý jej bol predaný syntaktickou analýzou, ako správne zredukovať výraz. V niektorých prípadoch sú načítané tokeny poslané ako parameter cez stack. Potom sa podľa terminálu nachádzajúceho sa na vrchole stacku a aktuálne načítaného tokenu, ktorý je získavaný pomocou funkcie **getToken**, prevádza operácia podľa precedenčnej tabuľky. V prípade symbolu < sa aktuálne načítaný token uloží na vrchol stacku a znovu sa pomocou funkcie **getToken** načíta ďalší. V prípade symbolu > sa skontroluje, či existuje pravidlo pre redukciu a ak áno, tak sa prevedie daná redukcia. Táto redukcia je uložená do štruktúry abstraktného derivačného stromu, ktorý je v prípade úspechu analýzy predaný späť syntaktickej analýze. Pokiaľ je symbol =, tak sa volá funkcia **getToken** a podľa toho, aký je aktuálne načítaný token, sa rozhodne či sa dajú zátvorky zredukovať. A v prípade symbolu ! sa ukončí precedenčná analýza neúspechom. To celé opakuje, až do chvíle, kedy nie je na vrchole stacku **$** ako jediný terminál a pokiaľ je aj aktuálne načítaný token $, ktorý je následne porovnaný podľa stavu, v ktorom je funkcia spustená, vyhodnotený ako správny, bola precedenčná syntaktická analýza úspešná.

Problémom pri implementácii bolo implementovanie zátvoriek, ktoré nám zabralo viac času. Postupovali sme podľa prednášok, z ktorých sme si ujasnili, ako funguje algoritmus pre precedenčnú analýzu.

2.3 Sémantická analýza

Sémantická kontrola je spojená z generovaním kódu, kde popri sémantických kontrolách sa robia aj typové kontroly, popri ktorých sa snažíme optimalizovať kód. Pri sémantickej analýze bolo naším hlavným problémom aké typy errorov vracať.

Na kontrolu sémantiky využívame **hashtable**, v ktorej máme uložené dáta o premenných a funkciách. Sémantická kontrola premenných spočíva v tom, či sú definované a inicializované, čo riešime vo funkcií **semCheckVar**. Sémantická kontrola funkcií spočíva tiež v kontrole definovania a inicializovania, a navyše obsahuje kontrolu počtu premenných. Ďalej máme semantickú kontrolu rezervovaných funkcií, kde kontrolujeme typ premennej, práve keď sa jedná o konštantu.

2.4 Generovanie kódu

Nakoľko je jazyk IFJ18 dynamickým jazykom, tak najväčšou výzvou bolo vymyslieť krátke typové kontroly pre jednotlivé aritmetické, alebo binárne operátory. Ďalej sme museli vymyslieť, ako vytvoriť rezervované funkcie a najväčšou prekážkou bol presun definícií premenných zo zanorených cyklov.

Pri generovaní kódu generujeme pseudokód, ktorý máme uložený v zreťazenom zozname, čo nám značne zjednodušuje prácu pri presune definícii premenných ako pri while tak isto aj pri if podmienke. Ďalej tam prebieha transformácia konštánt do cieľového kódu IFJcode18.

2.4.1 Generovanie výrazov

Pri generovaní výrazov využívame abstraktný derivačný strom, ktorý obsahuje výraz rozložený pomocou precedenčnej analýzy. Pri generovaní výrazov sa strom postupne skladá, čo nám uľahčuje generovanie výrazov, nakoľko sme to mohli vytvoriť pri použití while príkazu obsahujúceho switch.

2.4.2 Generovanie volania funkcií

Volanie funkcií je implementované, tým istým spôsobom pre všetky funkcie, okrem funkcie print, ktorá parametre dostane v stacku a počet parametrov dostane ako input do funkcie.

Generovanie funkcií využíva abstraktnú štruktúru stack, na ktorej je uložený názov a parametre funkcie, keď je potreba obsahuje aj premennú do ktorej sa uloží návratová hodnota funkcií. Túto metódu sme zvolili, z dôvodu, že nám uľahčil veľa práce.

