Dokumentace k projektu IFJ/IAL
Implementace interpretu imperativního jazyka IFJ12

9. prosince 2012

Tým číslo 17, varianta a/1/I

František Koláček(vedoucí)	xkolac12	20%
Veronika Nečasová	xnecas21	20%
Michaela Muzikářová	xmuzik04	20%
Petr Matyáš	xmatya03	20%
Dalibor Skácel	xskace11	20%

Obsah

1 Úvod

Tento dokument popisuje návrh a implementaci překladače imperativního jazyka IFJ12. Program načte zdrojový soubor a pokud je tento v pořádku, interpretuje kód a vrátí 0. Pokud v pořádku není, je vrácen kód chyby. Při naší práci nám pomohly materiály předmětů IFJ a IAL.

2 Zadání

Program má fungovat jako interpret vymyšleného programovacího jazyka IFJ12. Jazyk IFJ12 je podmnožinou programovacího jazyka Falcon, což je moderní skriptovací jazyk. Tento jazyk kombinuje několik programovacích paradigmat, a je to procedurální, objektově orientovný jazyk.

Jazyk IFJ12 je case-sensitive (záleží na velikosti písmen) a je dynamicky typovaný. Naše varianta zadání zahrnuje:

- Knuth-Morris-Prattův vyhledávací algoritmus
- řadicí metoda Quicksort
- tabulka symbolů, implementovaná binárním vyhledávacím stromem

Program funguje následovně:

1. načte zdrojový soubor, tedy kód programovacího jazyka IFJ12
2. interpret přímo vykonává načtené instrukce

Interpret má podporovat čtyři datové typy, a to číselný datový typ **numeric**, řetězec **string**, logický datový typ **boolean** a speciální datový typ **nil**, který se používá interně a připouští pouze hodnotu nil.

Dále interpret podporuje základní aritmetické operace s těmito typy, řádkové a blokové komentáře, větvení if – else, cyklus while a vestavěné i uživatelem vytvořené funkce. Vestavěné funkce jsou následující: `input()`, `numeric(id)`, `print(term1, term2,...)`, `typeOf(id)`, `len(id)`, `find(string, string)`, `sort(string)`.

3 Části

Interpret se skládá z několika hlavních částí. Tou nejdůležitější je **syntaktický analyzátor**. Syntaktický analyzátor kontroluje, zda řetězec tokenů, jemu zaslaný **lexikálním analyzátorem** reprezentuje syntakticky správně zapsaný program. Za předpokladu že je tento program správný, jej pak **interpret** převede do cílového programu. Jedná se o syntaxí řízený překlad. Moduly pracují s datovými strukturami binární vyhledávací strom, zásobník, seznam a hešovací tabulka.

3.1 Lexikální analyzátor

Úkolem lexikálního analyzátoru neboli scanneru je načíst ze zdrojového programu posloupnost lexémů, které jsou reprezentovány **tokeny**. Tokeny jsou prvky zdrojového souboru - číselné či řetězcové literály, identifikátory a další.

Dále také analyzátor a odstraňuje komentáře a bílé znaky. Zpracované lexémy jsou předávány syntaktickému analyzátoru. Lexikální analyzátor je implementován jako konečný automat o 41 stavech. Načítání a zpracování lexémů obstarává funkce `getToken()`. Příloha č.1 obsahuje model konečného automatu, na jehož základě probíhala samotná implementace lexikálního analyzátoru.

3.2 Syntaktický analyzátor

Další částí překladače je syntaktický analyzátor(parser). Parser je založen na rekurzivním sestupu shora dolů. Jeho úkolem je kontrola, zda řetězec tokenů reprezentuje syntakticky správně napsaný program. Pokud dojde při jeho činnosti k chybě, vypíše se kód chyby a interpretace se nespustí.

Parser žádá o tokeny scanner. Na základě tokenů pracuje s tabulkou symbolů. Syntaxi jazyka IFJ12 jsme popsali LL-gramatikou(příloha č.2). Tato gramatika Precedenční tabulka byla vytvořena dle zadání.

