Implementácia interpreta impertívneho jazyka IFJ08

Varianta:	**a/2/II**
Riešitelia:	Michal Ďuriš (xduris01) - vedúci teamu Martin Elis (xelism00) Juraj Červienka (xcervi09) Martin Herich (xheric00) Lukaš Bartoš (xbarto55)
Predmety:	IFJ - Formálni jazyky a prekladače IAL - Algoritmy
Akademický rok:	2008/2009 - zimný semester
Rozdelenie bodov:	xbarto55: 5 xcervi09: 20 xduris01: 30 xelism00: 25 xheric00: 20
Dátum odovzdania:	12.12.2008

Obsah

1. Úvod

Dokumentácia popisuje implementáciu imperatívneho jazyka IFJ08. Cieľom projektu bolo vytvoriť interpret imperatívneho jazyka IFJ08. Interpret je naimplmentovaný v jazyku C.

Program interpretu imperatívneho jzyka IFJ08 sa skladá z niektorých hlavných častí, ktoré nám zaisťujú jednotlivé činnosti:

- Lexikálna analýza
- Syntaktická analýza v ktorej je zahrnutá aj sémantická analýza a generovanie trojadresného kódu
- Interpret

Riešenie nášeho zadania sme si rozdelili na jednotlivé časti:

- ADT táto časť obsahuje hashovaciu tabuľku
- Lexikálna analýza, ktorá nám klasifikuje a rozčlenuje jednotlivé lexémy a ďalej posiela na
  spracovanie tokeny
- Syntaktická analýza je časť ktorá spracováva jednotlivé tokeny od lexikálnej analýzy a kontroluje syntax jazyka
- Precedenčná syntaktická analýza taktiež nazývaná syntaktická analýza zdola hore, v ktorej dochádza ku kontrole jednotlivých výrazov
- Sémantická analýza zabezpečuje kontrolu typov a sémantiky
- Interpret spracováva trojadresný kód, ktorý je generovaný samotnou syntaktickou analýzou
- Heap sort: algoritmus, ktorý služi na radenie a je súčasťou interpreta
- Knuth-Morris-Prathov algoritmus : je využívaný k vyhľadávaniu podreťazca v reťazci a je súčasťou interpreta

2. Spôsob riešenia interpretu

Program interpretu jayzka IFJ08 sa skladá z častí, ktoré spolu zabezpečujú správny chod interpreta.

2.1 Lexikálny analyzátor

Lexikálny analyzátor sme implementovali podľa postupu, prezentovaného na prednáškach predmetu IFJ. To predstavuje základnú myšlienku, že lexikálny analyzátor by mal pracovať na princípe deterministického konečného automatu (vid. príloha 1) a rozpoznávanie lexémov by malo prebiehať pomocou koncových stavov tohto automatu.

DKA jsme realizovali pomocou cyklu do-while, ktorý končí úspešným načítaním EOF, teda konca vstupného súboru. V každom kroku sa vykonáva načítanie jedného znaku (symbolu) zo vstupného súboru a porovnávanie tohto znaku so symbolmi validnými pre jazyk IFJ08. Pokiaľ načítaný symbol nie je validný, vracia lexikálna analýza chybu lexikálnej analýzy. V prípade, že je všetko v poriadku, lexikálna analýza vracia token, ktorý vyjadruje typ načítaného lexému. Rozpoznávanie typu tokenu je vykonávané tak, že sú načítané cyklom ďalšie symboly až do tej doby, pokiaľ nie je načítaný neočakávaný symbol. Pokiaľ sa tak stane a automat je v konečnom stave, tak sa identifikuje (podľa stavu) token a posledný načítaný neočakávaný symbol sa vráti späť do vstupného súboru pomocou funkcie Odeberznak.

DKA **preskakuje** biele znaky (medzery, \n, \t...) a komentáre.

DKA **prijíma** nasledujúci lexémy:

- **Identifikátory** a **kľúčové slová**

- rozlíšenie sa vykonáva pomocou funkcie strcmp, kedy sa porovnáva načítaný reťazec a prvky pola, v ktorom sú uložené jednotlivé kľúčové slová jazyka IFJ08
- za validné formy pokladáme neprázdne postupnosti znakov, číslic a písmen

- **Celé nezáporné čísla typu integer**
- **Desatinné čísla typu double** v tvare:

a) celá časť a desatinná časť oddelená bodkou
b) celá časť a exponent
c) celá časť a desatinná časť oddelená bodkou a exponentom

- **Reťazcové konštanty**

- reťazec znakov medzi párom úvodzoviek

- **Operátory**

a) aritmetické: + , - , * , / , ^
b) relačné: = , == , != , < , <= , > , >=

- **Ostatní:** (,) , { , } , ; , EOF, a čiarka (,)

Identifikátory a číselné konštanty sú ukladané do **tabuľky symbolov**. Tato bola implementovaná pomocou hashovacej tabuľky.

