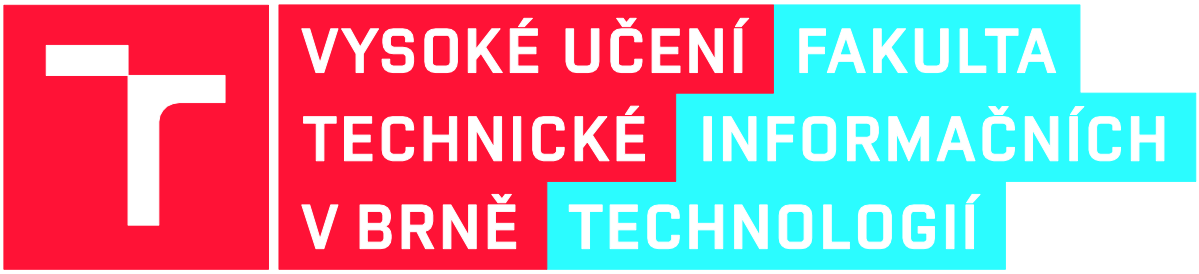


Dokumentace k projektu IFJ

Interpret jazyka IFJ16

Tým 046, Varianta b / 3 / II

12. prosinec 2016

Vilém Faigel (vedoucí) - 0%	xfaige00
Maroš Cocuľa - 25 %	xcocul00
Dominik Drdák - 25 %	xdrdak01
Martin Jaroš - 25 %	xjaros37
Rudolf Kučera - 25 %	xkucer91

Obsah

1. Úvod

Tato dokumentace přibližuje implementaci interpreta imperativního jazyka IFJ16, který je zjednodušenou podmnožinou jazyka Java SE 8. Projekt je vypracován pro předměty IFJ a IAL.

2. Řešení projektu

Projekt je rozdělen do čtyř hlavních bodů a to lexikální analýzy, syntaktické analýzy, precedenční analýzy a interpretu. Jednotlivé body budou popsány v následujících kapitolách dokumentace.

2.1 Lexikální analýza

Lexikální analýza, dál jenom (LA), je první modul našeho programu. Hlavní úkol LA je pročítat vstupní kód a rozdělit ho na základě jednotlivých lexikálních pravidel na lexémy (tokeny). LA je implementována jako konečný automat. Tento automat je nakreslen v příloze A.

LA úzko spolupracuje se syntaktickou analýzou. Syntaktická analýza volá funkci get(), která vrátí vždy jen jeden lexém. Token je tvořen strukturou, která obsahuje potřebné informace o lexému. A to konkrétní hodnotu lexému a číslo typu integer. Číslo lexému jednoznačně identifikuje typ lexému (například číslo, řetězec, klíčové slovo…). Pokud LA ve vstupním kódu najde chybu, skončí celý program a navrátí se chybový kód.

Jednotlivé typy tokenu a struktura pro token se nachází v souboru scaner.h.

2.2 Syntaktická analýza

Syntaktická analýza se skládá ze dvou hlavních částí:

1. Analýza kontextu jazyka
2. Analýza zpracování výrazů (precedenční analýza)

Na vstup syntaktické analýzy přicházejí jednotlivé tokeny z lexikální analýzy. Výstupem jsou naplněné tabulky symbolů a třiadresní kód instrukcí.

Při implementování syntaktické analýzy jsme využili metodu dvojího rekurzivního sestupu, založeného na LL-gramatice jazyka. Pomocí LL-gramatiky jsou porovnávány jednotlivé tokeny. Při nárazu na výraz v kódu se volí precedenční analýza. Ta zpracovává výrazy a zjednodušuje je. LL gramatika jazyka je popsána v příloze B. Tabulka precedenční analýzy je popsána v příloze C.

Při syntaktické analýze se kontrolují syntaktické, ale i sémantické chyby. Při výskytu chyby se vrátí návratová hodnota dané chyby.

2.3 Interpret

Interpret slouží k vykonání instrukcí. Spouští se jako poslední modul a pracuje s instrukční páskou. Instrukce se skládá ze třiadresního kódu, který generuje syntaktická analýza. Instrukce generuje i precedenční analýza.

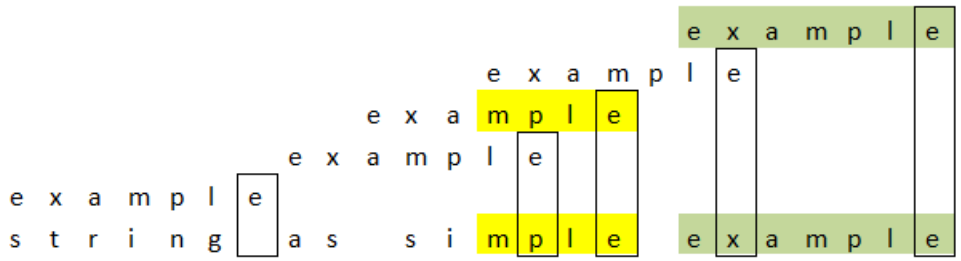
Po skončení syntaktické analýzy, když je vygenerována páska, se spouští interpret a prochází páskou až do konce a vykonává instrukce. Na základě instrukcí pak spouští konkrétní procesy pro vykonání.

