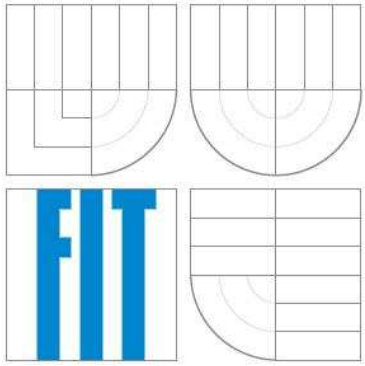
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta Informačních Technologií



FORMÁLNÍ JAZYKY A PŘEKLADAČE

2009/2010

Semestrální projekt

Implementace překladače imperativního jazyka IFJ09

Tým 2, varianta a/4/I

Rozšíření:

- Vestavěné funkce ve výrazech a výrazy v jejich parametrech – 1b
- Řádkový komentář – 0.5b

Autor:
Kubiš Radim, xkubis03 : 25%
Lukáš Radek, xlukas07 : 25%
Blaho Vojtěch, xblaho01 : 25%
Břenek Roman, xbrene02 : 25%

Obsah

Úvod

Kód obsahuje překladač, který interpretuje načtený zdrojový soubor v jazyce IFJ09. V případě bezchybného překladu se vrací výsledek programu. Jestliže však během zpracovávání došlo k chybě, vrací se návratová hodnota dle patřičné chyby.

V projektu jsme měli použít pro vyhledávání metodu Knuth-Morris-Prattova algoritmu. Dále bylo požadováno v určitých chvílích seřadit řetězec. Toto řazení je implementováno metodou Merge-sort. Tabulku symbolů jsme implementovali pomocí binárního vyhledávacího stromu.

1.1 Lexikální analyzátor

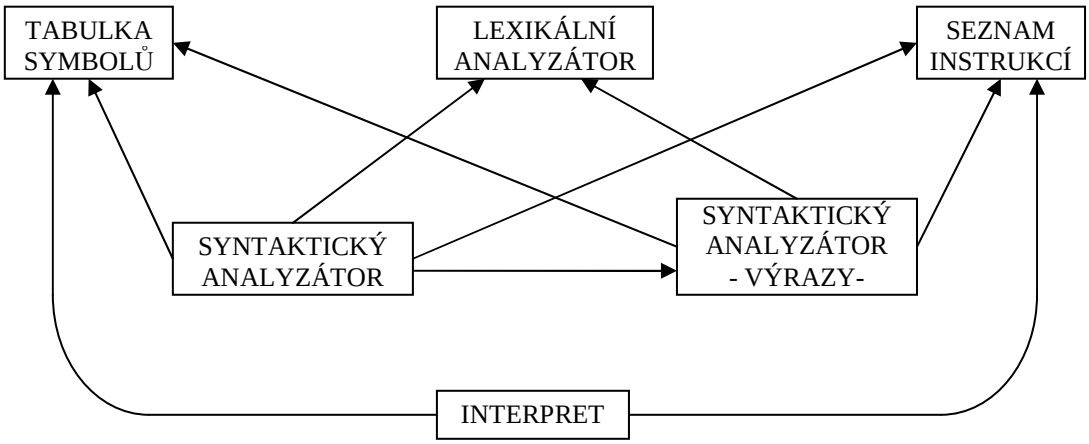
Samotný lexikální analyzátor načítá zdrojový kód ze zdrojového souboru v jazyce IJF09. Dále tento zdrojový kód dělí na jednotlivé lexémy (jsou reprezentovány tokeny), které později předá syntaktickému analyzátoru.

1.2 Syntaktický analyzátor

Syntaktický analyzátor bude dostávat od Lexikálního analyzátoru tokeny, ze kterých se pokusí sestavit derivační strom. Pokud se mu podaří sestavit derivační strom, program je korektní. Dále kontroluje program po stránce sémantické (kontrola datových typů, kontrola deklarací proměnných). Výstupem je abstraktní syntaktický strom.

