VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentácia k projektu do predmetov IFJ a IAL

Implementácia prekladača imperatívneho jazyka IFJ17

Tím 8, varianta II

6. prosince 2017

Kristián Liščinský (xlisci01) 25%
Matúš Liščinský (xlisci02) 25%
Vladimír Marcin (xmarci00) 25%
Šimon Stupinský, vedúci (xstupi10) 25%

Obsah

1 Úvod

Dokumentácia popisuje implementáciu prekladača imperatívneho jazyka IFJ17. Zvolili sme si variantu II, ktorá obsahovala implementáciu tabuľky symbolov pomocou tabuľky s rozptýlenými položkami. Implementáciu sme rozdelili do 3 hlavných častí, pričom každej z nich bude venovaná samostatná kapitola.

- lexikálna analýza
- synktatická a sémantická analýza
  - synktatická analýza (bez spracovania výrazov)
  - synktatická analýza spracovania výrazov
- generovanie vnútorného kódu

Dokumentácia taktiež zahrňuje 4 prílohy, ktoré obsahujú LL gramatiku, LL tabuľku, precedenčnú tabuľku a diagram konečného automatu, vytvorený pomocou [1], popisujúci lexikálny analyzátor.

2 Jazyk IFJ17

Jazyk IFJ17 je zjednodušenou podmnožinou jazyka FreeBASIC, ktorý stavia na legendárnom jazyku BASIC a je to staticky typovaný imperativný jazyk. V jazyku IFJ17 nezáleží na veľkosti písmen u identifikátorov a kľúčových slov, je teda case - insensitive. Je možné pracovať s celočíselnými, alebo desatinnými literálmi v základnom a exponenciálnom tvare. Zároveň je tiež možné využívať aj reťazcové literály, ktoré môžu obsahovať escape sekvencie zadané v dekadickom tvare. Podporovanými dátovými typmi jazyka IFJ17 sú typy **Integer**, **Double** a **String** a zároveň jazyk podporuje ako riadkové, tak aj blokové komentáre. Výrazy sú tvorené pomocou termov, zátvoriek a binárnych aritmetických, reťazcových a relačných operátoroch.

3 Štruktúra projektu

3.1 Lexikálna analýza

Lexikálna analýza je činnosť, ktorú vykonáva **lexikálny analyzátor** (scanner). Je to vstupná a najjednoduchšia časť prekladača. Lexikálny analyzátor rozdeľuje vstupnú postupnosť znakov, ktoré reprezentujú zdrojový program, na jednotlivé lexikálne jednotky - **lexémy**. Lexémy sú reprezentované vo forme tokénov a tie sú nasledne poskytované synktatickému analyzátoru. Jednotlivé tokeny generované lexikálnou analýzou sú implementované ako dvojice (sym, atr), kde sym je meno (identifikácia) symbolu a atr predsavuje prípadný atribut symbolu. U symbolov bez atribútu je atr prázdne [2]. Úlohou lexikálneho analyzátora je taktiež zbaviť zdrojový program bielych znakov a komentárov. V praxi je implementovaný pomocou konečného automatu (príloha C). V prípade, že načítame niečo, čo nezapadá do pravidiel programovacieho jazyka IFJ17 nastáva lexikálna chyba. Základnú kostru diagramu konečného automatu pre lexikálny analyzátor sme prebrali z minuloročného projektu jedného člena nášho tímu.

3.2 Synktatická a sémantická analýza

3.2.1 Synktatická analýza (bez spracovania výrazov)

Syntaktický analyzátor (parser) kontroluje množinu pravidiel, ktorá určuje prípustné konštrukcie daného jazyka. Syntaktická analýza je implementovaná rekurzívnym zostupom, ktorý je riadený pravidlami **LL-gramatiky** (príloha B). Terminálne symboly predstavujú tokeny ktoré sú na požiadanie vrácané lexikálnym analyzátorom. Na základe prijatých tokenov syntaktický analyzátor simuluje tvorbu derivačného stromu. Ak sa nepodarí nasimulovať derivačný strom, jedná sa o syntaktickú chybu a analýza sa ukončí.