3. Algoritmy a datové struktury pro IAL

3.1 Boyer-Moorův algoritmus

Vestavěná funkce find slouží k vyhledání podřetězce v řetězci. Funkce je implementována pomocí algoritmu Boyer-Moore s využitím algoritmu BMA. Pomocí algoritmu se dají přeskočit nevhodné znaky, tím pádem se výrazně zrychluje vyhledání podřetězce.

Vzorek se porovnává zprava doleva po jednotlivých znacích, dokud nedojde ke shodě všech znaků. Pokud ke shodě nedojede, vzorek se posune o celou jeho délku a porovnává se znovu zprava. Názorný příklad je na obrázku 1.



Obrázok 1 : Popis Boyer-Moore algoritmu pomocou BMA

3.2 Shell sort

Pomocí funkce sort jsme měli implementovat řídící algoritmus shell sort. Tento algoritmus je nestabilní a pomalejší než Quick či Heap sort. Algoritmus je podobný Insert sortu, s rozdílem, že využívá snižující přírůstek. To znamená, že neřadí jenom prvky vedle sebe, ale i prvky s mezerou, a to až do poloviny seřazovaného pole. V každém kroku je tato mezera zmenšována. Tím pádem jsou prvky vysokých a nízkých hodnot přemístěné rychle na správnou stranu pole. Jako optimální hodnota pro zmenšování mezery byla empiricky nalezená konstanta 2,2. Po každém kroku se mezera dělí touto konstantou. Použitím této konstanty je dosáhnuto časové složitosti O (n3/2).

Příklad shell sortu je na obrázku 2. V každém průchodu se porovnávají jen buňky stejné barvy.

průchod								
0	35	33	42	10	14	19	27	44
1	14	19	27	10	35	33	42	44
2	14	10	27	19	35	33	42	44
výsledek	10	14	19	27	33	35	42	44

Obrázok 2 : Shell sort

3.3 Tabulka symbolů

Pro uchování symbolů jsme použili tabulku s rozptýlenými položkami. Výhodou této varianty je rychlost vyhledání symbolů uložených v tabulce. Tabulku tvoří pole ukazatelů na jednosměrně vázaný seznam synonym. Každá položka seznamu obsahuje symbol a ukazatel na další prvek tabulky. Symbol se vyhledá na základě mapovací funkce, následně se prochází seznamem, dokud se prvek nenajde, anebo se nedojde na konec seznamu.

4. Rozdělení práce

Maroš Cocuľa – lexikální analýza, dokumentace, vestavěné funkce, tabulka symbolů
Dominik Drdák – interpret, návrh instrukcí, dokumentace
Vilém Faigel – precedenční analýza
Martin Jaroš – syntaktická analýza, dokumentace
Rudolf Kučera – interpret, návrh instrukci, dokumentace

