Dokumentace k projektu pro předměty IAL a IFJ

Implementace interpretu imperativního jazyka IFJ10

Tým 6, varianta b/2/I

2. listopad 2010

Tabulka řešitelů s údaji o rozdělení bodů.

Jméno	Příjmení	Přihlašovací jméno	Rozdělení bodů v procentech
Miroslav	Nemec	xnemec53	20,00%
Tomáš	Pop	xpopto01	20,00%
Lukáš	Langer	xlange02	20,00%
David	Molnár	xmolna02	20,00%
Tomáš	Mészáros	xmesza03	20,00%

Projekt má rozšíření o zjednodušený podmíněný příkaz If-Then bez části Else.

Autor: Miroslav Nemec, xnemec53@stud.fit.vutbr.cz

VUT FIT Brno

Obsah

1 Úvod....1
2 Struktura konečného automatu....2
3 LL-Gramatika....3
3.1 Návrh LL-Gramatiky....3
3.2 Implementace LL-Gramatiky....3
4 Popis způsobu řešení interpretu....4
4.1 Návrh interpretu....4
4.2 Implementace interpretu....4
4.3 Princip funkce interpretu....4
5 Implementace tabulky symbolů, řadícího algoritmu a vyhledávaní podřetězce v řetězci....5
5.1 Tabulka symbolů....5
5.2 Řadící algoritmus....5
5.3 Vyhledávaní podřetězce v řetězci....5
6 Rozdělení práce mezi členy týmu....6
7 Závěr....7
A Grafická podoba konečného automatu....8
B Pravidla LL-gramatiky....9
C Precedenční tabulka....10

Kapitola 1

1 Úvod

Před tím než budeme moct používat napsaný program ve zdrojovém jazyce je zapotřebí přejít několika fázemi, které jsou spolu provázány. Jedná se o lexikální analýzu, syntaktickou analýzu, sémantickou analýzu, generování vnitřního kódu, optimalizace a generování cílového kódu. Výsledkem těchto fází se ze zdrojového kódu vygeneruje cílový kód který potřebujeme k spuštění programu.

Podstatou této dokumentace je popsat postup a řešení problémů při vytváření Implementace interpretu imperativního jazyka IFJ10. Zejména se budeme věnovat struktuře konečného automatu, LL-gramatike, popisu způsobu řešení interpretu a radícího algoritmu.

Dokument obsahuje přílohy: grafická podoba konečného automatu, pravidla LL-gramatiky, precedenční tabulka.

Kapitola 2

2 Struktura konečného automatu

Grafickou podobu konečného automatu (viz. příloha) lze popsat následující tabulkou která objasňuje význam konečných stavů a tokenů které je reprezentují.

Konečné stavy	Tokeny reprezentující konečné stavy
f1	END_OF_LINE
f2	DIV_DOUBLE
f3	DIV_DOUBLE
f4	ID
f5	SEMICOLON
f6	LEFT_BRACKET
f7	RIGHT_BRACKET
f8	PLUS
f9	MINUS
f10	MULT
f11	DIV_INT
f12	EQ
f13	LEFT_LESS
f14	RIGHT_LESS
f15	NOT_EQ
f16	LEFT_LESS_EQ
f17	RIGHT_LESS_EQ
f18	COMMA
f19	CONST_STRING
f20	END_OF_FILE
f21	CONST_INT
f22	CONST_DOUBLE
f23	CONST_DOUBLE

Konečné stavy uvedené v této tabulce jsou funkčně ekvivalentní s konečnými stavy použitými v lexikálním analyzátoru. V implementaci se některé stavy (např. f5, f8 atd.) vypustily z důvodu optimalizace což znamená odstranění koncových stavů které jsou implementačně zbytečné. Jedná se zejména o koncové stavy vycházející z počátečního stavu které již dále nemají žádný další přechod. Například koncový stav f8, který reprezentuje plus nepotřebujeme implementovat protože se dá zahlásit již z počátečního stavu že jde o token plus. Konečný automat odstraňuje všechny komentáře a bílé znaky přičemž vrací jenom jeden konec řádku. Tímto je myšlen případ kdy několik řádku je prázdných nebo pokrytých bílými znaky a tím pádem je zbytečné vracet více tokenů signalizující konce řádku.