3.3 Sémantický analyzátor

Sémantický analyzátor je hlavní částí našeho překladače. Při sémantické analýze se kontrolují operace nad datovými typy a práce s nimi - například jejich přetypování. Jedná se buď o implicitní konverzi nebo ošetření nepovoleného přetypování. Veškeré kontroly se provádí na základě dat, uložených v derivačním stromu, a to při průchodu stromem zdola nahoru.

3.4 Generátor vnitřního kódu

Generátor vnitřního kódu na základě naplněné tabulky symbolů generuje tříadresný vnitřní kód ke zpracování interpretem. Tento tříadresný kód se generuje kvůli zjednodušení další práce a zjednodušení činnosti interpreteru.

3.5 Interpret

Interpret provádí interpretaci tříadresného kódu, generovaného syntaktickým analyzátorem. Tento modul se zavolá pro funkci main, a pokud jsou v něm volány další funkce, volá rekurzivně sám sebe. Tříadresný kód je uložen v instrukčním listu, který je implementován pomocí jednosměrně vázaného lineárního seznamu. Pro každou instrukci je tento kód reprezentován typem instrukce, adresou výsledku a adresami prvního a druhého operandu. Při zpracovávání aritmeticko-relačních operací provádí interpret sémantickou kontrolu.

4 Použité algoritmy

4.1 Knuth-Moris-Prattův algoritmus

Tento vyhledávací algoritmus využívá ke své práci konečný automat. Je optimalizací triviálního způsobu vyhledávání a jeho výhodou je, že se v prohledávaném řetězci nevrací. Automat načítá postupně znaky řetězce, než se dostane do koncového stavu. Pokud načtený znak je koncem řetězce a automat není v koncovém stavu, jedná se o chybu. Počátečním stavem je -1, konečný stav je délka řetězce.

4.2 Quicksort

Quicksort je jedním z nejrychlejších algoritmů pro řazení polí, založených na porovnávání prvků. Jedná se o nestabilní, nepřirozený algoritmus, jehož asymptotická časová složitost je linearitmická.

Základní myšlenkou je rozdělení posloupnosti řazených prvků na dvě přibližně stejně velké části(metoda rozděl a panuj). Je zvolen pivot, který je středem otáčení. Pokud budou obě části samostatně seřazeny, je seřazeno i celé pole. Obě části se rekurzivně řadí stejným postupem.

Nevýhodou tohoto algoritmu může být nutnost velmi pečlivé implementace. Základní Quicksort je nejpomalejší při třídění již setříděných nebo z větší části setříděných polí. Ve většině případů je ale tento algoritmus velice dobře využitelný.

4.3 Memory management unit

Memory management unit(MMU) je jednotka pro správu paměti a pro práci s ní. Zaznamenává a drží si informace o veškeré alokaci a uvolňování. MMU vnitřně využívá hešovací tabulku. V rámci našeho projektu jsme implementovali MMU pro lepší práci s pamětí a také jejím uvolňováním. Nejdůležitější součástí MMU je

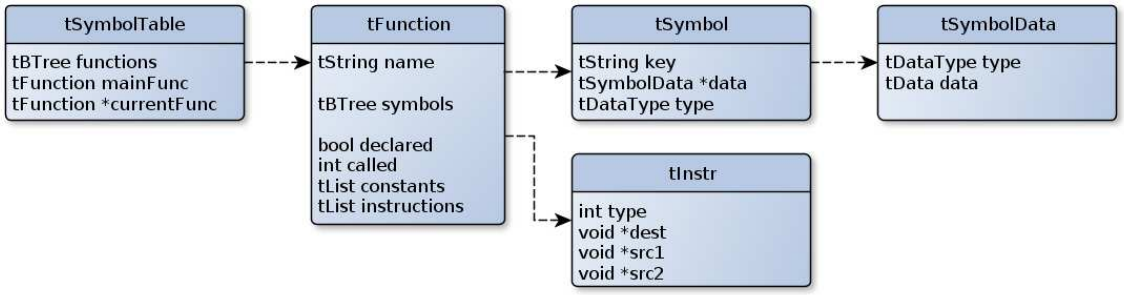
funkce `Globalfree()`. Tato funkce se stará o bezpečné uvolnění veškeré paměti, která nebyla dosud správně uvolněna, a také o otevřené soubory. Pokud se tedy s pamětí pracovalo správně a byla korektně uvolňována již v průběhu činnosti programu, neměla by tato funkce uvolnit již žádnou další paměť.

