FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Dokumentácia k projektu do predmetov IFJ a IAL
Implementácia interpretu jazyka IFJ16
Tým 039, varianta a/4/I

Vedúci týmu:	Sebastián Kisela	xkisel02	45%
Ďalší členovia:	Ondrej Svoreň	xsvore01	35%
	Daniel Rudík	xrudik00	20%

Obsah

1 Úvod

Táto dokumentácia popisuje implementáciu interpretu imperatívneho jazyka IFJ16, ktorá je zároveň zadaním projektu do predmetov IFJ a IAL. Jazyk IFJ16 je zjednodušenou podmnožinou jazyka Java SE 8. Projekt pozostáva zo štyroch hlavných častí:

- Lexikálna analýza
- Sytaktická analýza
- Sémantická analýza
- Interpret

2 Práca v tíme

2.1 Rozdelenie práce na jednotlivých častiach

- Sebastián Kisela – lexikálna analýza, spracovanie výrazov (precedenčná analýza) a volanie funkcií, interpret
- Ondrej Svoreň – syntaktický a sémantický analyzátor
- Daniel Rudík – vstavané funkcie, dokumentácia

2.2 Priebeh vývoja

Pre uchovanie už vytvoreného obsahu sme použili verzovací systém git na serveri jedného z členov tímu, kde sme dokumentovali dôležité zmeny v projekte, ktoré boli uložené v repozitári, do ktorého mali prístup všetci členovia tímu. Proces testovania prebiehal pomocou jednoduchého skriptu, ktorý spracovával nami vytvorené testy a upozorňoval na prípadné nedostatky programu. Problémom bolo, že aj napriek tomu, že sme začali pracovať dostatočne skoro, dvaja členovia ktorí mali na starosti implementáciu lexikálnej analýzy, to neboli schopní dokončiť po dlhú dobu, čo nás zdržalo.

3 Implementácia interpretu jazyka ifj16

Všetky vyššie uvedené časti interpretu boli vyvýjané relatívne nezávisle jedna na druhej. Najobjemnejšou a dá sa povedať najdôležitejšou častou projektu bol syntaktický analyzátor, ktorý vo výsledku spolupracuje s lexikálnym analyzátorom a komunikuje s ním pomocou tokenov reprezentujúcich lexémy načítavané zo zdrojového súboru. Na druhej strane komunikuje s precedenčnou analýzou v prípade potreby spracovania výrazu. V priebehu analýzy programu sa začína tvoriť inštrukčná páska programu, na ktorú sú ukladané všetky informácie potrebné pre vykonanie akcií uvedených v zdrojovom kóde, ktoré sú na záver interpretované pomocou samotného interpretu.

3.1 Lexikálna analýza

Scanner tvorí úvodnú časť interpretu. Funkcia **get_token()** vracia premennú typu Ttoken*, ktorá ukazuje na štruktúru obsahujúce potrebné informácie, ktoré token v sebe musí niesť pre ďalšie jeho spracovanie. Pri volaní tejto funkcie, je takisto volaná aj funkcia **pushToken**(Ttoken * token), ktorá pridá aktuálne vracaný token do zásobníka, pre prípadnú budúcu potrebu zo strany parseru. Parser može získať token zo zásobníka, ak niekde v programe pred týmto volaním je volaná funkcia **unget_token**(int num), kde parameter tejto funkcie je počet tokenov, o ktorý je potrebné sa vrátiť vzad.

3.2 Syntaktická a sémantická analýza

3.2.1 Syntaktická analýza

Syntaktická analýza tvorí základ interpretu. Jej hlavnou úlohou je komunikácia s lexikálnou analýzou prostredictvom tzv. tokenov ktoré nesú informáciu o danom spracovanom lexéme. Následne podľa gramatických pravidiel LL gramatiky kontroluje syntaktickú správnosť analyzovaného kódu na základe predikcie a vytvára tak abstraktný syntaktický strom.

3.2.2 Precedenčná analýza

Precedenčná analýza našeho interpretu je závislá na pravidlách jazyku IFJ16, podľa ktorého bola vytvorená precedenčná tabuľka. Spracovanie výrazu je riadené vzťahmi medzi jednotlivými symbolmi v precedenčnej tabuľke. Veľká časť vygenerovaného trojadresného kódu pre interpret je vygenerovaná v tejto časti a každá inštrukcia vytvorená pre interpret je uložená na inštrukčnú pásku práve spracovávanej funkcie, alebo v prípade globálnych premenných, na inštrukčnú pásku triedy.