2.2 Syntaktický analyzátor

V tejto casti je naimplementovaná syntaktická časť interpretu, ktorá sa rozdeľuje na tieto dve časti:

- **Syntaktická analýza zhora dole**
- **Syntaktická analýza zdola hore**

Táto časť interpretu sa považuje za najpodstatnejšiu časť interpreta a jej úlohou je kontrola syntaktickej gramatiky a sémantiky a taktiež vytvárať trojadresný kód, pomocou ktorého interpret vykonáva zadané príkazy.

Syntaktická analýza zhora dole

Prioritou v danej syntaktickej analýze je návrh LL-gramatiky a zostrojenie LL-tabuľky, podľa ktorej sa vykonáva kontrola správnosti syntaxe jazyka. Implementácia gramatiky bola vykonaná metódou rekurzívneho zostupu pre kontext jazyka založeného na LL-gramatike. Navrhnutá
LL-gramatika a LL-tabuľka sú v prílohe č.2

Sémantické kontroly boli vykonané v častiach pri deklarácii premenných, podmienke if else, cykle while a funkcii cout, cin a return. Pri funkciách cin, cout a return bol generovaný trojadresný kód. Podmienený príkaz if else a príkaz cyklu while sme implementovali pomocou návesti GOTO a LABEL.

Syntaktická analýza zhola hore

taktiež nazývaná ako precedenčná syntaktická analýza, ktorá je volaná len v prípade, že je nutné vyhodnotiť výraz. K tomuto využíva precedenčnú tabuľku viď. Príloha c.3.
V tabuľke sú nadefinované priority jednotlivých operátorov a taktiež zahŕňa asociativity operátorov. Tabuľka obsahuje jednotlivé symboly, ktoré vyjadrujú postup vyhodnotenia výrazov. Znaky v tabuľke vyjadrujú nasledovné :

< - načítaný symbol uložíme na zásobník
= - načítaný symbol uložíme na zásobník
> - nastáva redukcia podľa stanovených pravidiel

Ďalej sú aplikované jednotlivé pravidlá, podľa ktorých su redukované výrazy a s využitím zásobníka simulujeme vytváranie derivačného stromu.

V tejto časti je začlenená sémantická kontrola, ktorá kontroluje jednotlive typy operandov, v prípade nutnosti ich pretypuje alebo pri nekompatibilných typoch nám sémantickú chybu. Vyhodnotenie výrazov končí pri situácií .keď v zásobníku ostal na začiatku vložený znak $ a na je ukončovací znak výrazu, ktorý reprezentujú znaky ; } <<-

2.3 Interpret

Interpret posledný prvok v reťazci spracovania zdrojového kódu. Postupne spracováva inštrukcie uložené v zozname syntaktickou analýzou. Inštrukcia sa skladá z typu operácie, ukazateľov na prvý, druhý a cieľový operand a taktiež z ukazateľa na nasledujúcu inštrukciu. Ukazatele smerujú na štruktúry v tabuľke symbolov.

instrukcia [operacia, uk_na_zdroj1, uk_na_zdroj2, uk_na_ciel, uk_dalsia_instrukcia]

Interpret si načítava zoznam inštrukcií a postupne ním prechádza a súčasne vykonáva zadané inštrukcie. Operácie tvoria príkazy skoku, operácie porovnania, priradenia, aritmetické operácie, a vstavané funkcie.

Príkazy skoku implementujú príkazy vetvenia a cyklu, operácie porovnania vyhodnocujú operátory <,>,<=, >=, ==, !=. Operácie priradenia priraďujú jeden prvok do druhého (operátor = a pravidlo $E => i$), aritmetické operácie vykonávajú základné matematické operácie sčítanie (+), odčítanie (-), násobenie (*) a celočíslené delenie (/).

Vstavané funkcie predstavujú funkcie spojenia dvoch reťazcov (`concat()`), funkcia vyhľadania reťazca v podreťazci (`find()`), funkcia utriedenia reťazca (`sort()`) a funkcia, ktorá vráti dĺžku reťazca (`length()`).