4.4 Binární vyhledávací strom

Binární vyhledávací strom je datová struktura založená na binárním stromu, kde jsou jednotlivé prvky(uzly) uspořádány tak, aby v tomto stromu bylo možné rychle vyhledat danou hodnotu. Tabulka symbolů je implementována jako binární vyhledávací strom, který uchovává identifikátory funkcí a jejich proměnných, jejichž identifikátory jsou uloženy v dalších binárních vyhledávacích stromech.

4.5 Tabulka symbolů

Tabulka symbolů představuje strom funkcí a ukazatel na právě používanou funkci. Každá funkce obsahuje strom proměnných, seznam konstant a seznam instrukcí. Strom proměnných je binární vyhledávací strom. Seznam konstant a seznam instrukcí jsou jednosměrné seznamy. Pro lepší orientaci je na obr.1 diagram naší tabulky symbolů.



Obrázek 1: Diagram tabulky symbolů

5 Práce v týmu

Náš tým se poprvé setkal asi týden po zveřejnění zadání projektu. Nejprve byla vedoucím rozdělena práce mezi členy týmu a bylo určeno datum dalšího setkání. Takovýchto setkání se konalo 5 v průběhu řešení projektu, nejčastěji v prostorách naší fakulty. Zde se diskutovalo o řešení jednotlivých částí a o problémech, na které jsme při implementaci narazili. Při ladění a testování jsme také využívali program valgrind. Dále se ke komunikaci používal Skype a ke sdílení kódu repozitář GIT. Dokumentace byla napsána v LaTeXu.

