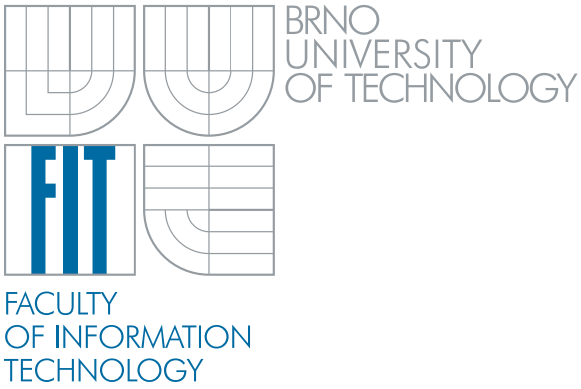
Vysoké učení technické v Brně
Fakulta informačných technológií



Implementácia interpretu imperatívneho jazyka IFJ14

Dokumentácia k projektu pre predmety IFJ a IAL

Tým: 100, varianta: a/2/II, rozšírenia: FUNEXP

11.12.2014

Autori:

Marek Minda	`xminda00`	20%	
Matej Mužila	`xmuzil00`	20%	
Tomáš Poremba	`xporem00`	20%	
Ladislav Šulák	`xsulak04`	20%	
Miloš Volný	`xvolny02`	20%	vedúci

Obsah

1 Úvod

Tento dokument popisuje návrh a implementáciu interpretu imperatívneho jazyka IFJ14. V našej variante bolo zadané implementovať tabuľku symbolov pomocou hashovacej tabuľky, pre metódu zoraďovania použiť algoritmus heapsort, a pre vyhľadávanie *Knuth-Moris-Prattov* algoritmus.

Interpret Jazyka IFJ14 sa skladá z 3 hlavných častí: lexikálna analýza, syntatická a sémantická analýza a interpret. V nasledujúcich kapitolách sú popísané všetky časti interpretu, vrátane vyššie uvedených algoritmov.

2 Zadanie

Úlohou bolo vytvoriť program v jazyku C, ktorý načíta zdrojový súbor zapísaný v jazyku IFJ14 a interpretuje ho. Jazyk IFJ14 je case-insensitive, staticky typovaný jazyk a je až na marginálne prípady podmnožinou jazyka Pascal. Interpret vyžaduje práve jeden argument na príkazovom riadku. Je ním názov súboru so vstupným programom v jazyku IFJ14. V prípade chyby vracia určenú návratovú hodnotu, inak vracia návratovú hodnotu 0.

3 Implementácia

V tejto kapitole je popísaná implementácia hlavných častí interpretu a algoritmov z našej varianty zadania.

V prvom kroku bol navrhnutý a následne implementovaný lexikálny analyzátor. Zároveň prebiehal návrh LL-tabuľky. Už v rannom štádiu implementácie lexikálneho analyzátora prebiehala implementácia rekurzívneho zostupu a tabuľky symbolov. Po úspešnom spojení týchto modulov bola implementovaná precedenčná tabuľka pre výrazy, a prebiehalo testovanie implementovaných modulov. V poslednom kroku zostávalo vytvoriť interpret.

3.1 Lexikálna analýza

Lexikálny analyzátor (scanner) spracováva načítaný zdrojový súbor, postupne spracováva lexémy a vracia lexémom odpovedajúci token. Množina tokenov bola definovaná jazykom IFJ14.

Prvý krok k vytvoreniu lexikálnej analýzy bol návrh konečného automatu. Skladá sa z konečnej množiny stavov a pravidiel, jedného počiatočného stavu a konečnou množinou koncových stavov. V prílohách k projetu v tomto dokumente je znázornená zjednodušená schéma konečného automatu.

Hlavnou funkciou v lexikálnom analyzátore je funkcia `getToken()`, ktorá po načítaní zdrojového súboru vráti nasledujúci token. V prípade identifikátorov, a dátových typov sa ukladá do štruktúry `string` aj hodnota atribútu. Štruktúra string je parametrom funkcie `getToken()`.

