



# Dokumentace k projektu IFJ/IAL

## Implementace interpretu imperativního jazyka IFJ13

9. prosince 2013

**Tým číslo 97, varianta a/1/I**

Řešitelé:

|                            |          |     |
|----------------------------|----------|-----|
| František Koláček(vedoucí) | xkolac12 | 25% |
| Lukáš Hermann              | xherma25 | 25% |
| Daniel Stodůlka            | xstodu06 | 25% |
| Tran Manh Hoang            | xtranh00 | 25% |

# Obsah

|  |   |
|--|---|
| 1. Úvod.....                             | 1 |
| 2. Popis řešení.....                     | 1 |
| 2.1 Lexikální analyzátor - scanner.....  | 2 |
| 2.2 Syntaktický analyzátor – parser..... | 2 |
| 2.3 Interpret.....                       | 2 |
| 2.4 Správa paměti.....                   | 3 |
| 3. Použité algoritmy .....               | 3 |
| 3.1 Knuth-Moris-Prattův algoritmus.....  | 3 |
| 3.2 Quicksort.....                       | 3 |
| 4. Práce v týmu.....                     | 4 |
| 5. Závěr .....                           | 4 |
| 6. Příloha.....                          | 5 |

# 1. Úvod

Tento dokument popisuje návrh a implementaci překladače imperativního jazyka IFJ13.

Námi vybrané zadání (**a/1/I**) obsahuje tyto **dané detaily implementace**

- ♣ **Knuth-Morris-Prattův** pro vyhledávání ve stringu
- ♣ Řadící algoritmus **Quicksort**
- ♣ **Binární strom** jako tabulka symbolů

Samotný překladač se skládá ze tří částí, implementovali jsme také vlastní správu paměti v podobě garbage collectoru. Jádrem překladače je **Syntaxtický analyzátor**, který má na starosti překlad zdrojového kódu. Pomocí **Lexikálního analyzátoru** načítá zdrojový kód a podle sémantiky jazyky IFJ13 jej překládá na posloupnost instrukcí. V případě úspěšného překladu se spustí **Intepretace**, který posloupnost instrukcí vykoná.

## 2. Popis řešení

### 2.1 Lexikální analyzátor - scanner

Definuje `struct_Token` a funkci `getToken()`, který na požádání syntaktického analyzátoru načte ze vstupního souboru jeden lexém, který je pak vrácen jako token, jež obsahuje jeho typ. Typem může být například číslo, klíčové slovo, řetězec apod.

Vlastní scanner je řešen jako konečný automat. Načítá znaky ze vstupního souboru pomocí funkce `getc()`. V některých případech načte scanner znak patřící k dalšímu tokenu. V těchto případech volá se funkce `ungetc()`, která vrátí znak do vstupního souboru. V případě neznámého lexému se vrátí příslušný chybový kód.

### 2.2 Syntaktický analyzátor – parser

Syntaktický analyzátor tvoří jádro celého překladače. Zajišťuje přeložení zdrojového kódu na posloupnost instrukcí. Při implementaci syntaktického analyzátoru jsme využili LL gramatiku (viz. Příloha 2), postupovali jsme metodou rekurzivního sestupu. Postupně jsme ukládali veškerá data, se kterými později bude pracovat i interpret, binární strom. Pro zpracovávání výrazů využíváme syntaktickou analýzu zdola horů.

Oba dva druhy analýz kooperují, hlavní řízení je podle rekurzivního sestupu, avšak když se narazí na výraz zavolá se analýza zdola nahoru. Podstatou syntaktické analýzy zdola nahoru je precedenční tabulka, na jejímž základě jsou řízeny veškeré redukce a vyhodnocování výrazu.

### 2.3 Interpret

Pro náš projekt jsme využili generátor tříadresného kódu, který používá námi vytvořenou instrukční sadu. Generátor rozlišuje plnohodnotný instrukci a instrukci skoku k zachování přehlednosti kódu. Tyto instrukce se pro následné využití interpretu ukládají do listu instrukcí. Interpret zpracovává instrukce lineárně s výjimkou instrukcí skoků nebo návratů z funkcí. Abychom předešli duplicitní kontrole dat, tak při aritmetických operacích interpret také plní funkci semantického analyzátoru.

## 2.4 Správa paměti

Jednotka pro správu paměti je nádstavbou nad základními funkcemi (malloc, realloc apod.) a také prací se soubory. Pro její implementaci jsme využili jednoduchý jednosměrný seznam.

