VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu

do předmětu IFJ a IAL

Implementace interpretu imperativního jazyka IFJ16

Tým 029, varianta b/3/I

Jiruška Adam, xjirus01, 50% - vedoucí
Janeček David, xjanec28, 50%
Kuba Michal, xkubam02, 0%
Karpíšek Miroslav, xkarpi05, 0%

Obsah

1. Úvod

Tato dokumentace popisuje tvorbu interpretu jazyka IFJ16, který je velmi zjednodušenou podmnožinou jazyku Java SE 8. Konkrétně se jedná o zadání b/3/I. V tomto zadání je implementace vyhledávání podřetězce v řetězci řešena pomocí Boyer-Moorova algoritmu a řazení řetězce pomocí metody Shell sort. Tabulka symbolů je implementována pomocí binárního vyhledávacího stromu.

2. Algoritmy

2.1 Boyer-Moore

Boyer-Mooreův algoritmus umožňuje velice rychlé vyhledávání podřetězce v řetězci. Základní myšlenkou algoritmu je, že některé znaky, které se nikdy nemohou rovnat hledanému řetězci, lze přeskočit. Využili jsme „Bad Character" heuristiku. Princip spočívá v tom, že když narazíme na znak řetězce, který není shodný se znakem v podřetězci, posuneme podřetězec tak, aby porovnávaný znak byl na úrovni posledního výskytu tohoto znaku v daném podřetězci. Podřetězec je tedy potřeba zpracovat dopředu a uložit takové pozice pro všechna písmena. Pokud se znak nevyskytuje v řetězci vůbec, pak posouváme o délku podřetězce.

2.2. Shell sort

Shell sort je řadící algoritmus, který neřadí pouze prvky vedle sebe, ale s určitou mezerou. V našem případě začínáme na mezeře rovné polovině celkové délky řetězce.
S každým průchodem se mezera snižuje na polovinu, až dosáhne vzdálenosti jedna.

2.3. Tabulka symbolů

Pro implementaci tabulky symbolů jsme využili binárních stromů (BS). Naši tabulku jsme rozdělili na několik částí. Nejdříve BS, ve kterém každý uzel obsahuje informace o třídě, ukazatele na synovské uzly a ukazatele na dva další binární stromy. Jeden z nich je BS statických proměnných definovaných v dané třídě, a druhý funkcí v dané třídě. BS funkcí obsahuje kromě základních informací ukazatel na BS lokálních proměnných dané funkce.

