Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ18

Tým 112, varianta II

Adrián Boros	**(xboros03)**	27 %
Adam Abrahám	(xabrah04)	27 %
Matouš Sloboda	(xslobo04)	19 %
Tomáš Žigo	(xzigot00)	27 %

5. decembra 2018

Obsah

1 Úvod

Táto dokumentácia popisuje návrh a implementáciu prekladača imperatívneho jazyka IFJ18, ktorý je zjednodušenou podmnožinou jazyka Ruby 2.0. Dokumentácia je rozdelená do niekoľkých sekcií, ktoré popisujú implementáciu jednotlivých častí projektu. Ďalej su pridané prílohy obsahujúce diagram konečného automatu, LL – gramatiky, LL – tabuľky a precedenčnej tabuľky.

2 Návrh a implementácia

Pri registráciách sme si vybrali variantu číslo II, ktorá spočíva v riešení projektu za pomoci tabuľky s rozptýlenými položkami. Projekt sme zostavili z niekoľkých častí, ktoré sú predstavené v tejto kapitole.

2.1 Lexikálna analýza

Návhr LA bol prvým krokom k vytvorení prekladača. Hlavnou funkciou tejto časti je `scan`, ktorá číta znak po znaku zdrojový kód, odstraňuje biele znaky a prevádza načítané znaky na štruktúru `Token`, ktorá sa skladá z typu a atribútu. Typy tokenov môžu byť: identifikátor, kľúčové slovo, celé číslo, desatinné číslo aj v exponenciálnom tvare, reťazec, aritmetický alebo relačný operátor, `EOL`, `EOF` atď. Atribút sa ďalej rozdeľuje podľa typu tokenu. V prípade, že token je reťazec, identifikátor alebo kľúčové slovo, atribútom je daný reťazec alebo dané kľúčové slovo. Ak sa jedná o celé alebo desatinné číslo, atribútom je číslo. Relačné alebo aritmetické operátory, tieto typy atribút nemajú. Takto vytvorené tokeny sú ďalej predávané ďalším častiam.

Celý lexikálny analyzátor je implementovaný ako deterministický konečný automat, ktorý sme si navrhli pred samotnou implementáciou a nachádza sa v prílohe 1. Lexikálny analyzátor obsahuje riadiaci `switch` s premennou `stateFlag` do ktorej sa ukladá aktuálny stav. Každý `case` odpovedá jednému stavu automatu. V prípade lexikálnej chyby program končí s návratovou hodnotou 1. V opačnom prípade sa načítavajú znaky až kým sa nám nevytvorí token ktorý môžeme predať syntaktikej analýze. Pri načítaní reťazca sa najprv skontroluje či daný reťazec nie je kľúčové slovo. Klúčové slová sú uložené v poli pevnej veľkosti. Ak sa reťazec nezhoduje s klúčovým slovom, jedná sa o identikátor. Kedže jazyk IFJ18 je `case-sensitive`, na konci prebehne kontrola či náhodou kľúčové slovo neobsahuje veľké písmeno.

2.2 Syntaktická analýza

Syntaktický analyzátor je jadrom prekladača. Kontroluje syntaktickú správnosť programu a zároveň riadi celý preklad. Syntaktická analýza sa riadi LL(1) – gramatikou, ktorá sa nachádza v prílohe 2, a metódou rekurzívneho zostupu podľa pravidiel v LL – tabuľke. Jednou z najväčších výziev bol návrh LL – gramatiky ktorú sme počas implementácie museli niekoľkokrát prerábať. Čistú LL(1) gramatiku sa nám nepodarilo vytvoriť pretože to zadanie neumožňuje. Problémová časť je tá, kde sa na základne jedného tokenu nedá rozhodnúť či sa jedná o priradenie alebo iba zadaný názov premennej. Vyriešili sme to načítaním ďalšieho tokenu, na základe ktorého sa rozhoduje do ktorého stavu sa prechádza. Pre každý neterminál sme si vytvorili vlastnú funkciu. V prípade potreby si funkcia žiada ďalší token pomocou volania funkcie `scan`. Ak program nie je zapísaný syntakticky správne, končí sa chybou 2.

2.2.1 Syntaktická analýza výrazov

Analýza výrazov je prevádzaná za pomoci precedenčnej tabuľky ktorá sa nachádza v prílohe 3. Kedže operátory `+` a `-` majú rovnakú prioritu, mohli sme si tabuľku zjednodušiť, takisto aj pri operátoroch `*` a `/`. Symbol `R` označuje relačné operátory `>, <, <=, >=`. Symbol `i` symbolizuje identifikátor, reťazec alebo číslo. Symbol `$` označuje všekty ostatné symboly, výraz nemôže obsahovať. Všetky spracovania výrazov sa vykonávajú v súbore `expression.c`, kde sa kontroluje aj správnosť aritmetických a logických výrazov. V tejto časti prevádzame aj možné konverzie z `int` na `float` alebo naopak. Aby sme rozlíšili kedy sa nachádzame v definícii funkcie alebo v hlavnom tele programu tak sme použili boolovskú premennú `inDef`.

