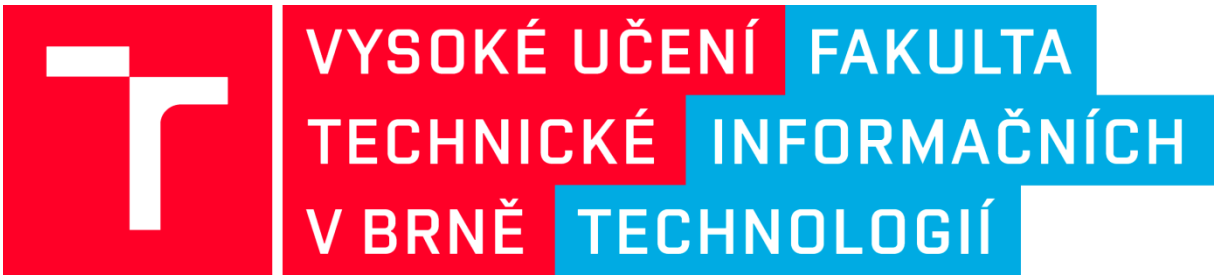
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentace k projektu do předmětů IFJ a IAL

Interpret jazyka IFJ15

Tým 89, varianta b/1/I
datum: 13.12.2015

Seznam autorů:
Novák David, xnovak1m, 20% - vedoucí
Trbola Martin, xtrbol00, 20%
Urban Šimon, xurban63, 20%
Juráček Ivo, xjurac05, 20%
Majer Marek, xmajer15, 20%

Rozšíření – BASE, WHILE, FUNEXP, SIMPLE

1. Úvod

Tato dokumentace popisuje návrh a implementaci interpretu jazyka IFJ15, který je zjednodušenou podmnožinou jazyka C++. Zvolili jsme si zadání b/1/l, kde bylo za úkol přidat do interpretu funkci **find**, která bude vyhledávat podřetězec v řetězci pomocí Boyer-Mooreova algoritmu, funkci **sort** (algoritmus Quicksort) a implementovat tabulku symbolů pomocí binárního vyhledávacího stromu.

Řešení interpretu jsme rozdělili na 4 hlavní úseky:

- Syntaktický analyzátor
- Sémantický analyzátor
- Lexikální analyzátor
- Interpret

2. Implementace částí

2.1 Lexikální Analyzátor

Lexikální Analyzátor (LA) je první a jediná část interpretu, která pracuje přímo se zdrojovým textem. Jeho hlavním úkolem je číst zdrojový text, zbavit ho nedůležitých částí (bílé znaky, komentáře) a převést lexémy na tokeny, které jsou dále zpracovávány syntaktickou analýzou. LA je implementován jako konečný automat, který zpracovává vstup znak po znaku a na základě stavu, ve kterém skončil, určí typ tokenu. LA končí, když v současném stavu nemůže zpracovat právě načtený znak. Tento znak je uschován a při pokusu o získání dalšího tokenu se u něj začíná. Pokud LA není schopen zpracovat tento znak, končí v chybovém stavu a vrací chybový kód 1. LA pro svou práci (konkrétně buffer) používá upravenou vzorovou knihovnu str.c ze stránek předmětu IFJ.

Celý LA je ovládán Syntaktickým Analyzátorem pomocí připravených funkcí:
init_scanner – připraví LA k práci (alokace bufferu, otevření souboru se zdrojovým textem)
get_token – pokusí se přečíst a převést aktuální lexém na token.
clean_scanner – uvolní všechny prostředky (zavření souboru, dealokace bufferu)

Konečný automat lexikálního analyzátoru:

(Ze všech stavů lze přejít do koncového stavu)

2.3 Sémantický Analyzátor

Sémantický Analyzátor nabízí funkce pro sémantické kontroly, které jsou volány syntaktickým analyzátorem. Využívá funkce pro práci s tabulkou symbolů. Mezi hlavní sémantické kontroly patří kontrola kompatibility datových typů, deklarací a definicí proměnných a funkcí. Ke své funkci používá kromě tabulky symbolů také zásobník, především pro kontroly parametrů funkcí.

2.4 Syntaktický Analyzátor

Syntaktický Analyzátor je jádrem celého interpretu a řídí všechny ostatní části. Samotná syntaktická analýza je řešena metodou rekurzivního sestupu. Syntaktická analýza řídí sémantické akce. Priorita vyhodnocování jednotlivých výrazů probíhá pomocí precedenční analýzy. Precedenční analýza si postupně ukládá jednotlivé tokeny na zásobník a pomocí precedenční tabulky určuje pořadí jednotlivých výrazů. Syntaktická analýza je zároveň generátor tří-adresného kódu pro interpret.

