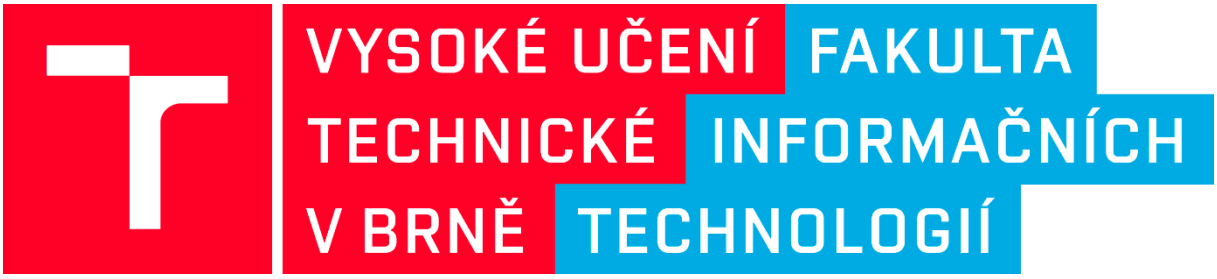
VYSOKÉ UCENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMACNÍCH TECHNOLOGIÍ



Dokumentace projektu do předmětů IFJ a IAL

Interpret jazyka IFJ15

Tým 094, varianta b/3//II

11. prosince 2015

Menšík Jakub (vedoucí)	xmensi03	20% z celkového hodnocení
Měchura Vojtěch	xmechu00	20% z celkového hodnocení
Moravec Matěj	xmorav32	20% z celkového hodnocení
Morávek Jan	xmorav33	20% z celkového hodnocení
Svoboda Jan	xsvobo0u	20% z celkového hodnocení

Obsah

1 Úvod

Tato dokumentace popisuje vývoj implementaci interpretu imperativního jazyka IFJ 2015, který je inspirován jazykem C++, jako projekt do předmětů *Algoritmy* a *Formální jazyky a překladače.* Úlohou interpretu je kontrola vstupního zdrojového kódu a jeho intepretaci v případě, že je vše v pořádku. V opačném případě program informuje o typu chyby.

Také jsou zde rozebrány nejvýznamnější algoritmy použité při implementaci. Dále rozdělení úkolů, způsob práce a problémy s tím spojené.

V závěru dokumentace je shrnutí naší práce a hodnocení a přínos celého projektu.

2 Zadání a návrh implementace

Jazyk IFJ15 je velmi zjednodušenou podmnožinou jazyka C++11. Jazyk IFJ15 je case-sensitive jazyk, to znamená, že záleží na velikosti písmen a je jazykem dynamicky typovaným, to znamená, že datový typ proměnné se určí na základě hodnoty do ní vložené.

Našim úkolem bylo pro tento jazyk naprogramovat interpret a to v jazyku C.

Po analýze problému implementace interpretu jsme jej rozdělili na pět částí, které bylo možné do jisté míry vyvíjet paralelně a nezávisle na sobě. Tyto části jsou:

1. Lexikální analyzátor
2. Syntakticky řízený překlad
   a) Syntaktický analyzátor
   b) Sémantický analyzátor
   c) Generování vnitřního kódu
3. Interpret
4. Vestavěné funkce
5. Správa paměti

Dále naše varianta b/3/II nám říká, jak máme konkrétní problémy řešit.

- Pro vyhledávácí funkci find použijte Boyer-Moorův algoritmus
- Pro řadící funkci sort použijte algoritmus Shell sort
- Tabulku symbolů implementuje pomocí tabulky s rozptýlenými položkami

3 Implementace

V této kapitole si rozebereme způsob implementace jednotlivých částí interpretu.

3.1 Lexikální analyzátor

Lexikální analyzátor zpracovává vstupní zdrojový kód v jazyce IFJ15 a jeho hlavním úkolem je čtení zdrojového souboru a na základě lexikálních pravidel jazyka jeho rozdělení na jednotlivé lexikální části, tak zvané lexémy. Lexikální analyzátor pracuje na žádost syntaktického analyzátoru, kdy přečte další lexém, který vrátí jako token. Tokeny jsou řešeny pomocí abstraktní datové struktury, která obsahuje informace o typu lexému a pro některé typy lexému i jejich hodnotu (například pro double).

Dalším důležitým úkolem lexikálního analyzátoru je odstranění veškerých komentářů a bílých znaků ze zdrojového souboru.

Pokud lexikální analyzátor narazí na neznámý lexém, ohlásí chybu a do konce zpracovávání programu zůstává ve stavu lexikální chyby.