Nejpodstatnější částí je funkce gcDispose(), která zajišťuje globální uvolnění veškeré použité paměti.

## 3. Použité algoritmy

### 3.1 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.

Implementuje také prefixovou funkci, která ještě před samotným vyhledáváním zajišťuje uložení informací o stringu a jeho chování při posunu hledaného stringu – díky tomuto nedochází ke vracení se k již kontrolovaným znakům.

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.

### 3. 2 Quicksort

Quicksort je jeden z nejrychlejších běžných algoritmů řazení založených na porovnávání prvků. Jedná se o nestabilní, nepřírozený algoritmus, jehož asymptotická časová složitost je lineární, výhodou je také jeho jednoduchost. Základní myšlenkou quicksortu je rozdělení řazené posloupnosti čísel na dvě přibližně stejně velké části (metoda rozděl a panuj). Kde v jedné části jsou čísla menší a ve druhé větší, pak je zvolen pivot, který je středem otáčení. Pokud budou obě části samostatně seřazeny, je seřazeno i celé pole. Obě části se pak rekurzivně řadí stejným postupem. Základní Quicksort je nejpomalejší při třídění již setříděných nebo z větší části setříděných polí.

## **4. Práce v týmu**

Práce v týmu probíhala již od počátečního zveřejnění zadání, kdy náš vedoucí rozdal úkoly a stanovil termíny jejich plnění. Díky společnému bytu našeho týmu komunikace nebyla problémem. Ke sdílení a práci se zdrojovými soubory jsme použili GIT.

## **5. Závěr**

V zadaném termínu odevzdání náš interpret zpracovává ukázkové příklady, další ladění probíhá v režii vlastních testovacích skriptů a také programu valgrind.

## 6. Příloha

### 1. LL gramatika

```
[TTYPE_VARIABLE]={ [TTYPE_VARIABLE]= 0 , 0 , '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_L_BRACKET]={ [TTYPE_VARIABLE]='<', '<', '=', '<', '<', '<', '=', '<', '<', '<', '<', '<',
'<', '<', '<', [TTYPE_SEMICOLON]= 0 },

[TTYPE_R_BRACKET]={ [TTYPE_VARIABLE]= 0 , 0
, '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_ADDITION]={ [TTYPE_VARIABLE]='<', '<', '>', '>', '>', '<', '>', '<', '>', '>', '>', '>',
'>', '>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_SUBTRACTION]={ [TTYPE_VARIABLE]='<', '<', '>', '>', '>', '<', '>', '<', '>', '>', '>', '>',
'>', '>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_MULTIPLICATION]={ [TTYPE_VARIABLE]='<', '<', '>', '>', '>', '>', '>', '>', '<', '>', '>',
'>', '>', '>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_DIVISION]={ [TTYPE_VARIABLE]='<', '<', '>', '>', '>', '>', '>', '>', '<', '>', '>', '>', '>',
'>', '>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_DOT]={ [TTYPE_VARIABLE]='<', '<', '>', '>', '>', '<', '>', '<', '>', '>', '>', '>', '>',
'>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_EQUAL]={ [TTYPE_VARIABLE]='<', '<', '>', '<', '<', '<', '>', '<', '>', '>', '>', '>',
'>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_NOT_EQUAL]={ [TTYPE_VARIABLE]='<', '<', '>', '<', '<', '<', '>', '<', '>', '>', '>', '>',
'>', '>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_LESSER]={ [TTYPE_VARIABLE]='<', '<', '>', '<', '<', '<', '>', '<', '>', '>', '>', '>', '>',
'>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_LESSER_EQUAL]={ [TTYPE_VARIABLE]='<', '<', '>', '<', '<', '<', '>', '<', '>', '>', '>', '>',
'>', '>', '>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_GREATER]={ [TTYPE_VARIABLE]='<', '<', '>', '<', '<', '<', '>', '<', '>', '>', '>', '>', '>',
'>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_GREATER_EQUAL]={ [TTYPE_VARIABLE]='<', '<', '>', '<', '<', '<', '>', '<', '>', '>', '>', '>',
'>', '>', '>', '>', [TTYPE_SEMICOLON]='>' },

[TTYPE_SEMICOLON]={ [TTYPE_VARIABLE]='<', '<', 0 , '<', '<', '<', 0
, '<', '<', '<', '<', '<', '<', '<', [TTYPE_SEMICOLON]='$' },
```

## 2. Konečný automat scanneru