	<	>	<=	>=	==	!=	+	-	*	/	(	)	inm	id	idf	,	=	$
<	>	>	>	>	>	>	<	<	<	<	<	>	<	<	<	>		>
>	>	>	>	>	>	>	<	<	<	<	<	>	<	<	<	>		>
<=	>	>	>	>	>	>	<	<	<	<	<	>	<	<	<	>		>
>=	>	>	>	>	>	>	<	<	<	<	<	>	<	<	<	>		>
==	>	>	>	>	>	>	<	<	<	<	<	>	<	<	<	>		>
!=	>	>	>	>	>	>	<	<	<	<	<	>	<	<	<	>		>
+	>	>	>	>	>	>	>	>	<	<	<	>	<	<	<	>		>
-	>	>	>	>	>	>	>	>	<	<	<	>	<	<	<	>		>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>		>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	>		>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	=		
)	>	>	>	>	>	>	>	>	>	>		>				>		>
inm	>	>	>	>	>	>	>	>	>	>		>				>		>
id	>	>	>	>	>	>	>	>	>	>		>				>	=	>
idf											=							
,	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	=		
=	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<			>
$	<	<	<	<	<	<	<	<	<	<	<		<	<	<			

Precedenční tabulka

LL gramatika použitá v syntaktickém analyzátoru:

P	->	*F P*
P	->	*EOF*
F	->	*type idf (AL) FD*
AL	->	*AD AL2*
AL	->	*ε*
AL2	->	*, AD AL2*
AL2	->	*ε*
AD	->	*type id*
FD	->	*;*
FD	->	*{ STL }*
STL	->	*S STL*
STL	->	*ε*
S	->	*D ;*
S	->	*{ STL }*
S	->	*if (E) S ELSE*
S	->	*for (D ; E ; E) S*
S	->	*while (E) S*
S	->	*do S while (E) ;*
S	->	*return E ;*
S	->	*cin >> id IN ;*
S	->	*cout << E OUT ;*
S	->	*E ;*
D	->	type id I
I	->	*= E*
I	->	*ε*
ELSE	->	*else S*
ELSE	->	*ε*
IN	->	*>> id IN*
IN	->	*ε*
OUT	->	*<< E OUT*
OUT	->	*ε*

2.2 Tabulka symbolů

Naše implementace obsahuje jednu globální tabulku symbolů pro funkce a lokální tabulky symbolů pro proměnné, které jsou umísťovány na zásobník lokálních tabulek. Obě tabulky jsou vytvořeny jako binární vyhledávací strom. Většina operací nad binárním stromem je implementována iterativně, kromě funkcí pro odstranění binárního stromu, které jsou implementovány rekurzivně, protože pro jejich rekurzivní volání není potřeba zásobník.

2.5 Interpret

Interpret je finální fází překladu programu a je spuštěn až při úspěšném ukončení syntaktické a sémantické analýzy. Interpret vykonává posloupnost tří-adresného kódu, který mu byl vygenerován syntaktickým analyzátor v předchozí části. Kvůli možnému rekurzivnímu chování funkcí si musí interpret ukládat aktuální hodnoty proměnných za běhu a nemůže používat tabulku symbolů, interpretační část proto pracuje s rámci, na které si ukládá hodnoty proměnných za běhu. Interpret zároveň kontroluje, jestli se nepracuje s neinicializovanou proměnnou.

2.6 Boyer-Moore Algoritmus

Boyer-Moore (dále BM), je algoritmus sloužící k vyhledávání podřetězce v jiném řetězci. BM se především hodí použít při hledání dlouhého podřetězce. V BM algoritmu jsou znaky v hledaném podřetězci zkoušeny zprava doleva a při neshodě, lze ve většině případů přeskočit velkou část řetězce. Pro implementaci lze použít dvě heurestiky „posun špatného znaku“ a „posun správné přípony“. V naší implementaci jsme využili obě, díky čemuž se zvyšuje efektivita algoritmu.

2.7 Quicksort

Quicksort je velmi rychlý a jednoduše implementovaný řadící algoritmus založený na principu „Rozděl a panuj“ s nejhorší možnou asymptotickou složitostií $O(n^2)$ a s očekávanou složitostí $O(n * log\ n)$. Algoritmus funguje na principu zvolení libovolného prvku v poli, *pivotu.* Naše implementace bere pivot jako pseudomedián, tedy

$$pivot = (levá\ hranice + pravá\ hranice)\ div(2),$$

kde *div* je operátor pro celočíselné dělení. Funkce Quicksort přehází pole tak, aby na jedné straně byly prvky větší než pivot, na druhé menší než pivot a pivot samotný byl umístěn přibližně mezi těmito částmi. Funkci potom rekurzivně zavoláme pro obě rozdělené části.