Spolu so syntaktickou analýzou je súčasne vykonávaná aj sémantická analýza. Pri definícii alebo deklarácii funkcie sa do globálnej tabuľky symbolov uloží dátová štruktúra reprezentujúca danú funkciu (v jazyku IFJ17 sú v globálnej tabuľke symbolov uložené iba funkcie) a súčasne sa vykonávajú semantické kontroly (napr.: či sa nejedná o redefiníciu funkcie apod.). V prípade definície funkcie sa následne spracováva telo funkcie. Priamo pri rekurzívnom zostupe sa vykonávajú všetky potrebné sémantické kontroly a napĺňa sa lokálna tabuľka symbolov príslušnej funkcie. Keď pri spracovávaní zdrojového súboru parser narazí na výraz alebo volanie funkcie, riadenie je predané modulu **expr** (volanie funkcie `expression(...)`), ktorý pomocou precedenčnej analýzy skontroluje správnosť výrazu a vyvolá potrebné sémantické kontroly a vráti riadenie späť parseru.

3.2.2 Synktatická analýza spracovania výrazov

Pre spracovanie výrazov sme podľa zadania použili **precedenčnú syntaktickú analýzu**, riadenú **precedenčnou tabuľkou**. Modul vykonávajúci spracovanie výrazov, je volaný syntaktickým analyzátorom pri dvoch situáciách. Tou prvou je samotné spracovanie výrazov a druhou je volanie užívateľských a vstavaných funkcií.

Parser načítava postupnosť tokenov a podľa pravidiel v LL-gramatike, v ktorých sa vyskytuje výraz, alebo volanie funkcie, predáva riadenie precedenčnej analýze. Vstupom tejto analýzy je postupnosť tokenov získaná lexikálnym analyzátorom a výstupom je postupnosť inštrukcií, ktoré sú zároveň vstupom pre dodaný interpret.

Spracovanie výrazov riadené precedečnou tabuľkou (príloha B) prebieha v dvoch krokoch. V prvom kroku je kontrolovaná správna postupnosť tokenov, ktoré sú prípustné v rámci syntaxe. V kladnom prípade, je výraz pomocou zásobníkovej štruktúry prevedený z infixovej notácie na výhodnú postfixovu. V druhom kroku je postfixová notácia prevádzaná na príslušné inštrukcie, zabezpečujúce vyhodnotenie výrazu. V tomto kroku zároveň prebieha kontrola typovej kompatibility operandov a prípadne sú generované inštrukcie implicitného pretypovania.

Volanie funkcií je obsiahnuté v kontrole typovej kompatibility parametrov a ich počtu. Na záver je kontrolovaná typová kompatibilita návratového typu volanej funkcie s typom premennej, do ktorej by mala byť priradená návratová hodnota funkcie.

3.3 Generovanie vnútorného kódu

Počas priebehu vykonávania syntaktickej a sémantickej analýzy sa generujú ekvivalentné inštrukcie kódu jazyka **IFJcode17**. Postupne sa tak inštrukciami plní jedna globálna inštrukčná páska implementovaná ako jednosmerne viazaný zoznam. Celá inštrukčná páska sa vypíše na štandardný výstup iba v prípade lexikálnej, syntaktickej i sémantickej správnosti zdrojového textu v jazyku IFJ17.

4 Dátové štruktúry

4.1 Zásobník

Zásobník sme implementovali pomocou dátovej štruktúry jednosmerného zoznamu. Štruktúra zásobníka obsahuje ukazateľ na položku zásobníka ktorá je aktuálne na vrchole zásobníka. Položka zásobníka predstavuje ukazateľ na dátový typ void a ukazateľ na nasledujúci prvok.

To že sme sa položku zásobníka rozhodli implementovať pomocou ukazateľa na dátový typ void, bolo ovplyvnené tým, že táto voľba umožňuje široké využitie zásobníka. Samotný zásobník vhodne využívame vo viacerých prípadoch. Prvým z nich je prevádzanie výrazov na postfixovu notáciu, pri ktorej využívame dva zásobníky. Prvý pre tvorbu výsledného výrazu a druhý pre pomocné ukladanie operátorov a zátvoriek podľa ich priorít. Zásobník je tiež využívaný pri tvorbe inštrukcií pre vyhodnocovanie výrazov a tiež pre ukladanie parametrov funkcie, kvôli uchovaniu príslušného poradia.