2.4 Riešenie algoritmov heap sort a Knuth-Morris-Pratt

Algoritmus radenia heap sort je využívaný vstavanou funkciu sort jazyka IFJ08.Algoritmus heapsort taktiež nazývaný aj ako halda. Základnou časťou algoritmu heap sort je implementácia ADT haldy, ktorá vie spraviť operácie vloženie prvku a výber najväčšieho prvku veľmi efektívne. Preto sa dá reťazec zoradiť jednoducho pomocou vloženia znakov do haldy a postupným vyberaním najväčšieho prvku.

Ako pomocný algoritmus som si implementoval algoritmus, ktorý dokáže predĺžiť haldu o jeden prvok. Výsledkom je, že pokiaľ všetky prvky s indexmi L+1 a R spĺňajú pravidlá haldy, tak po prevedení pomocného algoritmu budú spĺňať podmienky haldy prvky s indexmi L a R.

V našom prípade sme mali zoradiť reťazec tak, aby znak s menšou ordinálnou hodnotou predchádzal znak s väčšou.

Algoritmus vyhľadávania podreťazca v reťazci **Knuth-Morris-Pratt.** Využíva takzvaný konečný automat. Daný algoritmus nám vracia index začiatočného znaku nájdeného podreťazca ak daný reťazec neobsahuje podreťazec vraciame hodnotu -1.

3.Záver

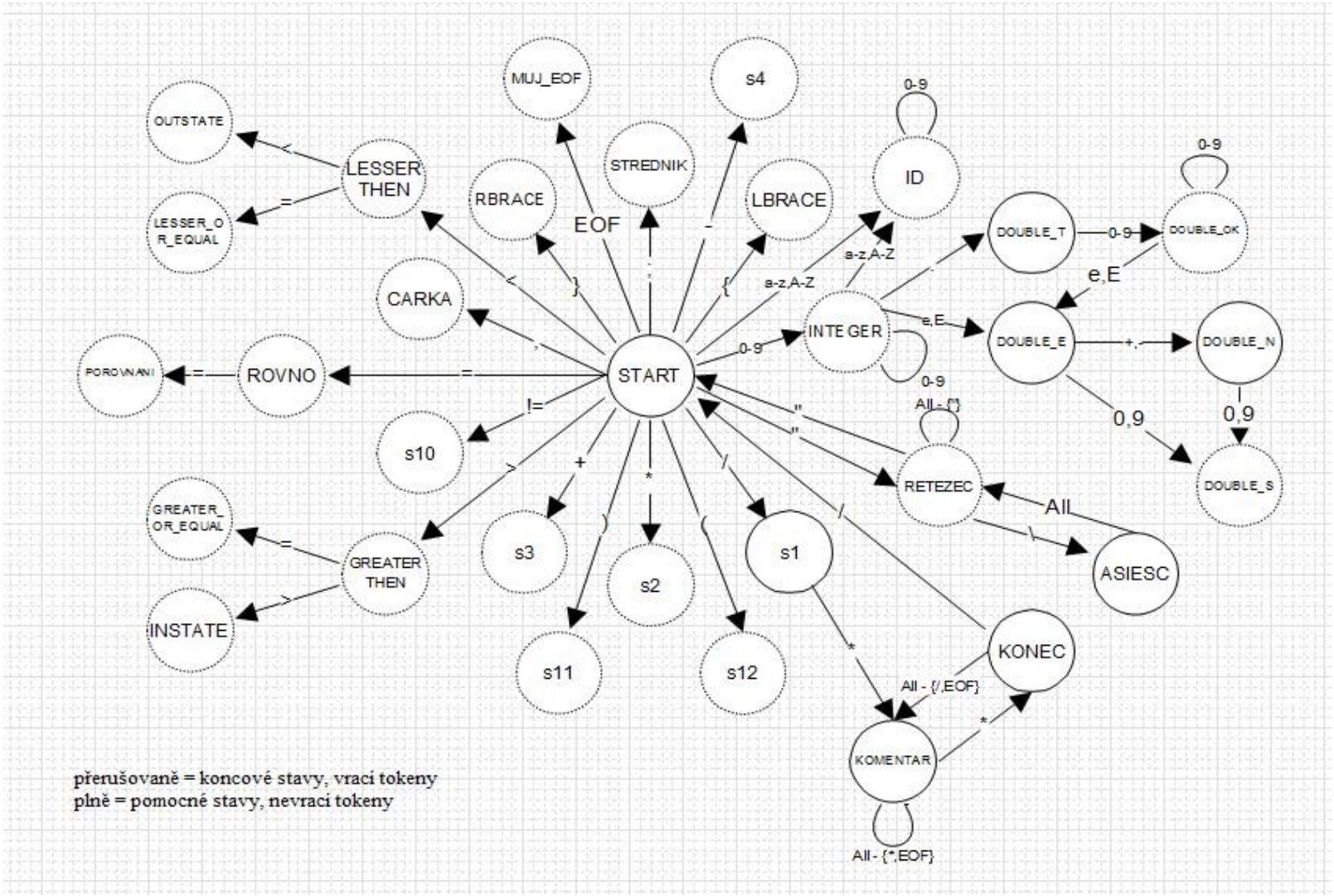
Tento projekt nám priniesol mnoho nových a cenných skúseností ako aj z oblasti formálnych jazykov tak aj z oblasti tímovej spolupráce a práce na rozsiahlejšom projekte. Naučili sme sa naimplementovať štyri základne časti interpreta a spoznali sme základnú problematiku ich tvorby.