Kapitola 3

3 LL-Gramatika

LL-gramatika tvoří jádro syntaktického analyzátoru tudíž se jedná o důležitou část tohoto projektu. Předmětem této kapitoly bude návrh a implementace této gramatiky.

3.1 Návrh LL-Gramatiky

Při návrhu bylo nutné se zamýšlet na tím zda použít méně či více pravidel. Pro přehlednost použití méně či více pravidel je vhodná tabulka výhod a nevýhod těchto metod:

více pravidel	**méně pravidel**
méně rekurzivních volání při rekurzivním sestupu	více rekurzivních volání při rekurzivním sestupu
nepřehledná gramatika	přehledná gramatika
funkce neterminálů jsou méně zanořené	funkce neterminálů jsou více zanořené

Pro naše řešení jsme se rozhodli jít střední cestou a to nepoužít ani málo ale ani moc pravidel.

3.2 Implementace LL-Gramatiky

Při implementaci LL-Gramatiky bylo nutné vyřešit spojení s precedenční analýzou. Řešení je následovné: Precedenční parser je reprezentovaný jako funkce v programu. LL parser tuto funkci považuje za funkci neterminálů <expression>.

V LL tabulce tohle řešení funguje tak že <expression> derivuje přímo neterminál na terminál čímž je následně možné vytvořit LL-Gramatiku. LL tabuka je tvořená množinou Fisrt, Empty, Follow a Predict.

LL-Gramatika je rozšířená o pravidlo které řeší zjednodušený podmínění příkaz If-Then bez části Else. Tohle pravidlo Funguje tak že neterminál <elseBranch> se může zderivovat na terminál else a sekvenci příkazů, nebo se zderivuje na epsilon čímž se vynechá část else.

Kapitola 4

4 Popis způsobu řešení interpretu

V této kapitole se budeme především zabírat návrhem, implementací a principem funkce interpretu. Také se budeme zabírat problémy při implementaci s následným řešením.

4.1 Návrh interpretu

Při návrhu jsme vycházeli z toho, že bude zapotřebí seznam instrukcí, které se při zavolání interpreta následovně vykonají. Každá instrukce je složená z tříadresního kódu. Tyto instrukce jsou generované při syntaxi řízeném překladu.

4.2 Implementace interpretu

Při implementaci interpreta jsme se setkali s množstvím problémů, z nichž nejvýznamnější bylo řešení rekurze. Mylně jsme používaly tabulku symbolů i pro vkládání hodnot proměnných což ztížilo řešení rekurze. Po konzultacích jsme dospěli k řešení tohoto problému a to za pomocí použití tří zásobníků.

První zásobník (programový) slouží pro ukládání: - lokálních a pomocných proměnných,
- návratových hodnot
- parametrů funkcí.

Druhý zásobník slouží pro zálohovaní hodnoty base pointeru pro případné volání funkcí.
Třetí zásobník slouží při volání funkce na zálohování adresy na následující funkci.

4.3 Princip funkce interpretu

Zásobníky obsahují pouze ukazatele na místa kde jsou data dynamicky alokované a uložené. V tříadresním kódu jsou uloženy adresy do lokálních tabulek symbolů, kde jsou následně uložené off-sety, neboli posuny v rámci aktuálního base pointeru. Pomocí off-setu se následovně alokuje na zásobníku místo pro uložení proměnných. Průběh interpretu lze popsat v následujících krocích :

