

**Dokumentace k projektu pro předměty IFJ a IAL**

**Implementace interpretu imperativního jazyka IFJ13**

Tým 005

Varianta a/3/I

13. prosince 2013

Řešitelé: Jan Hanák xhanak27 20% **vedoucí**

Michal Flax xflaxm00 20%

Cuong Duong Tuan xduong00 20%

Jakub Fišer xfiser10 20%

Vladimír Faltýn xfalty06 20%.

Obsah

[1 Úvod 3](#_Toc374710806)

[2 Zadání 3](#_Toc374710807)

[2.1 Varianta zadání 3](#_Toc374710808)

[3 Práce v týmu 3](#_Toc374710809)

[4 Implementace 4](#_Toc374710810)

[4.1 Knuth-Moris-Prattův algoritmus 4](#_Toc374710811)

[4.2 Shell sort 4](#_Toc374710812)

[4.3 Tabulka symbolů 4](#_Toc374710813)

[4.4 Lexikální analýza 4](#_Toc374710814)

[4.5 Syntaktická analýza 4](#_Toc374710815)

[4.6 Sémantická analýza 4](#_Toc374710816)

[4.7 Generátor tříadresného kódu 4](#_Toc374710817)

[5 Závěr 6](#_Toc374710818)

[6 Přílohy 7](#_Toc374710819)

[6.1 Konečný automat lexikálního analyzátoru 7](#_Toc374710820)

[6.2 LL gramatika 8](#_Toc374710821)

1 Úvod

Tato dokumentace popisuje implementaci interpretu pro imperativní jazyk IFJ13. Dokumentace je členěna do několika kapitol popisující konkrétní části překladače.

Přiložen je graf konečného automatu specifikující lexikální analyzátor a LL-gramatika pro syntaktický analyzátor.

2 Zadání

Jazyk IFJ13 je podmnožinou skriptovacího jazyka PHP. Jazyk je case-sensitive a je jazykem dynamicky typovaným. Zadáním je vytvořit program, který načte zdrojový soubor v jazyce IFJ13 a interpretuje jej, přičemž vrací návratovou hodnotu podle toho, jak proběhla činnost překladače.

2.1 Varianta zadání

Variantou zadání byla a/3/I, to konkrétně znamená:

- na věstavenou funkci *find\_string()* je použitý Knuth-Moris-Prattův algoritmus

- věstavěná funkce *sort\_string()* je implementována metodou shell sort

- tabulka symbolů je implementována pomocí binárního vyhledávacího stromu

3 Práce v týmu

Docházelo k nepravidelným schůzkám, práce se rozdělovala na několik částí. Jako první se napsal lexikální analyzátor, poté se navrhla LL-gramatika pro syntaktický analyzátor, který byl záhy napsán. Generátor tříadresného kódu byl napsaný před sémantickou analýzou, která byla napsána jako poslední. Časový plán kódování nebyl, spoléhalo se na svědomitý přístup všech členů týmu.

Kód byl psán jak v Linuxu, tak Windows, přičemž pro testování se využíval školní server Merlin. Ke sdílení kódu jsme využili repozitáře GIT a pro vzájemnou komunikaci jsme použili skupinový chat a skupinu v síti Facebook.

4 Implementace

4.1 Knuth-Moris-Prattův algoritmus

Věstavená funkce *find\_string()* využívá Knuth-Moris-Prattův algoritmus, který je založen na úvaze, že časové ztráty triviálního algoritmu jsou způsobeny tím, že v případě neshody vyhledávaného podřetězce v řetězci se podřetězec posune o jednu pozici doprava a znovu se celý zkontroluje, je tedy nutné se často při porovnání vracet zpět k pozicím, jež byly dříve zkontrolovány. Snahou tohoto algoritmu je v případě neshody hledaného podřetězce v řetězci posunout podřetězec tak, aby se nebylo nutné vracet ve vlastním řetězci. Jeho maximální složitost je O(m+n).

