Dokumentace IFJ 2016

Tým 15, varianta b/4/I

11. prosince 2016

Vedoucí: Marek Barvíř (xbarvi00), 20%
Členové: Petr Malaník (xmalan02), 20%
Karel Ondřej (xondre09), 20%
David Smékal (xsmeka13), 20%
Jakub Sochůrek (xsochu01), 20%

Obsah

1 Úvod **1**

2 Implementace **2**
2.1 Lexikální analyzátor 2
2.2 Syntaktický analyzátor 2
2.2.1 Precedenční analýza 2
2.3 Sémantický analyzátor 2
2.4 Interpret 2

3 Implementace algoritmů z IAL **3**
3.1 Vyhledávací Boyer-Moorerův algoritmus 3
3.2 Řadící List-Merge sort algoritmus 3
3.3 Tabulka symbolů 3

4 Práce v týmu **3**
4.1 Schůzky a komunikace 3
4.2 Rozdělení 4

5 Závěr **4**
5.1 Literatura 4

1 Úvod

Tato dokumentace popisuje projekt do předmětů Formální jazyky a překladače a Algoritmy. Projekt je implementace interpretu imperativního jazyka IFJ16. Jazyk IFJ16 je podmnožina jazyka Java.

Překladač se skládá z pěti hlavních částí, které jsou Lexikální analyzátor, Syntaktický analyzátor, Sémantický analyzátor a Interpret. Tyto části jsou níže popsány v dalších kapitolách.

2 Implementace

2.1 Lexikální analyzátor

Lexikální analyzátor rozděluje načtený zdrojový kód v jazyce IFJ16 na lexémy. Jednotlivé lexémy jsou reprezentovány jako tokeny, které nesou informace o typu lexému, číslo řádku, na kterém se nachází a u identifikátorů, čísel a řetězců také atribut. V případě načtení chybného lexému, vrací lexikální analyzátor chybu a chybový token s informací o chybě.

Činnost lexikálního analyzátoru řídí syntaktický analyzátor, který si říká o tokeny. Lexikální analyzátor je implementovám pomocí konečného stavového automatu (viz Obrázek 1).

2.2 Syntaktický analyzátor

Syntaktický analyzátor nám kontroluje syntaktickou správnost programu. Řídí se LL gramatikou(viz Tabulka 1), která definuje správnost konstrukce.

Vstupem syntaktického analyzátoru jsou řetězce tokenů, které mu zasílá lexikální analyzátor. Poté se provádí simulace derivačního stromu, pokud je simulace úspěšná tak vyhodnotí program jako správný, při neúspěchu si uloží chybu. Pokud se v řetězci objevý výraz, předá ho precedenčnímu anylyzátoru.

2.2.1 Precedenční analýza

Precedenční anylýza nám vyhodnotí výraz, který je předán ze syntaktického analyzátoru. Vstup je řetězec tokenů, lokální a globální tabulka. Předaný výraz si zpracujeme pomocí precedenční tabulky (viz Tabulka 2). Až je celý výraz zpracován tak se převede do postfixové notace. Pokud se převedení nepovede, uloží si chybu. Precedenční analýza dále vygeneruje 3 adresný kód pro daný výraz, a když je potřeba vytvoří pomocné proměnné.

2.3 Sémantický analyzátor

Sématický analyzátor prochází vygenerovanou pásku a kontroluje kompabilitu datových typů při dané operaci a doplňuje správný datový typ u pomocných proměnných, u kterých není datový typ v době prvního průchodu znám. Dále kontroluje správný počet a typ parametrů funkce a návratových hodnot. V případě kontroli inicializační pásky navíc kontroluje, zda byli statické promměnné inicializovány.

2.4 Interpret

Interpret vykonává sekvenčně jednotlivé instrukce na instrukční pásce. Nejdříve jsou zavedeny do paměti statické proměnné z inicializační pásky a následně se spouší samotná instrukční páska programu. Instrukční páska programu začíná instrukcí zavolání funkce *Main.run* a následně instrukcí ukončení programu.

