VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu z předmětu IFJ/IAL

Implementace jazyka IFJ14

Varianta b/3/I
Tým číslo 45

Tomáš Coufal xcoufa09, 25%, vedoucí týmu
Roman Halík xhalik01, 25%
Yurij Hladyuk xhlady00, 25%
Jakub Jochlík xjochl00, 25%

V Brně, 14. prosince 2014

OBSAH

1 ÚVOD

Tato dokumentace pojednává o tvorbě a úskalích týmového projektu, jehož výsledkem je implementace interpretu jazyka IFJ14, jenž je podmnožinou jazyka Pascal. Budeme se postupně zabývat jednotlivými částmi překladače a popíšeme řešené problémy při implementaci. Taktéž zmíníme způsob práce v týmu a organizaci práce.

2 ZADÁNÍ

Jazyk IFJ14 je podmnožinou jazyka Pascal, jedná se tedy o staticky typovaný jazyk, který je case insensitive. Nezáleží tedy na velikosti písmen. Pro každý tým bylo přiděleno specifické zadání. Pro náš tým konkrétně varianta b/3/I, která znamená následující:

- Implementace tabulky symbolů pomocí **binárního vyhledávacího stromu**
- Implementace řazení pomocí algoritmu **shell sort**
- Vyhledávání podřetězce v řetězci za využití **Boyer-Mooreova algoritmu**

Program načítá zdrojový soubor jako parametr příkazové řádky a vyhodnotí, zdali je kód syntakticky i sémanticky v pořádku, a jestliže ano, provede kód. V případě chyby vrací jako návratovou hodnotu kód chyby definovaný zadáním projektu. Překladač umí pracovat s datovými typy **integer**, **real**, **string** a **boolean**, zpracovávat výrazy s **aritmetickými** operátory a s operátory **porovnávání**. Dále umožňuje větvení **if-else** a cyklus **while**. Samozřejmostí jsou blokové komentáře. Navíc umožňuje pracovat se čtyřmi vestavěnými funkcemi:

- **Lenght** – vrací délku vstupního řetězce
- **Sort** – seřadí vstupní řetězec tak, že znak s nižní ordinární hodnou předchází znaku s vyšší ordinární hodnotou, vrací řetězec obsahující seřazené znaky
- **Find** – vyhledá ve vstupním řetězci první výskyt zadaného podřetězce, vrací jeho pozici (počítáno od 1)
- **Copy** – vrátí podřetězec vstupního řetězce, určený jeho délkou a počátkem

3 ŘEŠENÍ

3.1 Implemetace

3.1.1 Lexikální analýza

Lexikální analyzátor pro tento projekt IFJ je založen na deterministickém konečném automatu. Mezi jeho hlavní funkce patří rozpoznání a klasifikaci lexémů, které následně reprezentuje pomocí tokenů. Zároveň také odstraňuje komentáře a všechny bílé znaky. Token v našem projektu je formou dvouprvkové struktury. První prvek určuje samotný typ tokenu, druhý pak dodatečné informace, je-li potřeba.

Vstupem je zdrojový program v jazyce IFJ14 a výstupem je token. Ten je pak dále použit jako vstup pro syntaktický analyzátor.

Konečný automat pro námi vytvořený lexikální analyzátor je tvořen 35 stavy, z nichž 21 je stavů konečných. Z důvodu přehlednosti bude následující diagram rozdělen do dvou částí:

- Přerušovaný stav ‘’int’’ značí místo rozdělení

Hlavní tělo konečného automatu:

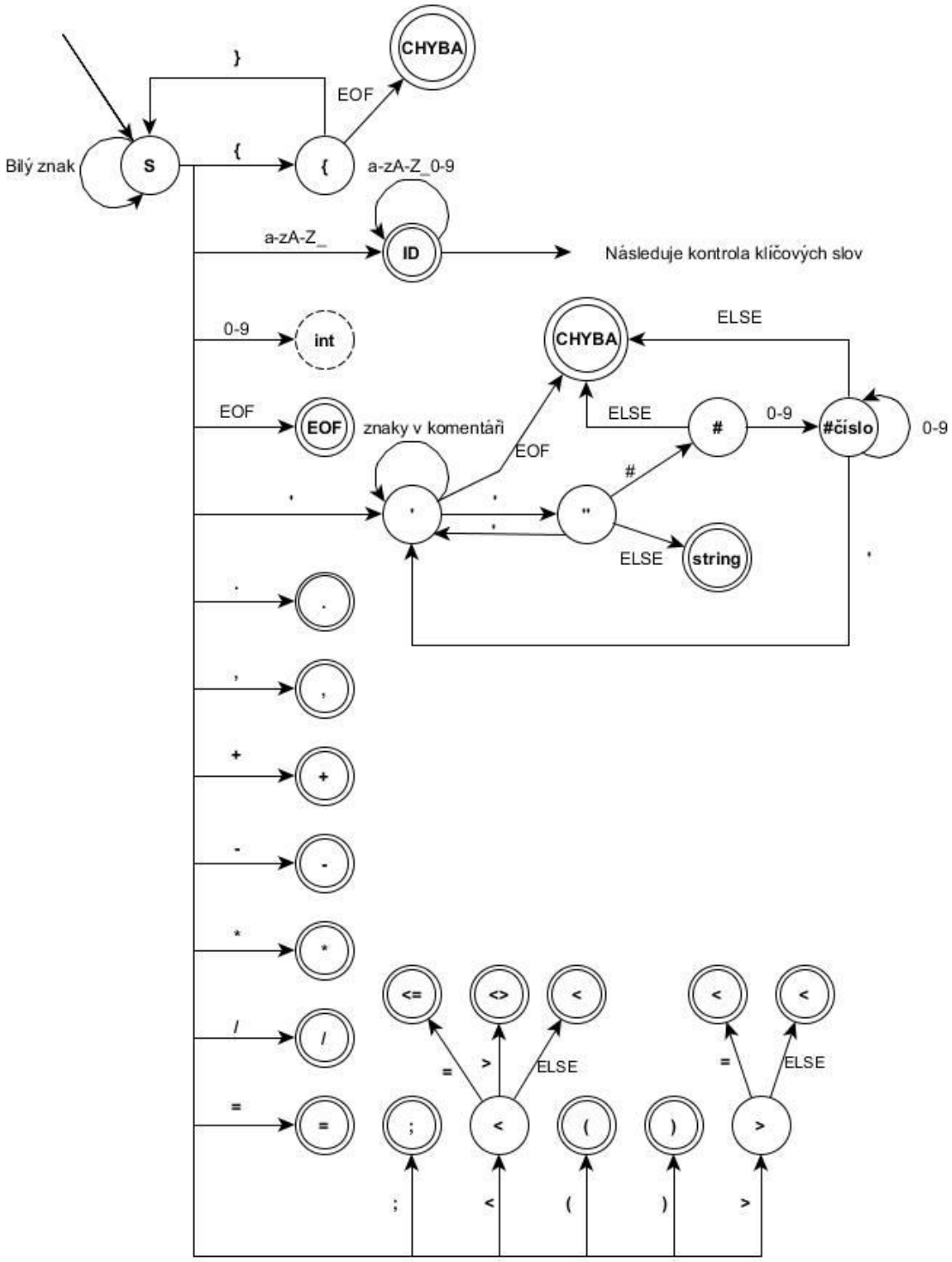
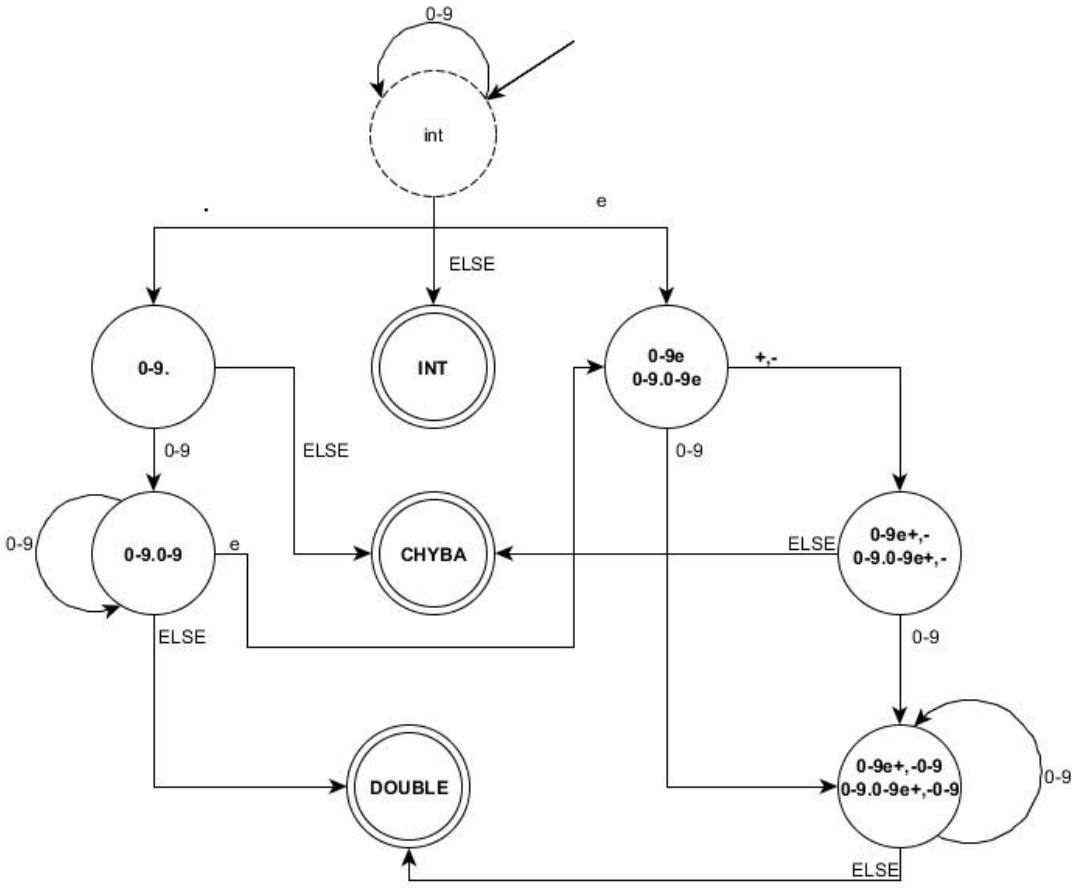