2.3 Sémantická analýza

Sémantická analýza sa prevádza súbežne so syntaktickou. Vo vytvorenej štruktúre `parserData` je uložená lokálna aj globálna tabuľka symbolov. Lokálna tabuľka je použitá pre premenné v tele funkcií. Globálna tabuľka je použitá pre hlavné telo programu a názvy funkcií. Tabuľky symbolov sú implementované ako tabuľky s náhodným rozptýlením. Tabuľky slúžia pre kontroly, či nedochádza k pretypovaniu danej premennej alebo či je daná premenná definovaná.

2.4 Generovanie cieľového kódu IFJcode18

Cieľový kód je vygenerovaný po úspešnom dokončení všetkých možných analýz, pričom jednotlivé inštrukcie sa generujú do dynamickej štruktúry a na konci programu je celý obsah štruktúry vypísaný na štandardný výstup. Vo vygenerovanom kóde sú použité komentáre pre zprehľadnenie výsledného kódu. Implementácia všetkých inštrukcií je umiestnená v súbore `generator.c`, ktorý obsahuje funkcie pre generovanie jednotlivých častí programu ako napr. generovanie začiatku alebo konca funkcie, volanie funkcie, generovanie cyklov alebo podmieneného výrazu atď. Používame dve dynamické štruktúry, jednu na generovanie kódu vo funkciách a druhú pre generovanie kódu v hlavnom tele programu. Je to z dôvodu že sa v tele programu môžu prelínať definície funkcií s hlavným telom programu. Týmto je zaručené vypísanie hlavného tela programu úplne na konci.

Na začiatku generovania je vygenerovaná potrebná hlavička, sú vygenerované globálne premenné v globálnych rámcoch a skok do hlavného tela programu. Ďalej sú vygenerované všetky vstavané funkcie `substr`, `ord`, `chr` a `lenght`.

Keďže IFJ18 je dynamicky typovaný jazyk, pri predávaní parametrov funkcie kontrolujeme či je možné predať parametre daného typu, takisto aj v tele funkcie pri aritmetických a relačných operáciách kontrolujeme kompatibilitu typov a ak je to možné previede sa konvertovanie na iný typ.

3 Tabuľka symbolov

3.1 Tabuľka s rozptýlenými položkami

Tabuľka symbolov je implementovaná ako tabuľka s rozptýlenými položkami s explicitným zreťazením synonym a nachádza sa v súbore `symtable.c`. Veľkosť mapovacieho poľa sme zvolili tak aby bolo rovné prvočíslu. Zvolili sme číslo 8641, tak aby nazaberalo veľa pamäte a zároveň nespôsobovalo veľa konfliktov.

Hashovacia funkcia má zásadný vplyv na výkon. Ako hashovaciu funkciu sme si vybrali `Murmur2`. Pre použitie tejto funkcie sme sa rozhodli na základe testu zo zdroja [1], kde sa porovnávali rýchlosti a spoľahlivosti rôznych hashovacích funkcií. Pod pojmom spoľahlivosť sa rozumie počet konfliktov. Implementácia hashovacej funkcie prebiehala na základe zdroja [2]. V prípade ideálnej hashovacej funkcie je prístupová doba k položkám konštantná a zvyšuje sa v prípade konfliktu. Každá položka obsahuje unikátny kľúč v podobe reťazca, ktorým je identifikátor premennej alebo funkcie. Obsahuje taktiež ukazateľ na ďalší prvok. Dátová časť obsahuje údaje o tom akého typu je symbol(`dtUndefined`, `dtInt`, `dtFloat`, `dtString`). Ďalej obsahuje boolovské premmenné značiace či je to funkcia alebo parameter.

4 Práca v tíme

4.1 Spôsob práce v tímu

Projekt sme sa snažili riešiť už hneď po uverejnení zadania. Avšak pre nedostatok znalostí sme si veľa vecí museli naštudovať samostatne. Celý projekt sme začali implementovať v strede októbra. Na začiatku sme nemali plán na rozdelenie práce ale všetko sme implementovali spoločne. Až postupom času, po nadobudnutí potrebných vedomostí sme si implementáciu rozdelili.

4.2 Komunikácia

Na začiatku sme sa stretávali raz do týždňa a to v priestoroch školy. Okrem osobných stretnutí sme komunikovali cez aplikáciu `Slack`. Ako verzovací systém sme používali `Github`. S blížiacim sa termínom odovzdania projektu sme sa stretávali častejšie, niekedy aj každý deň.

