



Dokumentace k projektu pro předměty IAL a IFJ  
Implementace intepretu jazyka IFJ15

13. prosince 2015

Řešitel	Login	Rozdělení bodů
Jakub Lužný (vedoucí)	xluzny00	25%
Jakub Lukáč	xlukac09	25%
Viktor Jančík	xjanci09	25%
Dušan Litvák	xlitvi01	25%

# Obsah

<b>1</b>	<b>Úvod</b>	<b>1</b>
1.1	Rozbor zadania . . . . .	1
<b>2</b>	<b>Implementácia</b>	<b>2</b>
2.1	Lexikálny analyzátor . . . . .	2
2.2	Syntaktický a sémantický analyzátor . . . . .	2
2.3	Precedenčná analýza . . . . .	4
2.4	Interpret . . . . .	5
2.5	Riešenie vybraných algoritmov . . . . .	5
2.5.1	Tabuľka symbolov . . . . .	5
2.5.2	Boyer-Moore . . . . .	5
2.5.3	Quick-sort . . . . .	5
2.6	Popis riešenia . . . . .	6
2.6.1	Rozdelenie práce . . . . .	6
2.6.2	Metriky kódu . . . . .	6
2.7	Záver . . . . .	6
2.8	Literatúra . . . . .	7

# 1 Úvod

Táto dokumentácia popisuje implementáciu interpretu imperatívneho jazyka IFJ15. Implementácia je projektom do predmetov *Formální jazyky a překladače* a *Algoritmy*. Úlohou tohoto interpreta je kontrola zdrojového kódu a jeho interpretácia, v prípade, že je zdrojový kód v poriadku. V opačnom prípade dôjde ku chybovému stavu.

Projekt je možné rozdeliť na niekoľko hlavných celkov:

- Lexikálny analyzátor – zo zdrojového kódu získava tokeny.
- Syntaktický analyzátor – skladá sa z analyzátora jazykových konštrukcií a analyzátora výrazov.
- Interpret.

Táto dokumentácia má za cieľ popísať proces konštrukcie interpretu z rozdielnych hľadísk.

## 1.1 Rozbor zadania

Vytvorenie programu v jazyku C, ktorý má fungovať ako interpret jazyka IFJ15, ktorý je podmnožinou jazyka C++11.

Interpret má podporovať nasledujúce dátové typy:

- string,
- int,
- double

Samozrejmosťou sú základné aritmetické operácie s uvedenými dátovými typmi. Okrem toho má interpret podporovať vetvenie **if** – **else** a cyklus **for**. Ďalším kľúčovým slovom je **return**. Jednou z nevyhnutných častí akéhokoľvek programovacieho jazyka sú komentáre, náš interpret ich musí podporovať vo forme riadkových komentárov ( `//` ), ale i vo forme viacriadkových komentárov ( `/* ...*/` ). Interpret by mal byť schopný spracovať základné aritmetické operátory `+`, `-`, `*`, `/` ako i relačné operátory `<`, `>`, `>=`, `<=`, `==`, `!=`. Interpret podporuje modifikátor **auto**

Naviac je schopný pracovať so vstavanými funkciami:

- `int length (string s)`
- `string substr (string s, int i, int n)`
- `string concat (string s1, string s2)`
- `int find (string s, string search)`
- `string sort (string s)`

## 2 Implementácia

### 2.1 Lexikálny analyzátor

Lexikálny analyzátor je prvou časťou interpreta, ktorá príde do kontaktu s kódom. Jeho úloha spočíva v čítaní zdrojového kódu, jeho rozdelenie na lexémy a ich následnú logickú reprezentáciu tzv. *tokenmi*, ktorí sú výstupmi lex. analyzátoru.

Jednotlivé tokeny obsahujú informácie o type posiadaného tokenu, ďalej v závislosti na type informáciu či sa jedná o aritmetický alebo relačný operátor, v prípade typu string samotný reťazec a v prípade double alebo int jeho hodnotu.

Nevyhnutnou úlohou lexikálneho analyzátoru je odstránenie akýchkoľvek komentárov a bielych znakov zo zdrojového programu. Lexikálny analyzátor taktiež kontroluje lexikálnu stránku kódu a odhaľuje lexikálne chyby.

Lexikálny analyzátor je implementovaný pomocou konečného automatu, ktorého grafického interpretácia je znázornená nižšie. Pre lepšie grafické znázornenie boli do schémy pridané niektoré stavy, reálne sa nenachádzajúce v zdrojovom kóde.

### 2.2 Syntaktický a sémantický analyzátor

Fukčným jadrom interpretu je syntaktický analyzátor. Jednou z úloh je kontrola syntaktickej správnosti programu. Syntax a jej pravidlá sú definované pomocou LL gramatiky, čiže syntaxe jazyka. Ďalej prebieha sémantická kontrola. Úlohou sémantickej kontroly je napr. kontrolovať korektné použitie dátových typov. Hlavnou úlohou syntaktického analyzátoru je v prípade správnosti zdrojového programu jeho preklad na postupnosť pseudoinštrukcií. V opačnom prípade dôjde ku syntaktickej chybe.