Diagram 1.1

K-A pro číselný literál:



Poznámka: Znak “e” zastupuje jak velké “E” tak malé “e”

Diagram 1.2

3.1.2 Syntaktický analyzátor

Syntaktický analyzátor neboli parser tvoří jádro celého překladače. Na vstupu syntaktického analyzátoru je očekávám soubor obsahující zdrojový kód. Analyzátor přijímá pomoci lexikálního analyzátoru tokeny, které dále zpracovává jedním ze dvou způsobů.

Prvním způsobem je rekurzivní sestup založený na LL gramatice (tabulka 1). LL gramatika je založená na postupném ověřování tokenu pomoci těchto pravidel. Funkce jsou implementovány tak, že postupně ověřují aktuální token a poté požádají o následující. Ne vždy se dá rozhodnout na základě jednoho tokenu a často se předávají tokeny mezi funkcemi.

<program> -> <decareList> <funciton> <body> <EOF>
<declareList> -> var <DeclareListContent>
<declareListContent> -> id : type ; <declareListContent>
<declareListContent> -> eps
<paramsList> -> id : type ; <paramsList>
<paramsList> -> id : type
<function> -> function id (<paramList>) : typ ; <forward>
<function> -> eps
<forward> -> forward ;
<forward> -> <declareList> <body> ; <function>
<paramsCall> -> id , <paramsCall>
<paramsCall> -> id
<callFunction> -> id (<paramsCall>)
<state> -> id := <evalExpression>
<state> -> id := <callFunction>
<state> -> if <evalExpression> then <body> ; else <body>
<state> -> while <evalExpression> do <body>
<state> -> write (<type>)
<state> -> readln (<type>)
<state> -> id := <evalExpression>
<state> -> id := <call>
<state> -> if <evalExpression> then <body> ; else <body>
<state> -> while <evalExpression> do <body>
<state> -> write (<type>)
<state> -> readln (<type>)
<statements> -> <state> ; <statements>
<statements> -> <state>
<body> -> <statements> end
<body> -> end

Tabulka 1: LL gramatika

Precedenční syntaktickou analýzou jsou řešeny výrazy. Využívá se zde tabulka priorit (Tabulka 2). Námi naimplementovaná funkce si ukládá tokeny na zásobník a poté podle těchto pravidel vytváří tři adresný kód pro interpret. Ověřuje se zde i sémantika přímo při zpracovávání výrazu. Porovnává se zde kompatibilita mezi prvním, druhým operandem a adresou, do které se má uložit výsledek.

	+	-	*	/	>	<	>=	<=	=	<>	(	)	i	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
=	<	<	<	<	<	<	<	<	>	>	<	>	<	>
<>	<	<	<	<	<	<	<	<	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	x
)	>	>	>	>	>	>	>	>	>	>	x	>	x	>
i	>	>	>	>	>	>	>	>	>	>	x	>	x	>
$	<	<	<	<	<	<	<	<	<	<	<	x	<	x

Tabulka 2: Tabulka priorit

3.1.3 Sémantická analýza

Semantická analýza má za úkol vyhodnotit výrazy a typovou správnost operandů. Také kontroluje, zda-li souhlasí datový typ l-hodnoty a další. Pro vyhodnocování výrazů byla použita precedenční analýza na základě tabulky priorit (viz. výše). Ta pomocí implementovaného zásobníku rozhoduje o prioritě operací a kterých operandů ve výrazu se tato operace bude týkat. Prioritu operací ovlivnují také závorky a proto s nimi při vyhodnocování výrazu také počítáme a zohledňujeme je. Dále při vyhodnocování výrazů musíme řešit typové konverze (například při dělení nebo také při počítání mezivýsledků, které mohou mít jiný typ než výsledek finální).