- na vrchol programového zásobníku se rezervuje místo pro návratovou hodnotu
- starý base pointer se uloží do druhého zásobníku .
- nový base pointer se nastaví tak aby ukazoval na návratovou hodnotu
- na programový zásobník se uloží parametry funkcí.
- adresa následující instrukce se uloží na třetí zásobník
- skočí se na návěští funkce
- vykoná se kód funkce kde jsou například alokované lokální proměnné podle off-setu z lokální tabulky
- při ukončení funkce se návratová hodnota uloží na aktuální base pointr
- odalokuje se všechno co je nad aktuálním base pointrem
- obnoví se base pointer, který je zálohovaný na druhém zásobníku
- z třetího zásobníku se obnoví adresa následující instrukce

Výsledkem těchto kroků je návratová hodnota uložená na vrcholu programového zásobníku.

Kapitola 5

5 Implementace tabulky symbolů, řadícího algoritmu a vyhledávaní podřetězce v řetězci

V této kapitole se budeme zajímat postupy a použitými algoritmy pro implementaci tabulky symbolů, řadícího algoritmu a vyhledávaní podřetězce v řetězci.

5.1 Tabulka symbolů

Tabulka symbolů je implementovaná pomocí binárního stromu. Nad tabulkou symbolů je zapotřebí jen několik základních operací a to inicializace, vkládání a mazání celé tabulky. Ostatní operace, které jsou běžně implementované nad binárním stromem jsou pro tabulku symbolů nepotřebné. Tabulky symbolů dělíme na globální a lokální. V globální tabulce symbolů se ukládají deklarace proměnných pro hlavní program a deklarace funkcí. V lokálních tabulkách symbolů se ukládají parametre funkcí a pomocné proměnné generované kompilátorem. Každá funkce má svoji lokální tabulku symbolů. Tabulky symbolů jsou ukládány do seznamu přičemž poslední položka je globální tabulka.

5.2 Řadící algoritmus

Jako řadící algoritmus je použit heap sort. Tomuto algoritmu se také říká řazení hromadou. V našem případě se jedná o binární hromadu, která je založená na binárním stromu. Podstatou heap sortu je implementace hromady polem. Heap sort umí efektivně provést operaci vkládání prvku a operaci výběr největšího prvku. V případě porušení hromady je zapotřebí znovu ustanovit porušenou hromadu. Této operaci se říká prosetí jejíž výsledkem je prvek z kořene přesunutý postupnými výměnami na své místo a kořen se správným prvkem. Postupným prosévaním a zapisováním do výstupného pole vzniká seřazená posloupnost.

5.3 Vyhledávaní podřetězce v řetězci

Vyhledávaní podřetězce v řetězci je realizováno Boyer-Mooreůvým algoritmem. Boyer-Mooreův algoritmus prochází znaky ve vzorku zprava doleva. V případě, že procházený znak není součástí hledaného vzorku skočíme dopředu o celou délku vzorku. Jak je známo z předmětu IAL Boyer-Mooreův algoritmus používá dva heuristické principy pro postup v prohledávaném řetězci. Pro naše řešení jsme použili první heuristiku, která je podrobně popsána v opoře k předmětu IAL.

Kapitola 6

6 Rozdělení práce mezi členy týmu

Tabulka rozdělení prací v týmu:

Jméno	Příjmení	Přihlašovací jméno	Práce
Miroslav	Nemec	xnemec53	tabulka symbolů, dokumentace
Tomáš	Pop	xpopto01	precedenční tabulka
Lukáš	Langer	xlange02	interpret, lexikální analýza
David	Molnár	xmolna02	testování, vestavěné funkce find a sort
Tomáš	Mészáros	xmesza03	syntaktická analýza

Nedá se přesně říct kdo na čem pracoval protože projekt byl týmový. Jednotlivé části jsme pravidelně konzultovali a řešili při pravidelných schůzkách. Pro vzdálenou komunikaci jsme používaly vygenerované fórum a SVN.