4.3 Rozdelenie práce

Projekt sa vypracovával väčšinou spoločne. Pre dané bodové rozdelenie sme sa rozhodli z dôvodu, že po rozdelení práce na projekte sa však viac do implementácie zapojili hlavne prví traja študenti.

Člen tímu	Práca
Adrián Boros	vedúci tímu, lexikálna analýza, generovanie cieľového kódu, návrh LL-gramatiky, testovanie, dokumentácia
Adam Abrahám	lexikálna analýza, syntaktická analýza, sémantická analýza, syntaktická analýza výrazov, tabuľka symbolov, pomocné dátové štruktúry
Tomáš Žigo	lexikálna analýza, generovanie cieľového kódu, syntaktická analýza, sémantická analýza návrh LL-gramatiky
Matouš Sloboda	lexikálna analýza, testovanie, sprevádzkovanie komunikačných a verzovacích kanálov

Tabuľka 1: Rozdelenie práce

5 Záver

Projekt bol pre nás niečím novým, nakoľko to bol prvý tímový projekt ktorý sme absolvovali. Prvýkrát sme museli konzultovať naše návrhy na implementáciu a museli sme vytvoriť spolupracujúce časti. Počas práce na tomto projekte sme získali skúsenosti s prácou na rozsiahlom projekte. Oboznámili sme sa s fázami, od tvorby návrhu až po testovanie. Využili sme pokusné odovzdanie ktoré nám pomohlo odhaliť slabé miesta v našom projekte. Projekt sme sa snažili vypracovať podľa zadania. Najťažšie boli posledné týždne a dni pred finálnym odovzdaním.

