VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ


Dokumentace projektu do předmětu IFJ a IAL

# Interpret jazyka IFJ15

Tým 043, varianta b/2/I

Martin Honza (xhonza03) - 20%
Patrik Jurnečka (xjurne03) - 20%
Hana Slámová (xslamo00) - 20%
Frantisek Šumšal (xsumsa01) - 20%
Adam Švidroň (xsvidr00) - 20%

14. prosince 2015

# Obsah

# 1 Úvod

Dokumentace popisuje návrh a implementaci projektu Interpret jazyka IFJ15 do předmětu IFJ (Formální jazyky a překladače) a IAL (Algoritmy). Vybrali jsme si zadání b/2/I, které nám udávalo, jáký algoritmus máme pro daný problém využít. Pro vyhledávání **Boyer-Mooreův algoritmus**, pro řazení algoritmus **Heapsort**, který byl využit ve vestavěné funkci **sort**. Poslední ze specifikových pravidel použití, jsme měli využít k implementaci tabulky symbolů **binární vyhledávací strom**.

Cílem projektu bylo vytvořit program, který interpretuje jazyk IFJ15, což je velmi zjednodušenou podmnožinou jazyka C++11.

# 2 Struktura projektu

Překladač je rozdělen do tří hlavních celků. Pomocí **lexikálního analyzátoru** načítá zdrojový kód. **Syntaktický analyzátor** zažádá lexikální o token a ověří syntaxi. Po bezchybné kontrole se spustí **interpret** s kontrolou sémantiky jazyka.

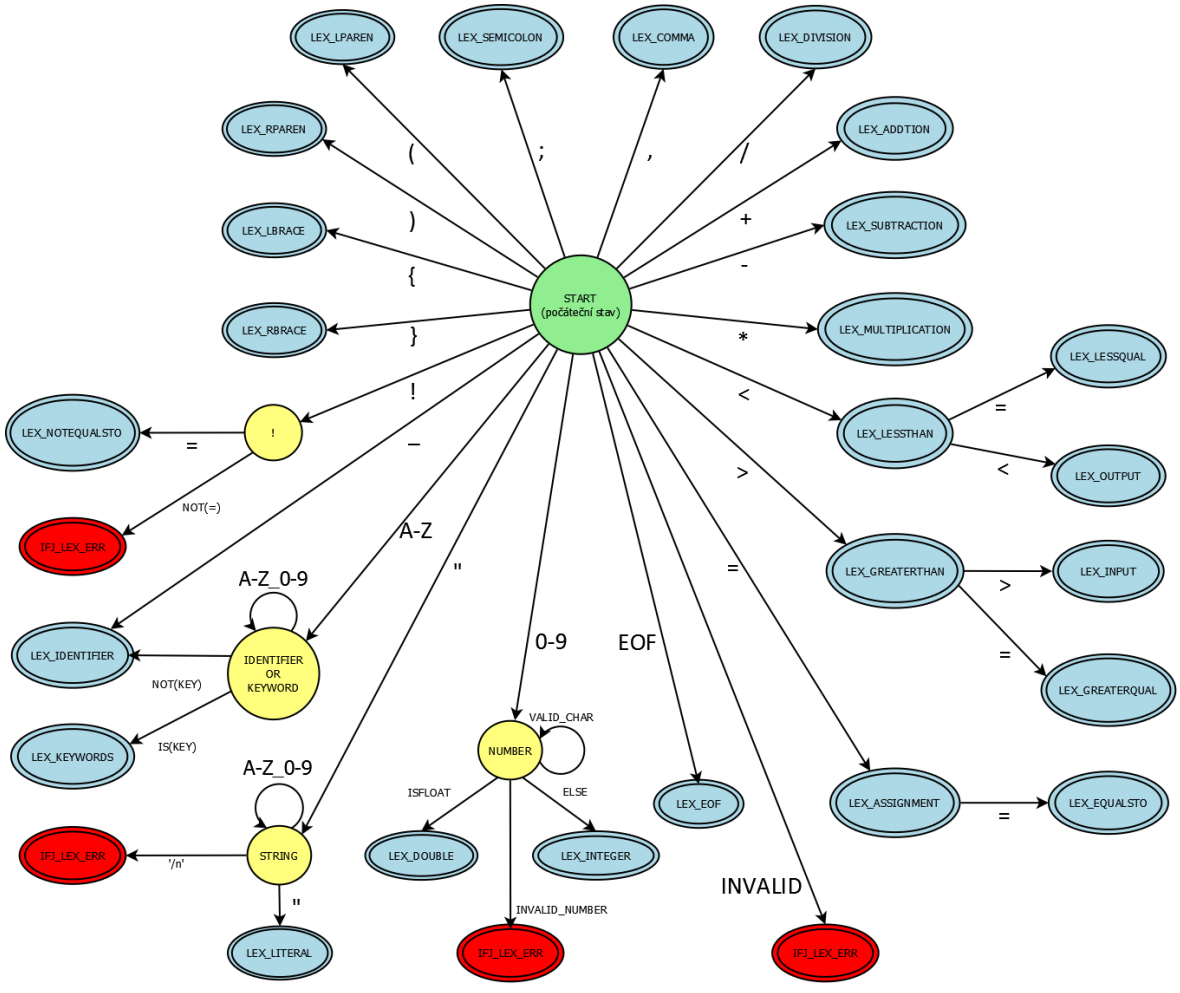
# 3 Lexikální analizátor (scanner)