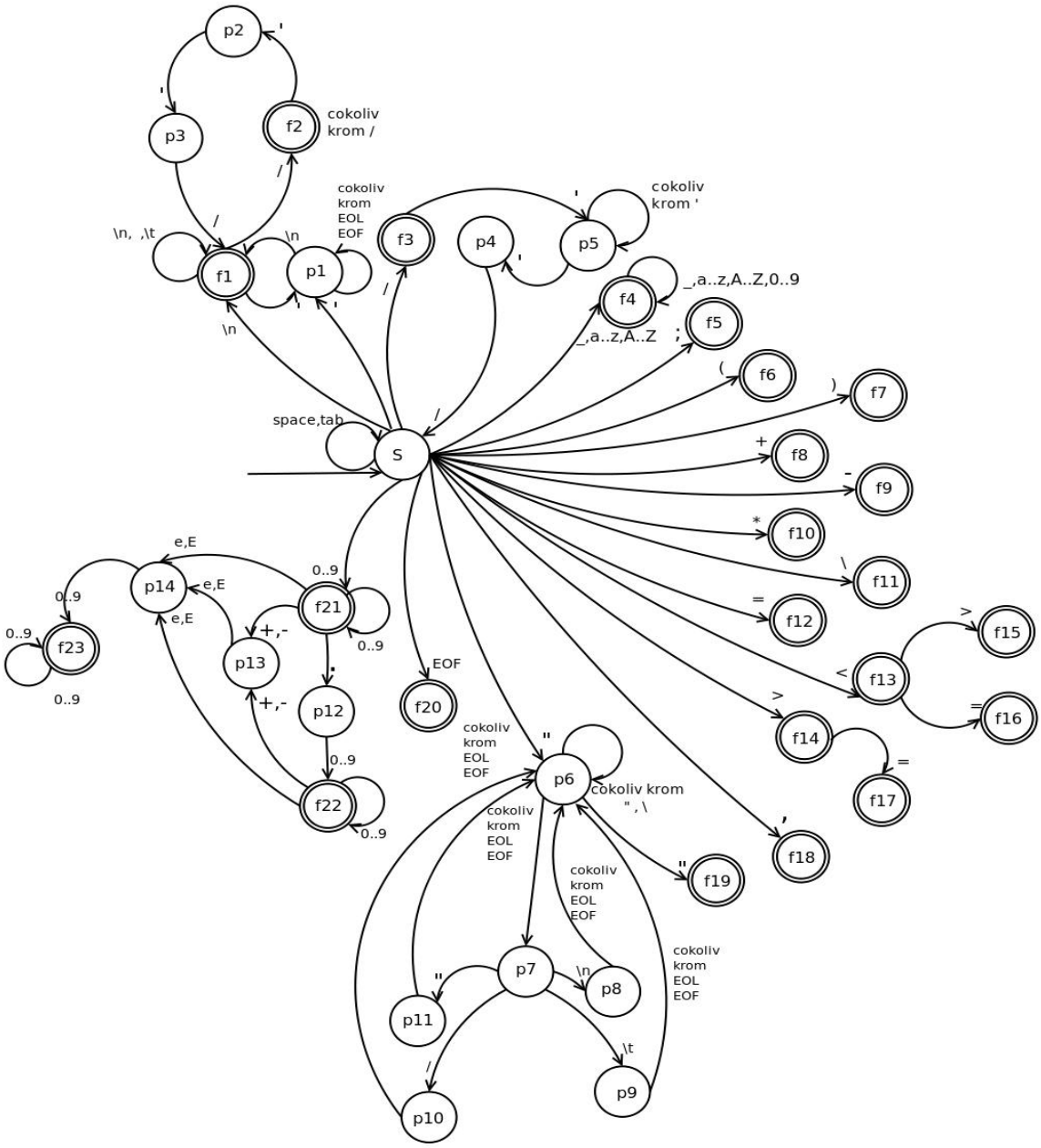
Kapitola 7

7 Závěr

Projekt pro předměty IAL a IFJ byl zatím největší výzvou na této škole. Dlouhými týdny práce jsme dosáhli požadovaného řešení. Díky tomuto projektu se nejen zlepšili naše programátorské schopnosti ale také schopnosti práce v týmu.