4.2 Shell sort

Věstavená řadící funkce *sort\_string()* je založena na Shell sortu, který patří mezi nestabilní řadící metody. Je založený na Bubble sortu. V několika průchodech se řadí prvky ve stejné vzdálenosti od sebe pomocí Bubble sortu, vzdálenost se postupně snižuje, obecně se začíná na n/2, která se nadále snižuje dělením 2. Výhodou je např. to, že na rozdíl od Heap sortu není třeba halda a na rozdíl od Quick sortu není potřeba zásobník.

4.3 Tabulka symbolů

Tabulka symbolů byla implementována pomocí binárního vyhledávacího stromu.

Jeden prvek stromu je definován pomocí struktury *NODE*, která se skládá z klíče, dat a ukazatele na levý a pravý podstrom. Tato struktura se používá jak pro proměnné, tak pro funkce, proto byl přidán i ukazatel na proměnné funkce, pokud nějaké má. Nad tabulkou symbolů jsou implementovány základní funkce jako inicializace stromu, přidání nebo vyhledání prvku. Taky byla speciálně pro rekurzi vytvořena funkce pro kopírování stromu a zásobník pro prvky stromu.

4.4 Lexikální analýza

Lexikální analyzátor je v principu implementován jako série podmínek, vycházejících z přiloženého konečného automatu. Implementace je založena na postupném čtení vstupního souboru po řádcích, které jsou ukládány do pomocného bufferu. Při každém požadavku na nový token je zkoumán začátek řetězce v tomto bufferu, pokud vyhovuje podmínce na tvar nějakého tokenu, je začátek řetězce posunut na znak následující v bufferu po právě zpracovaném tokenu a odeslán token s příslušným číslem. Pokud začátek řetězce nevyhovuje žádné podmínce, je situace vyhodnocena jako lexikální chyba.

4.5 Syntaktická analýza

Naše syntaktická analýza je založena na LL-gramatice (příloha 2). Pro zpracování vstupu bylo vytvořeno dynamické pole tokenů reprezentující jednotlivé řádky. Toto pole bylo při sestavování kontrolováno podle gramatiky, vstupní program byl tedy zpracováván metodou rekurzivního sestupu. Syntaktická analýza je tvořena hlavní smyčkou programu obsahující především zanořené podmínkové výrazy. Výrazy byly zpracovávány zdola nahoru s využitím zásobníku.

4.6 Sémantická analýza

Při zpracování celého řádku v syntaktické analýze je zavolána syntaktická analýza. Ta má za úkol vytvářet tabulku symbolů, kontrolovat definice funkcí, počet parametrů při jejich volání a základní typovou kompatibilitu ve výrazech. Sémantická analýza také zpracovává veškeré výrazy převodem do postfixové notace. Další sémantické chyby nebylo možné v tomto stádiu interpretu odhalit a jsou tedy detekovány až v interpretu. V sémantické analýze je obsaženo i generování tří-adresného kódu.

4.7 Interpret

Tato část je implementována pomocí postupného vyhodnocování seznamu instrukcí 3AK, který je vytvořen před spuštěním interpretu v rámci sémantické analýzy. Dále interpret detekuje chyby v rámci programu, které je možné detekovat až za běhu programu. Algoritmus postupně načítá instrukce v předem daném pořadí, vykoná příslušné operace a uloží výsledky, případně jedná-li se o instrukce funkcí, podmínek čí cyklů, provede příslušné operace a je-li to vyžadováno, nepokračuje bezprostředně následující instrukcí, ale instrukcí která je uvedena u právě vyhodnocované instrukce.

Dále je vhodné se samostatně zmínit činnosti interpretu v souvislosti s funkcemi. Pokud je zpracováváno volání uživatelské funkce, je uložena aktuální tabulka symbolů na zásobník a jako aktuální je nastavena kopie té tabulky symbolů, která přísluší právě volané funkci. Kopii je nutné vytvořit, aby v případě že je funkce volána vícekrát během běhu programu, nesmí se přepisovat původní záznamy v tabulce symbolů, které jsou uloženy v rámci syntaktické/sémantické analýzy. Toto je implementováno z důvodu, že každé volání funkce musí začínat s identickým stavem tabulky symbolů příslušejících dané funkci.