2.4.3 Generovanie definície funkcií

Je tak isto riešené cez abstraktnú štruktúru stack, ktorý obsahuje názov funkcie, spolu s potrebnými parametrami. Funkcia je vložená do kódu a obsahuje jump, ktorý ju preskočí aby definície funkcií neovplyvnilo executeovanie inštrukcií.

2.4.4 Generovanie príkazu while

Pri generovaní príkazu while sa vygeneruje podmienkový výraz, ktorého pravdivostná hodnota rozhodne o jeho ukončení, alebo pokračovaní v ňom. Po vykonaní poslednej inštrukcie v tele while-u skočíme pred vygenerovaný podmienkový výraz a opakujeme kontrolu.

2.4.5 Generovanie príkazu if

Ako pri generovaní while-u, takisto aj pri if príkaze sa vygeneruje podmienkový výraz, ktorý rozhodne o tom, ktoré inštrukcie v ife sa vykonajú.

3. Špeciálne algoritmy a dátove štruktúry

3.1 Abstraktný derivačný strom

Vybrali sme si ho pre zjednodušenie generovania kódu u výrazov. Pri vymýšľaní tohto stromu sme si ho najprv potrebovali nakresliť. Ďalej sme prišli na to, že budeme potrebovať stack, práve na zjednodušenie skladania daného stromu. Skladanie abstraktného derivačného stromu prebieha po node-och, pomocou ktorého sa vygeneruje kód, ktorý nám vráti premennú, v ktorej je uložený výsledok. Ďalej sa tento node nahradí danou premennou. Tento postup opakujeme, pokiaľ nám neostane posledný node.

3.2 Hashtable

Hashtable sme implementovali pomocou metódy Double hashing, ktorá spočíva v tom, že pri nájdení kolízie hashujeme dva krát a druhý hash je vždy násobený počtom rehashovaní. Táto metóda je najefektívnejšia, keď sa zvolia správne prvočísla, ktoré sú použité pri hashovaní a spolu s tým sa zvolí aj správny rozsah zaplnenia, ktorý je u nás 65%. Tento nami vybraný rozsah zaplnenia znižuje riziko kolízie.

Tento typ hash table, sme si vybrali z dôvodu jeho efektivity, je síce náročnejší na implementáciu, ale to nás aj tak neodradilo.

3.3 Abstraktný stack

Inšpiráciu sme čerpali z originálnej štruktúry stack, ktorý sme obohatili o funkcionalitu.

Funkcionalita spočíva v tom, že môžeme získavať všetky položky v celom stacku, bez toho aby sme museli popovať všetky prvky pred danou položkou. Ďalšou špeciálnou funkciou je popovanie položky na danom indexe. Položky nášho abstraktného stacku obsahujú 3 druhy informácii: integer, string a void pointer.

Tento spôsob implementácie sme si zvolili preto, lebo je možné použiť ho vo viacerých častiach a rôznymi možnosťami v našom prekladači.

4. Práca v tíme

4.1 Spôsob práce v time

Na projekte sme začali pracovať v podstate od zadania projektu. Po zadaní projektu sme si dohodli stretnutie, kde sme sa dohodli na spôsobe akým sa budeme riadiť. Pracovalo sa po častiach, a vždy keď sa jednotlivé časti dokončili dali sa dokopy a otestovali sa. Na jednotlivých častiach pracoval väčšinou jeden človek, pričom niekedy došlo k pomoci od iného člena tímu.

4.1.1 Verzovací systém

Pre správu projektu sme používali verzovací systém Git. Ako vzdialený repozitár sme používali GitHub. Pokiaľ žiadal niekedy pomoc člen tímu od niekoho iného, poslal svoj súbor na privátny Discord server, kde popísal svoju chybu a niekto z ostatných členov sa na to rýchlo pozrel a pomohol mu.

4.1.2 Komunikácia

Ako bolo písané vyššie tak od prvého stretnutia sme sa stretávali osobne prakticky každý týždeň, niekedy aj viackrát. Pokiaľ sme riešili projekt a potrebovali sme komunikovať, že sme prišli na nejakú chybu alebo sme si nevedeli rady, tak sme používali spomínaný Discord, kde sme odpoveď od ostatných členov dostali pomerne rýchlo.

