

# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

## FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentace k projektu do předmětů IFJ a IAL  
Implementace interpretu jazyka IFJ15  
Tým 052, varianta *a/2/II*

Vedoucí týmu:	Postolka Matěj	xposto02	25 %
Další členové:	Osadský Lukáš	xosads00	25 %
	Plaskoň Pavol	xplask00	25 %
	Pospíšil Pavel	xpospi88	25 %

# Obsah

<b>1</b>	<b>Úvod</b>	<b>2</b>
<b>2</b>	<b>Práce v týmu</b>	<b>2</b>
2.1	Rozdělení práce na jednotlivých částech . . . . .	2
2.2	Průběh vývoje . . . . .	2
<b>3</b>	<b>Implementace interpretu jazyka IFJ15</b>	<b>3</b>
3.1	Lexikální analýza . . . . .	3
3.2	Syntaktická a sémantická analýza . . . . .	3
3.2.1	Zpracování jazykových konstrukcí . . . . .	3
3.2.2	Zpracování výrazů a volání funkcí . . . . .	4
3.3	Interpret . . . . .	4
3.3.1	Volání funkcí . . . . .	4
3.4	Datové struktury . . . . .	5
3.4.1	Zásobník . . . . .	5
3.4.2	Řetězec . . . . .	5
3.4.3	Tabulka s rozptýlenými položkami . . . . .	5
3.5	Algoritmy . . . . .	5
3.5.1	Řadící algoritmus – Heap Sort . . . . .	5
3.5.2	Vyhledávání podřetězce v řetězci – Knuth-Morris-Pratt . . . . .	5
<b>4</b>	<b>Přílohy</b>	<b>6</b>
4.A	Diagram konečného automatu lexikální analýzy [2] . . . . .	6
4.B	LL-gramatika . . . . .	7
4.B.1	Část první . . . . .	7
4.B.2	Část druhá . . . . .	8
4.C	Precedenční tabulka . . . . .	9
4.D	Instrukční sada trojadresného kódu . . . . .	10
4.D.1	Část první . . . . .	10
4.D.2	Část druhá . . . . .	11
<b>5</b>	<b>Reference</b>	<b>11</b>

# 1 Úvod

Dokumentace popisuje implementaci interpretu jazyka IFJ15, který je podmnožinou jazyka C++11. Interpret se skládá ze čtyřech částí popsanych v následujících kapitolách.

- Lexikální analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Interpret

## 2 Práce v týmu

### 2.1 Rozdělení práce na jednotlivých částech

- Osadský Lukáš – Lexikální analyzátor
- Postolka Matěj – Sémantický a syntaktický analyzátor
- Pospíšil Pavel – Zpracování výrazů a volání funkcí, dokumentace
- Plaskoň Pavol – Interpret, vestavěné funkce

### 2.2 Průběh vývoje

Projekt je řešený čtyřčlenným týmem, bylo tedy potřebné zvolit vhodný systém správy zdrojových souborů. Přes téměř nulové zkušenosti většiny členů týmu jsme k těmto účelům využili verzovací systém `git` na privátním serveru vedoucího člena. Konzultace probíhaly jednou týdně, obsahovaly zhodnocení aktuálních výsledků a stanovení dalšího postupu. Nejdříve tedy každý člen pracoval sám, po několika týdnech práce proběhly dvě schůzky na kterých jsme programovali společně. V průběhu celého procesu členové týmu doplňovali krátké testovací ukázky kódu, kterými bylo následně možné, pomocí skriptu napsaného v jazyce `Python`, testovat dosavadní stabilitu celku. Při testování se nám též osvědčil nástroj `gcovr`.

## 3 Implementace interpretu jazyka IFJ15

### 3.1 Lexikální analýza

Lexikální analyzátor je vstupní částí interpretu. Je založen na deterministickém konečném automatu, jehož hlavním úkolem je čtení zdrojového souboru a na základě lexikálních pravidel jazyka rozdělit jednotlivé posloupnosti znaků souboru na lexikální části – lexémy.

Rozpoznané lexikální jednotky jsou reprezentovány strukturou `token`, která obsahuje informace o typu tokenu a jeho data. Data do tokenu jsou zapisována za pomoci funkcí z námi implementované knihovny `string.h`.