6 Závěr

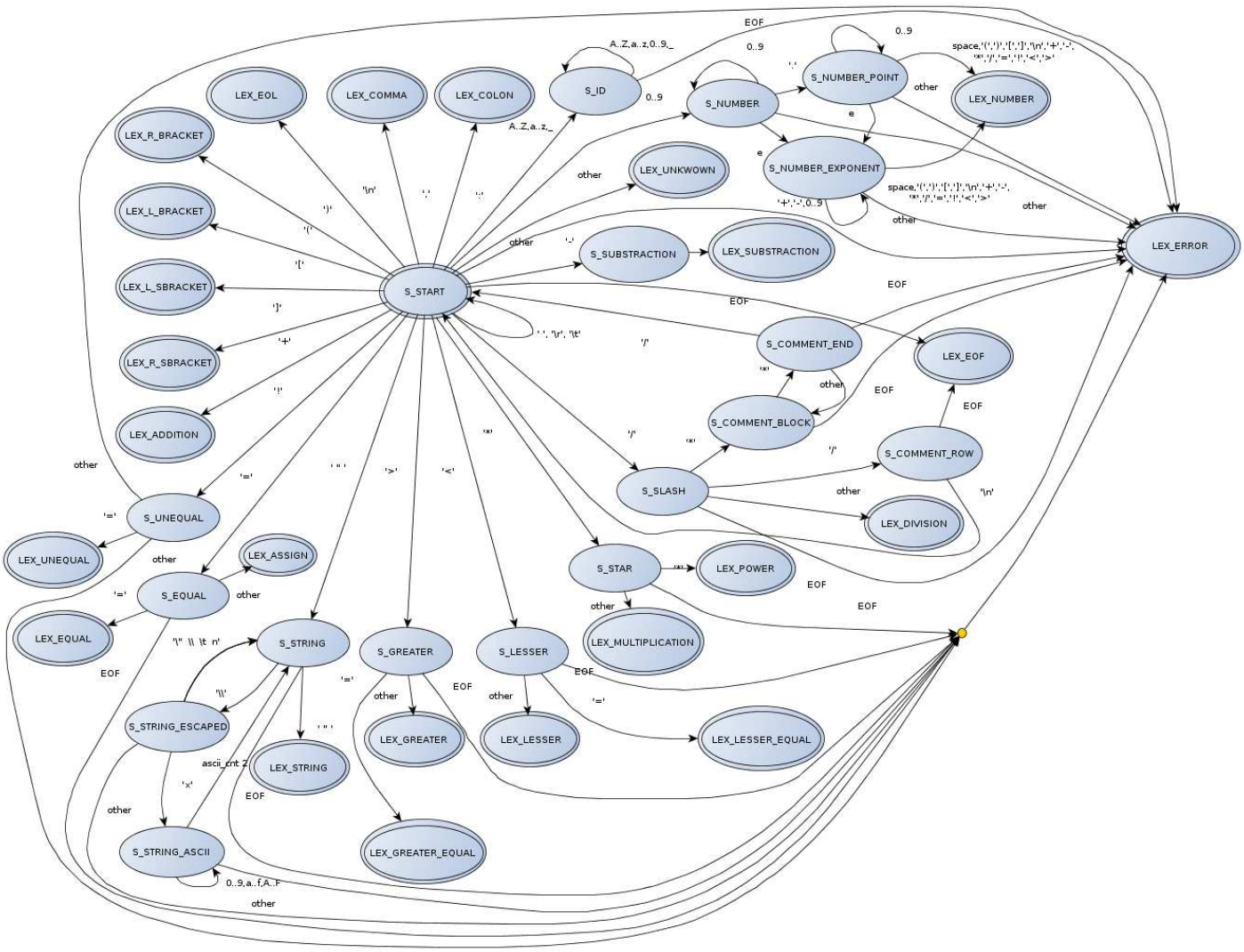
Náš interpret jazyka IFJ12 po dlouhé, usilovné týmové práci funguje. Testování probíhalo za pomoci dostupných příkladů a testů, sloužících pro rychlejší a efektivnější otestování programu, které napsal člen týmu. Interpret důsledně dodržuje veškeré požadavky zadání na vstupní a výstupní formáty dat. Na tomto projektu jsme si vyzkoušeli implementaci některých zajímavých datových struktur a algoritmů, teorii formálních jazyků v praxi a spolupráci v menším týmu.

7 Metriky kódu

Počet řádků kódu: 5523
Počet zdrojových souborů:31
Velikost spustitelného souboru:88 751 bytů (Linux 64bit)

8 Přílohy

8.1 Graf konečného automatu LA



8.2 LL pravidla

< program > → < body _ program >
< program > → < body _ program >
< program > → < body _ program >
< body_program > → < def_function> < body_program>
< body_program > → < def_function> < body_program>
< body_program> → < command> < body_program>
< body_program> → ε
< def_function > → function idFunction (< params>) EOL < stat_list> end EOL
< stat_list> → ε
< stat_list > → < command> < stat_list>
< params > → id < params_n >
< params > → ε
< params_n > → , id < params_n >
< params_n > → ε
< command > → id = < assign>
< command > → if expression EOL < stat_list> else EOL < stat_list> end EOL
< command > → while expression EOL < stat_list> end EOL
< command > → return expression EOL
< assign > → expression
< assign > → idFunction(< params >)
< assign > → input()
< assign > → numeric(id)
< assign > → print(< term >)
< assign > → typeof(id)
< assign > → len(id)
< assign > → find(string, string) || find (id, id)
< assign > → sort(string) || sort (id)
< assign > → string[< num> : < num >] EOL
< num > → ε
< num > → num
< term > → id
< term > → < value >
< value > → num
< value > → string
< value > → logic