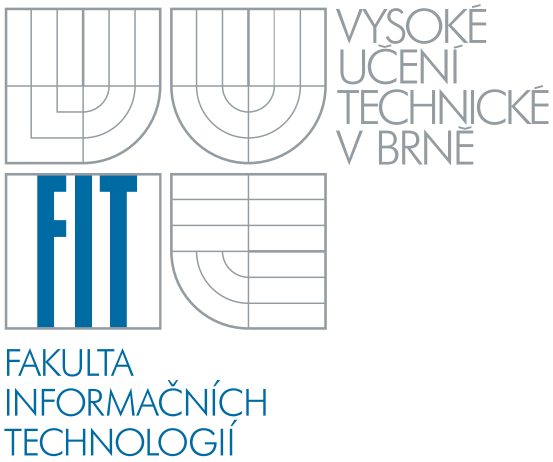


Dokumentace k projektu IFJ/IAL

Implementace interpretu imperativního jazyka IFJ14

11. prosince 2014

Tým 004, varianta a/2/II

Norbert Ďurčanský - vedoucí (xdurca01), 20%
Jindřich Dudek (xdudek04), 20%
Jiří Dostál (xdosta40), 20%
Ján Jusko (xjusko00), 20%
Natalya Loginova (xlogin00), 20%

Obsah

Úvod

Tato dokumentace obsahuje popis implementace interpretu imperativního jazyka IFJ14. Program načítá a interpretuje zdrojový soubor zapsaný v jazyce IFJ14. Jestliže interpret vyhodnotí kód jako spravný, pak se vrací návratová hodnota 0. V opačném případě program vrací kód chyby.

Zadání

Jazyk IFJ14 je podmnožinou jazyka Pascal, což je staticky typovaný (tj. jednotlivé proměnné mají předem určen datový typ svou definicí) procedurální jazyk. Jazyk je case insensitive (nezáleží na velikosti písmen).

Interpret se skládá ze tří hlavních částí: (1) lexikální analyzátor (neboli scanner), (2) syntaktický analyzátor (parser), který je nejdůležitější částí programu, (3) interpret. Scanner načítá zdrojový soubor a rozpoznává ve zdrojovém kódu jednotlivé lexikální jednotky, parser pak zkontroluje sintaxi, provede sémantické akce a vygeneruje 3-adresný kód, který interpret vykoná.

Kromě toho interpret poskytuje některé vestavěné funkce včetně vyhledávání podřetězce v řetězci a řazení znaků v řetězci.

Náš tým řešil variantu zadání projektu a/2/II:

- Knuth-Morris-Prattův vyhledávací algoritmus
- Řazení metodou Heap sort
- Implementace tabulky symbolů pomocí hashovací tabulky

Implementace

V této kapitole je popsána implementace jednotlivých částí programu.

3.1 Lexikální analyzátor

Lexikální analyzátor (scanner) rozpoznává jednotlivé lexémy (lexikální jednotky) zdrojového programů a vytváří korespondující tokeny, přičemž komentáře a bílé znaky jsou vynechány (nikam se neukládají). Tokeny odpovídají lexikálním prvkům zdrojového kódu (například identifikátory, datové

typy, klíčová slova, literály atd.) a kromě typu lexému obsahují důležité doplňující informace pro sémantickou analýzu a interpretaci (např. jméno identifikátoru, konkrétní hodnotu konstanty apod.).

Implementace lexikálního analyzátoru je založena na využití konečného automatu, který je realizován pomocí řídící konstrukce switch spolu s řídící proměnnou, která uchovává aktuální stav automatu. Scanner vrací na požádání jeden token, který následuje za předešlým již načteným tokenem. Implementace je prováděna funkcí `getnextToken`, která přijímá ukazatel na strukturu reprezentující poslední načtený token a vrací celé číslo reprezentující buď kód chyby, v případě, že na chybu narazí, nebo typ aktuálního tokenu a současně strukturu jej reprezentující (prostřednictvím parametru) v případě úspěchu.

Diagram konečného automatu, který specifikuje lexikální analyzátor je příloze č. 1.