Analyzátor si pomocou funkcie `token_t get_next_token(scanner* s)` načíta jednotlivé tokeny (dodávané lexikálnym analyzátorom) zo zdrojového súboru. Tie následne spracováva pomocou metód syntaktickej analýzy a vytvára pseudokód.

V našej implementácii používame dve rôzne tabuľky symbolov. Jedna je globálna tabuľka funkcií, druhá je zásobník tabuliek premenných. So zásobníkom pracujeme tak, že pri vstupe do nového bloku vytvoríme novú tabuľku premenných pre ten blok, a vložíme ju na vrchol zásobníku. Pri odchode z bloku, tabuľku premenných na vrchole zásobníku odstránime a dealokujeme.

Syntaktická analýza prebieha metódou rekurzívneho zostupu a využíva pravidlá LL gramatiky.

```
LL gramatika
TODO jebou epsilon
TODO jebou epsilon
TODO jebou epsilon
TODO jebou epsilon
TODO jebou epsilon
TODO jebou epsilon
TODO jebou epsilon
TODO jebou epsilon
TODO jebou epsilon
TODO jebou epsilon
TODO jebou epsilon
```



	<>=	+-	*/	(	)	id	\$
<>=	>	<	<	<	>	<	>
+-	>	>	<	<	>	<	>
*/	>	>	>	<	>	<	>
(	<	<	<	<	=	<	E
)	>	>	>	E	>	E	>
id	>	>	>	E	>	E	>
\$	<	<	<	<	E	<	E

Tabulka 1: Precedenční tabulka

TODO jebou epsilon  
 TODO jebou epsilon

Vstupný bod syntaktického analyzátoru je funkcia `parse_program`, ktorá volá funkciu `parse_funcs`, ktorá slúži ako vstupný bod pre LL gramatiku. Následne pomocou rekurzívnych volaní funkcií `parse_*`, kde `*` zodpovedá názvom produkcií v LL gramatike. Zároveň s syntaktickou analýzou sa vykonáva sémantická analýza v príslušných funkciách produkcií. Funkcia `parse_expr` slúži na syntaktickú a sémantickú analýzu výrazov a zároveň ako vstupný bod pre precedenčný syntaktický analyzátor. Zároveň so syntaktickou a sémantickou analýzou sa vykonáva generovanie kódu.

## 2.3 Precedenčná analýza

Precedenčná analýza slúži ku vyhodnocovaniu výrazov. Kedykoľvek syntakt. analyzátor narazí v pravidle na výraz, je volaná preced. analýza, ktorá mu následne vracia spracovaný výraz.

Ku svojej činnosti potrebuje a využíva lexikálny analyzátor, ktorý jej dodáva tokeny. Pri aplikácii precedenčných pravidiel sú priamo generované inštrukcie. Syntaktické pravidlá pre precedenčnú analýzu sú jej základom a spolu s precedenčnou tabuľkou sú zadefinované nižšie.

```

Expression grammar (missing function calls, rel ops)
expr -> expr < expr | expr > expr | expr <= expr | expr >= expr | expr != expr
expr -> expr + expr | expr - expr
expr -> expr * expr | expr / expr
expr -> ( expr )
expr -> int | double | string | id

```

Samotný algoritmus je tvorený, postupným načítavaním jednotlivých tokenov a vykonávaním operácií určenej precedenčnou tabuľkou s využitím zásobníku. Jednotlivé znaky v tabuľke vyjadrujú operácie `push(<)`, `reduce(>)`, `push only(=)` a nedovolenú kombináciu (`E`). Operácia je z tabuľky vybraná na základe načítaného terminálu a posledného terminálu na zásobníku. Takže dochádza ku postupnému vyhodnocovaniu a redukovaniu symbolov na zásobníku pomocou pravidiel precedenčnej tabuľky. Pre potreby algoritmu bol implementovaný zásobník, ktorý predstavuje pomocnú dátovú štruktúru.

## 2.4 Interpret

Úlohou interpretu je vykonávanie programu, zapísaného pomocou pseudoinštrukcií. Interpret postupne lineárne prechádza a vykonáva postupnosť inštrukcií. Ako pomocnú dátovú štruktúru využíva zásobník, kde uchováva potrebné dáta, premenné, medzivýsledky operácií a riadiace hodnoty. Navyše umožňuje aj nelineárny posun vo vektore inštrukcií pomocou skokových inštrukcií. Jazyk IFJ15 umožňuje aj volanie funkcií, z toho dôvodu bolo potrebné implementovať mechanizmus volania ako súčasť interpreteru. Výsledné riešenie vytvorí nový funkčný rámec na zásobníku dát a následne opäť lineárne vykonáva postupnosť pseudoinštrukcií. Pri návrate z funkcie je obnovený funkčný rámec volajúcej funkcie a návratová hodnota je predaná na vrchole zásobníku. Vstupná funkcia do interpretovaného programu je funkcia `main`, ktorej adresa je počiatočnou adresou pseudoinštrukcie pri začatí vykonávania programu.