Špeciálne využitie zásobníka predstavuje modul pre uvoľnenie alokovanej pamäte počas behu programu. Ukazatele na alokovanú pamäť sa ukladajú na zásobník a pred ukončením behu programu sú tieto ukazetele korektne uvoľnené. Tento modul ma tak podobnú funkčnosť ako známy garbage collector.

4.2 Tabuľka s rozptýlenými položkami

Tabuľka s rozptýlenými položkami tvorí základ tabuľky symbolov. Slúži na ukladanie dvojice kľúč-hodnota a jej výhodou je rýchlosť vyhľadávania položiek. Základom je pole ukazateľov na jednotlivé položky. Jedna položka obsahuje svoj kľúč, data a ukazateľ na ďalšiu položku, aby mohli byť zreťazené v jednosmerne viazanom zozname synonym.

V našom prípade data jednej položky predstavujú ukazateľ na štruktúru reprezentujúcu premennú alebo funkciu. Hashovacia funkcia má zásadný vplyv na výkon tabuľky. Nami vybraná hashovacia funkcia, ktorú sme použili, je uvedená v referenciách [3]. V prípade ideálnej hashovacej funkcie je čas prístupu k položkám konštantný, pričom v prípade konfliktu sa čas nájdenia položky predlží o dobu prehľadania zoznamu synonym. V našom programe sa nachádza jedna globálna tabuľka symbolov, v ktorej sú uložené informácie o funkciách, pričom každá funkcia ma svoju vlastnú tabuľku symbolov, ktorá obsahuje informácie o parametroch alebo lokálnych premenných funkcie. Tabuľka symbolov je implementovaná v súboroch `symtable.c` a `symtable.h`.

5 Vývoj

Tím o 4 ľuďoch už vyžaduje opatrenia, aby bol vývoj riadený, aby každý vedel, čo treba robiť a čo robia ostatní. Tieto opatrenia bolo potrebné zaviesť hneď na začiatku, ešte predtým ako sa začne vývojový cyklus.

5.1 Rozdelenie práce

- **Kristián Liščinský:** lexikálna analýza, dokumentácia, testovanie
- **Matúš Liščinský:** generovanie kódu, pomocné moduly pre prácu s reťazcami, garbage collector, testovanie
- **Vladimír Marcin:** návrh LL-gramatiky, syntaktická analýza bez spracovavania výrazov, sémantické kontroly, návrh tabuľky symbolov, testovanie
- **Šimon Stupinský:** návrh LL-gramatiky, syntaktická analýza so spracovavaním výrazov, sémantické kontroly, implementácia zásobníka, automatické testy

5.2 Komunikácia a použité prostriedky

Komunikácia nášho tímu prebiehala prevažne osobne v pravidelných intervaloch, ktoré sa s blížiacim príchodom dňa odovzdania znižovali. Okrem toho sme tiež využívali online komunikáciu v podobe skupinového chatu na facebooku.

Keďže práca celého tímu bola závislá na jedných a tých istých súboroch, pre väčší komfort sme sa rozhodli využiť distribuovaný verzovací systém git.

6 Záver

Projekt bol naším prvým tímovým projektom, prvykrát sme museli vzájomne komunikovať a konzultovať jednotlivé možnosti implementácie a zmeny v zdrojových kódoch. Počas práce na ňom sme si rozšírili svoje znalosti a získali sme skúsenosti s prácou na rozsiahlejšiom projekte a prácou v tíme. Oboznámili sme sa s náročnosťou rôznych fáz tvorby projektu od návrhu až po testovanie. Využili sme možnosť pokusného odovzdania, ktoré nám pomohlo získať predstavu o stave jednotlivých častí nášho projektu. Na základe týchto informácií sme sa vedeli zamerať na slabšie miesta našej implementácie. Snažili sme sa vytvoriť program, ktorý bude plne vyhovovať zadaniu. Na konci sme dospeli k názoru, že práca na projekte nás posunula vpred nielen v programátorskom, ale aj ľudskom smere.