Analyzátor je implementován v souborech *lex.c* a *lex.h*. Lexikální analyzátor načítá zdrojový kód po znacích a převádí jej na tokeny. Veškerá komunikace s dalšími knihovnami probíhá pomocí funkce *lex_get_token*. Analyzátor je implementován pomocí konečného automatu (Obrázek 1), přičemž veškeré komentáře ve zdrojovém kódu ignoruje. Funkce je volána vždy, když syntaktický analyzátor zažádá o token.

Načítání znaků zajišťuje funkce *fgetc()*. Na základě načteného lexému se rozhodne o jaký typ tokenu se jedná. V některých případech je potřeba načtený znak vrátit do souboru pomocí funkce *ungetc()*.

Token definuje jednotlivé prvky kódu a určuje jak se k němu má při překladu přistupovat. Analyzátor prochází tabulku klíčových slov (*lex_kw_t keywords[]*), zda-li nejde o rezervované klíčové slovo pro speciální funkci.

## 3.1 Konečný automat




Obrázek 1: Konečný automat

# 4 Syntaktická analizátor (parser)

Syntaktický analyzátor kontroluje syntaktickou správnost zdrojového kódu. Postupně volá tokeny z lexikálního analyzátoru, které jsou zpracovávaný dvěma způsoby. Pomocí LL gramatiky (Tabulka 1) a precedenční syntaktické analýzy (Tabulka 2), které jsou jádrem celého SA. Při úspěšném dokončení syntaktické analýzy se generuje instrukční páska pro interpret.

## 4.1 Obecný syntaktický analyzátor (LL gramatika)

LL gramatika postupně ověřuje rekurzivním sestupem příchozí tokeny. Funkce jsou navrženy tak, aby ověřili aktuální token nebo požádali lexikální analyzátor, funkcí *lex_get_token* o následující. Nedá-li se rozhodnout na základě pouze jednoho tokenu zažádá si o další.