3.2 Syntaktický analyzátor

Syntaktický analýzátor (parser) je srdcem překladače. Jeho úkolem je zkontrolovat syntaxi vstupního programu, provést korespondující sémantické akce a vygenerovat tříadresný kód (tj. posloupnost tříadresných instrukcí), jež bude následně interpretován interpretem.

Parser načíta posloupnost tokenů ze zdrojového kódu na vstupu pomocí lexikálního analyzátoru. Syntaktická analýza jazykových konstrukcí (deklarace/definice proměnných a funkcí, řídící/přiřazovací příkazy apod.) je implementována pomocí rekurzivního sestupu podle pravidel LL gramatiky metodou shora dolů. Výrazy se zpracovávají pomocí precedenčního syntaktického analyzátoru, pracujícího zdola nahoru. Pak se pomocí funkce `generate_inst` generuje instrukční list.

Pravidla LL gramatiky jsou v příloze č. 2, precedenční tabulka je v příloze č. 3.

3.3 Interpret

Základní funkce interpretu je vykonání vnitřního kódu vygenerovaného syntaktickým analyzátorem. Interpret tedy podle instrukcí zpracuje vstup programu a vygeneruje výstup.

Implementace je založena na tom, že interpret prochází instrukčním listem vnitřního kódu (který je implementován pomocí nekonečného pole) a postupně vykoná jednotlivé instrukce. Podpora instrukcí skoku je zajištěna použitím zásobníku a tabulek symbolů.

Samotná instrukce je reprezentována strukturou, která obsahuje operandy instrukce a některé další pomocné informace.

3.4 Vyhledávání podřetězce v řetězci

Podle varianty zadání vyhledávání podřetězce v řetězci je implementováno pomocí Knuth-Morris-Prattova vyhledávacího algoritmu (verze z přednášek IAL).

Knuth-Morris-Prattův algoritmus (KMP) využívá ke své práci konečný automat, který je reprezentován vzorkem P a vektorem FAIL, který má prvky typu integer, reprezentující cílový index zpětné šipky. Z každého uzlu automatu vychází dvě hrany - ANO (shoda znaku) a NE (neshoda). Automat načítá znaky, pokud dojde ke shodě, posune se na další stav, v případě neshody se vrátí na stav specifikovaný vektorem FAIL. Dosažení koncového stavu znamená nalezení vzorku.

Algoritmus má složitost O(n+m), kde n je délka prohledávaného textu, m je délka vyhledávaného vzorku.

3.5 Řazení

Podle varianty zadání řazení je implementováno pomocí algoritmu Heap Sort (verze z přednášek IAL).

Základní myšlenkou řadící metody Heap Sort je využití hromady (heap) - struktury, založené na binárním stromu, u níž pro všechny uzly platí, že mezi otcovským uzlem a všemi jeho synovskými uzly je stejná relace uspořádání. Pomocí této struktury lze seřadit dodaná data prostě pomocí jejich vložení do hromady a následného postupného vybírání největšího/nejmenšího prvku. Podstatou řadicí metody je implementace hromady polem, která stanoví implicitní zřetězení prvků hromady.

Metoda Heap Sort má linearitmickou složitost O(N*log N), je nestabilní a není přirozená.

3.6 Tabulka symbolů

Tabulka symbolů v našem řešení byla realizována pomocí hashovací tabulky, která uchovává identifikátory funkcí, proměnných a další důležité informace s nimi spojené. K vyhledávání a vkládaní dat do tabulky je využita hashovací funkce, která generuje pozici uložení v hashovací tabulce na základě názvu identifikátoru, který tak zároveň slouží jako klíč pro vyhledávání. S tabulkou symbolů se pracuje pomocí funkcí uvozených názvem `hashtable_`, které lze nalézt v modulu ial.c.

Práce v týmu

Ke komunikaci v týmu jsme využívali pravidelné schůze, kde jsme diskutovali řešení projektu, a vedoucí kontroloval stav plnění jednotlivých úkolů členy týmu. Kromě toho se používaly i jiné komunikační nástroje jako Skype a soukromá diskusní skupina.

Pro management zdrojového kódu jsme využívali systém správy verzí Git.

