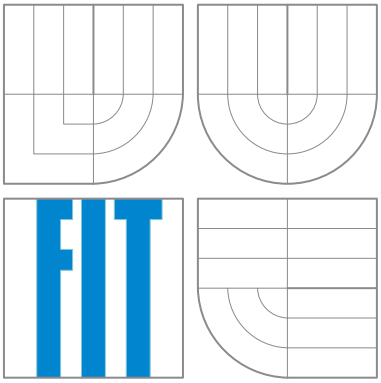
Vysoké učení technické v Brně
Fakulta informačných technológií



Implementace interpretu imperativního jazyka IFJ13

Dokumentace k projektu pre předměty IFJ a IAL

Tým 013, varianta b/3/I

15. prosince 2013

Vedoucí týmu: Tomáš Bank	19 %	xbankt00@stud.fit.vutbr.cz
Mark Birger	24 %	xbirge00@stud.fit.vutbr.cz
Roland Botka	19 %	xbotka00@stud.fit.vutbr.cz
Zdenko Brandejs	19 %	xbrand06@stud.fit.vutbr.cz
Daniil Khudiakov	19 %	xkhudi00@stud.fit.vutbr.cz

Obsah

1 Úvod

Tento dokument popisuje návrh a implementaci interpretu imperatívního jazyka IFJ13. Varianta b/3/I udává 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. Kapitola 3 obsahuje postup naši práce. Další kapitola 4 obsahuje popis řešení. Popis práce v týmu popisuje kapitola 5. Na závěr dokumentu 6 je stručné shrnutí celé práce. V kapitole 8 jsou přidány přílohy obsahující strukturu konečného automatu a LL-gramatiku, protože se jedná o zásadní prvky interpretu.

2 Zadání 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 předem určená návratová hodnota.

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

3 Postup práce

Při vývoji interpretu jsme pracovali postupně. V úvodu jsme definovali seznam tokenů a navrhli jsme konečný automat. Podle konečného automatu jsme navrhli lexer a pracovali na návrhu LL-tabulky. Poté jsme implementovali rekurzivní sestup, binární vyhledávací strom a sadu funkcí pro práci s ním. Dalším krokem bylo navrhnutí tabulky priorit operací a implementování precedenční analýzy. Implementovali jsme jednosměrný seznam instrukcí, pro který jsme navrhli instrukce. Další fází bylo generování instrukcí v parseru. Na závěr jsme začali psát interpert, popsali jsme instrukce skoku, implementovali rekurzi přes jednosměrné seznamy. Během implementace jsme opravovali chyby, které se objevily při průběžných testech. Dokumentaci jsme psali během implementace jednotlivých kroků, na závěr jsme dopsali poslední části dokumentace.

4 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.

4.1 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.

4.1.1 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 ukázaly během řešení jiných částí, proto byl automat ještě několikrát upraven. Návrh automatu je v příloze.

4.1.2 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.

4.2 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.

4.3 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 instrukcií.

4.4 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 operandů 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

4.5 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 vyhledává svoji duplikaci v řetězci *txt*. Při úspěšném nalezení se vrátí index do pole *txt*, kde je počátek podřetězce. Při neúspěchu se vrátí hodnota *-1*.

4.6 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.

4.7 Tabulka symbolů

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

5 Práce v týmu

Náš tým měl nepravidelné schůzky v průměru jednou za dva týdny. Scházeli jsme se na fakultě informačních technologií v knihovně nebo na fakultě elektrotechniky a komunikačních technologií taktéž v knihovně. Zde jsme probírali implementační problémy, možná řešení a organizaci další práce. Mimo osobních schůzek jsme konzultovali hlavně prostřednictvím aplikace Skype, přes společnou konverzaci. Zdrojové kódy jsme sdíleli prostřednictvím GitHubu, což je nejznámější server pro hosting open-source projektů.

Práce na projektu nám ukázala, jak těžké je pracovat v týmu, jak se projekt vyvíjí od návrhu, jak důležité je testování a zodpovědná práce každého člena. Důležité bylo také psát čitelný a dobře komentovaný program, což napomáhalo přehlednosti celého projektu. Velmi nám pomohlo pokusné odevzdání, do kterého jsme měli mít funkční projekt a ukázalo nám, na čem máme ještě zapracovat.

5.1 Rozdělení práce

- **Tomáš Bank** - lexer (spolu s Markem), interpret (spolu s Rolandem)
- **Mark Birger** - rekurzivní sestup, lexer (spolu s Tomášem), precedenční analýza (spolu s Daniilem) spojování jednotlivých částí
- **Roland Botka** - interpret (spolu s Tomášem), dokumentace
- **Zdenko Brandejs** - Shell sort algoritmus, Boyer-Moorův algoritmus
- **Daniil Khudiakov** - vyhledávací strom, precedenční analýza (spolu s Markem)

6 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 metody teórií formálních jazyků v praxi, implementace algoritmů abstraktních datových typů a spolupráci v malém týmu. Prohloubili jsme si znalost rekurze a rekurzivní volání funkcí, porozuměli jsme základům interpretů a překladačů. Naučili jsme se pracovat v týmu a prezentovat svou práci a nápady.