| | | | |
|---|---|---|---|
| 1. | <program> | → | <declrList> EOF |
| 2. | <declrList> | → | <funcDeclr> <declrList> \| EPSILON |
| 3. | <funcDeclr> | → | <typeSpec> ID ( <params> ) <funcStmt> |
| 4. | <funcStmt> | → | <compoundStmt> \| ; |
| 5. | <params> | → | <paramItem> \| EPSILON |
| 6. | <paramItem> | → | <typeSpec> ID , <paramItem> \| <typeSpec> ID |
| 7. | <typeSpec> | → | int \| double \| string |
| 8. | <statement> | → | <compoundStatement> \| <ifStmt> \| <forStmt> \| <assignEndStmt> \| <callStmt> \| <returnStmt> \| <inputStmt> \| <outputStmt> \| <varDeclr> \| <expressionStmt> |
| 9. | <compoundStmt> | → | <stmtList> |
| 10. | <stmtList> | → | <statement> <stmtList> \| EPSILON |
| 11. | <varDeclr> | → | <typeSpec> <varDeclrItem> ; \| auto <varInitialize> ; |
| 12. | <varDelcrItem> | → | ID \| <varInitialize> |
| 13. | <varInitialize> | → | ID = <expression> |
| 14. | <ifStmt> | → | if ( <expression> ) <compoundStmt> else <compoundStmt> |
| 15. | <forStmt> | → | for ( <varDeclr> ; <expression> ; <assignStmt> ) <compoundStmt> |
| 16. | <assignEndStmt> | → | <assignStmt> ; |
| 17. | <assignStmt> | → | ID = <expression> \| ID = <call> |
| 18. | <callStmt> | → | ID = <call> |
| 19. | <call> | → | ID ( <callParams> ) |
| 20. | <callParams> | → | <callParam>, <callParams> \| <callParam> \| EPSILON |
| 21. | <callParam> | → | ID \| LITERAL |
| 22. | <returnStmt> | → | return <expression> ; |
| 23. | <inputStmt> | → | cin ≫ ID <inputArgs> ; |
| 24. | <inputArgs> | → | ≫ ID \| EPSILON |
| 25. | <outputStmt> | → | cout ≪ <callParam> <outputArgs> ; |
| 26. | <outputArgs> | → | ≪ <callParam> \| EPSILON |

Tabulka 1: LL gramatika

### 4.2 Syntaktický analizátor výrazů (precedenční syntaktická analýza)

Syntaktická analýza zdola nahoru. Dle Zadání jsme analýzu implementovali za pomocí precedenční tabulky (Tabulka 2) a níže uvedenou gramatikou, která nám slouží k vyhodnocení vstupních výrazů. Podle zadání projektu jsme vytvořili precedenční tabulku ze zadaných priorit operátorů. SA výrazů je volán vždy, když obecný SA narazí na výraz. Funkce si ukládá jednotlivé příchozí tokeny na zásobník a poté je v závislostí na prioritách jednotlivých prvků vyhodnocuje.

$$\begin{array}{llllll} E \to \text{id} & E \to (E) & E \to E * E & E \to E \div E & E \to E{+}E & E \to E{-}E \\ E \to E < E & E \to E \leq E & E \to E > E & E \to E \geq E & E \to E == E & E \to E \neq E \end{array}$$

| | + | − | * | / | < | > | <= | >= | == | != | ( | ) | id | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| + | > | > | < | < | > | > | > | > | > | > | < | > | < | > |
| − | > | > | < | < | > | > | > | > | > | > | < | > | < | > |
| * | > | > | > | > | > | > | > | > | > | > | < | > | < | > |
| / | > | > | > | > | > | > | > | > | > | > | < | > | < | > |
| < | < | < | < | < | < | > | > | > | > | > | < | > | < | > |
| > | < | < | < | < | < | > | > | > | > | > | < | > | < | > |
| <= | < | < | < | < | < | > | > | > | > | > | < | > | < | > |
| >= | < | < | < | < | < | > | > | > | > | > | < | > | < | > |
| == | < | < | < | < | < | > | > | > | > | > | < | > | < | > |
| != | < | < | < | < | > | > | > | > | > | > | < | > | < | > |
| ( | < | < | < | < | < | < | < | < | < | < | < | = | < | |
| ) | > | > | > | > | > | > | > | > | > | > | | > | | > |
| id | > | > | > | > | > | > | > | > | > | > | | > | | > |
| $ | < | < | < | < | < | < | < | < | < | < | < | | < | = |