1.3 Interpret

Vlastní Interpret přijímá jednotlivé instrukce, které následně vykonává.



Obr.1 Vývojový cyklus

Lexikální analyzátor

Funkcí lexikálního analyzátoru je rozdělit zdrojový program (sled příkazů) na jednotlivé lexémy, které jsou reprezentovány tokeny.

2.1 Postup

Snažili jsme se vytvořit deterministický konečný automat, dále jen DKA, složený z grafické a kódové části. DKA přijímá všechny identifikátory, klíčová slova, celočíselné konstanty, desetinné konstanty, řetězcové konstanty, komentáře, výrazy a operátory.

2.2 Grafická část

Základ grafické části byl založen na výpisu všech pravidel, který bude konečný automat, dále jen KA, přijímat. Podařilo se nám vytvořit 17 pravidel, díky nimž jsme zkonstruovali KA. Tato pravidla nebyla definitivní. V průběhu vytváření a testování interpretu byly zjišťovány nedostatky v těchto pravidlech a musela být nutně změněna. Dalším krokem při vytváření grafické části bylo odstranění stejných přechodů typu pq->q a získání DKA.

2.3 Kódová část

Vytvoření kódové části bylo snazší. Při tvorbě jsme postupovali dle přílohy A. Syntaktický analyzátor volá funkci getNextToken, která čte zdrojový program po znacích v cyklu. Porovnávacím příkazem switch, dle stavu, ve kterém se nachází, provádí následný kód. Tento kód může ukládat načtený znak do atributu attr, může aktuální stav změnit, načtený znak vrátit či vrátit příkazem return odpovídající hodnotu zpět syntaktickému analyzátoru.

2.4 Vlastní implementace

Při vytváření lexikálního analyzátoru jsme neměli větší problém. Vycházeli jsme z loňských zkušeností a nebyl větší problém tuto část implementovat. Setkali jsme se jen se dvěma drobnými problémy. Jedením bylo chybné parsrování číselných konstant a druhým bylo nesprávné načítání středníků. Oby tyto nedostatky jsme ale bez větších problémů odstranily.

Syntaktický analyzátor

Syntaktický analyzátor je hlavní část překladače. Kontroluje gramatiku daného jazyka. V našem případě se jedná o LL gramatiku, která je popsána níže. Vstupním souborem jsou jednotlivé tokeny, které přijímá od lexikálního analyzátoru a výstupem pak abstraktní syntaktický strom.

Kromě syntaktické kontroly jazyka, se zde provádí i některé sémantické kontroly a vytváří se trojadresný kód jednotlivých instrukcí pro interpret.

Syntaktická analýza (SA) se dá rozdělit na dvě části:

3.1 Hlavní syntaktická analýza

3.1.1 Postup

Tuto SA jsme řešili metodou rekurzivního sestupu, tj. shora dolů. Nejdříve jsme napsali binární vyhledávací strom, kterým je reprezentována tabulka symbolů a upravili vzorovou instrukční tabulku. Poté jsme se vrhli na samotnou analýzu. Sepsali jsme si všechny příkazy, které se v jazyku IFJ09 mohou vyskytovat a napsali pro ně pravidla (soubor těchto pravidel je v příloze). Podle těchto pravidel jsme pak začali psát samotný kód. Poté jsme přidali některé sémantické kontroly a tvorbu trojadresného kódu.

3.1.2 Komplikace

Syntaktický analyzátor byl společně se syntaktickým analyzátorem pro výrazy jednoznačně nejnáročnější částí celého překladače a při jeho implementaci jsme řešili nejvíce problémů. Tyto problémy vycházeli především ze zvyku pracovat s programovacím jazykem C a vytvářet překladač pro jazyk Pascal pro nás bylo o něco obtížnější právě kvůli jeho syntaxi. Během programování těchto částí byl kladen velký důraz na komunikaci členů týmu, kteří se snažily obě části napsat tak, aby se mezi sebou správně dorozumívali.