5. Závěr

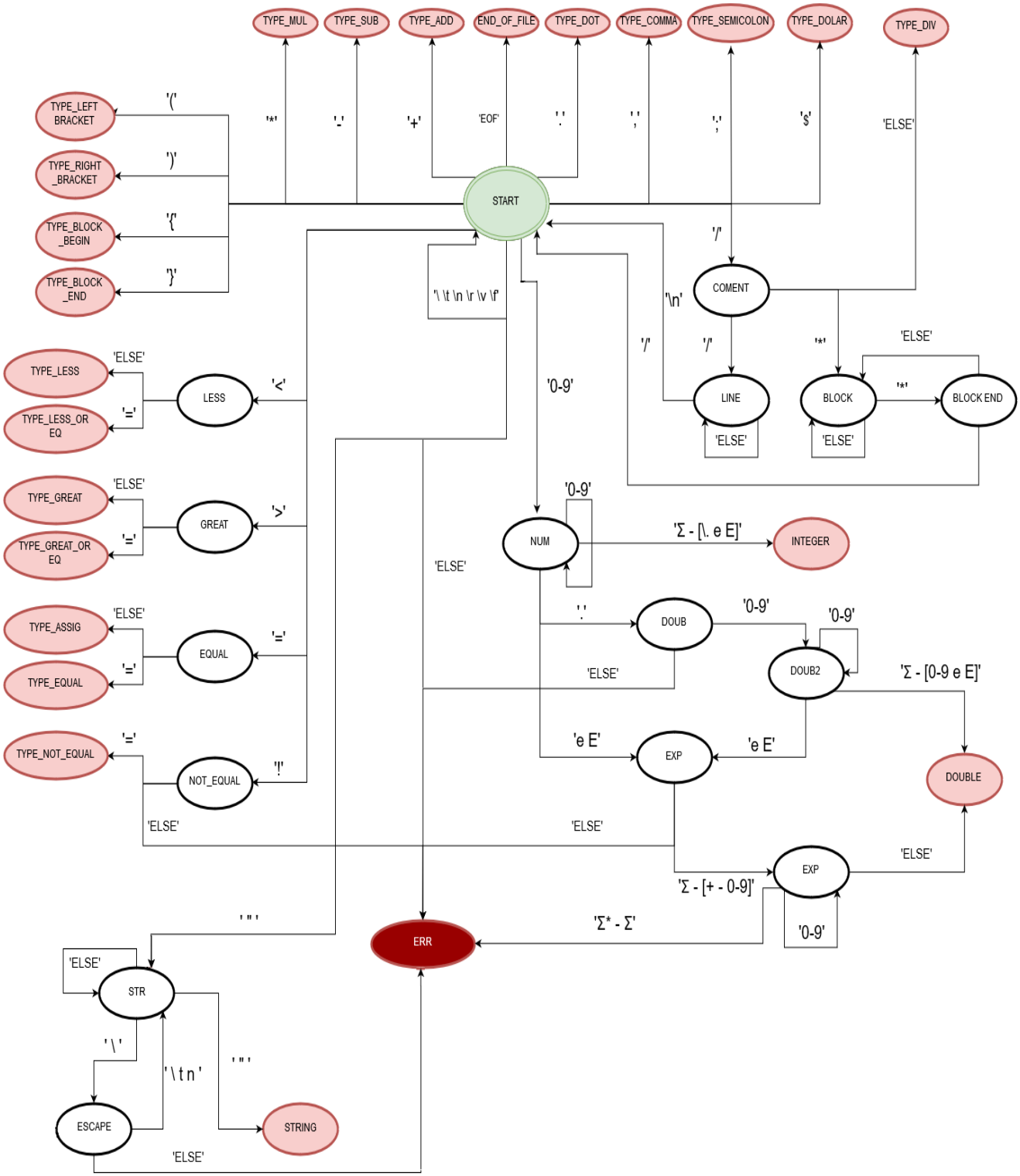
Na práci s projektem jsme se naučili pracovat v týmu na nejrozsáhlejším projektu v bakalářském studiu. Byla to pro nás velká zkušenost. Dále jsme se naučili pracovat s verzovacím systémem Git a základy principu funkce interpretu a jeho implementace. Bohužel jsme se při práci v týmu naučily novou zkušenost, a tou je odlišná funkcionalita jednoho z členu teamu. Tím náš kód utrpěl ztrátu z pohledu precedenční analýzy. To znamená, že kód není plně funkční tak jak by jsme si přály.

6. Literatura

[1] Honzík, Ján. Algoritmy – studijní opora. Brno: Vysoké učení technické, 2016.
[2] Meduna, Alexander. Formální jazyky a překladače – studijní opora. Brno: Vysoké učení technické, 2015

Přílohy

A Konečný automat lexikálního analyzátoru



B LL Gramatika

LL gramatika

1. <main> -> class ID { <main_body> }
2. <main> -> eps

3. <main_body> -> static type id <prom>
4. <main_body> -> eps

5. <prom> -> ; <main_body>
6. <prom> -> (<prom_in>) { <func_body> }
7. <prom_in> -> eps
8. <prom_in> type id
9. <prom_in> type id,

10. <func_body> -> type id ; <func_body>
11. <func_body> -> id <prikaz> ; <func_body>
12. <func_body> -> eps
13. <func_body> -> if (<exp>){ <func_body> } else { <func_body> } <func_body>
14. <func_body> -> while (<exp>){ <func_body> } <func_body>

15. <prikaz> -> = id (<param_list>)
16. <prikaz> -> eps
17. <prikay> -> = id <exp>

18. <param_list> -> eps
19. <param_list> -> type id <param>
20. <param> -> , <param_list>
21. <param> -> eps

Gramatika - výrazy

1. <exp> -> (<exp>)
2. <exp> -><exp> + <exp>
3. <exp> -><exp> - <exp>
4. <exp> -><exp> * <exp>
5. <exp> -><exp> / <exp>
6. <exp> -><exp>><exp>
7. <exp> -><exp><<exp>
8. <exp> -><exp> == <exp>
9. <exp> -><exp>>= <exp>
10. <exp> -><exp><= <exp>
11. <exp>-> eps

C Precedenční tabulka

	+	-	*	/	>	<	>=	<=	==	<>	(	)	i	$	other
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>	ERR
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>	ERR
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>	ERR
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>	ERR
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>	ERR
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>	ERR
>=	<	<	<	<	>	>	>	>	>	>	<	>	<	>	ERR
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>	ERR
==	<	<	<	<	<	<	<	<	>	>	<	>	<	>	ERR
<>	<	<	<	<	<	<	<	<	>	>	<	>	<	>	ERR
(	<	<	<	<	<	<	<	<	<	<	<	=	<	ERR	ERR
)	>	>	>	>	>	>	>	>	>	>	ERR	>	ERR	>	ERR
i	>	>	>	>	>	>	>	>	>	>	ERR	>	ERR	>	ERR
$	<	<	<	<	<	<	<	<	<	<	<	ERR	<	ERR	ERR
other	This will never happen														