Vedlejším úkolem lexikální analýzy je odstraňování všech komentářů a bílých znaků, neboť nejsou podstatné v dalších krocích interpretace. Vstupní souboru může obsahovat dva typy komentářů – řádkový, uvozený znaky `//` a blokový ohraničený sekvencemi znaků `/*` a `*/`. Po ukončení čtení komentáře automat opět přejde do počátečního stavu.

Činnost lexikálního analyzátoru je přímo řízena syntaktickým analyzátozem, který postupně žádá o jednotlivé tokeny. Princip fungování lexikální analýzy reprezentuje příloha 4.A, ve které je zobrazeno její schéma.

### 3.2 Syntaktická a sémantická analýza

Syntaktický a sémantický analyzátor, neboli `parser`, představuje ústřední část naší implementace interpretu jazyka IFJ15. Parser se volá prakticky ihned po spuštění programu a přejímá řízení do doby, než dojde k úplnému zpracování zdrojového souboru.

#### 3.2.1 Zpracování jazykových konstrukcí

Syntaktická analýza je implementována rekurzivním sestupem, který je řízen pravidly naší LL-gramatiky uvedenými v příloze 4.B. Neterminální symboly představují tokeny přijaté od lexikálního analyzátoru. Ten je volán přímo z parseru vždy, když je třeba zpracovat další token. Se syntaktickou analýzou je současně vykonávána také analýza sémantická. Při deklaraci nebo definici funkce se do globální tabulky symbolů ukládá datová struktura reprezentující danou funkci – jazyk IFJ15 podporuje v globálním prostoru pouze funkce.

V případě definice funkce poté dochází ke zpracování těla dané funkce. Přímo během rekurzivního sestupu se tak vykonávají všechny potřebné sémantické kontroly a naplňuje se lokální tabulka symbolů. Taktéž se generují vnitřní instrukce, které se ukládají do instrukčního seznamu příslušné funkce. Pokud se během syntaktické analýzy narazí na výraz, je řízení programu předáno modulu pro vyhodnocování výrazů `expr`, který pomocí precedenční analýzy provede vyhodnocení daného výrazu a poté předá řízení zpět parseru.

Po zpracování celého zdrojového souboru se provádí závěrečné sémantické kontroly. Kontroluje se například, zda došlo během zpracování zdrojového souboru k definici všech deklarovaných funkcí a přesná signatura funkce `main`. Tímto je syntaktická a sémantická analýza ukončena a parser předá řízení interpretu.

### 3.2.2 Zpracování výrazů a volání funkcí

Zpracování výrazů je voláno v několika rozličných situacích. Existují situace, kdy se však na místě výrazu může objevit volání funkce. Volání funkcí i zpracovávání výrazů jsou tedy v naší implementaci součástí jednoho modulu.

Zpracování výrazů řízené precedenční tabulkou uvedenou v příloze 4.C probíhá ve dvou krocích. V prvním kroku je za pomoci zásobníkové struktury výraz převeden z infixové na postfixovou notaci. V tomto kroku je kontrolována správná posloupnost operátorů, operandů a závorek.

V kroku druhém je vyhodnocena postfixová notace a vygenerovány příslušné instrukce. V této fázi běhu interpretu se kontroluje datová kompatibilita operandů a nastavují odvozené datové typy proměnným s modifikátorem `auto`.

Při výskytu volání funkce je mimo jiné kontrolována kompatibilita návratového typu funkce a typu proměnné pro přiřazení navracené hodnoty.

## 3.3 Interpret

Interpret ke své práci využívá globální zásobník, lokální zásobník pro aktuální funkci a tabulky s rozptýlenými položkami pro proměnné. Každý blok příkazů interpretovaného programu má vlastní tabulku proměnných – zabezpečení viditelnosti proměnných pouze v rámci jejich bloku. Na začátku se v globální tabulce symbolů vyhledá funkce `main`, vytvoří se pro ni lokální rámec a instrukční ukazatel se nastaví na začátek instrukčního listu – jednosměrně vázaného lineárního seznamu. Přesun na další položku seznamu představuje přechod na další instrukci. Při instrukci skoku se instrukční ukazatel nastaví na danou instrukci `INS_LAB` označující návěští.