3.2 Syntaktická analýza pro výrazy

Pokud hlavní syntaktická analýza narazí na výraz, zavolá na jeho vyřešení syntaktickou analýzu pro výrazy. Zde se pracuje metodou zdola nahoru. Generování trojadresného kódu pro všechny instrukce, vyjma CIN, se děje právě zde.

3.2.1 Postup

Pro zpracování výrazů ve zdrojovém souboru jazyka IFJ09 bylo potřeba vytvořit podprogram hlavního syntaktického analyzátoru, který pracuje metodou zdola nahoru. Tato metoda zaručuje správné vyhodnocení výrazu podle asociativity a priority jednotlivých operátorů, proto jsme si museli vytvořit precedenční tabulku. Protože tento syntaktický analyzátor pracuje se zásobníkem, museli jsme jej také vytvořit. V rámci řešení výrazu se také generuje trojadresný kód pro interpret a dělají se sémantické kontroly. Redukce jednotlivých operací na mezivýsledky probíhala podle navržených pravidel.

3.2.2 Komplikace

Při řešení této části jsme se nejdříve snažili vzít loňský program, ten poupravit a použít pro letošní zadání. Po dlouhé době úsilí a nefunkčnosti jsme se rozhodli celý syntaktický analyzátor pro výrazy napsat znovu. Později se toto rozhodnutí ukázalo jako rozumné, protože po napsání nového kódu jsme konečně zprovoznili danou část programu.

3.3 Pravidla pro LL gramatiku

```
<parse> <declList> <statList> <EOF>
```

```
<declList> var <declList>
<declList> id : <type_of_id>
   <type_of_id> string ; <decllist>
   <type_of_id> integer ; <decllist>
   <type_of_id> double ; <decllist>
<declList> begin <statList>
```

```
<statList> <stat> <statList>
<statList> begin <statList>
<statList> end
```

```
<stat> ; <stat>
<stat> <vyraz>
<stat> readln ( <stroke_or_right_parenthesis>
   <stroke_or_right_parenthesis> , id <stroke_or_right_parenthesis>
   <stroke_or_right_parenthesis> )
<stat> write ( <stroke_or_expr>
   <stroke_or_expr> <vyraz> , <stroke_or_expr>
   <stroke_or_expr> )
<stat> while <vyraz> do <stat>
<stat> while <vyraz> do begin <statList>
<stat> if <vyraz> then <stat> else <stat>
<stat> if <vyraz> then begin <statList> else <stat>
<stat> if <vyraz> then <stat> else begin <statList>
<stat> if <vyraz> then begin <statList> else begin <statList>
```

Interpret

Interpret je konečné zařízení, které pouze vykonává samotné instrukce. Spolupracuje s tabulkou instrukcí a s tabulkou symbolů. Z těchto dvou tabulek vezme vždy patřičnou instrukci a symboly (proměnné) a tyto vlastnosti vyhodnotíme.

4.1 Knuth-Morris-Prattův algoritmus

V projektu jsme měli použít pro vyhledávání metodu Knuth-Morris-Prattova algoritmu. Metoda slouží pro vyhledání podřetězce v řetězci. Je rychlejší než náhodný algoritmus, jelikož se nevrací v textu zpět.

Zakládá se na vytvoření pomocného pole (vektoru), který uchovává čísla. Čísla určují o kolik míst se může posunout "ukazatel" v prohledávaném řetězci jestliže po pár (jednom a více) shodných prvcích nastala nějaká neshoda u prvků řetězce a podřetězce a současně dosud nebyl nalezen hledaný podřetězec.