3.3 Interpret

Interpret pracuje s trojadresným kódom, vytvoreným precedenčnou analýzou a syntaktickým analyzátorom. Pracuje hlavne so zásobníkmi funkcií a so zásobníkmi globálnych premenných, v ktorých podľa potreby hľadá konkrétne premenné. Náš interpret používa 21 inštrukcií, z ktorých aritmetické inštrukcie sú spracovávané vo funkcií **math()**, v ktorej sú zároveň uskutočňované sémantické kontroly, kontroly inicializácie premennej alebo delenie nulou. Sémantické a inicializačné kontroly premenných su vykonávané aj v ďalších funkciách, napr. kontrola správnosti typu parametrov používaných vo funkciách. Pri volaní funkcie je spustená inštrukcia `INS_PUSH_TABLE`, ktorá zaistí vytvorenie kópie parametrov, pre zachovanie povodných hodnot premenných.

3.4 Algoritmy

3.4.1 Radiaci algoritmus - list-merge sort

Funkcia **ifj16.sort** je implementovaná ako algoritmus list-merge sort. Tento algoritmus je založený na postupnom rozdeľovaní reťazca na dve časti podľa stredového indexu, čím sa vytvoria polia o dĺžke 1. Tieto polia su ďalej predané pomocnej funkcii **merge**, kde sú spájané do väčších polí, až kým sa nevytvorí jedno pole dĺžky rovnej pôvodnému reťazcu. Počas spájania týchto polí sú porvnávané ich jednotlivé prvky a adekvátne zoraďované.

3.4.2 Vyhľadávanie podreťazca v reťazci - Knuth-Morris-Prattov algoritmus

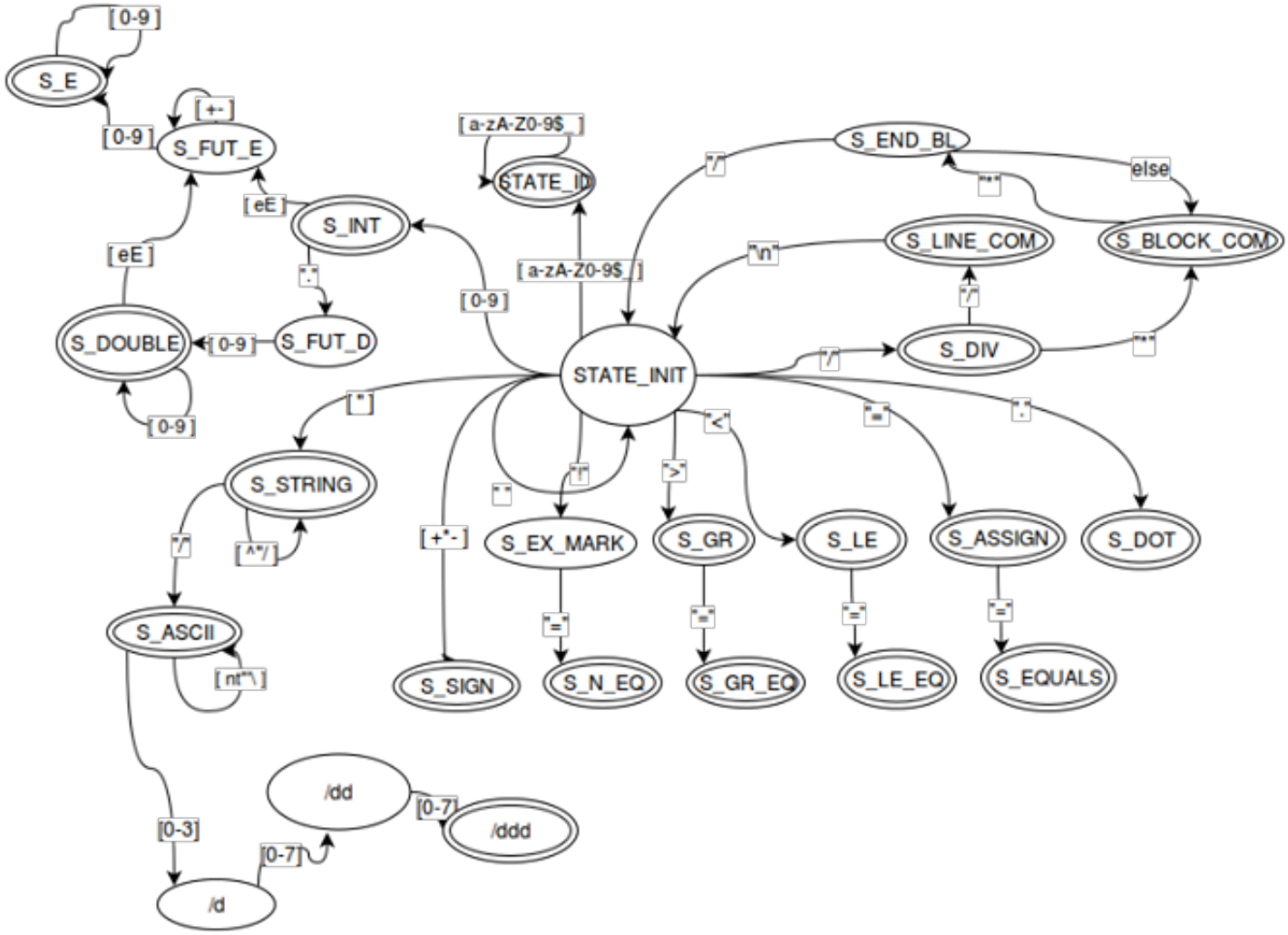
Funkcia **ifj16.find** je implementovaná pomocou Knuth-Morris-Prattovho alogritmu. Ten je založený na vytvorení tzv. fail vektoru - `fail_v` - čo je pole celých čísel, v ktorom je ku každému znaku z daného podreťazca priradené číslo, ktoré určuje index na ktorom má vyhľadávanie pokračovať v prípade nezhody znakov.