### 3.3.1 Volání funkcí

Před provedením volání funkce jsou její parametry uloženy na pomocný zásobník instrukcí `INS_PUSH_PARAM`. Dále se do globálního zásobníku uloží ukazatel na aktuální instrukci a aktuální rámec, vytvoří se nový lokální rámec a instrukční ukazatel se nastaví na začátek instrukčního listu volané funkce. Při návratu z funkce se její návratová hodnota uloží do vyhrazené proměnné ve struktuře reprezentující funkci v globální tabulce symbolů. Tím je interpret oddělen od režie návratových hodnot. Menší nevýhodou této implementace je generování návratové proměnné pro každou funkci. Vše je zabezpečeno při generování kódu. Z globálního rámce se načte původní rámec a instrukce, odkud byla funkce volána, a pokračuje se instrukcí další.

## 3.4 Datové struktury

### 3.4.1 Zásobník

Zásobník je dynamická datová struktura. Položkami zásobníku jsou ukazatele na typ `void`, což umožňuje jeho široké využití. Uplatňuje se v `parseru` i `interpretu`. Má důležitou úlohu například při převodu infixového zápisu výrazů na postfixový.

### 3.4.2 Řetězec

V naší implementaci je řetězec v podstatě **vektor**, nebo-li dynamické pole znaků. Dle potřeby mění svoji kapacitu. Námi implementovaný datový typ `TString` se značně liší od datového typu `string` ve vyšších programovacích jazycích.

### 3.4.3 Tabulka s rozptýlenými položkami

Tato datová struktura je použita pro tabulky symbolů. Její výhodou je rychlost vyhledávání položek. Základem je pole ukazatelů na jednotlivé položky. Položky obsahují svůj klíč, data a ukazatel na další položku, aby mohly být propojené v jednosměrně explicitně vázaný lineární seznam – seznam synonym. V případě ideální hashovací funkce není propojení v seznam potřebné a čas přístupu k položkám je konstantní. Nalezení takové funkce není triviální. Zvolili jsme dostatečnou funkci `sbdm[1]`. V případě konfliktu se čas nalezení položky prodlužuje o dobu prohledání seznamu synonym.

## 3.5 Algoritmy

Následující kapitola se zabývá dvěma nejzajímavějšími algoritmy naší implementace, první z nich slouží pro řazení, druhý pro vyhledávání podřetězce v řetězci.

### 3.5.1 Řadící algoritmus – Heap Sort

Heap – hromada je struktura stromového typu. Nejčastěji, i v naší implementaci, se jedná o uspořádaný binární strom. Významnou operací nad hromadou je `siftDown` – prosetí, která zrekonstruuje hromadu porušenou v kořeni. Prvek v kořeni se postupně přesune na správné místo. V nejhorším případě (přesun z kořene až do listu) bude složitost  $\log_2 n$ . Podstatou řazení je implementace hromady polem velikosti  $N$ , kde uzel umístěný na indexu  $i$  má svého levého syna na indexu  $2i + 1 - base$ <sup>1</sup> a pravého syna na indexu  $2i + 2 - base$ . Index nejpravějšího neterminálního uzlu na nejnižší úrovni je  $N/2 - 1 + base$ . Vlastní cyklus heap sortu má lineární složitost, přičemž v každém kroku se vymění kořen s posledním prvkem a *proseje* se hromada. Tím dostáváme lineární časovou složitost.