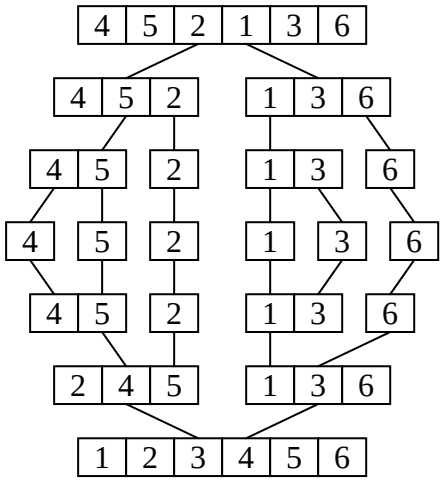
Až po vytvoření tohoto vektoru může dojít na vlastní vyhledávání. Zakládá se na průchodu polem, kde se porovnávají jednotlivé prvky prohledávaného pole s prvním prvkem pole hledaného. V případě, že se tyto prvky rovnají, začnou se porovnávat i zbylé prvky podřetězce s následujícími prvky prohledávaného pole. Jestliže nastane nějaká neshoda u prohledávání dalších prvků zjistíme na kterém prvku to bylo a poté zjistíme z dříve vytvořeného vektoru, o kolik se můžeme posunout.

1	2	3	4	5	6	7	8	9	…
a	b	a	b	a	b	a	b	a	
a	b	**a**	b	a	c				
		a	b	**a**	b	a	c		
				a	b	**a**	b	a	c
						a	b	**a**	b
								a	b

Obr.4 Schéma činnosti řadícího algoritmu (H.W.Lang, 2001)

4.2 Merge-sort

Je založen na myšlence "Rozděl a panuj". Řetězec, který má být seřazen, je rozdělen na dvě poloviny. Jednotlivé poloviny jsou na sobě nezávisle seřazeny. Následně jsou seřazené poloviny zatříděny do sebe. My jsme jej implementovali následujícím způsobem: řetězec je dělen na dvě poloviny tak dlouho, dokud neobsahuje jen jeden znak. Následně jsou tyto "podřetězce" zatříděny rekurzivně do sebe. K zatřiďování využíváme pomocného pole.



Obr.5 Schéma činnosti Merge-sort (Micka, 2008)

4.3 Vlastní implementace

Při implementaci této části projektu jsme vycházeli ze vzorového zadání na internetu. Metody na vyhledávání a třídění jsme nastudovali taktéž na webu. Vlastní implementace těchto metod nám nečinila větší problémy. Měli jsme spíše problémy s tím, že jsme nevěděli, v jaké formě nám syntaktický analyzátor pošle instrukce, neboť v době, když jsme pracovali na této části kódu, syntaktický analyzátor ještě nebyl zcela dokončen. Proto jsme museli tuto část několikrát opravovat.

Závěr

Jelikož tento předmět opakujeme a náš tým se skládá ze stejných členů jako minulý rok vycházeli jsme při práci převážně z loňského řešení. Z předchozího roku jsme se již poučili, že se nevyplácí dělat práci na poslední chvíli, a proto jsme se rozhodli začít na projektu pracovat hned, jak bude zadání oficiálně vyvěšeno.

Po zveřejnění projektu se náš tým sešel a každý člen si vzal nějaký základní úkol, který se byl nezbytně nutný k nastudování před začátkem vlastního programování, nebo se dal vytvořit samostatně bez funkčnosti ostatních částí. Mezi tyto úkoly patřily: tvorba precedenční tabulky, pravidla pro LL gramatiku, naprogramování řadícího a vyhledávacího algoritmu atd. Tyto malé schůzky jsme později asi třikrát zopakovali než jsme zahájili vlastní programování.