3.2 Syntaktická a sémantická analýza

Syntaktický a sémantický analyzátor (parser) je najdôležitejšou časťou celého interpretu. Postupne spracúva zdrojový kód na základe tokenov, ktoré dostáva od lexikálneho analyzátora. Podľa navrhnutej LL gramatiky kontroluje, či postupnosť tokenov reprezentuje syntakticky a sémanticky správne napísaný zdrojový program a generuje inštrukcie vnútorného kódu na základe ktorých interpret spracováva celý program.

3.2.1 Oddelenie lokálnych symbolov od symbolov globálnych

Globálne premenné a funkcie sú uložené v globálnej tabuľke symbolov. Každá funkcia, uložená v tejto tabuľke obsahuje prototyp svojej lokálnej tabuľky symbolov obsahujúcej jej lokálne premenné a li-

terály. Pri zavolaní funkcie interpret vytvorí kópiu tohto prototypu lokálnej tabuľky symbolov s ktorou následne pracuje.

3.2.2 Spracovanie riadiacich štruktúr

Pri spracovaní riadiacich štruktúr nie je možné sa zaobísť bez skokov. Kód teda nie je možné spracovávať lineárne.

Riadiace štruktúry v jazyku *IFJ14* obsahujú vždy riadiaci výraz typu `bool` a jeden alebo dva bloky príkazov. Tento výraz je vyčíslovaný za pomoci precedenčnej analýzy a teda môže obsahovať aj funkcie. Syntaktický analyzátor vkladá do 3-adresného interného kódu skokové inštrukcie a návestia, ktoré zabezpečia korektný prechod programom.

Spracovanie blokov kódu (ohraničených kľúčovými slovami `begin` a `end`) je vyriešené za pomoci rekurzívneho zostupu. Blok je navonok chápaný ako jeden príkaz. Blok samotný môže obsahovať postupnosť príkazov.

3.2.3 Spracovanie výrazov

Spracovanie výrazov je vyriešené za pomoci precedenčnej analýzy. Precedenčná analýza prevádza výraz do postfixovej notácie za pomoci precedenčnej tabuľky. Tiež sa stará o vytvorenie pomocných premenných, potrebných na vyčíslenie daného výrazu. Následne sú generované inštrukcie v internom 3-adresnom kóde, ktoré reprezentujú daný výraz. Pri tomto prevode je kontrolovaná aj kompatibilita typov jednotlivých premenných a literálov.

Za výraz považujeme aj funkciu, respektíve jej návratovú hodnotu. Za výraz taktiež považujeme aj parametre funkcií. Tento prístup nám umožňuje použiť výrazy ako parametre funkcií, funkcie ako parametre funkcií a funkcie ako súčasť výrazov. Náš program teda spĺňa podmienky rozšírenia *FUNEXP*.

Precedenčná tabuľka je vzhľadom na svoju jednoduchosť reprezentovaná poľom, obsahujúcim prioritu každého z operátorov.

3.2.4 Spracovanie funkcií

Syntaktický analyzátor sa tiež stará o spracovanie definícií a deklarácií funkcí, premenných a literálov, spracovanie riadiacich štruktúr programu, blokov a jednotlivých príkazov.

Pri deklarácií funkcií je syntaktický a sémantický analyzátor zodpovedný za vytvorenie prototypu lokálnej tabuľky symbolov pre danú funkciu, naplnenie tejto tabuľky symbolov premennými reprezentujúcimi parametre funkcie a jej naávratovú premennú.

Pri definícií funkcií je syntaktický a sémantický analyzátor zodpovedný navyše aj za vloženie jej lokálnych premenných do protorypu lokálnej tabuľky symbolov a za spracovanie samotného kódu reprezentujúceho túto funkciu.

Pri volaní funkcie syntaktický a sémantický analyzátor vyčísli zadané argumenty danej funkcie, skontroluje ich typ a počet. Ak všetky typy a ich počet súhlasia s tým ako bola funkcia deklarovaná, pridá do zoznamu inštrukcií interného kódu pre každý z argumentov inštrukciu `PUSH` (má obdobný význam ako inštrukcia `PUSH` pri procesoroch *Intel x86*) a inštrukciu `CALL` ktorou prikáže interpretu skočiť na miesto v kóde reprezentujúce začiatok danej funkcie a vykonať ju.