Celý princip lexikálního analyzátoru je založen na konečném automatu (viz obrázek 1), jehož stav po konci běhu analyzátoru oznámí, jaký lexém byl detekován. Implementace se řídila tímto návrhem, avšak v konečné implementaci všechny úspěšné běhy tohoto automatu vedou do jednoho konečného stavy a všechny neúspěšné do

druhého. V obrázku jsou modře označeny stavy, které určí typ lexému, které však dále směřují do dříve zmíněného zcela konečného stavu.

3.2 Syntaktický analyzátor

Syntaktický analyzátor s pomocí lexikálního analyzátoru kontroluje syntaktickou správnost programu. Modul `syntax.c` je založen na metodě rekurzivního sestupu pomocí LL gramatiky (syntaxe jazyka) a řeší analýzu bez výrazů. Modul `vyrazy.c` je založen na precedenční syntaktické analýze a je volán parserem pro výrazy. Současně probíhá jak syntaktická, tak i sémantická kontrola a generování vnitřního kódu.

3.2.1 Syntaktický analyzátor a sémantický analyzátor

Syntaktický analyzátor byl implementován pomocí rekurzivního sestupu podle LL gramatiky, kterou jsem si vytvořili. Analyzátor zároveň kontroluje definice a deklarace funkcí. Pokud narazí na výraz, je zavolána funkce, která daný výraz vyhodnotí a zároveň vytvoří potřebné instrukce tří adresného kódu.

3.2.2 Generování vnitřního kódu

Syntaktický analyzátor také provádí sémantické kontroly jazyka a generuje fragmenty tří adresného kódu.

Vstupem syntaktického analyzátoru je řetězec tokenů, které mu průběžně zasílá lexikální analyzátor. Výstupem syntaktického analyzátoru po analýze vstupu s vyhodnocenými výrazy z precedenční analýzy jsou informace o tom, zda je program syntakticky a sémanticky správně napsán.

3.3 Interpret

V této části se nejprve vloží do globální pásky instrukce, které vygenerovala funkce main, poté již probíhá intepretace tří adresného kódu z instrukční pásky.

3.4 Vestavěné funkce

Zde si blíže popíšeme princip dvou důležitějších vestavěných funkcí.

3.4.1 Find

Vestavěnou funkci `find` jsme dle naší varianty zadání měli implementovat pomocí vyhledávacího Boyer-Moorova algoritmu. Tento algoritmus je velmi rozšířeným způsobem pro hledání podřetězce v řetězci. U tohoto algoritmu je možno přeskakovat velké množství znaků a tím značně urychlit vyhledávání. My jsme využili tzv. „bad character shift“ heuristiku.

Při porovnávání vzoru s textem jsme narazili na znak c v textu, který se nachází na pozici p, který se liší od právě porovnávaného znaku vzoru v místě m. Pak postupujeme takto:

1. Znak c se ve vzoru vůbec nenachází, nemá smysl se tímto dále zabývat a posouváme začátek vzoru na pozici p+1.
2. Znak c se ve vzoru nachází pouze nalevo od porovnávaného znaku vzoru. Proto nejbližší levý shodný znak posuneme na pozici p. Znaky mezi nimi nás nezajímají.
3. Znak c se ve vzoru vyskytuje jak nalevo, tak i napravo, proto jej posuneme pouze o jednu pozici doprava.

3.4.2 Sort

Vestavěnou funkci `sort` jsme dle naší varianty zadání měli implementovat pomocí řadícího algoritmu Shell sort. Tento řadící algoritmus je podobný Insert sortu, avšak neporovnává sousední prvky, které jsou přímo vedle sebe, ale prvky, mezi kterými je určitá mezera. V každém kroku je poté mezera zmenšena. Na několika krocích je mezera zmenšena na jedna a Shell sort pracuje stejně jako Insert sort. Výhodou tohoto přístupu je, že prvky s vysokými a nízkými hodnotami jsou přemístěny na odpovídající stranu pole a v poslední iteraci se přesouvá minimum prvků.

3.5 Správa paměti

Neustálá dynamická alokace paměti a následné ověřování úspěšnosti se jeví jako komplexní záležitost, z tohoto důvodu byl námi implementován modul, který tuto problematiků řeší. Tento modul si ukládá ukazatele na veškerou alokovanou paměť, díky tomu je možné kdykoli uvolnit veškerou, interpretem alokovanou paměť a tím jej připravit na ukončení.