Tabulka 2: Precenenční tabulka syntaktické analýzy výrazů

## 5 Interpret

Poslední částí Interpretu je vlastní překlad, který nastane pouze pokud zdrojový kód úspěšně projde lexikální a syntaktickou analýzou. Interpretační část ma za úkol zpracovat instrukční sadu založenou na 3AC (tříadresný kód). Generování 3AC probíhá v souborech *interpret_gen.c* a *interpret_gen.h*. Samotnou interpretaci lze nalézt v *interpret.c* a *interpret.h*.

Tříadresný kód, může obsahovat několik instrukcí (Tabulka 3), které jsou předány k interpretaci. Každá uvedená instrukce obsahuje tři ukazatele. První ukazatel (*addr1*) udává výsledek dáne operace, zbylé dva jsou operandy (*addr2* a *addr3*). Instrukce očekávají přesný typ ukazatele, pakliže jsou jiného typu než očekává jedná se o sémantickou chybu. **Sémantická analýza** je prováděna během interpretace.

| | | | |
|---|---|---|---|
| INSTR_HALT | INSTR_CALL_LENGTH | INSTR_CALL_SUBSTR | INSTR_CALL_CONCAT |
| INSTR_CALL_FIND | INSTR_CALL_SORT | INSTR_CIN | INSTR_COUT |
| INSTR_CALL | INSTR_RET | INSTR_PUSHF | INSTR_PUSHP |
| INSTR_MOVI | INSTR_MOVD | INSTR_MOVS | INSTR_ADD |
| INSTR_SUB | INSTR_MUL | INSTR_DIV | INSTR_LT |
| INSTR_GT | INSTR_LTE | INSTR_GTE | INSTR_EQ |
| INSTR_NEQ | INSTR_JMP | INSTR_JMPC | INSTR_LAB |

Tabulka 3: Instrukce

# 6 Algoritmy z předmětu IAL

## 6.1 Řazení (algoritmus Heapsort)

Heapsort neboli „řazení hromadou“ je struktura stromového typu, u niž pro všechny členy platí, že mezi otcovským a všemi jeho synovskými uzly je stejná relace uspořádání. Nejčastějším případem je binární hromada založena na binárním stromu. Princip řazení je stejný jak pro čísla, tak pro řetězce.

Heapsort je řadící metoda s lineární složitostí. Algoritmus patří mezi nestabilní algoritmy, protože přesouvá prvky s příliš velkými skoky a také se nechová přirozeně. Heapsort je asi dvakrát pomalejší než Quicksort.

## 6.2 Vyhledávání podřetězce v řetězci (Boyer-Mooreův algoritmus)

Dle zadání jsem měli vestavěnou funkci **find**, která by používala pro vyhledání Boyer-Mooreův algoritmus. Boyer-Mooreův algoritmus je jedním velmi efektivním algoritmem pro vyhledávání podřetězců v řetězcích.

Algoritmus prochází textem a porovnává vzorek zprava doleva a pro posuv bere ten z výsledků dvou heuristik, který je výhodnější. Efektivita algoritmu spočívá v časové složitosti. Na rozdíl od jiných algoritmů nemusí porovnávat znak po znaku, ale dle podmínek může několik znaků přeskočit. Pokud algoritmus narazí v řetězci na znak, který vyhledávaný podřetězec neobsahuje, může posunout vyhledávání o celou jeho délku. Čím delší je vyhledávaný podřetězec, tím je kratší doba. Funkce je implementována podle skript z kurzu IAL.

- ComputerJumps()

  Stanovení hodnot pole *CharJump*, které určují posuv vzorku. Použití pole CharJump pro postup vzorku činí tento algoritmus mnohem rychlejší než algoritmus Knuth-Morris-Prattův.

  Jednotlivé symboly řetězce mají uloženou svojí ASCII hodnotu.