Před voláním funkce se uloží parametry funkce do dočasného rámce a při instrukci volání funkce se uloží dočasný rámec na vrchol datového uložiště, které je implementované jako zásobník. V rámci se vytvoří prostor pro lokální proměnné, informace o proměnné, kam se má uložit výsledek, a ukazatel, kde se má po skončení funkce pokračovat na pásce. Následně se skočí na místo na pásce, kde se nachází volaná funkce a pokračuje v programu. Při ukončení

funkce se uloží výsledek do návratové proměnné, uvolní se rámec v datovém uložišti a provede se skok na uložené místo na pásce.

3 Implementace algoritmů z IAL

3.1 Vyhledávací Boyer-Moorerův algoritmus

Funkci *find*, která slouží pro vyhledávání podřetězce v řetězci jsme implementovali pomocí Boyer-Moorova algoritmu. Tento algoritmus nám nabízí několik heuristik. Pro naši implementaci jsme si vybrali heuristiku "bad-character".

Tato heuristika využívá znalost nejpravějšího výskytu znaku. Od tohoto znaku se odvodí o kolik se bude moci porovnávaný řetězec posunout. Při neshodě na prvním porovnávaném znaku se posune o celou délku porovnávaného řetězce. Pokud se neshoda vyskytne dál posune se za pozici špatného znaku.

3.2 Řadící List-Merge sort algoritmus

List-Merge sort je metoda založená na slučování s využitím řazení bez přesunu položek. V prvním kroku si vytvoříme pomocné pole, obsahující index na další člen neklesající posloupnosti a začátky posloupností si uložíme do seznamu. V následujích kroku si do každé položky seznamu začátků, doplníme seznam následujících položek posloupnosti. Dále ze seznamu vyzvedneme začátky dvou posloupností a setřídíme je, tím získáme jednu setřízenou posloupnost, kterou vložíme nakonec seznamu posloupností. Algoritmus opakujeme tak dlouho, dokud nebudeme mít pouze jeden seznam neklesající posloupnosti. Až tohoto dosáhneme řazení je hotovo.

3.3 Tabulka symbolů

Tabulka symbolů je implementována pomocí binárního stromu. Jednotlivé funkce pro práci z binárním stromem jsou napsány rekurzivně. Pří zařazování jednotlivých prvků do stromu se rozhoduje podle názvu proměnné. V jednotlivých prvcích jsou uloženy informace o typu proměnné nebo funkce, inicializaci a definování proměnné a velikosti alokované paměti.

V programu vytváříme dvě tabulky. Jedna tabulka je globální, do které se uloží název classy a pomocí full id se uloží funkce a globální proměnné. Druhá tabulka je lokální a do té si ukládáme lokální proměnné.

4 Práce v týmu

4.1 Schůzky a komunikace

Během našeho projektu probíhaly schůzky každý týden. Na schůzkách se řešilo kdo co udělal a popřípadě se přerozdělovali úkoly. Mezi komunikační kanál na dálku jsme volili Facebook a Skype.

4.2 Rozdělení

Marek Barvíř - Parser, vestavěné funkce, lokální a globální tabulka, IAL algoritmy
Petr Malaník - Garbage collector, komplementace, testování
Karel Ondřej - Lexikální analyzátor, interpret, uložiště, sémantická analýza
David Smékal - Precedenční analýza, vestavěné funkce, IAL algoritmy, prezentace
Jakub Sochůrek - Precedenční analýza, IAL algoritmy, dokumentace

5 Závěr

Práce na projektu nám přinesla mnoho zkušeností. Mezi hlavní patří vývoj aplikace v týmu. V projektu se neobjevili žádné větší problémy, když se nějáké vyskytly tak jsme si dovedli poradit. Projekt byl časově posunut trošku dopředu, tak jsme si museli látku studovat dopředu.

Také jsme chtěli využít možnost pokusného odevzdání, což nám při prvním pokusu časově vyšlo Také by jsme chtěli poděkovat za možnost druhého pokusného odevzdání, které pro nás bylo velkým přínosem.