3.6 Tabulka s rozptýlenými položkami

Tabulku symbolů jsme podle zadání měli řešit jako tabulku s rozptýlenými položkami. Při implementaci jsme se inspirovali domácím úkolem z předmětu IAL, avšak jsme museli všechny funkce upravit tak, aby kooperovali s našim projektem. Tabulka symbolů slouží k uchovávání proměnných, výhodou je rychlé hledání. Základem tabulky je pole ukazatelů na jednosměrně vázané seznamy. Proměnné se do tabulky vkládají pomocí hashovací funkce, která vrací index, na který uložíme v poli danou proměnnou.

4 Vývoj

Zde popíšeme, jak probíhal vývoj projektu a používané prostředky

4.1 Rozdělení práce

Menšík Jakub – vedoucí, syntaktický analyzátor, interpret
Měchura Vojtěch – Lexikální analyzátor, dokumentace
Moravec Matěj – Interpret, vestavěné funkce
Morávek Jan – Lexikální analyzátor, testovací soubory
Svoboda Jan – Vyhodnocení výrazů, interpret

4.2 Použité prostředky

Pro online komunikaci jsme používali platformu Facebook, kde pro naše účely byla vytvořena uzavřená skupina. Také jsme se každý druhý týden setkávali, za účelem prezentace již dosažených výsledků a následného plánování úkolů na další dva týdny.

Pro společný vývoj programové části byla jako hosting použita služba GitHub, pro zabezpečení kódu jsme zvolili maskovací pojmenování.

5 Závěr

Projekt byl časově náročný, práce na něm trvala přes dva měsíce. Díky své komplexnosti nás naučil mnohému, ať schopnosti týmové spolupráce, komunikace či plánování.

Při práci na projektu jsme spoléhali na samostudium, neboť práci na projektu jsme chtěli provádět paralelně a k tomu nám nevyhovoval harmonogram přednášek s potřebnými informacemi.

5.1 Literatura

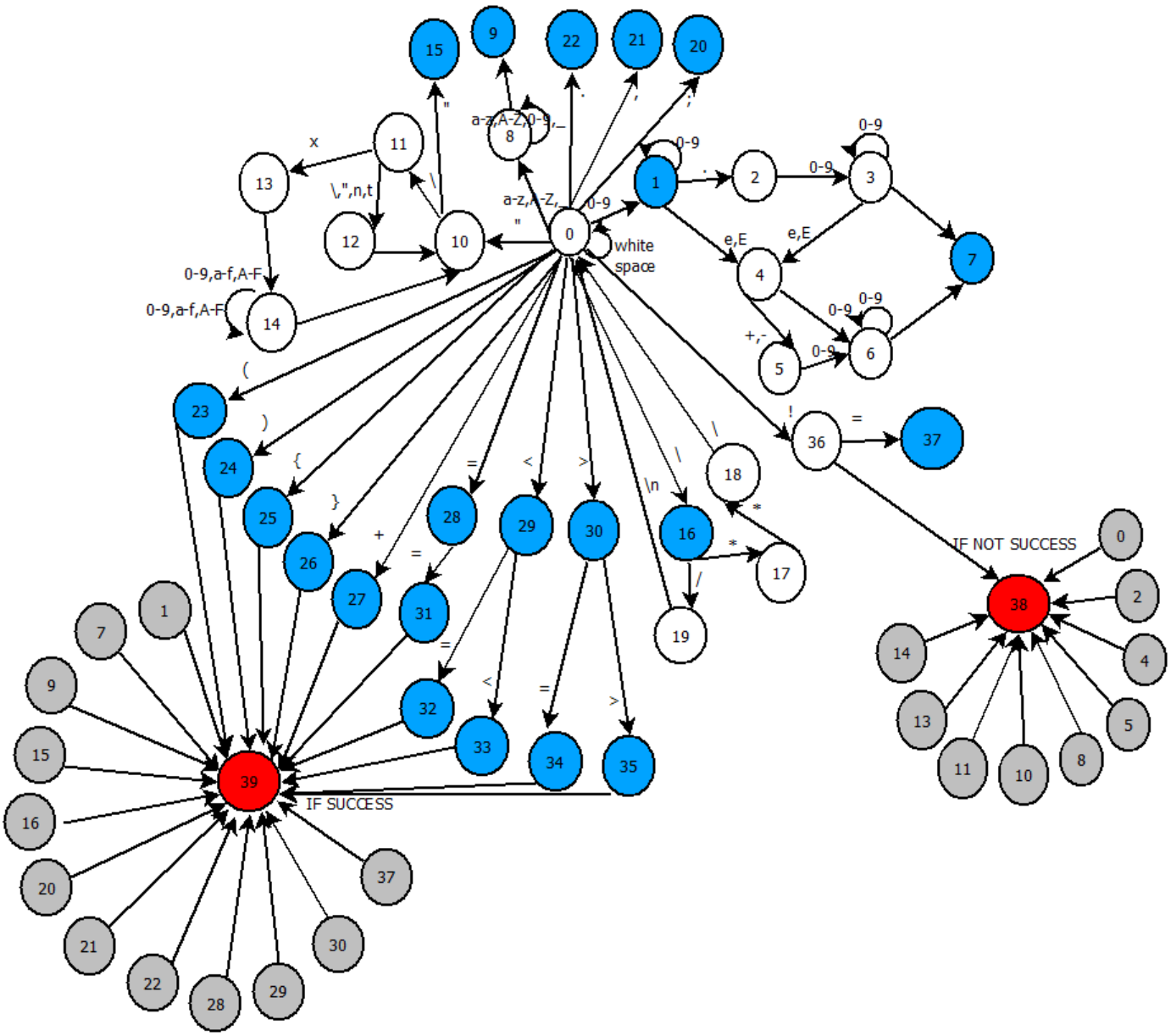
[1] Přednášky, skripta a podklady k předmětu IFJ

