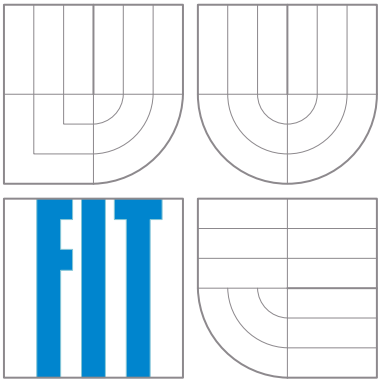
Vysoké učení technické v Brně
Fakulta informačných technológií



Implementace interpretu imperativního jazyka IFJ13

Dokumentace k projektu pre předměty IFJ a IAL

Tým 013, varianta b/3/I

13. prosince 2013

Tomáš Bank xbankt00@stud.fit.vutbr.cz
Mark Birger xbirge00@stud.fit.vutbr.cz
Roland Botka xbotka00@stud.fit.vutbr.cz
Zdenko Brandejs xbrand06@stud.fit.vutbr.cz
Daniil Khudiakov xkhudi00@stud.fit.vutbr.cz

Obsah

1 Úvod

Tento dokument popisuje návrh a implementaci interpretu imperatívního jazyka IFJ13. Varianta b/3/I udáva použít pro vyhledávaní Boyer-Mooreův algoritmus (libovolný typ heurestiky), pro řazení algoritmus Shell sort a tabulku symbolů implementovánu pomocí binárního vyhledávacího stromu.

Interpret je složen ze tří hlavních částí: lexikální analyzátor, syntaktický analyzátor a interpret. Všechny tyto části jsou popsány v tomto dokumentu i se zadáním problému, jeho analýzou a popisem řešení.

Dokument se skládá z několika částí. V kapitole 2 je popsané zadání problému. Ďalší kapitola 4 obsahuje popis řešení. Na závěr 4 dokumentu je struční shrnutí celé práce. V kapitole 6 jsou přidány přílohy obsahující LL-gramatiku a strukturu konečného automatu, protože se jedná o zásadní prvky interpretu.

2 Zadaní problému

Naším úkolem bylo vytvořit program, který načte zdrojový soubor zapsaný v jazyce IFJ13 a interpretuje jej. Jestliže činnost interpretu proběhne bez chyb, vrací se návratová hodnota 0(nula). Jestliže došlo k nějaké chybě, vrací se vopred určená návratová hodnota.

Jméno souboru s řídicím programem v jazyce IFJ13 bude předáno jako první a jediný parametr na příkazové řádce. Program bude příjímat vstupy ze standardního vstupu, směrovat všechny své výstupy na standardní výstup.

3 Popis řešení

V této části se zabýváme podrobnějším popisem řešení projektu. Důraz klademe na 3 hlavní části (lexikální analýza, syntaktická analýza a interpret), ale i na konečný automat, boyer-moorův algoritmus, shell sort algoritmus a tabulku symbolů.

3.1 Lexikální analýza

Je to první část interpretu. Na vstupu lexikálního analyzátoru (scanneru) je zdrojový text překládaného programu. Zdrojový program je rozdělen na posloupnost tokenů (lexémy). Lexémy jsou logicky oddělené lexikální jednotky, tyto lexémy jsou dále zpracovány v syntaktickém analyzátoru.

3.1.1 Konečný automat

První krok k vytvoření lexikální analýzy je návrh konečného automatu. KA se skládá z konečné množiny stavů nad vstupní abecedou, konečné množiny pravidel, jednoho počátečního stavu a množiny koncových stavů.

V první fázi jsme vytvořili návrh konečného automatu, který nebyl úplně v souladu se zadáním. K výslednému konečnému automatu jsme se dostali úpravou prvního návrhu. Některé chyby se ukázali během řešení jiných částí, proto byl automat ještě několikrát upraven. Návrh automatu je v příloze.

3.1.2 Implementace lexikální analýzy

Hlavní funkcí v lexeru je funkce *getToken()*. Tahle funkce generuje následující token. Ve funkci je vlastně implementován konečný automat dle návrhu. Zdrojový text se čte po znacích funkcí *getc()*, na základě načítaného znaku a aktuálního stavu se rozhoduje o konečném stavu. Lexer vrací následující token typu string. Struktura string obsahuje pole znaků reprezentující hodnotu atributu, délku tohoto atributu a velikost alokované paměti. Pokud načítaná posloupnost znaků neodpovídá žádnému stavu (syntaktická chyba), vypisuje se chybová hláška s informací, na kterém řádku k chybě došlo.