Další nemilou zkušeností z minulého roku bylo, že jsme měli neuvěřitelný problém s verzemi jednotlivých naprogramovaných částí. Proto jsme se rozhodli se scházet a pracovat na projektu společně. Tyto schůzky byly zpočátku dost nepravidelné, avšak jak se blížil čas odevzdání, scházeli jsme přibližně dvakrát týdně. Tyto schůzky trvaly přibližně šest hodin a společně jsme na nich ve dvojicích vytvářeli výsledný kód. Tím se nám podařilo z velké části problémy s verzemi odbourat. Bohužel i tak jsme častokrát řešili problém, která naprogramovaná část je nejaktuálnější a často se i stávalo, že na jedné části pracovalo více lidí současně.

Jako dalším pomocným krokem k přehlednosti při vytváření a zálohování projektu nám bylo vytvoření datového skladu, ve kterém jsme si uchovávali jednotlivé části programu. K tomuto datovému skladu měli všichni členové týmu přístup a proto si mohli kdykoliv stáhnout takřka nejnovější naprogramované části.

I přes brzký začátek jsem projekt dolaďovali i v posledních dnech těsně před odevzdáním. Velice nám pomohlo, že jsme znali spoustu věcí z loňských let. Na druhou stranu tím, že jsme při řešení vycházeli z loňského řešení pro nás byly některé věci těžko opravitelné a možná by bylo lepší celý kód napsat znovu, než vycházet z již rozepsaného programu.

5.1 Rozšíření

Nejdříve jsme se snažili správně napsat náš základní část překladače a pořádně náš projekt otestovat. Po těchto testech jsme se ještě rozhodli implementovat do naší práce nějaké rozšíření. Celkově se nám povedly správně zapsat dvě rozšíření a to: *Vestavěné funkce ve výrazech a výrazy v jejich parametrech a řádkový komentář.*

5.2 Informační zdroje

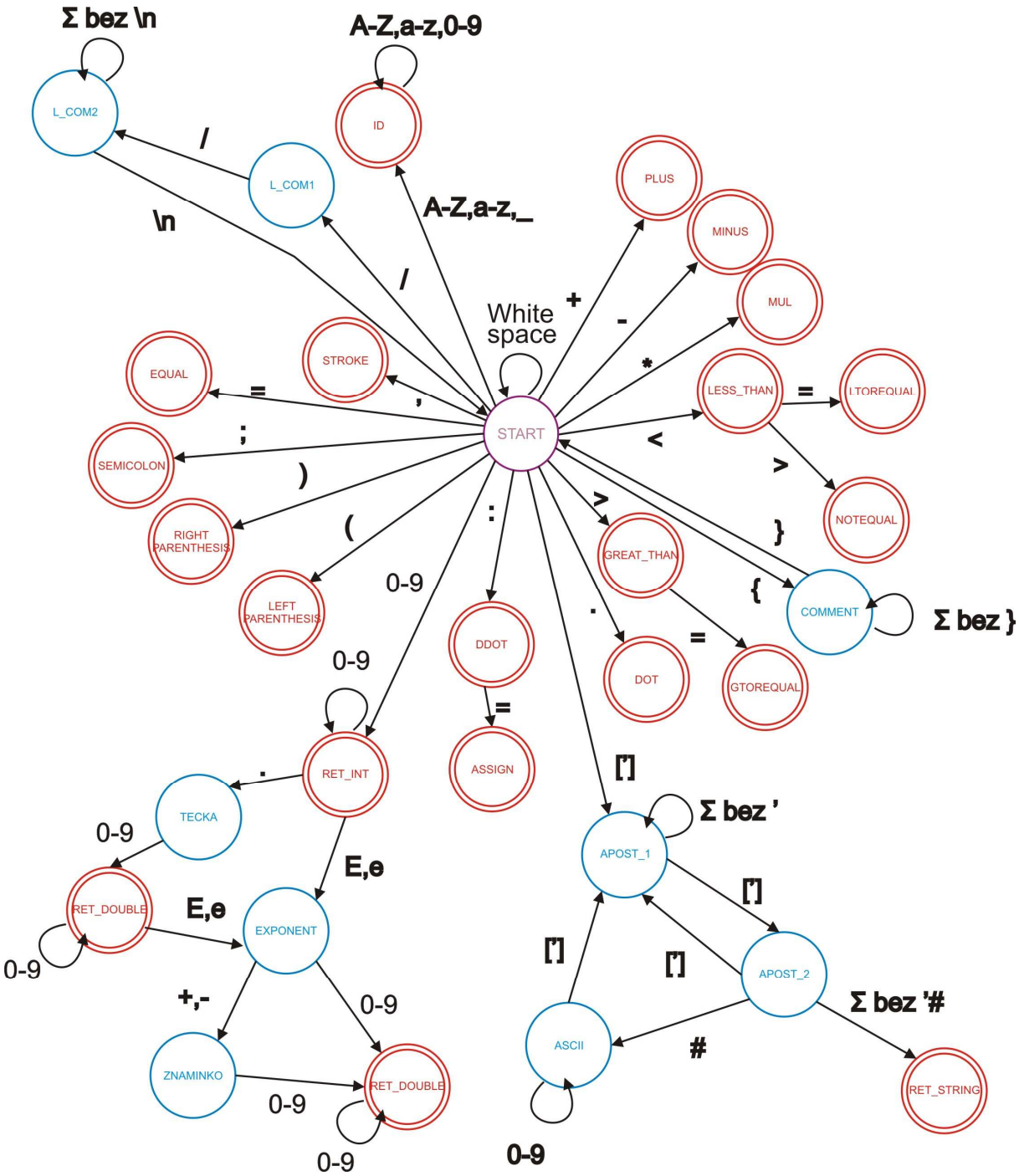
H.W.Lang. *Mergesort* [online]. 2000-01-19, 2008-01-25 [citováno 2009-12-09]. Dostupné z WWW: <http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm>