3.1 Metrika kódu

Počet súborov: 19
Veľkosť dát: 120,1 KB
Veľkosť spustiteľného súboru: 82,2 KB

4.Prílohy

Deterministický konečný automat lexikálnej analýzy:



LL - Gramatika pre syntaktickú analýzu:

1. <START> -> double id ; <START>
2. <START> -> string id ; <START>
3. <START> -> int <HEAD>
4. <HEAD> -> id; <START>
5. <HEAD> -> main() {<TeloP> EOF
6. <TeloP> -> <prikaz> <TeloP>
7. <TeloP> -> }
8. <prikaz> -> cin >> id <zoznam cin>
9. <zoznam cin> -> >> id <zoznam cin>
10. <zoznam cin> -> ;
11. <prikaz> -> cout << <vyraz> <zoznam cout>
12. <zoznam cout> -> << <vyraz> <zoznam cout>
13. <zoznam cout> -> ;
14. <prikaz> -> <vyraz>;
15. <prikaz> -> if <vyraz> { TeloP else { TeloP
16. <prikaz> -> while <vyraz> { TeloP
17. <prikaz> -> return <vyraz>;

Pravidlá syntaktickej analýzy z dola hore:

E	->	E + E	E	->	E = E
E	->	E - E	E	->	E == E
E	->	E * E	E	->	E != E
E	->	E / E	E	->	(E)
E	->	E < E	E	->	i
E	->	E > E	E	->	f(E)
E	->	E <= E	E	->	f(E,E)
E	->	E >= E			

Tabuľka pre syntaktickú analýzu zhora dole:

	double	string	int	main	{	cin	cout	if	while	return	;	id	<<	>>	}	(	concat	find	length	sort
START	1	2	3																	
HEAD				5								4								
TeloP						6	6	6	6	6		6			7	6	6	6	6	6
prikaz						8	11	15	16	17		14				14	14	14	14	14
zoznam cin											10			9						
zoznam cout											13		12							

Precedenčná tabuľka - syntaktická analýza z dola hore:

	+	-	*	/	=	(	)	i	<	>	<=	>=	==	!=	f	,	<<	{	;
+	>	>	<	<	>	<	>	<	>	>	>	>	>	>	<	>	>	>	>
-	>	>	<	<	>	<	>	<	>	>	>	>	>	>	<	>	>	>	>
*	>	>	>	>	>	<	>	<	>	>	>	>	>	>	<	>	>	>	>
/	>	>	>	>	>	<	>	<	>	>	>	>	>	>	<	>	>	>	>
=	<	<	<	<	<	<	>	<	<	<	<	<	<	<	<	>	>	>	>
(	<	<	<	<	<	<	=	<	<	<	<	<	<	<	<	=			
)	>	>	>	>	>		>		>	>	>	>	>	>		>	>	>	>
i	>	>	>	>	>		>		>	>	>	>	>	>		>	>	>	>
<	<	<	<	<	>	<	>	<	>	>	>	>	>	>	<	>	>	>	>
>	<	<	<	<	>	<	>	<	>	>	>	>	>	>	<	>	>	>	>
<=	<	<	<	<	>	<	>	<	>	>	>	>	>	>	<	>	>	>	>
>=	<	<	<	<	>	<	>	<	>	>	>	>	>	>	<	>	>	>	>
==	<	<	<	<	>	<	>	<	<	<	<	<	>	>	<	>	>	>	>
!=	<	<	<	<	>	<	>	<	<	<	<	<	>	>	<	>	>	>	>
f						=													
,	<	<	<	<	<	<	=	<	<	<	<	<	<	<	<				
$	<	<	<	<	<	<		<	<	<	<	<	<	<	<				