3.4.3 Tabuľka symbolov (vyhľadávanie pomocou binárneho stromu)

Tabuľka je tvorená binárnym stromom, pričom sú jednotlivé tabuľky symbolov reprezentované uzlami stromu, kde každý uzol obsahuje ukazateľ na ľavý podstrom, pravý podstrom a ukazateľ na koreňový uzol stromu resp. podriadenú tabuľku (napríklad ukazateľ tabuľky triedy na koreň tabuľky funkcie definovanej v danej triede a pod.).

4 Prílohy

4.1 Diagram konečného automatu



4.2 LL-gramatika

<starter> -> **class** id <class_body> <class_next>
<class_next> -> ϵ
<class_next> -> <starter>
<class_body> -> ϵ
<class_body> -> **static** <type> id <global_definition>
<global_definition> ;
<global_definition> -> = <constant_value>;
<global_definition> -> (<params>)<body_of_function>
<params> -> ϵ
<params> -> <type> id <params_next>
<params_next> -> ϵ
<params_next> -> , <type> id <params_next>
<body_of_function> -> ϵ
<body_of_function> -> <type> id <local_definition>
<local_definition> -> = <expression> ;
<body_of_function> -> <func_call>
<body_of_function> -> **if** (<condition>) { <body> } **else** { <body> }

<body_of_function> -> **while** (<condition>) { <body> }
<body_of_function> -> **return** <expression> ;
<body> -> ϵ
<body> -> id = <expression> ;
<body> -> <func_call>
<body> -> **if** (<condition>) { <body> } **else** { <body> }
<body> -> **while** (<condition>) { <body> }

	id	;	=	return	if	while	class	static	ϵ	,	(	func_call
<class_next>									2			
<class_body>								5	4			
<global_definition>		6	7								8	
<params>	10								9			
<body>	22				24	25			21			23
<params_next>									11	12		
<body_of_function>	14			20	18	19			13			17
<local_definition>		15	16									

4.3 Precedenční tabulka

stack \ input	+	-	*	/	<	<=	>	>=	==	!=	(	)	func	i	,	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	<	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	<	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>
<	<	<	<	<	$	$	$	$	$	$	<	>	<	<	<	>
<=	<	<	<	<	$	$	$	$	$	$	<	>	<	<	<	>
>	<	<	<	<	$	$	$	$	$	$	<	>	<	<	<	>
>=	<	<	<	<	$	$	$	$	$	$	<	>	<	<	<	>
==	<	<	<	<	$	$	$	$	$	$	<	>	<	<	<	>
!=	<	<	<	<	$	$	$	$	$	$	<	>	<	<	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	<	=	$
)	>	>	>	>	>	>	>	>	>	>	$	>	$	$	>	>
func	$	$	$	$	$	$	$	$	$	$	=	$	$	$	$	$
i	>	>	>	>	>	>	>	>	>	>	$	>	>	$	>	>
,	<	<	<	<	<	<	<	<	<	<	<	=	<	<	,	$
$	<	<	<	<	<	<	<	<	<	<	<	$	<	<	<	$

5 Referencie

[1] Prof. Ing. Jan Maxmilián Honzík, CSc. *Algoritmy IAL: Sudijní opora* [online]. Verze 14-N. 2014-01-06. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php?file=%2Fcourse%2FIAL-IT%2Flectures%2FOpora-IAL-2014-verze-14-N.pdf&cid=11418f