3.3 Interpret

Po úspešnej analýze programu sa prechádza k interpretácii vygenerovaných inštrukcií. Jednotlivé inštrukcie sa spracúvajú za sebou v danom poradí, až na výnimky spôsobené inštrukciami `CALL`, `RET`, `JUMP` a `JUMP_ELSE`. Predchádzajúca analýza totiž nemá prostriedky na to, aby tieto zmeny detekovala.

Jednotlivé inštrukcie sú reprezentované funkciami, ktoré pracujú s tabuľkou symbolov, zoznamom inštrukcií, zásobníkom pre lokálne tabuľky symbolov volaných funkcií, zásobníkom návratových adries a zásobníkom argumentov, ktoré sa predávajú volaným funkciám.

Volanie funkcií je najzložitejšou časťou interpretácie. Samotné volanie je reprezentované skokom na prvú inštrukciu volanej funkcie. Tomu predchádza uloženie argumentov na zásobník, kontrola ich inicializácie, vytvorenie lokálnej tabuľky symbolov, ktorá náleží danej funkcii a jej naplnenie (tzn. priradenie hodnôt parametrom funkcie). Pokračuje to uložením vytvorenej tabuľky symbolov na zásobník lokálnych tabuliek a uložením návratovej adresy. Táto je symbolizovaná inštrukciou, ktorá nasleduje po volaní funkcie. Až v tomto momente sa môže predať riadenie programu volanej funkcii.

Podobne to funguje pri návrate z funkcie. Návratová hodnota funkcie sa uloží do predpripravenej premennej v globálnej tabuľke, z vrcholu zásobníka lokálnych tabuliek sa odstráni tabuľka funkcie ktorá sa vykonávala a z vrcholu zásobníka návratových adries sa vyberie inštrukcia. Touto inštrukciou sa potom pokračuje pri vykonávaní programu.

3.4 Knuth-Morris-Prattov algoritmus

Funkcia `find()` bola implementovaná pomocou *Knuth-Morris-Prattovho* algoritmu. Tento algoritmus pri vyhľadávaní podreťazca v reťazci využíva znalosti, že pri čiastkovom neúspechu nesie podreťazec dostatočné množstvo informácií na určenie začiatku nového hľadania. Potrebuje teda pomocné pole, ktoré si upraví ešte pred samotným vyhľadávaním.

3.5 Heap Sort algoritmus

Funkcia `sort()` bola implementovaná algoritmom heapsort. Heapsort využíva stromovú štruktúru nazývanú halda – heap. V prvom kroku heapsort vytvorí zo zadaného slova porušenú haldu, ktorú upraví aby spĺňala dané pravidlá. V koreni tak budeme mať najäčší prvok a ten vymeníme s prvkom na poslednom indexe poľa. Znova ustanovíme haldu, no teraz už len nad nezoradeným poľom. To je v našom prípade o jeden prvok kratšie, keďže najväčší prvok už je na svojom mieste. Tento cyklus opakujeme, až bude nezoradená časť poľa prázdna.

3.6 Tabuľka symbolov

Tabuľka symbolov je implementovaná za pomoci hashovacej tabuľky. Tabuľka je pripravená na prácu s poľami, avšak táto funkcionalita nebola zatiaľ implementovaná v zbytku programu. Hashovacia tabuľka pozostáva z poľa ukazateľov na zoznam položiek. Veľkosť thoto poľa je nastavená na 193. Pri takomto veľkom poli sú pamäťové nároky únosné a rýchlosť je pre naše účely dostatočná.