2.8 Vestavěné funkce

2.8.1 Length

Funkce vrací délku textové řetězce. Tato funkce má jeden vstupní parametr, a to je textový řetězec, nad kterým se iteruje. V každé iteraci se inkrementuje čítač, který symbolizuje počet znaků v řetězci. Jakmile je nalezen znak konce řetězce, iterace se ukončí a vrací se hodnota čítače.

2.8.2 Substr

Tato funkce přijímá tři vstupní parametry. Prvním parametrem je textový řetězec, ze kterého chceme získat podřetězec, druhým je celočíselná hodnota, značící pozici znaku, od kterého znaku se začne počítat podřetězec a třetím je počet znaků vpravo od hodnoty získané ve druhém parametru této funkce. Funkce taktéž testuje, jestli jsou hodnoty ohraničení podřetězce validní. Je-li hodnota třetího parametru natolik vysoká, že by součet druhého a třetího parametru byl vyšší, než velikost prvního parametru, tak funkce vrátí podřetězec, začínající na pozici druhého parametru a končící na konci prvního parametru. Druhý a třetí parametr nesmí být záporný.

2.8.3 Concat

Funkce, která vrací konkatenaci dvou řetězců, přijímá dva textové řetězce. Textový řetězec, získaný druhým parametrem funkce, je vložen za textový řetězec, získaný prvním parametrem funkce.

2.8.4 Sort

Tato funkce přijímá pouze jeden textový řetězec. Vzestupně seřadí řetězec podle ordinální hodnoty jednotlivých znaků pomocí algoritmu Quicksort. Výsledný seřazený řetězec je touto funkcí vrácen.

2.8.5 Find

Poslední implementovaná vestavěná funkce, která má dva vstupní parametry. Prvním je řetězec, ve kterém chceme nalézt jiný řetězec (vzorek), který je zadán ve druhém parametru této funkce. Funkce používá algoritmus Boyer-Moore, pouze pokud se nemá hledat jeden znak, zde je rychlejší použít klasický průchod řetězcem znak po znaku a testovat, jestli se vyskytuje testovaný znak v řetězci. Nevyskytuje-li se vzorek v zextovém řetězci, funkce vrací hodnotu -1.

3. Práce v týmu

3.1 Rozdělení práce

- David Novák: Vedoucí týmu, lexikální analyzátor, git, Makefile, dokumentace
- Martin Trbola: Syntaktický analyzátor, interpret, testování
- Šimon Urban: Sémantický analyzátor, tabulka symbolů, testování
- Ivo Juráček: Řadící algoritmus Quicksort, Booyer-Moore a vestavěné funkce, dokumentace
- Majer Marek: Interpret, tříadresný kód, dokumentace

3.2 Komunikace

Ze začátku projektu jsme se scházeli každé úterý v knihovně. Na prvním srazu jsme si rozdělili jednotlivé úlohy a s přibývajícími vědomostmi z předmětu IFJ jsme na nich mohli začít pracovat. Později jsme týdenní schůzky zrušili a domluva probíhala pouze pomocí chatu a gitu, na který jsme si ukládali aktuální výsledky.

3.3 Metriky kódu

- Řádků kódu : 6654
- Počet souborů : 29

3.4 Poznámky k implementaci

Pro práci se stringy, jsme použili upravenou vzorovou knihovnu str.c ze stránek IFJ.

4. Závěr

Fungování intepretu odpovídá požadovanému zadání a je zároveň rozšířeno o několik rozšíření. Práce na interpretu přinesla spoustu užititečných zkušeností, především z hlediska vývoje aplikací v týmu. Projekt byl časově náročný, ale testovací verzi jsme stihli vytvořit před prvním pokusným odevzdáním, na kterém bohužel nedostala moc bodů. Důvodem byla špatně fungující interpretující část. Chyby jsme odstranili a na druhém pokusném odevzdání už náš interpret fungoval mnohem lépe.

5. Literatura

Kilpelainen, P. *Biosequence Algorithms, Spring 2005 Lecture 3: Boyer-Moore Matching[online].* cit 12. prosince 2015. Dostupné na: http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides03.pdf

Quicksort. *algoritmy.net.* [online]. [2015] [cit. 2015-12-13]. Dostupné na: *https://www.algoritmy.net/article/10/Quicksort*