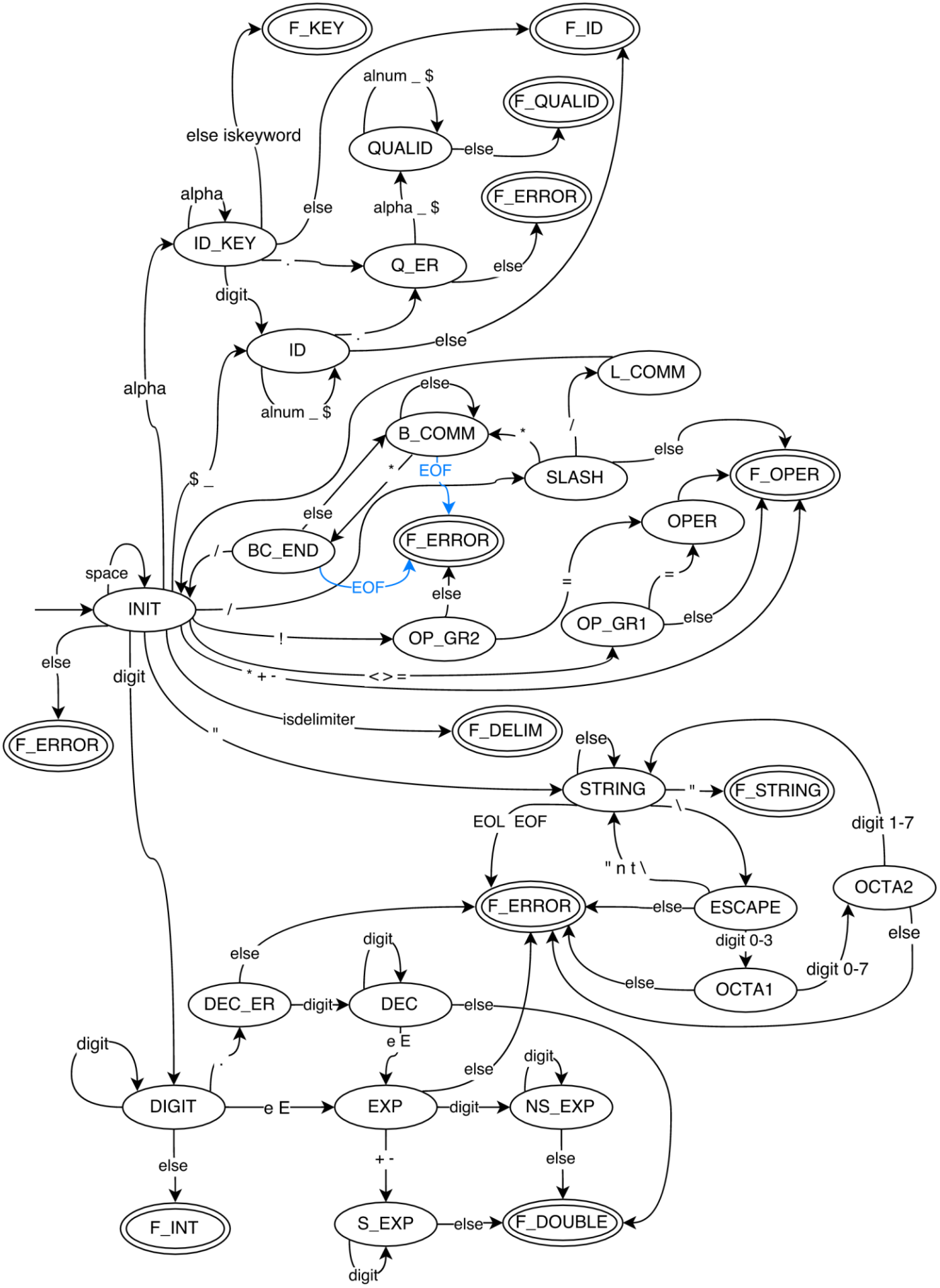
3. Implementace interpretu

3.1 Lexikální analyzátor

Lexikální analyzátor (scanner) je založen na deterministickém konečném automatu (viz 3.1.1.) a je to jediná část interpretu pracující přímo se vstupním souborem. Jeho úkolem je ze vstupního textu odstranit nepotřebné části (bílé znaky, komentáře) a zbylé lexémy reprezentovat pomocí tokenů.

Scanner je ovládán syntaktickým analyzátorem, který si žádá o tokeny. Při implementaci jsme využili vzorovou knihovnu str.c z ukázky jednoduchého interpretu.

3.1.1 Deterministický konečný automat



3.2 Syntaktický analyzátor

Syntaktický analyzátor funguje na principu rekurzívního sestupu. Podle tokenu, který obdrží od lexikálního analyzátoru, se rozhoduje, které z pravidel LL gramatiky má použít. Při tvorbě gramatiky (viz 3.2.1) bylo potřeba zohlednit, aby šlo pravidlo vybrat vždy jednoznačně. Další problém, který bylo potřeba vyřešit, byla možnost použití proměnné před její definicí. Nejprve jsme zkoušeli do tabulky symbolů ukládat záznamy o použití a následně je provázat s definicí, ale nakonec jsme zvolili dvouprůchodovou verzi syntaktické analýzy.

Další částí syntaktického analyzátoru je precedenční analýza, používaná ke zpracování výrazů. Ta pracuje se zásobníkem neterminálů a terminálů, a také s precedenční tabulkou (viz 3.2.2). Podle této tabulky se provádějí jednotlivá pravidla a tvoří kód pro provedení výrazu.