## 2.5 Riešenie vybraných algoritmov

### 2.5.1 Tabuľka symbolov

Tabuľka symbolov je implementovaná formou hešovacej tabuľky. Tabuľka sa skladá z poľa zretazených zoznamov. Prvok zretazeného zoznamu je `symbol_t` ktorý obsahuje všetky potrebné informácie o danej premennej / funkcii. Ako hešovací algoritmus sme použili modulárne hešovanie. Každý znak v reťazci sa zahešuje ako jeho ASCII hodnota plus  $31 \cdot$  heš kód predošlých znakov v reťazci. Celkový heš kód reťazca sa potom dá do rozsahu veľkosti hešovacej tabuľky, pomocou modula veľkosti hešovacej tabuľky. Hešovacia tabuľka sa automaticky zväčšuje keď priemerná dĺžka zretazeného zoznamu nadobudne dĺžku väčšiu ako 5 prvkov za predpokladu uniformného hešovania. Veľkosti hešovacej tabuľky sú vždy prvočísla, aby sa predišlo situáciám kde heš kód nejakého stringu má spoločný faktor s veľkosti hešovacej tabuľky. Toto opatrenie zlepšuje uniformnosť výsledných hešov.

### 2.5.2 Boyer-Moore

Tento algoritmus je považovaný za najefektívnejší v bežných aplikáciách. Algoritmus porovnáva vzor s textom sprava doľava, začínajúc prvkom najviac vpravo. Postup v prípade nezhody sa odvíja od použitej heuristiky. V našom projekte sme použili tzv. Bad Character Shift. Táto heuristika si pred vyhľadávaním zostaví tabuľku, kde je pre každý znak údaj o tom, ako ďaleko je jeho ďalší najbližší výskyt. Následne sa pri nezhode vždy posunie o tento počet znakov, namiesto jedného ako by to robil naivný algoritmus.

### 2.5.3 Quick-sort

Ide o radiaci algoritmus s najmenšou priemernou zložitou na reálnych dátach. Na začiatku sa taktiež zvolí pivot. Jeho voľba má zásadný vplyv na rýchlosť algoritmu. V ideálnom prípade by sa malo jednať o medián. V našej implementácii sa ako pivot volí prostredný prvok poľa. Podľa tohoto pivota sa pole rozdelí na dve polovice a prvky prehádzame tak, aby na jednej strane boli menšie ako pivot a na druhej strane naopak väčšie. Tent postup opakujeme pre obidve rozdelené časti (bez pivota, ktorý už je umiestnený na správnom mieste). Procedúru opakujeme až kým nenarazíme na všetky triviálne riešiteľné podproblémy, tzn. pole o veľkosti 1.

## 2.6 Popis riešenia

Pretože na zložitosť projektu a jeho náročnosť na čas, ale i schopnosti sme boli upozornení dopredu, v dostatočnom predstihu sme si zložili tím. Mesiac po začiatku semestra sme si stanovili pravidelné týždenné schôdzky, kde sme riešili vykonanú prácu, analyzovali postup práce a jednotlivé problémy.

Praktická tvorba kódu prebiehala s využitím verzovacieho nástroja Git, na komunikáciu slúžila uzavretá skupina na Facebook-u. Komunikačný kanál nebol zvolený vhodne, čo sa ukázalo v okamihu keď sa zväčšil objem komunikácie a stala sa neprehľadnou. Následne sme prešli na Slack. Pre potreby zdieľania informácií sme využili Google Docs.

Práca na vybraných jednotlivých častiach projektu prebiehala paralelne, naopak iné časti na seba boli úzko naviazané a s vývojom sa čakalo, resp. vývoj týchto častí mal na starosti jeden človek pre uľahčenie tvorby kódu a celistvosť práce.

Mnohé časti implementácie boli predmetom diskusií (napr. forma a obsah tokenov predávaných lexikálnym analyzátorom), naopak iné časti boli plne v kompetencii daného člena tímu.

### 2.6.1 Rozdelenie práce

**Dušan Litvák** lexikálny analyzátor, dokumentácia

**Jakub Lužný** vstavané funkcie, vedenie projektu

**Jakub Lukáč** syntaktická a sémantická analýza výrazov, interpret

**Viktor Jančík** návrh a implementácia dátových štruktúr, syntaktická a sémantická analýza mimo výrazov

### 2.6.2 Metriky kódu

Počet súborov: 31

Počet riadkov zdrojového kódu: 5279 //TODO

Veľkosť statických dát:

Veľkosť spustiteľného súboru:

## 2.7 Záver

Hoci šlo len o školský projekt, museli sme sa vysporiadať s rovnakými problémami aké sa vyskytujú v každom komerčnom projekte, či lepšie povedané v reálnom prostredí. Stratégia vývojového procesu, medziľudská komunikácia, uchovávanie a zdieľanie informácií a mnoho ďalších problémov nám prinieslo mnoho užitočných skúseností.



## 2.8 Literatúra

### Reference

[IFJop] *Studijní opora IFJ.*

[IFJpr] *Přednášky IFJ.*

[BM] *Boyer-Moore algorithm.* (1997).  
<http://www-igm.univ-mlv.fr/~lecroq/string/node14.html>