- ComputerMatchJumps()

  Funkce pro samotný výpočet pole *MatchJump*.

- Boyer_Moor_Alg()

  Stanovení indexu polohy vzorku nalezeného v řetězci.

## 6.3 Tabulka symbolů

Tabulku symbolů jsme implementovali pomocí **binárního vyhledávacího stromu (BVS**). Jedná se o nejpoužívanější implementaci pro dynamické vyhledávací tabulky. Algoritmus je implementován rekurzivně, dle opory kurzu IAL (algoritmy). Vyhledávání v BVS je velmi podobné binárnímu vyhledávání v seřazeném poli.

Vytvořili jsme vlastní knihovnu funkcí. Implementace knihovny je v souborech *bst.c* a *bst.h*.

Rozhraní bst poskytuje funkčnost specifickou pro tabulku symbolů, která je implementována v souborech *stable.c* a *stable.h*. Tabulka symbolů slouží k ukládání informací o funkcích a proměnných.

# 7 Chybové stavy

Pro chybové stavy jsme implementovali vlastní funkci, která je vždy volána při nalezení chyby v kódu, jak lexikální, syntaktické nebo při interpretaci, kde je kontrolována i sémantika kódu. Návratové chybové kódy byly definovány v zadání projektu.

# 8 Práce v týmu

Komunikace v týmu probíhala za pomocí IRC Freenode. Pořádali jsme nepravidelná osobní setkání, která se uskutečňovala průměrně jednou za čtrnáct dní nebo po domluvě v učebně CVT. Na osobním setkání bylo objasněno, co vše je zapotřebí udělat. Rozdělování prací nebylo striktně určeno, všichni členové týmu si navzájem pomáhali.

Ke sdílení zdrojových kódů jsme využili GitHub s vlastním soukromým kanálem. Každý si vytvořil vlastní větev a pracoval v ní, aby se nezasahovalo do hlavní větve, kde se nacházeli pouze závěrečné otestované kódy.

## 8.1 Rozdělení práce

- **Martin Honza:** Syntaktická analýza výrazů, pomocné práce
- **Patrik Jurnečka:** Dokumentace, prezentace, pomocné práce při tvorbě modulů
- **Hana Slámová:** Vestavěné funkce, pomocné práce při tvorbě modulů
- **Frantisek Šumšal:** Lexikální analyzátor, syntaktický analyzátor, interpret, organizace a vedení týmu
- **Adam Švidroň:** Vestavěné funkce, pomocné práce při tvorbě modulů

# 9 Závěr

Na projektu se pracovalo téměř dva měsíce, i když jsme začali včas, dokončit projekt jsme nestíhali do pokusného odevzdání. V posledním týdnu před odevzdáním jsme dodělávali interpret a snažili se vše zkompletovat, aby fungoval projekt dle zadání. Dokumentace se tvořila průběžně při dokončení některého logického celku.

Projekt byl velmi náročný, nejen ze strany programové, ale také hlavní roli hrála domluva týmu. Velkou roli hrála i neznalost principů všech dílčích částí překladače, které jsme museli studovat velmi dopředu, ale s tím se dalo počítat, byly jsme na to upozorněni v kurzu IFJ. Nakonec se to zvládlo dokončit. Z projektu jsme si odnesli velké zkušenosti, jak efektivně organizovat a pracovat v týmu.

Navržená implementace byla nakonec otestována v prostředí operačních systémů Linux.

Dokumentace i prezentace jsme psali v jazyce LaTeX.

## 9.1 Metriky kódu

- Počet souborů: 29
- Celkový počet řádků kódu: 5 676
- Velikost spustitelného souboru: 121kB (OS Linux 32 bitová architektura)

# Reference

[1] *Přednášky, skripta a podklady k předmětu IFJ a IAL.*

[2] AHO, A. V. *Compilers: principles, techniques,*. 2nd ed. Boston: Pearson/Addison Wesley, c2007. ISBN 03-214-8681-1.