Rozdělení práce mezi členy týmu:
Norbert Ďurčanský (vedoucí) - vedoucí programátor, analýza a realizace algoritmů.
Jindřich Dudek - pomocný programátor.
Jiří Dostál - návrh, tvorba a aplikace testů.
Ján Jusko - pomocný programátor.
Natalya Loginova - vestavěné funkce, dokumentace.

Literatura

1. CORMEN, Thomas H. Introduction to algorithms. 3rd ed. Cambridge: MIT Press, 2009. ISBN 978-0-262-03384-8.
2. MEDUNA, Alexander. Formal languages and computation: models and their applications. Boca Raton: CRC Press, 2014. ISBN 978-1-4665-1345-7.

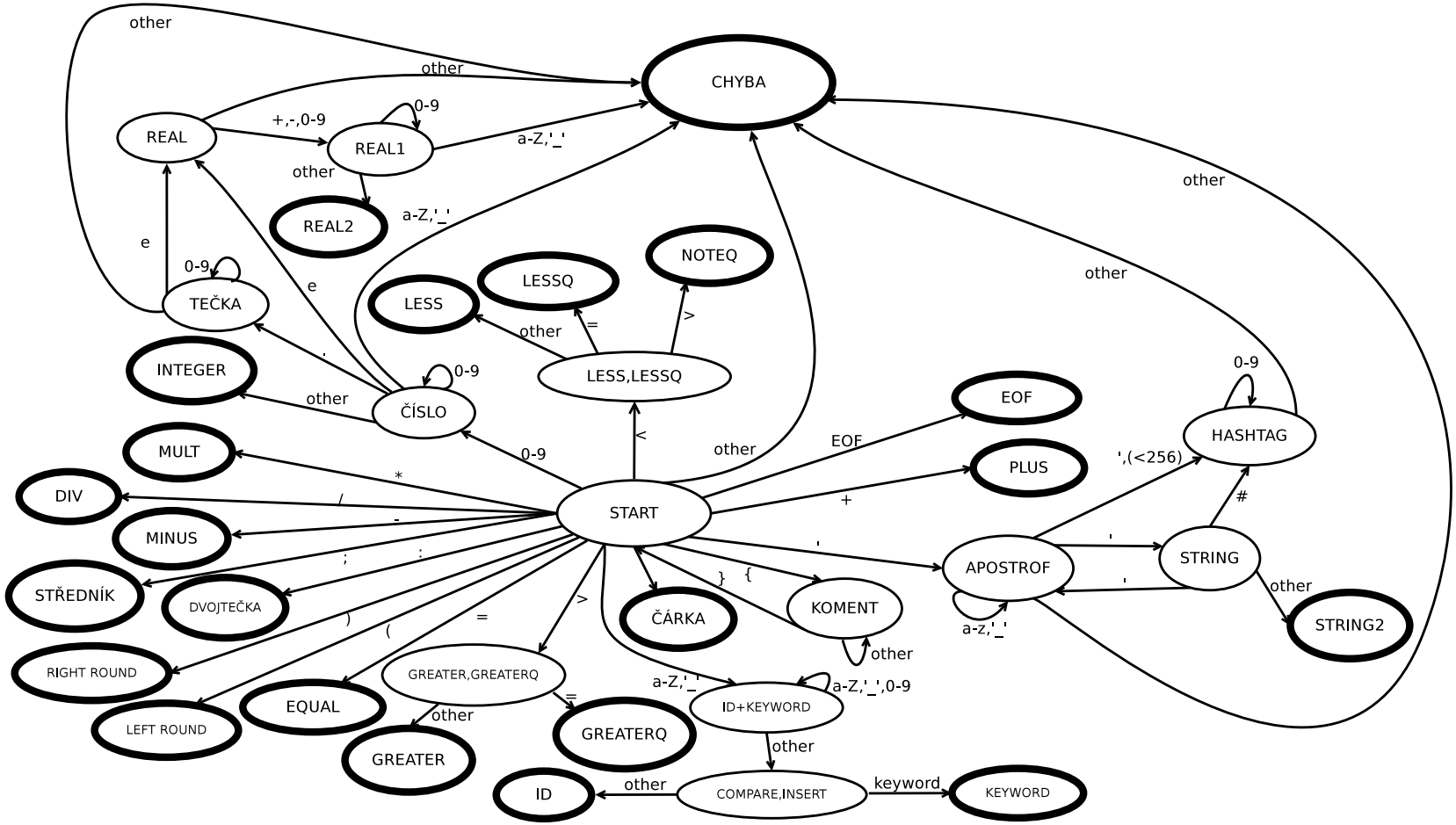
Přílohy

A.1 Pravidla LL gramatiky

1. <PROGRAM>→ var <DECLARELIST>begin <PROG>
2. <PROGRAM>→ begin <PROG>
3. <PROGRAM>→ function <FUNCTION>forward;
4. <PROGRAM>→ function <FUNCTION>; <PROGFUNCTION>
5. <FUNCTION>→ id(<FUN_PARAMS>);
6. <FUN_PARAMS>→ id:<DTYPE>; <FUN_PARAMS>
7. <FUN_PARAMS>→ ε
8. <DTYPE>→ integer
9. <DTYPE>→ real
10. <DTYPE>→ string
11. <DTYPE>→ boolean
12. <PROGFUNCTION>→ begin <PROGCONDITION>
13. <PROGFUNCTION>→ id <COMMAND>
14. <PROGFUNCTION>→ while <COMMAND>
15. <PROGFUNCTION>→ if <COMMAND>
16. <PROGFUNCTION>→ write <COMMAND>
17. <PROGFUNCTION>→ readln <COMMAND>
18. <COMMAND>→ id:=
19. <COMMAND>→ readln(id)
20. <COMMAND>→ write(<TERM>)
21. <TERM>→ id
22. <TERM>→ const_string
23. <TERM>→ const
24. <COMMAND>→ while <PRECEDENCE>do begin <PROGCONDITION>