Příloha A

A Grafická podoba konečného automatu



Příloha B

B Pravidla LL-gramatiky

<prog>	->	<decList>	scope	eol	<varList>	<statList>	end	scope	eol				
<decList>	->	<decl>	eol	<decList>									
<decList>	->	<def>	eol	<decList>									
<decList>	->	eps											
<decl>	->	declare	function	id	(	<paramList>	)	as	<type>				
<def>	->	function	id	(	<paramList>	)	as	<type>	eol	<varList>	<statList>	end	function
<type>	->	integer											
<type>	->	double											
<type>	->	string											
<paramList>	->	<param>	<paramList>										
<paramList>	->	,	<paramList>										
<paramList>	->	eps											
<param>	->	id	as	<type>									
<varList>	->	<var>	eol	<varList>									
<varList>	->	eps											
<var>	->	dim	id	as	<type>								
<itemList>	->	id	<itemList>										
<itemList>	->	,	<itemList>										
<itemList>	->	eps											
<statList>	->	<statement>	eol	<statList>									
<statList>	->	eps											
<statement>	->	input	;	<itemList>									
<statement>	->	print	<expression>	;	<exprList>								
<statement>	->	id	=	<expression>									
<statement>	->	if	<expression>	then	eol	<statList>	<elseBranch>	end	if				
<statement>	->	do	while	<expression>	eol	<statList>	loop						
<statement>	->	return	<expression>										
<exprList>	->	<expression>	;	<exprList>									
<exprList>	->	eps											
<elseBranch>	->	else	eol	<statList>									
<elseBranch>	->	eps											
<expression>	->	e											

Příloha C

C Precedenční tabulka

	i	c_int	c_d	c_str	+	-	*	\ d.	/	=	<>	<	<=	>	>=	(	)	,	$
i					>	>	>	>	>	>	>	>	>	>	>	=	>	>	>
c_int					>	>	>	>	>	>	>	>	>	>	>		>	>	>
c_d.					>	>	>	>	>	>	>	>	>	>	>		>	>	>
c_str					>	>	>	>	>	>	>	>	>	>	>		>	>	>
+	<	<	<	<	>	>	<	<	<	>	>	>	>	>	>	<	>		>
-	<	<	<	<	<	>	<	<	<	>	>	>	>	>	>	<	>		>
*	<	<	<	<	>	>	>	>	>	>	>	>	>	>	>	<	>		>
\ d.	<	<	<	<	>	>	<	>	<	>	>	>	>	>	>	<	>		>
/	<	<	<	<	>	>	<	>	>	>	>	>	>	>	>	<	>		>
=	<	<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	>		>
<>	<	<	<	<	<	<	<	<	<	<	>	>	>	>	>	<	>		>
<	<	<	<	<	<	<	<	<	<	<	<	>	>	>	>	<	>		>
<=	<	<	<	<	<	<	<	<	<	<	<	<	>	>	>	<	>		>
>	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	<	>		>
>=	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	>		>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)					>	>	>	>	>	>	>	>	>	>	>		>		>
,	<	<	<	<													>	>	
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<			KON.

Pravidla

1.)	E->i()	4.)	E->E=E	10.)	E->E+E
2.)	E->i(E)	5.)	E->E<>E	11.)	E->E-E
3.)	E->E,E	6.)	E->E<E	12.)	E->E*E
		7.)	E->E<=E	13.)	E->E/E
		8.)	E->E>E	14.)	E->E\E
		9.)	E->E>=E		

15.)	E->(E)
16.)	E->i
17.)	E->c_int
18.)	E->c_d.
19.)	E->c_str