6.1 Metriky

- Počet súborov: **23**
- Počet riadkov zdrojového kódu: **2493**
- Počet Git commitov: **120**

7 Referencie

[1] JGraph Ltd. *Online diagram software for making flowcharts, process diagrams, org charts, UML, ER and network diagrams* [online]. 2005-2017 [cit. 2017-12-06]. Dostupné z: `http://draw.io`

[2] Prednášky *Slajdy a další studijní materíály nejen k přednáškám* [online]. [cit. 2017-12-01]. Dostupné z: `https://www.fit.vutbr.cz/study/courses/IFJ/public/materials/`

[3] *Hashovacia funkcia* [online]. [cit. 2017-11-20]. Dostupné z: `http://www.cse.yorku.ca/~oz/hash.html`

Prílohy

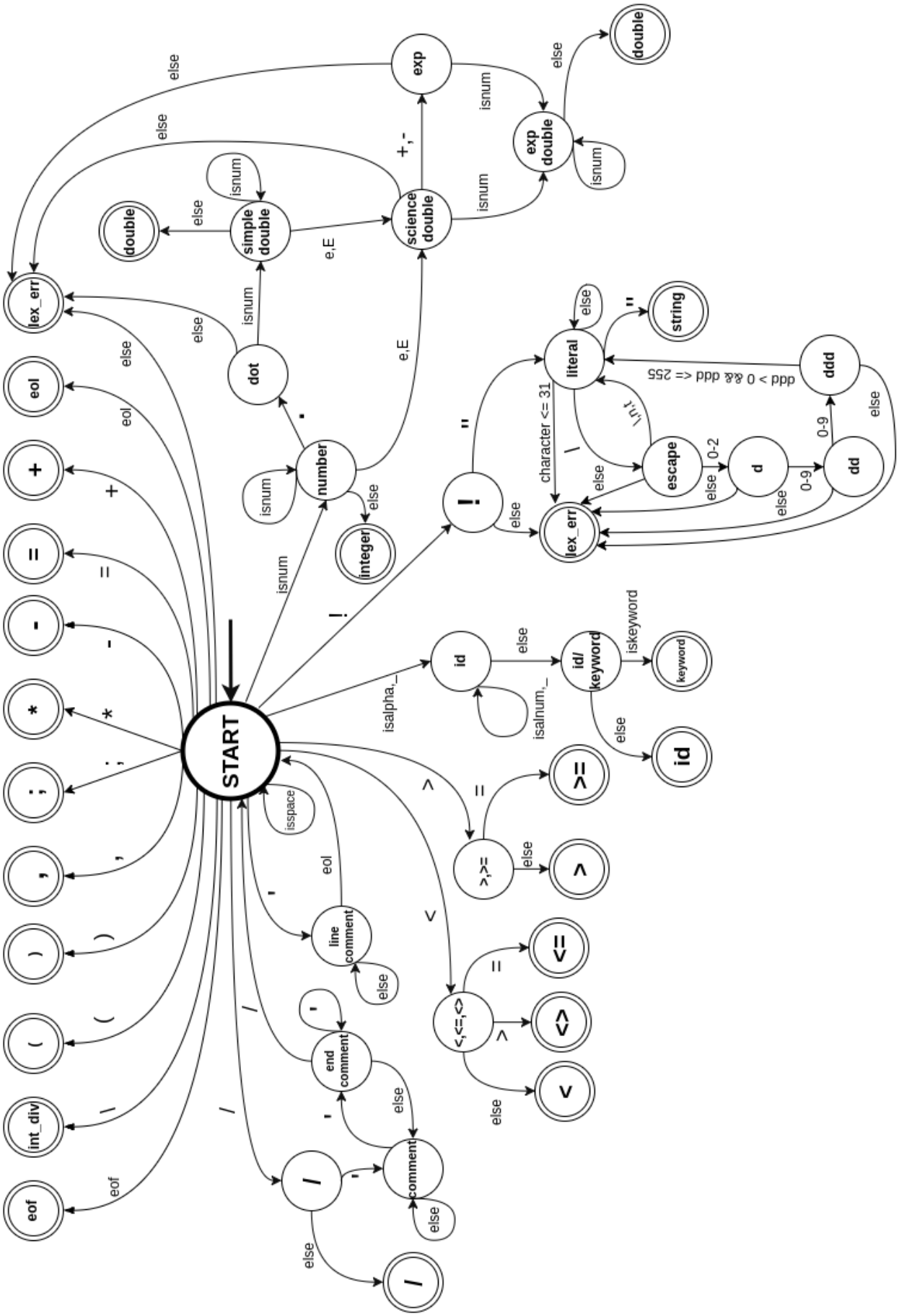
A LL gramatika

1	<prog>	->	<declare-function> EOL <prog>
2	<prog>	->	<define-function> EOL <prog>
3	<prog>	->	<main-function>
4	<prog>	->	EOL <prog>
5	<declare-function>	->	Declare Function id (<param-list>) As <data-type>
6	<define-function>	->	Function id (<param-list>) As <data-type> EOL <function-element> End Function
7	<main-function>	->	Scope EOL <function-element> End Scope
8	<function-element>	->	epsilon
9	<function-element>	->	Dim id As <data-type> <value> EOL <function-element>
10	<function-element>	->	<element-list>
11	<element-list>	->	<statement> <function-element>
12	<statement>	->	EOL
13	<statement>	->	id = <call-assign> EOL
14	<statement>	->	Print <E> ; <exp-to-print> EOL
15	<statement>	->	Input id EOL
16	<statement>	->	Return <E> EOL
17	<statement>	->	Do While <E> EOL <stat-list> Loop EOL
18	<statement>	->	If <E> Then EOL <stat-list> <else-branch> End If EOL
19	<value>	->	epsilon
20	<value>	->	= <E>
21	<else-branch>	->	epsilon
22	<else-branch>	->	Else EOL <stat-list>
23	<stat-list>	->	<statement> <stat-list>
24	<stat-list>	->	epsilon
25	<exp-to-print>	->	<E> ; <exp-to-print>
26	<exp-to-print>	->	epsilon
27	<call-assign>	->	<E>
28	<call-assign>	->	id(<param-value>)
29	<param-value>	->	epsilon