Každá položka pozostáva zo svojho unikátneho kľúča reprezentovaného štruktúrou `string` (respektíve jej položkou str). Ďalej obashuje záznamy špecifikujúce jej typ, stav (inicializovaná / neinicializovaná), počiatočný index, dĺžku a ukazateľ na dáta. Ukazateľ je typu void *, takže dochádza k pretypovaniu na typ, ktorý sa v danej položke nachádza. Nie je reprezentovaný pomocou `union`, práve kôli zamýšľanému rozšíreniu zahŕňajúcemu polia.

Do ukazateľa na dáta sa uloží vždy ukazateľ na dátový typ uvedený v položke. Pri funkciách a stringoch sa jedná o abstraktné dátové typy, pri ostatných o vstavané dátové typy jazyka C.

Miesto na dáta je alokované hňeď po pridaní záznamu do tabuľky. Predchádza sa tak možným chybám pri možnom zápise na adresu, ktorá nám nebola pridelená operačným systémom.

4 Testovanie

Jednotlivé časti projektu boli testované ako samostatne, ako počas vývoja jednotlivých modulov, tak aj v rámci funkčnosti celého projektu. Na testovaní sa podieľal každý člen týmu, no najväčšiu zásluhu majú členovia Matej Mužila, Tomáš Poremba a Marek Minda. Snažili sme sa zamerať na dodržanie

presnej špecifikácie jazyka IFJ14, ale i na čo najviac extrémnych hodnôt, výrazov a konštrukcií. V prípade akýchkoľvek nejasností prebiehala konzultácia v týme, alebo sme sa obrátili na fórum k projektu.

5 Práca v týme

Schôdzky v týme boli veľmi nepravidelné, za celý semester sme sa stretli všetci asi 3 krát. Čiastkové problémy sa diskutovali osobne medzi členmi týmu ktorý sa podieľali na danom module, alebo s ním súviseli. Komunikácia väčšinou prebiehala prostredníctvom sociálnej siete *Facebook*. Zdrojové kódy sme zdielali prostredníctvom *GitHub*-u, ktorý zároveň slúžil i ako diskusia k jednotlivým častiam implementácie.

Práca na našom projekte nás naučila ako ťažké je pracovať v týme, keď aj v menšom. Naučili sme sa ako sa sofistikovanejší projekt vyvíja od návrhu k implementácií, aké dôležité je testovanie a dodržanie presnej špecifikácie zadania. Nerozumná deľba práce mohla spôsobiť značné problémy na úspešnosti celého projektu, teda sme sa už na začiatku semestra snažili úlohy rozdeliť čo najrozumnejšie a najvyváženejšie medzi jednotlivými členmi týmu.

5.1 Rozdelenie práce

- **Miloš Volný** - syntaktická analýza, návrh LL gramatiky
- **Marek Minda** - syntaktická analýza, testovanie
- **Tomáš Poremba** - lexikálna analýza, interpret, algoritmy, dokumentácia, testovanie
- **Matej Mužila** - syntaktická a sémantická analýza, precedenčná analýza, tabuľka symbolov, dokumentácia, testovanie
- **Ladislav Šulák** - lexikálna analýza, testovanie, dokumentácia

6 Záver

Interpret imperatívneho jazyka IFJ14, ktorý je až na marginálne prípady podmnožinou jazyka Pascal sa nám úspešne podarilo implementovať. Autori sa inšpirovali vzorovým príkladom Jednoduchý interpret, ktorý bol dostupný na stránkach predmetu IFJ. Z tohto príkladu sme prevzali a značne upravili modul zaobstarávajúci prácu so zoznamom inštrukcií vnútorného kódu a modul zaobstarávajúci prácu s reťazcami (takmer) ľubovoľnej dĺžky.

Pri implementácií projektu sme si vyskúšali implementáciu rôznych algoritmov, abstraktných dátových typov, ale predovšetkým aj metódy formálnych jazykov v praxi a spoluprácu v menšom týme.