3.2 Boyer-Moorův algoritmus

Boyer-Moorův algoritmus je využit ve funkci *find_string()* jazyka IFJ13, která vyhledává podřetězec v řetězci a pokud nalezla, vrací jeho pozici (počítano od nuly). Konkrétně to znamená index pole, ve kterém je uložen řetězec a obsah tohoto indexu se shoduje se začátkem podřetězce. První parametr *txt*, je řetězec, ve kterém se bude podřetězec vyhledávat. Druhý parametr *pat*, je podřetězec, který vhledává svoji duplikaci v řetezci *txt*. Při úspěšném nalezení se vrátí index do pole *txt*, kde je počátek podřetězce. Při neúspěchu se vrátí hodnota *-1*.

3.3 Shell Sort algoritmus

Shell Sort algoritmus využívá funkce *sort_string()* jazyka IFJ13, která seřadí znaky v zadaném řetězci tak, aby znak s nižší ordinální hodnotou vždy předcházel znaku s vyšší ordinální hodnotou. Toto řazení pracuje na principu bublinového vkládaní. Funkce vrátí řetězec seřazených znaků.

3.4 Tabulka symbolů

Implemetovaná pomocí binárního vyhledávacího stromu. Klíčem je struktura *string* pro podporu nekonečně dlouhých identifikátorů proměnných a funkcí. Hodnotu z binárniho vyhledávacího stromu využíva interpret při interpretaci instrukcí.

3.5 Syntaktická analýza

Na vstup syntaktický analyzátor (parser) dostává výstup lexikálního analyzátoru, tedy posloupnost tokenů. Parser kontroluje, zda řetězec tokenů reprezentuje syntakticky správně napsaný program První část syntaktické analýzy je rekurzivní sestup podle LL-gramatiky. Každá různá levá strana LL-gramatiky je funkce. Druhá část syntaktické analýzy je precedenční analýza podle precedenční tabulky, která je implementována pomocí dvou zásobníků. Syntaktická kontrola výrazu je součastí precedenční analýzy. Kontrola konstrukcí jazyku je součástí rekurzivního sestupu. Během syntaktické analýzy se generuje seznam instrukcí pro interpret.

3.6 Sémantická analýza

Úlohy sémantické analýzy jsou přidruženy k syntaktickým pravidlům v syntaktickém analyzátoru. Některé sémantické kontroly nelze v syntaktickém analyzátoru provést, proto je provádí interpret při vykonávání instrukcí.

3.7 Interpret

Umožňuje přímo interpretovat zdrojový text jiného programu. V případě korektní syntaktické analýzy dostává interpret na vstup blok instrukcí. Každá instrukce je reprezentována navrhnutým tříadresným kódem, který může obsahovat tyto instrukce:

All instruction definition

Common instruction	Logical instructions	Special instruction for string
I_STOP	*I_C_IS*	*I_STR_LEN*
	I_C_IS_NOT	*I_SUB_STR*
Goto instructions		
I_GOTO	*I_C_LESS*	*I_FIND_STR*
I_GOTO_IF	*I_C_LESS_EQ*	*I_SORT_STR*
	I_C_MORE	Function instruction
Operation instruction		*I_RETURN*
I_ASSIGN	*I_C_MORE_EQ*	*I_CALL*
I_PLUS		
I_MINUS	Datatype convertation instruction	
I_MULTIPLY	*I_CONVERT*	
I_DIVIDE		
I_CONCATEN	IO instructions	
	I_READ	
	I_WRITE	

Každá instrukce obsahuje tři adresy, a to dvě adresy operadnů a jednu adresu výsledku. Každá instrukce má přesně dané adresy, se kterými pracuje.
Hlavní funkcí interpretu je *interpreterStart()* , která je řešená pomocí přepínače switch, který vybírá provádění obdržené instrukce

4 Závěr

Před samotnou implementací jazyka IFJ13 jsme se inspirovali vzorovým „Jednoduchým interpretem“, který byl dostupný na stránkach předmětu IFJ. Implementovali jsme dle specifikace v zadání a upřesnění na fóru. Při návrhu a implementaci jsme vycházeli s poznatků z předmětů IFJ a IAL.

Na projektu jsme si vyzkoušeli teorii formálních jazyků v praxi, implementace algoritmů a spolupráci v malém týmu.

5 Použité zdroje

- Jan M. Honzík: Studijní opora pro předmět Algoritmy,2012
- Alexander Meduna, Roman Lukáš: Studijní opora pro předmět Formálny jazyky a překladače,2012