7 Použité zdroje

- Jan M. Honzík: Studijní opora pro předmět Algoritmy, 2012
- Alexander Meduna, Roman Lukáš: Studijní opora pro předmět Formálny jazyky a překladače, 2012
- Zbynek Křivka, Roman Lukáš, Lukáš Rychnovský: Příručka pro studenty předmětu Formální jazyky a překladače Jak na projekt, 2007
- Lukáš Rychnovský, Zbynek Křivka: Příručka pro studenty předmětu Formální jazyky a překladače Práce v týmu, 2007

8 Přílohy

8.1 Metriky kódu

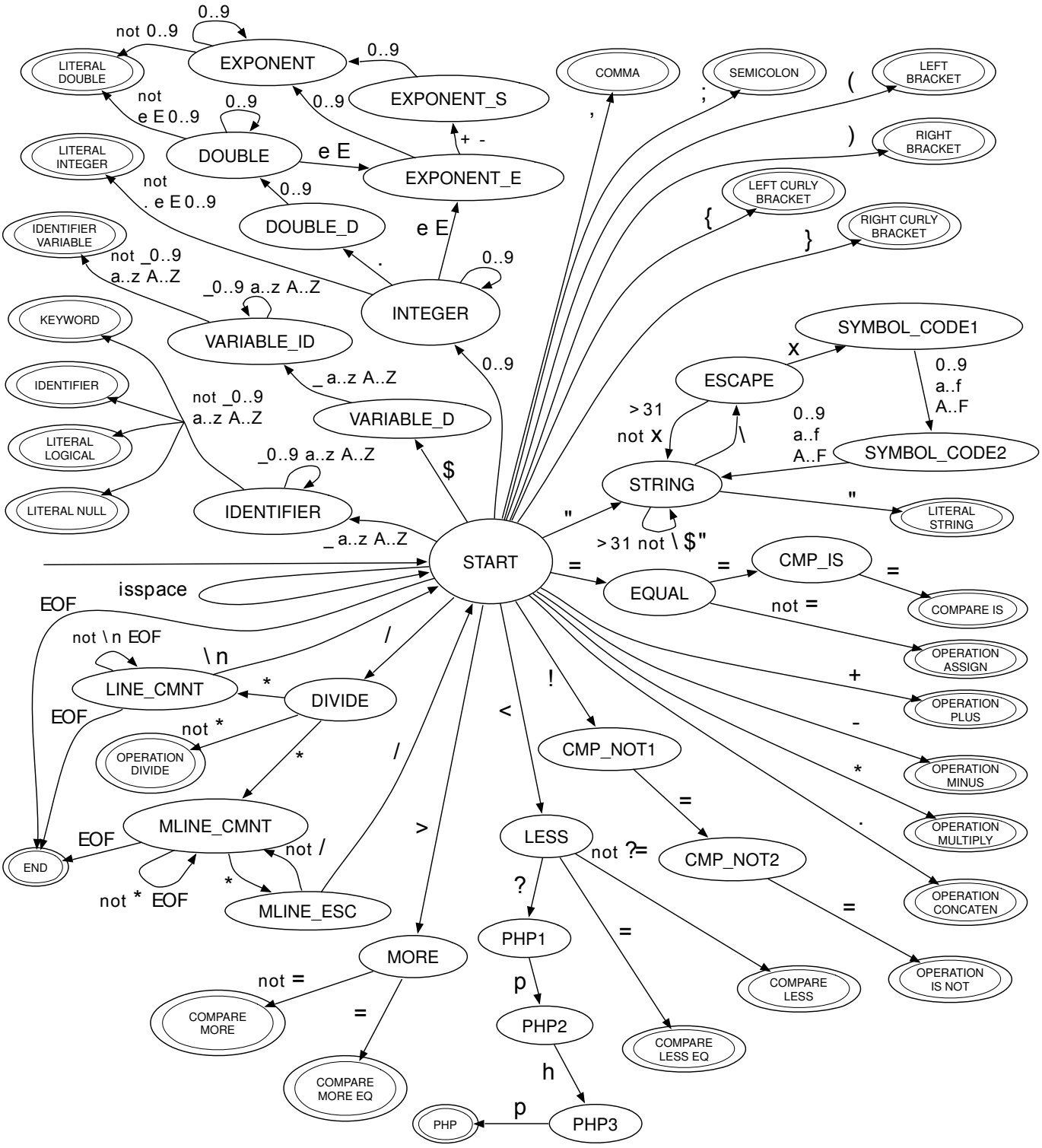
Počet souborů: 22

Počet řádků zdrojového textu: 4468

Velikost statických dat: 129 198 B

Velikost spustitelného souboru: 56 871 B

8.2 Konečný automat



Slouží pro lexikální analýzu

8.3 LL-gramatika

1	<**program**>	→	PHP → <?php
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**>	→	ε
7	<**params**>	→	$id <*params_more*>
8	<**params_more**>	→	, $id <*params_more*>
9	<**params_more**>	→	ε
10	<**cmd_sequence**>	→	ε
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**>	→	ε
18	<**input**>	→	<*expression*> <*input_more*>
18	<**input_more**>	→	, <*expression*> <*input_more*>
19	<**input_more**>	→	ε

Slouží pro rekurzivní sestup