4.2 Rozdelenie práce medzi členmi

Prácu v tíme sme si rozdelili tak, aby každý člen tímu mal čas robiť si aj ostatné projekty. Body sme si rozdelili každý po 25%.

Člen tímu	Pridelená práca
Pavel Podlužanský	Vedenie a organizácia tímu, testovanie, dokumentácia, prezentácia, časť sémantickej analýzy, projektové knižnice (string)
Adam Richter	Hlavný tester, lexikálna analýza, dokumentácia
Matúš Škúta	Syntaktická analýza, testovanie, ostatné projektové knižnice a generátor kódu spolu so sémantickou analýzou.
Jaromír Hradil	Precedenčná analýza, testovanie, projektové knižnice (stack)

5. Záver

Na projekte sme začali robiť relatívne skoro, začali sme najprv s hľadaním informácií k projektu, spracovaní konečného automatu a LL-gramatiky, resp. tabuľky. Popri vytváraní prekladača, sme zistili, že potrebujeme vytvoriť pomocné knižnice, ktoré sme implementovali postupne. Keďže sme ešte nevedeli veľa vecí, ktoré sme sa dozvedeli až na prednáškach, implementácia prebiehala postupne s prednáškami.

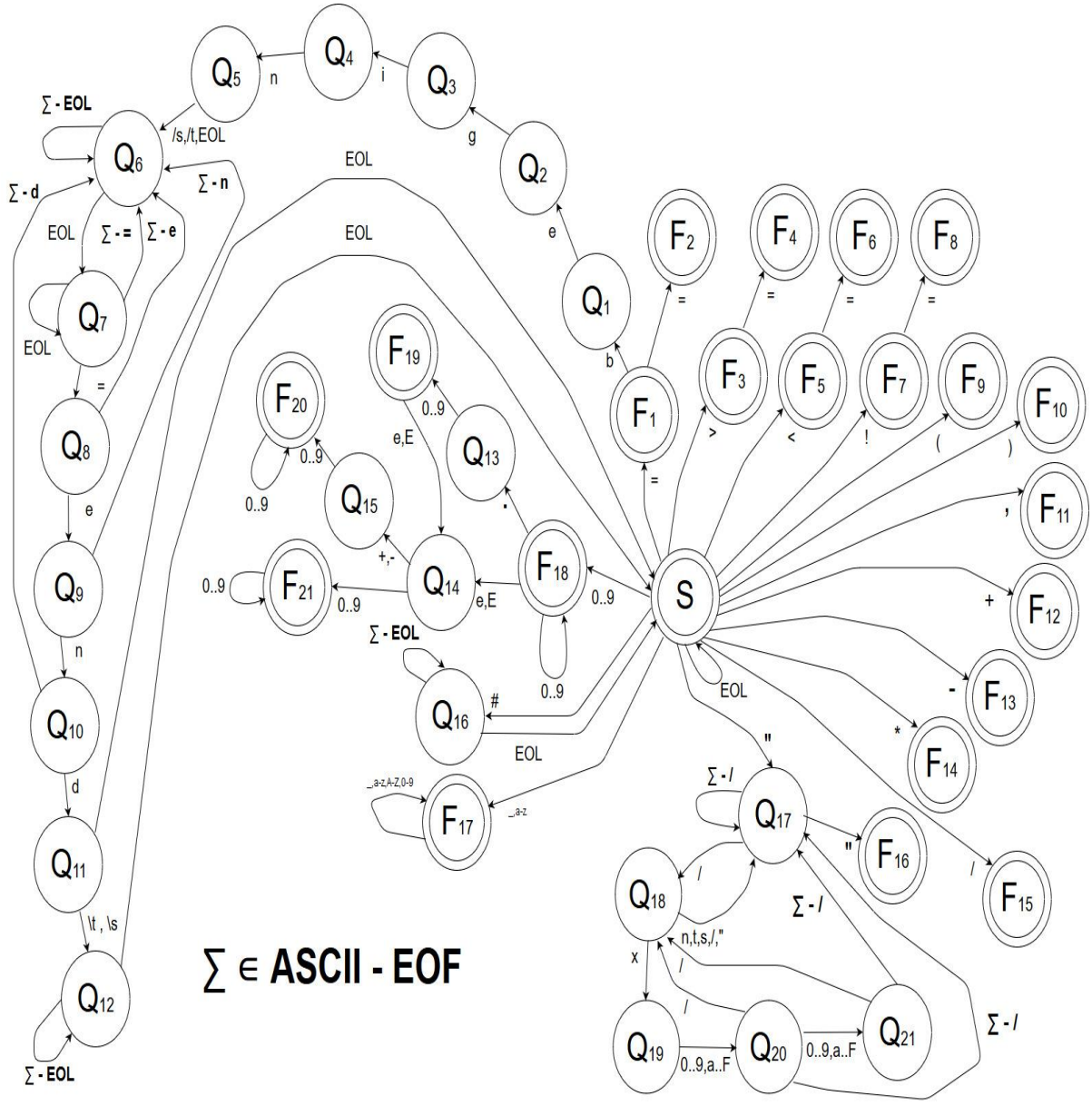
Počas projektu sme prišli na viacero problémov, ktoré sme najprv riešili na spoločných stretnutiach. Ak sme ich náhodou nevedeli, zvyčajne nám pomohlo fórum, alebo sme sami položili otázku na fórum.