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. Communications of ACM, 20(10):762-772, 1977

[3] Meduna, A.: Automata and Languages. London, Springer, 2000.

6 Příloha

6.1 Konečný automat pro implementaci lexikálního analyzátoru



Poznámka: Šedé symboly zastupují stavy, které se již diagramu vyskytují, avšak jejich výskyt je duplicitní pro ušetření zbloudilých čar, které by způsobily značnou nepřehlednost.

6.2 Precedenční tabulka

	+	-	*	/	==	!=	<=	>=	<	>	(	)	ID	$
+	V	V	M	M	V	V	V	V	V	V	M	V	M	V
-	V	V	M	M	V	V	V	V	V	V	M	V	M	V
*	V	V	V	V	V	V	V	V	V	V	M	V	M	V
==	V	V	V	V	V	V	V	V	V	V	M	V	M	V
!=	M	M	M	M	V	V	V	V	V	V	M	V	M	V
<=	M	M	M	M	V	V	V	V	V	V	M	V	M	V
>=	M	M	M	M	V	V	V	V	V	V	M	V	M	V
<	M	M	M	M	V	V	V	V	V	V	M	V	M	V
>	M	M	M	M	V	V	V	V	V	V	M	V	M	V
(	M	M	M	M	M	M	M	M	M	M	M	R	M	C
)	V	V	V	V	V	V	V	V	V	V	C	V	C	V
ID	V	V	V	V	V	V	V	V	V	V	C	V	C	V
$	M	M	M	M	M	M	M	M	M	M	M	C	M	C

M - menší
V - větší
R - rovno
C - syntaktická chyba

6.3 LL gramatika

<SYNTAX> → <FUNKCE>

<FUNKCE> → <TYP> id (<PARAM>) {<STAT>} <FUNKCE>

<FUNKCE> → <TYP> id (<PARAM>); <FUNKCE>

<FUNKCE> → epsilon

<PARAM> → <TYP> id <PARAM2>

<PARAM> → epsilon

<PARAM2> → , <TYP> id <PARAM2>

<PARAM2> →epsilon

<STAT> → if (<EXP>) { <STAT> } <ELSE>

<STAT> → { <STAT>} <STAT>

<STAT> → return <EXP>; <STAT>

<STAT> → for (<TYP> id = <EXP>; <EXP>; id = <EXP>) {<STAT>} <STAT>

<STAT> → id = <EXP>

<STAT> → id = id (<ARG>)

<STAT> → int id

<STAT> → int id = <EXP>

<STAT> → int id = id (<ARG>)

<STAT> → double id

<STAT> → double id = <EXP>

<STAT> → double id = id (<ARG>)

<STAT> → string id

<STAT> → string id = <EXP>

<STAT> → string id = id (<ARG>)

<STAT> → auto id = <EXP>

<STAT> → auto = id (<ARG>)

<STAT> → cin <CIN>

<STAT> → cout <COUT>

<STAT> → epsilon

<STAT> → int length (<VEST>)

<STAT> → int find (<VEST>)

<STAT> → string substr (<VEST>)

<STAT> → string concat (<VEST>)

<STAT> → string sort (<VEST>)

<ELSE> → else {<STAT>} <STAT>

<ARG> → var <ARG2>

<ARG> → epsilon

<ARG2> → , var <ARG2>

<ARG2> → epsilon

<CIN> → >> id <CIN2>

<CIN2> → >> id <CIN2>

<CIN2> → epsilon

<COUT> → << var <COUT2>

<COUT2> → << var <COUT2>

<COUT2> → epsilon