25. <COMMAND>→ if <PRECEDENCE>then begin <PROGCONDITION>
26. <PROGCONDITION>→ <COMMAND>else begin <PROGFUNCTION>
27. <LIBRARYFUNCTION>→ length(id)
28. <LIBRARYFUNCTION>→ length(const_string)
29. <LIBRARYFUNCTION>→ copy(id,id,id)
30. <LIBRARYFUNCTION>→ copy(id,const,id)
31. <LIBRARYFUNCTION>→ copy(id,id,const)
32. <LIBRARYFUNCTION>→ copy(id,const,const)
33. <LIBRARYFUNCTION>→ copy(conststr,id,id)
34. <LIBRARYFUNCTION>→ copy(conststr,id,const)
35. <LIBRARYFUNCTION>→ copy(conststr,const,id)
36. <LIBRARYFUNCTION>→ copy(conststr,const,const)
37. <LIBRARYFUNCTION>→ find(id,id)
38. <LIBRARYFUNCTION>→ find(id,conststr)
39. <LIBRARYFUNCTION>→ find(conststr,id)

40. <LIBRARYFUNCTION>→ find(conststr,conststr)
41. <LIBRARYFUNCTION>→ sort(id)
42. <LIBRARYFUNCTION>→ sort(conststr)

A.2 Konečný automat



A.3 Precedenční tabulka

	+	-	*	/	ID	(	)	<	>	≥	≤	$	<>
+	>	>	<	<	<	<	>	>	>	>	>	>	>
-	>	>	<	<	<	<	>	>	>	>	>	>	>
*	>	>	>	>	<	<	>	>	>	>	>	>	>
/	>	>	>	>	<	<	>	>	>	>	>	>	>
ID	>	>	>	>			>	>	>	>	>	>	>
(	<	<	<	<	<	<	=	<	<	<	<		<
)	>	>	>	>			>	>	>	>	>	>	>
<	<	<	<	<	<	<	>					>	
>	<	<	<	<	<	<	>					>	
≥	<	<	<	<	<	<	>					>	
≤	<	<	<	<	<	<	>					>	
$	<	<	<	<	<	<	<	<	<	<	<	OK	<
<>	<	<	<	<	<	<	>					>	

A.4 Metriky kódu

Počet souborů: 20 souborů

Počet řádků zdrojového textu: 6564 řádků

Velikost spustitelného souboru: 124.1 kB (systém Linux, 32 bitová architektura, při překladu bez ladicích informací)