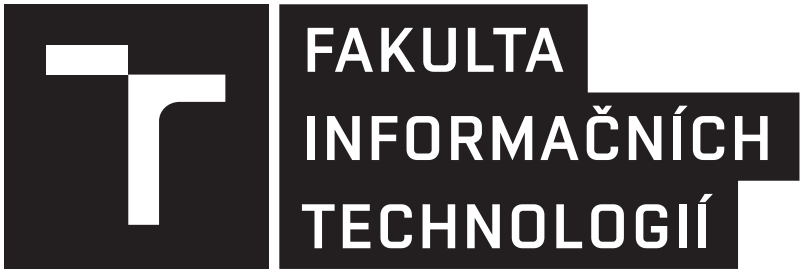
Vysoké učení technické v Brně
Fakulta informačních technologií



Dokumentace 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

1 Úvod

Tato dokumentace popsiuje implementaci interpretu jazyka IFJ15,který je zjednodušenou podmnožinou jazyka C++11. Interpret se skládá ze čtyřech částí popsaných v následujících kapitolách.

- Lexikální analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Interpret

2 Práce v týmu

2.1 Rozdělení práce na jednotlivých částech

- Lexer Lukáš – lexikální analyzátor
- Matěj Postolka – Sémantický a syntaktický analyzátor
- Pavel Pospíšil – Zpracování výrazů a volání funkcí
- Pavel Plaskoň – Interpret, vstavěné 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 jsme využili verzovací systém `Git` na privátním serveru vedoucího člena.

`Inkrementánlí vývojový cyklus !!!`

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 kódy, kterými bylo následně možné, pomocí skriptu napsaného v jazyce `Python`, testovat dosavadní stabilitu celku. Při testování se nám též osvědčil program `gprof`.

`Něco o valgrindu?`

3 Implementace interpretu jazyka IFJ15

3.1 Lexikální analýza

*** Lukáš Osadský
unget_token
o komentářích (návrat do startu)
pouívání stringu

Lexikální analyzátor je vstupní část překladače. 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 lexémy jsou reprezenované strukturou token, která obsahuje informace o typu tokenu a jeho data. Jeho vedlejší úlohou je odstraňování všech komentářů a bílých znaků, neboť nejsou potřebné pro následné zpracování. Princip fungování lexikálního analyzátoru reprezuntuje příloha 4.A, ve které je zobrazeno jeho schéma. Činnost lexikálního analyzátoru je přímo řízena syntaktickým analyzátorem, který postupně žádá o jednotlivé tokeny.

3.2 Syntaktická a sémantická analýza

3.2.1 Zpracování jazykových konsturkcí

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.

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 – jazyk IFJ15 podporuje v globálním prostoru pouze funkce – se do globální tabulky symbolů ukládá datová struktura reprezentující danou funkci.

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, přesná signatura fce main. Tímto je syntaktická a sémantická analýza ukončena a parser předá řízení interpretu.

3.2.2 Zpracování výrazů a volání funkcí

Zpracování výrazu je voláno v několika rozličných situacích. Existují situace, kdy se však na místě výrazu může objevit volání funkce. Volání funkcí i zpracovávání výrazů jsou 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 kontrolují datové typy operandů a nastavují odvozené datové typy proměnným s modifikátorem `auto`.

Při výskytu volání funkce je mimo jiné kontrolován datový typ proměnné, kterou tato funkce nastavuje svojí návratovou hodnotou.

3.2.3 Sémantická analýza

Sémantická analýza probíhá pralelně se syntaktickou analýzou v rámci rekurzivního sestupu i precedenční analýzy výrazů. Kontroluje se definice a deklarace funkcí i deklarace proměnných.

3.3 Interpret

```
*** Pavol Plaskoň
Text
```

3.4 Datové struktury

```
Úvodní text ke sktrukturám? Vypadalo by to podle mě lépe
```

3.4.1 Zásobník

```
*** Tvůrce??
Text
Umí přístup přes indexy, je udělanej jako pole... myslím
```

3.4.2 Řetězec

```
*** Tvůrce??
Text
```

3.4.3 Tabulka s rozptýlenými položkami

```
*** Pavol Plaskoň
používáme sdbm
explicitně řazené položky
```

Datová struktura použitá 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 vázaný lineární seznam – seznam synonym. V případě ideální hashovací funkce není propojení v seznam potřebné a čas přístupu k položkám konstantní. Nalezení takové funkce není triviální. V případě konfliktu se čas nalezení položky prodloužuje o dobu prohledání lineárního seznamu.