### 3.5.2 Vyhledávání podřetězce v řetězci – Knuth-Morris-Pratt

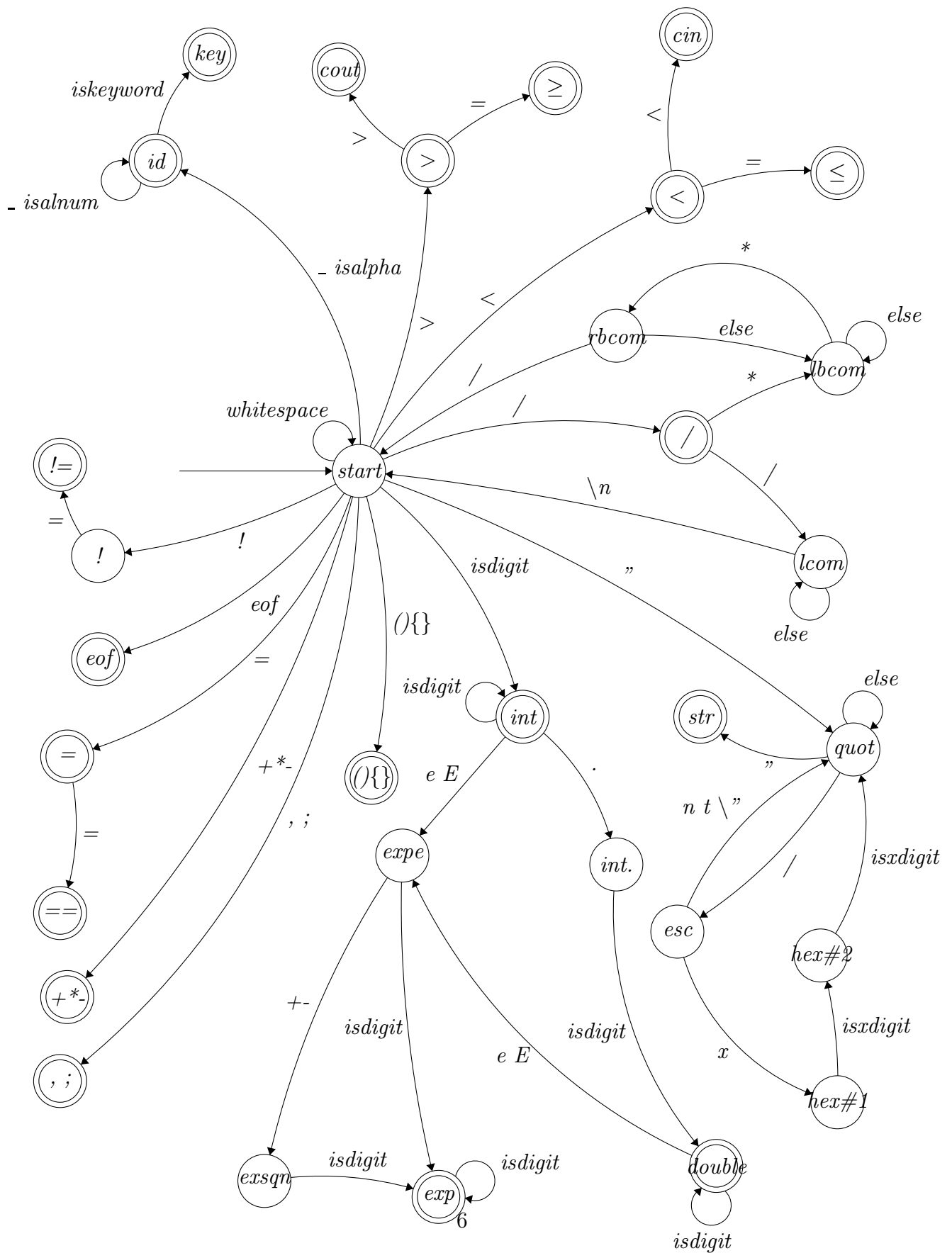
Vyhledání podřetězce v řetězci ve vestavěné funkci `find` je řešeno algoritmem Knuth-Morris-Pratt. Základem algoritmu je vytvoření masky, tzv. `Fail vector`. Jedná se o pole celých čísel, délka pole je totožná s délkou hledaného textu. Ke každému písmenu hledaného podřetězce je přiřazeno číslo, které určuje index pro začátek dalšího hledání v případě neshody znaků.

---

<sup>1</sup>*base* – index první položky pole

## 4 Přílohy

#### 4.A Diagram konečného automatu lexikální analýzy [2]



## 4.B LL-gramatika

### 4.B.1 Část první

PROG->FUNCTION\_DECL PROG

PROG->eps

FUNCTION\_DECL->DATA\_TYPE t\_identifier t\_lround\_bracket FUNC\_DECL\_PARAMS t\_rround\_bracket NESTED\_BLOCK

DATA\_TYPE->t\_int

DATA\_TYPE->t\_double

DATA\_TYPE->t\_string

FUNC\_DECL\_PARAMS->DATA\_TYPE t\_identifier FUNC\_DECL\_PARAMS\_NEXT

FUNC\_DECL\_PARAMS\_NEXT->t\_comma FUNC\_DECL\_PARAMS

FUNC\_DECL\_PARAMS->eps

FUNC\_DECL\_PARAMS\_NEXT->eps

NESTED\_BLOCK->t\_lcurly\_bracket NBC t\_rcurly\_bracket

NBC->DECL\_OR\_ASSIGN NBC

DECL\_OR\_ASSIGN->DATA\_TYPE t\_identifier DECL\_ASSIGN t\_semicolon

~ DECL\_OR\_ASSIGN->t\_auto t\_identifier t\_assign EXPRESSION t\_semicolon

DECL\_ASSIGN->t\_assign EXPRESSION

DECL\_ASSIGN->eps

NBC->FCALL\_OR\_ASSIGN NBC

FCALL\_OR\_ASSIGN->t\_identifier FOA\_PART2

FOA\_PART2->t\_lround\_bracket FUNCTION\_CALL\_PARAMS t\_rround\_bracket t\_semicolon