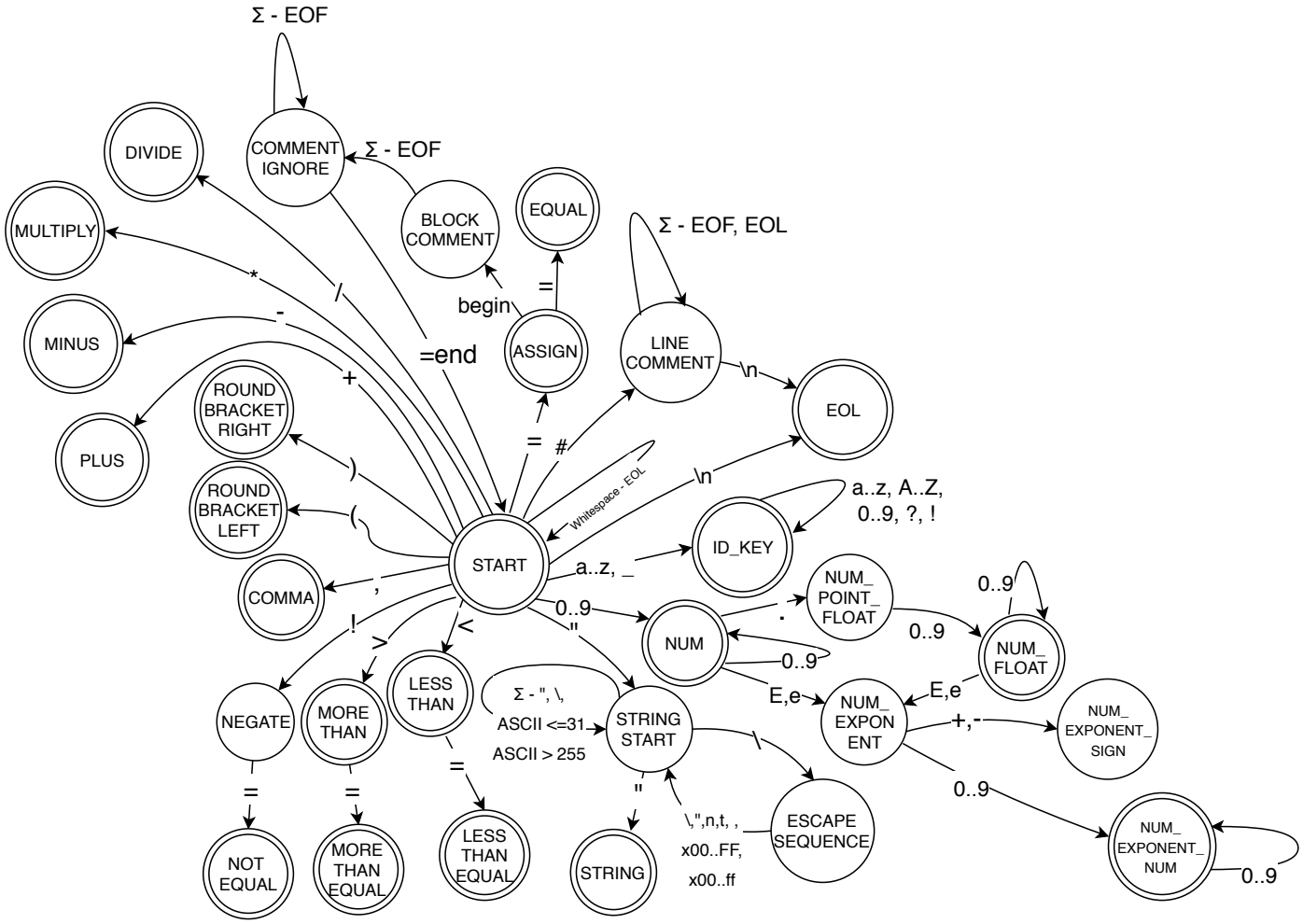
Literatúra

[1] StacExchange: *Which hashing algorithm is best for uniqueness and speed?* Feb. 2011 https://tinyurl.com/yb2u8ma8

[2] MurmurHash2. Jún 2011 https://github.com/abrandoned/murmur2/blob/master/MurmurHash2.c

6 Prílohy

A Diagram konečného automatu



Obr. 1: Diagram konečného automatu špecifikujúci lexikálny analyzátor

Poznámky k diagramu

Za zmienku stojí zakreslenie spracovania blokových komentárov, kde komentár začína reťazcom `=begin` a končí `=end`, ktoré však musia byť umiestnené na začiatku riadka. V diagrame sa nám tieto podmienky nepodarilo presne zachytiť. V implementácii sa to rieši boolovskou premennou `LineBegin` ktorá určuje či sa nachádzame na začiatku riadka alebo nie. Po prijatí reťazca `=end` sa ďalej odignorujú znaky až do konca riadka.

Takisto problémom bolo zachytiť možnosť, že znaky `?` a `!` sa môžu vyskytovať iba na konci identifikátorov funkcií. Kontrola či je tomu tak, sa však prevádza až v syntaktickej analýze.

B LL – gramatika

1. <prog> -> <program_body>
2. <prog> -> <def_id>
3. <prog> -> EOF
4. <prog> -> EOL <prog>
5. <program-body> -> <command> <prog>
6. <def_id> -> def FunctId (<param_list>) EOL <command> end
7. <param_list> -> ID <param>
8. <param_list> -> ε
9. <params> -> , ID <param>
10. <params> -> ε
11. <value> -> INT_VALUE
12. <value> -> FLOAT_VALUE
13. <value> -> STRING_VALUE
14. <value> -> ID
15. <command> -> ID = <def_value> EOL <command>
16. <command> -> ID EOL <command>
17. <command> -> if <expression> then EOL <command> else EOL <command> end EOL <command>
18. <command> -> while <expression> do EOL <command end EOL <command>
19. <command> -> ε
20. <command> -> <def_value> EOL <command>
21. <def_value> -> FunctId <arg>
22. <def_value> -> print <arg>
23. <def_value> -> = <expression>
24. <def_value> -> inputs <arg>
25. <def_value> -> inputi <arg>
26. <def_value> -> inputf <arg>
27. <def_value> -> lenght <arg>
28. <def_value> -> substr <arg>
29. <def_value> -> ord <arg>
30. <def_value> -> chr <arg>
31. <arg> -> <value> <arg_n>
32. <arg> -> (<arg_n2>)
33. <arg> -> ε
34. <arg_n2> -> <value> <arg_n>
35. <arg_n2> -> ε
36. <arg_n> -> <value> <arg_n>
37. <arg_n> -> ε

Tabuľka 2: LL gramatika

C Precedenčná tabuľka

	+-	*/	R	== !=	(	)	i	$
+-	>	<	>	>	<	>	<	>
***/**	>	>	>	>	<	>	<	>
R	<	<		>	<	>	<	>
== !=	<	<	<		<	>	<	>
(	<	<	<	<	<	=	<	
)	>	>	>	>		>		>
i	>	>	>	>		>		>
$	<	<	<	<	<		<	

Tabuľka 3: Precedenčná tabuľka použitá pri analýze výrazov

D LL – tabulka

	DEF	EOF	EOL	END	,	(	INT_VAL	FLOAT_VAL	STRING_VAL	ID	FUNCTID	IF	WHILE	PRINT	INPUTS	INPUTI	INPUTF	LENGTH	SUBSTR	ORD	CHR	$
<prog>	2	3	4							1	1	1	1	1	1	1	1	1	1	1	1	1
<program_body>										5	5	5	5	5	5	5	5	5	5	5	5	5
<def_id>	6																					
<param_list>										7												8
<param>					9																	10
<value>							11	12	13	14												
<command>										15,16	20	17	18	20	20	20	20	20	20	20	20	19
<def_value>											21			22	24	25	26	27	28	29	30	23
<arg>						32	31	31	31	31												33
<arg_n2>							34	34	34	34												35
<arg_n>					36																	37

Tabuľka 4: LL – tabulka