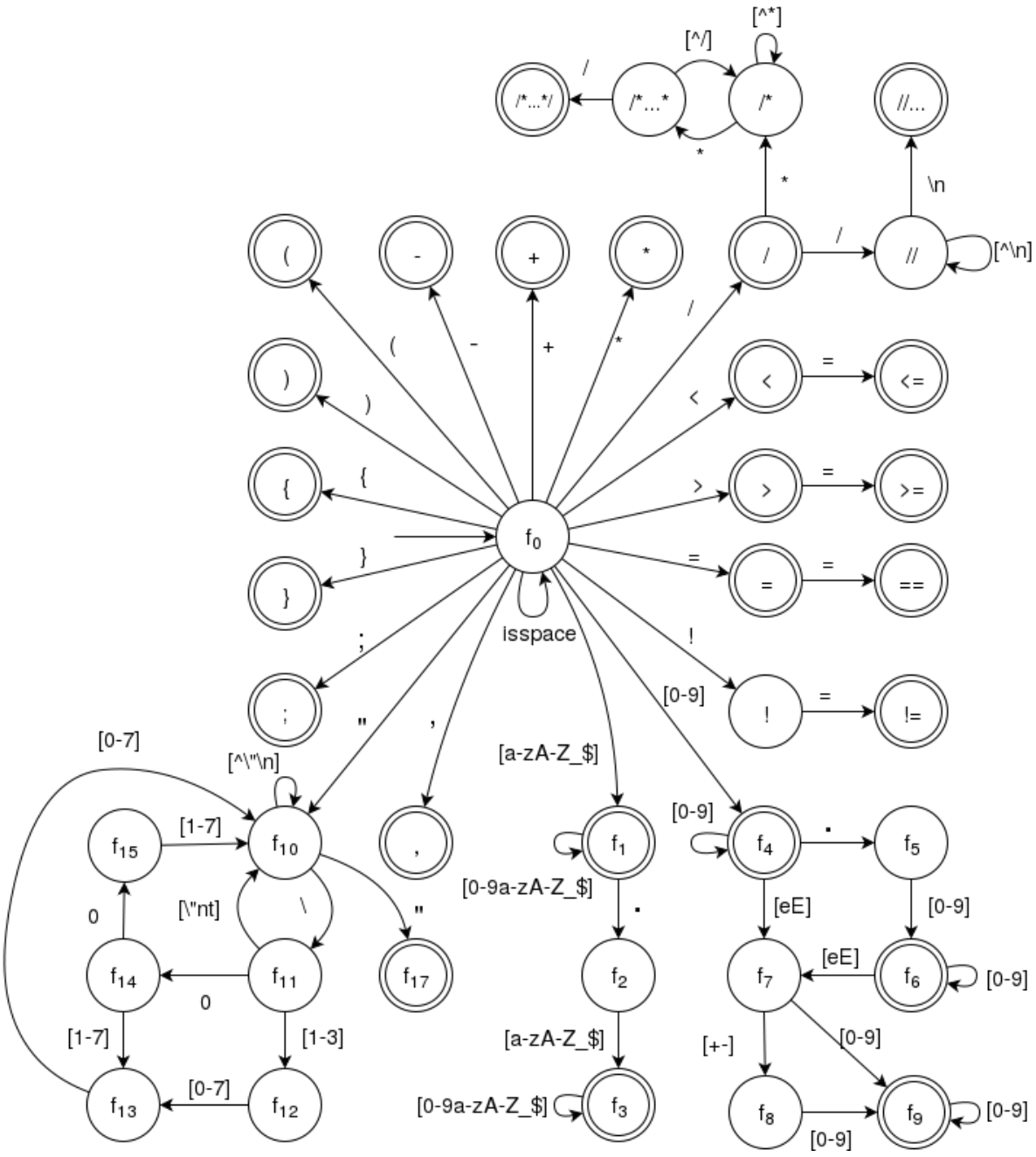
5.1 Literatura

[1] Skripta k předmětu IFJ

[2] Skripta k předmětu IAL

[3] Generátor 3AK: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/

Přílohy:



Obrázek 1: Stavový automat

<program> → "class" id "{" <class_body> "}" <program>
<program> → End of file

<class_body> → ϵ
<class_body> → "static" <type>id ";" <class_body>
<class_body> → "static" <type>id -" <expression> ";" <class_body>
<class_body> → "static" <type>id "(" <param> ")" "{" <func_body> "}" <class_body>

<type> → "int"
<type> → "String"
<type> → "double"
<type> → "void"

<var_type> → "int"
<var_type> → "double"
<var_type> → "String"