FOA\_PART2->t\_assign EXPRESSION t\_semicolon

HARD\_VALUE->t\_int\_value

HARD\_VALUE->t\_double\_value

HARD\_VALUE->t\_string\_value

FUNCTION\_CALL\_PARAMS->FUNCTION\_CALL\_PARAM FUNCTION\_CALL\_PARAMS\_NEXT

FUNCTION\_CALL\_PARAMS->eps

FUNCTION\_CALL\_PARAM->t\_identifier

FUNCTION\_CALL\_PARAM->HARD\_VALUE

FUNCTION\_CALL\_PARAMS\_NEXT->t\_comma FUNCTION\_CALL\_PARAMS

FUNCTION\_CALL\_PARAMS\_NEXT->eps



#### 4.B.2 Část druhá

NBC->BUILTIN\_CALL NBC  
BUILTIN\_CALL->BUILTIN\_FUNC t\_lround\_bracket FUNCTION\_CALL\_PARAMS t\_rround\_bracket t\_semicolon  
BUILTIN\_FUNC->token\_bf\_length  
BUILTIN\_FUNC->token\_bf\_substr  
BUILTIN\_FUNC->token\_bf\_concat  
BUILTIN\_FUNC->token\_bf\_find  
BUILTIN\_FUNC->token\_bf\_sort  
NBC->IF\_STATEMENT NBC  
IF\_STATEMENT->t\_if t\_lround\_bracket EXPRESSION t\_rround\_bracket NESTED\_BLOCK ELSE\_STATEMENT  
ELSE\_STATEMENT->t\_else NESTED\_BLOCK  
ELSE\_STATEMENT->eps  
NBC->COUT NBC  
COUT->t\_cout t\_cout\_bracket COUT\_OUTPUT COUT\_NEXT t\_semicolon  
COUT\_OUTPUT->t\_identifier  
COUT\_OUTPUT->HARD\_VALUE  
COUT\_NEXT->t\_cout\_bracket COUT\_OUTPUT COUT\_NEXT  
COUT\_NEXT->eps  
NBC->CIN NBC  
CIN->t\_cin t\_cin\_bracket t\_identifier CIN\_NEXT t\_semicolon  
CIN\_NEXT->t\_cin\_bracket t\_identifier CIN\_NEXT  
CIN\_NEXT->eps  
NBC->FOR\_STATEMENT NBC  
FOR\_STATEMENT->t\_for t\_lround\_bracket FOR\_DECLARATION FOR\_EXPR FOR\_ASSIGN t\_rround\_bracket NESTED\_BLOCK  
FOR\_DECLARATION->DATA\_TYPE t\_identifier DECL\_ASSIGN t\_semicolon  
FOR\_DECLARATION->t\_auto t\_identifier t\_assign EXPRESSION t\_semicolon  
FOR\_EXPR->EXPRESSION t\_semicolon  
FOR\_ASSIGN->t\_identifier t\_assign EXPRESSION  
NBC->NESTED\_BLOCK NBC  
NBC->RETURN  
RETURN->t\_return EXPRESSION t\_semicolon  
NBC->eps

∞

## 4.C Precedenční tabulka

Stack \ Input	+	-	*	/	(	)	id	<	>	<=	>=	==	!=
+	>	>	<	<	<	>	<	>	>	>	>	>	>
-	>	>	<	<	<	>	<	>	>	>	>	>	>
*	>	>	>	>	<	>	<	>	>	>	>	>	>
/	>	>	>	>	<	>	<	>	>	>	>	>	>
(	<	<	<	<	<	=	<	<	<	<	<	<	<
)	>	>	>	>	!	>	!	>	>	>	>	>	>
id	>	>	>	>	!	>	!	>	>	>	>	>	>
<	<	<	<	<	<	>	<	>	>	>	>	>	>
>	<	<	<	<	<	>	<	>	>	>	>	>	>
<=	<	<	<	<	<	>	<	>	>	>	>	>	>
>=	<	<	<	<	<	>	<	>	>	>	>	>	>
==	<	<	<	<	<	>	<	<	<	<	<	>	>
!=	<	<	<	<	<	>	<	<	<	<	<	>	>