V sémantické analýze se, stejně jako v analýze syntaktické, generují instrukce pro interpret, na jejichž základě jsou výrazy vyhodnoceny při běhu programu.

3.1.4 Interpret

Úkolem interpretu je vykonávání instrukcí v podobě tří-adresného kódu, který přijímá od parseru (modul obsluhující syntaktickou a sémantickou analýzu). Samotný interpret je vlastně instrukční sada obsahující aritmetické instrukce, instrukce porovnávání, instrukce vestavěných funkcí a speciální instrukce – například instrukce skoku nebo přiřazení.

Interpret nejprve načte jednu instrukci, vykoná ji a načte instrukci následující, dokud není konec programu. Výjimkou je instrukce skoku, která provádí skok v seznamu instrukcí na danou adresu – návěští.

3.1.5 Binární vyhledávací strom

V našem zadání jsme imlementovali tabulku symbolů pomocí binárního stromu. V naší impementaci binární strom používá jako klíč identifikátor proměnné či funkce a jejich umístění se řídí lexikografickým porovnáním. Existují zde globální tabulky a tabulky lokání pro každou funkci. V nich jsou obsazené proměnné definované v daném kontextu. Dále pro každé volání funkce se vytváří

„instance“ lokální tabulky, kde se ukládají hodnoty pro to konkrétní volání. V případě rekurze tedy nedochází ke konfliktům. Každý uzel proměnné v tabulce obsahuje její identifikátor, datový typ a hodnotu. U uzlu funkce pak hodnotou myslíme návratovou hodnotu a navíc zde jsou informace o typu návratové hodnoty, odkaz na list instrukcí dané funkce, seznam parametrů a odkaz na tabulku symbolů pro funkci.

3.2 Algoritmy

3.2.1 Shell sort

Shell sort, neboli algoritmus řazení se snižujícím se přírůstkem, je řadící algoritmus založený na principu záměny dvou prvků vzdálených o stejný krok, který je na počátku řazení roven polovině délky řetězce, který řadíme. S každým průchodem se délka kroku půlí. Implementace algoritmu proběhla na základě znalostí nabytých v předmětu IAL.

3.2.2 Boyer-Mooreův algoritmus

Boyer-Mooreův algoritmus je metoda umožňující rychlé vyhledání podřetězce v zadaném řetězci. Základní specifikum tohoto algoritmu spočívá v úvaze, že některé znaky, které se nikdy nemohou rovnat vyhledávanému vzorku, lze přeskočit. Touto metodikou lze velmi urychlit vyhledávání.

Samotný princip spočívá v průchodu řetězcem zleva doprava a porovnáváním posledního znaku vzorku s řetězcem, přičemž při shodě se dále porovnávají další znaky vzorku, směrem doleva. Jestliže je neshoda nalezena na i-tém znaku vzorku, posun v rámci řetězce se provede o délku vzorku - i + 1, tedy přesně za znak, který byl odlišný. Jinak se provádí posun o celou délku vzorku. Jestliže je nalezena shoda, algoritmus končí a vrací pozici nalezeného podřetězce (indexováno od 1).

Při implementaci algoritmu byly taktéž využity znalosti získané na přednáškách a v materiálech předmětu IAL.

3.2 Testování

Testování probíhalo průběžně. Snažili jsme se navzájem kontrolovat práci druhých a ošetřit všechny možné vstupy.

3.4 Práce v týmu

Práci jsme si rozdělili přibližně následujícím způsobem:

- Tomáš Coufal – sémantická analýza, binární strom
- Roman Halík – interpret
- Yurij Hladyuk – syntaktická analýza
- Jakub Jochlík – lexikální analýza

Doplňkové záležitosti se rozdělily dle potřeby a aktuálního vytížení členů týmu. Na vše přitom dohlížel náš vedoucí, který celý projekt „tmelil“ ve funkční celek. Ke sdílení kódu byl využit repozitář GIT.

4 METRIKY KÓDU

- Řádků kódu: 3555
- Počet zdrojových souborů: 23
- Velikost spustitelného souboru:

5 ZÁVĚR

Projekt byl vyvíjen přibližně po dobu dvou měsíců, z toho poslední dva týdny velmi intenzivně, díky čemuž interpret splňuje veškeré formální požadavky a pracuje dle zadání. Nebyla však implementována žádná rozšíření.

6 LITERATURA

- HONZÍK, Jan M. Algoritmy: Studijní opora. 2014. verze 14-Q.