Prvé pokusné odovzdanie bolo veľmi prínosné, nakoľko nám ukázalo značný počet chýb. Za druhé pokusné odovzdanie sme boli teda veľmi vďační a ukázalo nám, že sa nám podarilo opraviť značnú mieru chýb.

Tento projekt bol veľmi prínosný, naučil nás ako tímovej spolupráci, tak aj riešeniu komplikovanejších problémov a objasnil nám funkcionalitu prekladačov.

6. Prílohy

P1 Diagram konečného automatu



Obr.1. Konečný automat špecifikujúci lexikálny analyzátor

P2 LL- gramatika

1. <program> -> <function> <statement> <program>
2. < program > -> EOL < program >
3. < program > -> EOF
4. < function > -> def ID (<params>) EOL <statement> end EOL
5. < function > -> ε
6. < params > -> ID < params_more>
7. < params > -> ε
8. < params_more> -> , ID < params_more>
9. < params_more> -> ε
10. < statement > -> if <expression> then EOL <statement> else EOL < statement > end EOL
11. < statement > -> while <expression> <do> EOL <statement> end EOL
12. < statement > -> print (< print_params >) EOL
13. < statement > -> id <id_assign> EOL
14. < statement > -> < expression > EOL
15. < statement > -> ε
16. < print_params > -> < term > < print_params_more>
17. < print_params_more> -> , < term > < print_params_more>
18. < print_params_more> -> ε
19. <id_assign> -> = <expression >
20. < id_assign > -> <params>
21. < id_assign> -> (<params>)

Obr.2. LL-gramatika

P3 LL- tabulka

	EOL	EOF	def	ID	if	while	print	,	<expression>	<term>	(	)	=	$
< program >	2	3	1	1	1	1	1		5					
< function >	5		4	5	5	5	5		5					
< params >	7			6								7		
< params_more >	9							8				9		
< statement >	15	15		13	10	11	12		14					15
< print_params >										16				
< print_params_more >	18							17				18		
< Id_assign >	20			20							21		19	

Obr.3. LL tabuľka

P4 Precedenčná tabuľka

	+	-	*	/	(	)	<	>	<=	>=	=	==	!=	id	$
+	>	>	<	<	<	>	>	>	>	>	!	>	>	<	>
-	>	>	<	<	<	>	>	>	>	>	!	>	>	<	>
*	>	>	>	>	<	>	>	>	>	>	!	>	>	<	>
/	>	>	>	>	<	>	>	>	>	>	!	>	>	<	>
(	<	<	<	<	<	=	<	<	<	<	<	<	<	<	!
)	>	>	>	>	!	>	>	>	>	>	!	>	>	!	>
<	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
>	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
<=	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
>=	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
=	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
==	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
!=	<	<	<	<	<	>	!	!	!	!	!	!	!	<	>
id	>	>	>	>	!	>	>	>	>	>	>	>	>	!	>
$	<	<	<	<	<	!	<	<	<	<	<	<	<	<	!

Obr.4 Precedenčná tabuľka