6 Prílohy

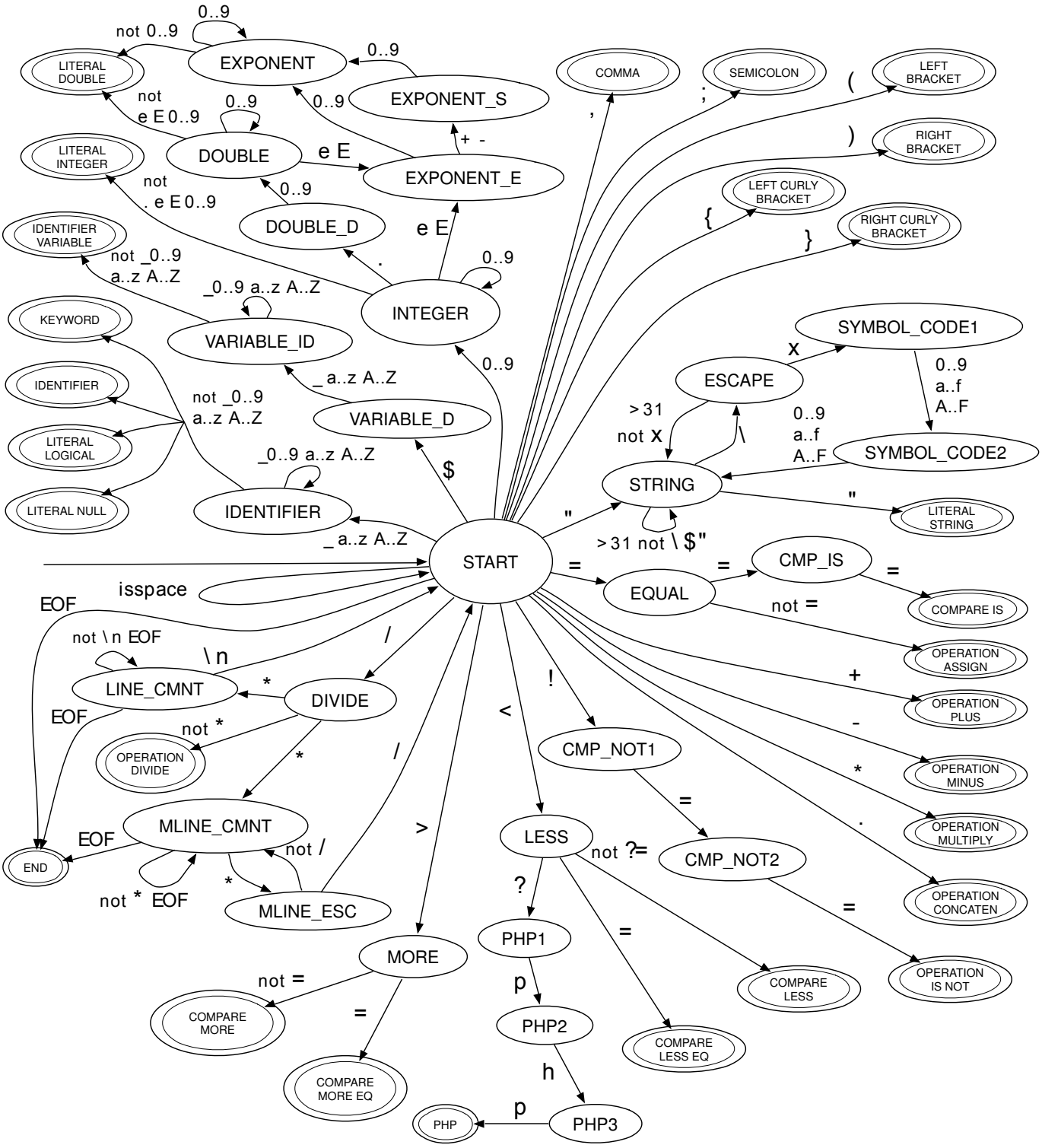
6.1 Metriky kódu

Počet souborů:
Počet řádků zdrojového textu:

Velikost statických dat:

Velikost spustitelného souboru:

6.2 Konečný automat



6.3 LL-gramatika

1	<**program**>	→	PHP <*program_units*>
2	<**program_units**>	→	<*func_define*> <*program_units*>
3	<**program_units**>	→	<*cmd_sequence*> <*program_units*>
4	<**program_units**>	→	EOF
5	<**func_define**>	→	function id (<*params*>) { <*cmd_sequence*> }
6	<**params**>	→	EOF
7	<**params**>	→	$id <*params_more*>
8	<**params_more**>	→	, $id <*params_more*>
9	<**params_more**>	→	EOF
10	<**cmd_sequence**>	→	EOF
11	<**cmd_sequence**>	→	<*cmd*> <*cmd_sequence*>
12	<**cmd**>	→	$id = <*expression*> ;
13	<**cmd**>	→	**if** (<*expression*>) { <*cmd_sequence*> } **else** { <*cmd_sequence*> }
14	<**cmd**>	→	**while** (<*expression*>) { <*cmd_sequence*> }
15	<**cmd**>	→	**return** <*expression*> ;
16	<**cmd**>	→	$id = id (<*input*>) ;
17	<**input**>	→	EOF
18	<**input**>	→	<*expression*> <*input_more*>
18	<**input_more**>	→	, <*expression*> <*input_more*>
19	<**input_more**>	→	EOF