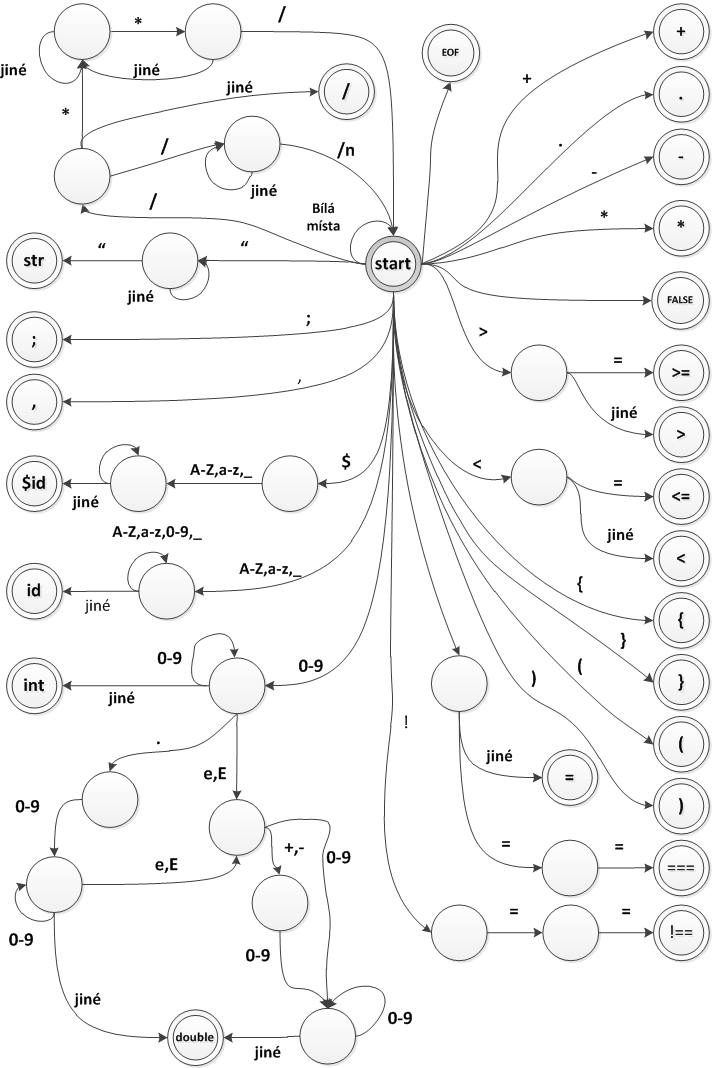
Naopak pokud je zpracováváno ukončení funkce, je smazána právě aktuální tabulka symbolů a jako aktuální tabulka je nastavena tak, která se nachází na vrcholu zásobníku tabulek instrukcí.

5 Závěr

// doplnit

6 Přílohy

6.1 Konečný automat lexikálního analyzátoru



6.2 LL gramatika

G = (N,T,P,S)

N ... neterminály

T ... terminály

P ... pravidla

S ... počáteční stav

N = {Expr, Sign, Term, ID}

T = {else, false, function, if, return, true, while, for, id, $id, integer,

double, string, null, +, -, \*, /, (, ), {, }, string, number, <, >, =,

<=, >=, ===, !==, ','(==čárka), ;, EOL }

P = {

S -> <if> Expr { S } EOL

S -> <else> { S } EOL

S -> <elseif> Expr { S } EOL

S -> <while> Expr { S } EOL

S -> <for> (Expr; Expr; Expr) { S } EOL

S -> <return> Expr ;

S -> <function> id ( ID ) { S } EOL

S -> $id = Expr ;

S -> Ø

ID -> $id

ID -> ID , ID

ID -> Ø

Expr -> id ( ID )

Expr -> ( Expr )

Expr -> Expr Sign Expr

Expr -> Term

Term -> $id

Term -> integer

Term -> double

Term -> string

Term -> boolean

Term -> null

Term -> Ø (prázdná množina)

Sign -> +

Sign -> -

Sign -> \*

Sign -> /

Sign -> <

Sign -> >

Sign -> ===

Sign -> <=

Sign -> >=

Sign -> !==

Sign -> .

}