H.W.Lang. *Knuth-Morris-Pratt* [online]. 2001-05-16, 2008-01-27 [citováno 2009-12-09]. Dostupné z WWW: <http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm>

MEDUNA, Alexander. *Doplňující informace k předmětu IFJ* [online]. 2008-09-21 [citováno 2009-12-09]. Dostupné z WWW: <https://www.fit.vutbr.cz/study/courses/IFJ/public/project/>

Příloha A

Konečný automat pro lexikální analyzátor



Příloha B

Precedenční tabulka

	i	+	-	*	div	(	)	f	,	<	>	<=	>=	=	<>	:=	$
i		>	>	>	>		>		>	>	>	>	>	>	>	>	>
+	<	>	>	<	<	<	>	<	>	>	>	>	>	>	>	>	>
-	<	>	>	<	<	<	>	<	>	>	>	>	>	>	>	>	>
*	<	>	>	>	>	<	>	<	>	>	>	>	>	>	>	>	>
div	<	>	>	>	>	<	>	<	>	>	>	>	>	>	>	>	>
(	<	<	<	<	<	<	=	<	=	<	<	<	<	<	<	<	
)		>	>	>	>		>		>	>	>	>	>	>	>	>	>
f						=											
,	<	<	<	<	<	<	=	<		<	<	<	<	<	<	<	
<	<	<	<	<	<	<	>	<	>	>	>	>	>	>	>	>	>
>	<	<	<	<	<	<	>	<	>	>	>	>	>	>	>	>	>
<=	<	<	<	<	<	<	>	<	>	>	>	>	>	>	>	>	>
>=	<	<	<	<	<	<	>	<	>	>	>	>	>	>	>	>	>
=	<	<	<	<	<	<	>	<	>	<	<	<	<	>	>	>	>
<>	<	<	<	<	<	<	>	<	>	<	<	<	<	>	>	>	>
:=	<	<	<	<	<	<	>	<	>	<	<	<	<	<	<	<	>
$	<	<	<	<	<	<		<		<	<	<	<	<	<	<	