30	<param-value>	->	<E> <next-param-value>
31	<next-param-value>	->	, <E> <next-param-value>
32	<next-param-value>	->	epsilon
33	<param-list>	->	<param> <next-param>
34	<next-param>	->	, <param> <next-param>
35	<param-list>	->	epsilon
36	<next-param>	->	epsilon
37	<param>	->	id As <data-type>
38	<data-type>	->	Integer
39	<data-type>	->	String
40	<data-type>	->	Double

B LL tabuľka

	=	,	Declare	Dim	Do	Double	Else	End	EOL	Function	ID	If	Input	Integer	(	Loop	Print	Return	)	Scope	String	E
<prog>			1						4	2										3		
<declare_function>			5																			
<define_function>										6												
<main_function>																				7		
<fuction_element>				9	10			8	10		10	10	10				10	10				
<element_list>					11				11		11	11	11				11	11				
<statement>					17				12		13	18	15				14	16				
<value>	20								19													
<else>							22	21														
<stat_list>					23		24	24	23		23	23	23			23	23	23				
<expression>									26													25
<call_assign>											28											27
<param_value>																			29			30
<next_par_value>		31																	32			
<param_list>											33								35			
<next_param>		34																	36			
<param>											37											
<data_type						40								38							39	
<E>																						

C Diagram konečného automatu lexikálneho analyzátora



D Precedenčná tabuľka

	+	-	*	/	(	)	\	<	>	<=	>=	=	<>	id	lit	$
+	>	>	<	<	<	>	<	>	>	>	>	>	>	<	<	>
-	>	>	<	<	<	>	<	>	>	>	>	>	>	<	<	>
*	>	>	>	>	<	>	>	>	>	>	>	>	>	<	<	>
/	>	>	>	>	<	>	>	>	>	>	>	>	>	<	<	>
(	<	<	<	<	<	=	<	<	<	<	<	<	<	<	<	
)	>	>	>	>		>	>	>	>	>	>	>	>			>
\	>	>	<	<	<	>	>	>	>	>	>	>	>	<	<	>
<	<	<	<	<	<	>	<							<	<	>
>	<	<	<	<	<	>	<							<	<	>
<=	<	<	<	<	<	>	<							<	<	>
>=	<	<	<	<	<	>	<							<	<	>
=	<	<	<	<	<	>	<							<	<	>
<>	<	<	<	<	<	>	<							<	<	>
id	>	>	>	>		>	>	>	>	>	>	>	>			>
lit	>	>	>	>		>	>	>	>	>	>	>	>			>
$	<	<	<	<	<		<	<	<	<	<	<	<	<	<	