7 Použitá literatúra

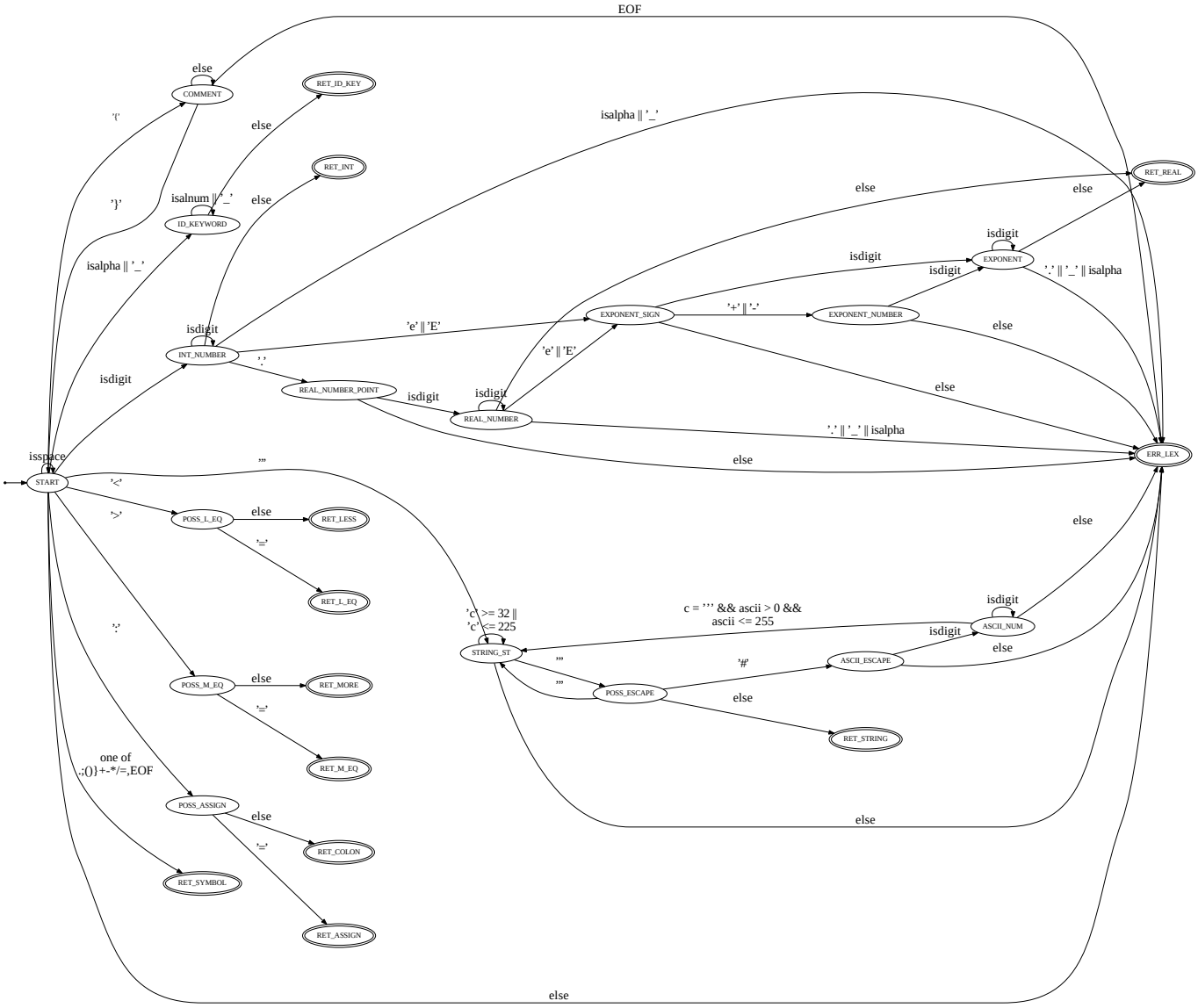
- Honzík, J. *Algoritmy IAL: Studijní opora*. 2014.
- Meduna, A., Lukáš R. *Formální jazyky a překladače IFJ: Studijní opora*. 2006. revízia 2009-2012.
- Lang H. W. *Knuth-Morris-Pratt algorithm* [online]. 2001-01. Dostupné na: http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/kmpen.htm.