<param> → ϵ
<param> → <param_n>

<param_n> → <var_type>id
<param_n> → <var_type>id "," <param_n>

<func_body> → ϵ
<func_body> → <var_type>id ";" <func_body>
<func_body> → <var_type>id -" <expression> ";" <func_body>
<func_body> → <var_type>id -" <call_func> ";" <func_body>
<func_body> → <while><func_body>
<func_body> → <if><func_body>
<func_body> → "return" <expression> ";" <func_body>
<func_body> → <call_func> ";" <func_body>
<func_body> → id -" <expression> ";" <func_body>
<func_body> → id -" <call_param> ";" <func_body>

<while> → "while" "(" <condition> ")" "{" <if_while_body> "}"

<if> → if "(" <condition> ")" "{" <if_while_body> "}" "else" "{" <if_while_body> "}"

<if_while_body> → ϵ
<if_while_body> → <while><if_while_body>
<if_while_body> → <if><if_while_body>
<if_while_body> → "return" <expression> ";" <if_while_body>
<if_while_body> → <call_func> ";" <if_while_body>
<if_while_body> → id -" <expression> ";" <if_while_body>
<if_while_body> → id -" <call_func> ";" <if_while_body>

<call_func> → id "(" <call_param> ")"
<call_func> → full_id "(" <call_param> ")"

<call_param> → ϵ
<call_param> → <call_param_n>

<call_param_n> → <call_param_type>
<call_param_n> → <call_param_type> "," <call_param_n>

<call_param_type> → id
<call_param_type> → full_id
<call_param_type> → integer
<call_param_type> → String
<call_param_type> → double

<expression> → <expression> "+" <expression>
<expression> → <expression> "-" <expression>
<expression> → <expression> "*" <expression>
<expression> → <expression> "/" <expression>
<expression> → "(" <expression> ")"
<expression> → integer
<expression> → double
<expression> → string
<expression> → id
<expression> → full_id

<condition> → "(" <condition> ")"
<condition> → <condition> "+" <condition>
<condition> → <condition> "-" <condition>
<condition> → <condition> "*" <condition>
<condition> → <condition> "/" <condition>
<condition> → <condition> "<" <condition>
<condition> → <condition> ">" <condition>
<condition> → <condition> "<=" <condition>
<condition> → <condition> ">=" <condition>
<condition> → <condition> "!=" <condition>
<condition> → <condition> "==" <condition>
<condition> → integer
<condition> → double
<condition> → string
<condition> → id
<condition> → full_id

Tabulka 1: LL-gramatika

	EM	IN	DO	ST	SI	FQ	=	*	/	+	-	<	<=	>	>=	!=	==	(	)	{	}	;	,
EM																							
IN								>	>	>	>	>	>	>	>	>	>		>			>	
DO								>	>	>	>	>	>	>	>	>	>		>			>	
ST								>	>	>	>	>	>	>	>	>	>		>			>	
SI								>	>	>	>	>	>	>	>	>	>	F	>			>	
FQ								>	>	>	>	>	>	>	>	>	>	F	>			>	
=																							
*		<	<	<	<	<		>	>	>	>	>	>	>	>	>	>	<	>			>	
/		<	<	<	<	<		>	>	>	>	>	>	>	>	>	>	<	>			>	
+		<	<	<	<	<		<	<	>	>	>	>	>	>	>	>	<	>			>	
-		<	<	<	<			<	<	>	>	>	>	>	>	>	>	<	>			>	
<		<	<	<	<	<		<	<	<	<	=	=	=	=	>	>	<	>			>	
<=		<	<	<	<	<		<	<	<	<	=	=	=	=	>	>	<	>			>	
>		<	<	<	<	<		<	<	<	<	=	=	=	=	>	>	<	>			>	
>=		<	<	<	<	<		<	<	<	<	=	=	=	=	>	>	<	>			>	
!=		<	<	<	<	<		<	<	<	<	<	<	<	<	=	=	<	>			>	
==		<	<	<	<	<		<	<	<	<	<	<	<	<	=	=	<	>			>	
(		<	<	<	<	<		<	<	<	<	<	<	<	<	<	<	<	=				
)								>	>	>	>	>	>	>	>	>	>		>			>	
{																							
}																							
;		<	<	<	<	<		<	<	<	<	<	<	<	<	<	<	<					
,																							

Tabulka 2: Precedenční tabulka