3.5 Algoritmy

`Úvodní text k algoritmům?`

3.5.1 Řadíci algoritmus – Heap Sort

`*** Tvůrce??`
`Text`

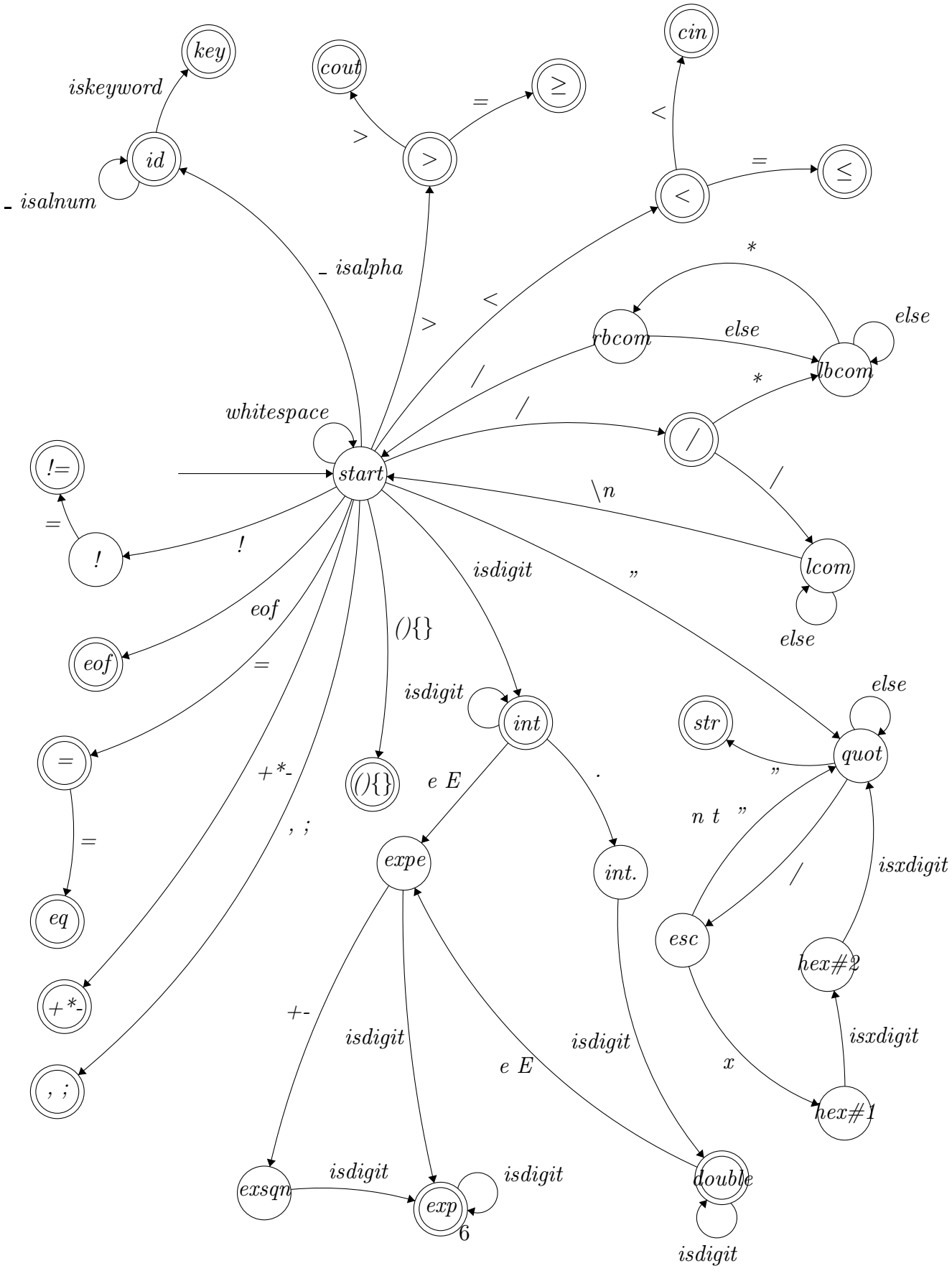
Funkce pro seřazení prvků v poli.

3.5.2 Vyhledávaní podřetězce – 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 délky hledaného textu. Ke každému písmenu hledaného řetězce je přiřazeno číslo, které určuje index pro návrat programu v případě neshody znaků.

4 Přílohy

4.A Diagram konečného autommlatu lexikální analýzy



4.B LL-gramatika

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
```

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

`INS_ASSIGN dest, src1`
přiradí hodnotu proměnnej 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`
vydelí 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, je-li(hm..advanced czech?)src1 větší než src2, výsledek uloží do dest

`INS_GREATEQ dest, src1, src2`
testuje, je-li src1 větší, nebo roven src2, výsledek uloží do dest

`INS_LESSER dest, src1, src2`
testuje, je-li src1 menší než src2, výsledek uloží do dest

`INS_LESSEQ dest, src1, src2`
testuje, je-li src1 menší, nebo roven src2, výsledek uloží do dest

`INS_JMP label`
nepodmíněný skok na návěští

`INS_CJMP cond label`
podmíněný skok na návěští label na základe hodnoty cond

`INS_LAB label`
návěští pro skok

`INS_PUSH_PARAM src`
uloží na pomocnej zásobník parameter pro volání funkce

`INS_CALL func`
volání funkce func

`INS_RET`
ukončení provádení funkce

`INS_PUSH_TAB src1`
vytvoření nového rámce pro vnořený blok src1

`INS_POP_TAB src1`
zrušení rámce jednoho bloku programu

`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 funcke `sort` s parametrem src1, výsledek uloží do dest

`INS_CIN src1`

vypíše na standardní výstup src1

`INS_COUT dest`

načítá do dest určitou? hodnotu ze standardního vstupu

5 Zdroje

`http://madebyevan.com/fsm/`

`sdbm algoritmus`