A Metriky kódu

Počet súborov: 19
Počet riadkov zdrojového textu: 6998 + 54
Veľkosť spustiteľného súboru: 72426 B

B Prílohy

B.1 Konečný automat



B.2 LL-gramatika

```
PROGRAM - program v jazyku IFJ14
VARDEF - definícia premenných
FUNDEF - definícia užívateľských funkcií
FWD_FUNDEC - dopredná deklarácia funkcie
BODY - hlavne telo programu
STATEMENT_LIST - sekvencia čiastkových príkazov
EXPR - výraz
PARAM_LIST - zoznam parametrov pri užívateľských funckiách
id - identifikátor premennej
fun_id - identifikátor funkcie
operator - množina platných operátorov jazyka IFJ14
type - množina platných typov jazyka IFJ14 (real, boolean, integer, string)

PRAVIDLÁ
################################################################################

<VARDEF> -> var <ID> : <TYPE> ; <MOREVARDEF>
<VARDEF> -> ϵ
<MOREVARDEF> -> ϵ
<MOREVARDEF> -> <ID> : <TYPE> ; <MOREVARDEF>

################################################################################

<BODY> -> begin <STATEMENT_LIST> end.

################################################################################

<FUNDEF> -> <FWD_FUNDEC> <FUNDEF> <MOREFUNDEF>
<FUNDEF> -> function <FUN_ID> (<PARAM_LIST>) : <TYPE> ;
<VARDEF>
begin
<STATEMENT_LIST>
end;
<FUNDEF> -> ϵ
<MOREFUNDEF> -> <FUNDEF>
<MOREFUNDEF> -> ϵ

################################################################################

<PARAM_LIST> -> <PARAMS> <PARAM>
<PARAM_LIST> -> ϵ
<PARAMS> -> <PARAM>; <PARAMS>
<PARAMS> -> ϵ
<PARAM> -> <id> : <type>

################################################################################

<FWD_FUNDEC> -> function <FUN_ID> (<PARAM_LIST>) : <TYPE> ; forward ;
```

<FWD_FUNDEC> -> ϵ

##

<PROGRAM> -> <VARDEF>
<FUNDEF>
<BODY>

##

<STATEMENT_LIST> -> <STATEMENTS> <STATEMENT>
<STATEMENT_LIST> -> ϵ
<STATEMENTS> -> <STATEMENT>; <STATEMENTS>
<STATEMENTS> -> ϵ

##

<STATEMENT> -> if <EXPR> then <COMP_STATE> else <COMP_STATE>
<STATEMENT> -> <id> := <EXPR>
<STATEMENT> -> while <EXPR> do <COMP_STATE>
<STATEMENT> -> <id> := <fun_id> (<EXPR>)
<STATEMENT> -> readln(<id>)
<STATEMENT> -> write(<term_list>)
<STATEMENT> -> <COMP_STATE>

##

<COMP_STATE> -> begin <STATEMENT_LIST> end

##
<EXPR> -> <EXPR_PART> <EXPR2>
<EXPR2> -> <operator> <EXPR2>
<EXPR2> -> ϵ
<EXPR_PART> -> <fun_id>(<EXPR>)
<EXPR_PART> -> <constant>
<EXPR_PART> -> <id>

B.3 Precedenčná tabuľka

	+	−	*	/	<	>	=	<>	>=	<=	(	)
+	=	=	<	<	>	>	>	>	>	>	<	<
−	=	=	<	<	>	>	>	>	>	>	<	<
*	>	>	=	=	>	>	>	>	>	>	<	<
/	>	>	=	=	>	>	>	>	>	>	<	<
<	<	<	<	<	=	=	=	=	=	=	<	<
>	<	<	<	<	=	=	=	=	=	=	<	<
=	<	<	<	<	=	=	=	=	=	=	<	<
<>	<	<	<	<	=	=	=	=	=	=	<	<
>=	<	<	<	<	=	=	=	=	=	=	<	<
<=	<	<	<	<	=	=	=	=	=	=	<	<
(	>	>	>	>	>	>	>	>	>	>	=	=
)	>	>	>	>	>	>	>	>	>	>	=	=