## 4.D Instrukční sada trojadresného kódu

### 4.D.1 Část první

- **INS\_ASSIGN** *dest*, *src1*  
přiřadí hodnotu proměnné *src1* do *dest*
- **INS\_ADD** *dest*, *src1*, *src2*  
sčítá *src1* a *src2*, výsledek uloží do *dest*
- **INS\_SUB** *dest*, *src1*, *src2*  
odečítá *src1* od *src2*, výsledek uloží do *dest*
- **INS\_MUL** *dest*, *src1*, *src2*  
vynásobí *src1* a *src2*, výsledek uloží do *dest*
- **INS\_DIV** *dest*, *src1*, *src2*  
vydělí *src2* a *src1*, výsledek uloží do *dest*
- **INS\_EQ** *dest*, *src1*, *src2*  
testuje rovnost *src1* a *src2*, výsledek uloží do *dest*
- **INS\_NEQ** *dest*, *src1*, *src2*  
testuje nerovnost *src1* a *src2*, výsledek uloží do *dest*
- **INS\_GREATER** *dest*, *src1*, *src2*  
testuje, zda je *src1* větší než *src2*, výsledek uloží do *dest*
- **INS\_GREATEQ** *dest*, *src1*, *src2*  
testuje, zda je *src1* větší nebo roven *src2*, výsledek uloží do *dest*
- **INS\_LESSER** *dest*, *src1*, *src2*  
testuje, zda je *src1* menší než *src2*, výsledek uloží do *dest*
- **INS\_LESSEQ** *dest*, *src1*, *src2*  
testuje, zda je *src1* menší nebo roven *src2*, výsledek uloží do *dest*
- **INS\_JMP** *label*  
nepodmíněný skok na návěští *label*
- **INS\_CJMP** *cond* *label*  
podmíněný skok na návěští *label* na základě hodnoty *cond*
- **INS\_LAB** *label*  
definice návěští *label* pro skok
- **INS\_PUSH\_PARAM** *src*  
uloží na pomocný zásobník parametr *src* pro volání funkce
- **INS\_CALL** *func*  
volání funkce *func*

#### 4.D.2 Část druhá

- **INS\_RET**  
ukončení provádění funkce
- **INS\_PUSH\_TAB src1**  
vytvoření nového rámce pro vnořený blok *src1*
- **INS\_POP\_TAB src1**  
zrušení rámce vnořeného bloku *src1*
- **INS\_LENGTH dest, src1**  
volání vestavěné funkce *length* s parametrem *src1*, výsledek uloží do *dest*
- **INS\_SUBSTR dest**  
volání vestavěné funkce *substr* s předem uloženými parametry, výsledek uloží do *dest*
- **INS\_CONCAT dest, src1, src2**  
volání vestavěné funkce *concat* s parametry *src1* a *src2*, výsledek uloží do *dest*
- **INS\_FIND dest, src1, src2**  
volání vestavěné funkce *find* s parametry *src1* a *src2*, výsledek uloží do *dest*
- **INS\_SORT dest, src1**  
volání vestavěné funkce *sort* s parametrem *src1*, výsledek uloží do *dest*
- **INS\_CIN src1**  
načítá do *src1* hodnotu ze standardního vstupu
- **INS\_COUT dest**  
vypíše na standardní výstup *dest*

## 5 Reference

- [1] *Hash Functions* [online]. [cit. 2015-12-13]. Dostupné z: <http://www.cse.yorku.ca/~oz/hash.html>
- [2] WALLACE, E. *Finite State Machine Designer* [online]. 2010 [cit. 2015-12-13]. Dostupné z: <http://madebyevan.com/fsm/>
- [3] Prof. Ing. Jan Maxmilián Honzík, CSc. *Algoritmy IAL: Sudijní opora* [online]. Verze 14R. 2015-12-11 [cit. 2015-12-13]. Dostupné z: <https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IAL-IT/texts/Opora-IAL-2014-verze-14R.pdf>