3.2.1 LL gramatika

<prog> -> CLASS <class> <classes>
<classes> -> CLASS <class> <classes>
<classes> -> EPS
<class> -> ID { <def_list> }
<def_list> -> STATIC <definition> <def_list>
<def_list> -> EPS
<definition> -> DATA_TYPE ID <definition_rest>
<definition_rest> -> ;
<definition_rest> -> = <expression> ;
<definition_rest> -> (<param_list>) <body>
<param_list> -> DATA_TYPE ID <param_rest>
<param_rest> -> , DATA_TYPE ID <param_rest>
<param_rest> -> EPS
<body> -> { <stat_list> }
<stat_list> -> <stat> <stat_list>
<stat_list> -> EPS
<stat> -> EPS
<stat> -> DATA_TYPE ID <definition_rest>
<stat> -> ID <stat_rest>
<stat> -> { <stat_list> }
<stat> -> IF (<expression>) <body> ELSE <body>
<stat> -> WHILE (<expression>) { <stat_list> }
<stat> -> RETURN <expression> ;
<arguments> -> <arg> <arguments_rest>
<arguments_rest> -> , <arg> <arguments_rest>
<arguments_rest> -> EPS
<arg> -> TERM

3.2.2 Precedenční tabulka

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

3.3 Sémantický analyzátor

Sémantický analyzátor obsahuje funkce pro kontrolu sémantiky. Tyto funkce jsou volány syntaktickým analyzátorem. V sémantické analýze je využívána tabulka symbolů. Nejčastějšími kontrolami jsou kontroly deklarací a definicí funkcí a proměnných, kompatibility datových typů. Sémantický analyzátor se nám bohužel nepodařilo dotáhnout do konce.

3.4 Interpret

Interpret je část programu vykonávající kód, který byl vygenerován v průběhu syntaktické analýzy. V naší implementaci je interní kód rozdělen na pásky pro jednotlivé funkce, mezi kterými se interpret přepíná při provádění programu. Při volání funkcí je také důležité vytvořit si rámec funkce, kde jsou uložené hodnoty lokálních proměnných pro danou funkci, také je tam návratová adresa a místo, kam se vrací hodnota. Tento rámec se vkládá na zásobník rámců, kde vždy na vrcholu je rámec aktuálně prováděné funkce. K dispozici jsou také rámce poslední prováděné funkce a funkce, která bude zavolána (připravovaný rámec).

4. Práce v týmu

Z pohledu práce v týmu si z tohoto projektu bohužel odnášíme velice negativní zkušenost. První chybou bylo, že jsme promrhali září a říjen, kdy jsme se nemohli domluvit na termínech schůzek a moc to neřešili, protože bylo „stále dost času“. S příchodem listopadu si většina z nás uvědomila, že je nejvyšší čas začít něco dělat. Práci jsme si rozdělili následovně:

xjirus01: interpret
xjanec28: lexikální analýza + co bude potřeba
xkubam02: sémantická analýza
xkarpi05: syntaktická analýza

Hned při domluvě komunikace jednotlivých částí interpretu nastaly hlavní problémy. Míra, který měl dělat syntaktickou analýzu, na všechny dotazy a návrhy odpovídal, že je jedno jak to uděláme, že on poté synt. analýzu udělá tak, aby to fungovalo. To se nám příliš nelíbilo. Po čase jsme u něho stále neviděli žádný pokrok. Všichni jsme mu psali zprávy a chtěli se sejít a domluvit. On schůzky neustále oddaloval. Když jsme se konečně dohodnuli na termínu, tak nepřišel. Poté neodpovídal už ani na zprávy a nakonec nám řekl, že mu „ujel vlak“ a „IFJ si zopakuje za rok“. Do projektu nevypracoval vůbec nic. Michal řekl, že ve třech nemáme šanci projekt dokončit a také skončil. Odeslal nám nedokončený Boyer-Mooreův algoritmus. Zůstali jsme tedy, na celý projekt, dva… Ke společné práci jsme využívali github, skype, facebook.

5. Závěr

Z projektů se kterými jsme se na FIT VUT doposud setkali, byl tento rozhodně nejobtížnější. Především jeho obsáhlost a celková časová náročnost. Toto bylo umocněno výpadkem poloviny týmu. Podle toho vypadá i výsledek celého projektu, ze kterého jsme nestihnuli odstranit množství chyb a nedostatků. Odnesli jsme si mnohá ponaučení. Především to, že při práci ve skupině je třeba si stanovit jasně daná pravidla, podle kterých se budou všichni členové postupovat.