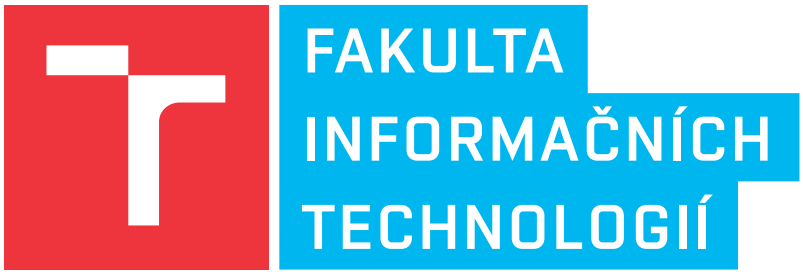
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Implementácia interpretu imperatívneho jazyka

IFJ16

Tím 036, variant a/2/II
Rozšírenia: SIMPLE, BOOLOP

Jakub Semrič
Petr Rusiňák
Kryštof Rykala
Martin Mikan
Martin Polakovič

11. prosince 2016

Obsah

1 Úvod

Tento dokument popisuje návrh a implementáciu interpretu imperatívneho jazyka *IFJ16*, ktorý je podmnožinou jazyka *JAVA SE 8*. Program funguje ako konzolová aplikácia, ktorá načíta zdrojový súbor programu v jazyku IFJ16 a následne ho interpretuje. V prípade výskytu chyby, program vracia kód chyby.

Dokument sa skladá z viacerých častí. V kapitole 2 je stručne popísaný vývoj projektu, v časti 3 je popísané riešenie jednotlivých častí interpretu, v kapitole 4 sa venujeme použitým algoritmom a v nasledujúcej kapitole 5 dátovým štruktúram. Kapitola 6 dokumentuje vypracované rozšírenia, kapitola 7 rozoberá náš postup pri testovaní interpretu.

2 Práca v tíme

Keďže sa jednalo o tímový projekt, na ktorom pracoval tím o 5 členoch, bolo potrebné si určiť opatrenia a metódy spolupráce. Zaviedli sme konvenciu písania kódu, ktorá nám zabezpečila prehľadnosť a jednotnosť projektu. Taktiež sme zaviedli konvenciu písania komentárov, aby spĺňali normu pre vygenerovanie dokumentácie v nástroji *doxygen*. K spolupráci viacerých ľudí na jednom projekte sme využili vytvorenie vzdialeného privátneho repozitára na serveri *GitHub, Inc.*.

Tímovú komuníkaciu a kolaboráciu sme riešili osobnými stretnutiami, ktoré boli približne raz za tri týždňe. Stanovili sme si ciele, ktoré museli byť do najbližšieho stretnutia dosiahnuté.

2.1 Rozdelenie úloh

Implementovaná časť	**Člen týmu**
Lexikálny analyzátor	Kryštof Rykala
Parser	Jakub Semrič
Precedenčná analýza	Peter Rusiňák
Algoritmy, tabuľka symbolov	Martin Polakovič
Interpret	Martin Mikan

2.2 Priebeh vývoja

Pre nedostatok vedomostí týmu a nestabilitu návrhu, vývoj prebiehal v niekoľkých iteráciach. Na začiatku každej iterácie sa navrhovalo s primeranou mierou, v ďalšej fáze sa implementovalo a na konci iterácie sa zamýšľalo ako riešiť novo vzniknuté problémy.

3 Popis riešenia a implementácia jazyka IFJ16

Projekt sme implementovali z niekoľkých funkčných celkov. V nasledujúcich podkapitolách je popis jednotlivých modulov.

3.1 Lexikálny analyzátor

Lexikálny analyzátor tvorí vstupnú časť interpretu. Model tvorí deterministický konečný automat. Spracováva zdrojový súbor a identifikuje v ňom platné lexémy, ktoré posiela syntaktickému analyzátoru vo forme tokenov. Token obsahuje typ lexémy a v prípade aj hodnotu, ak sa jedná o celočíselný, desatinný alebo reťazcový literál.

Konečný automat je implementovaný tak, aby vedel spracovať aj jednoduché (`//`) a blokové (`/* */`) komentáre. Po ukončení komentáru automat prejde do začínajúceho stavu a začne spracovávať nasledujúcu lexému alebo ďalší komentár.

Model konečného automatu môžete nájsť v prílohe 9.1.

3.2 Syntaktický a sémantický analyzátor

Syntaktický analýzátor (parser) je hlavná časť interpretu. Parser dostáva riadenie hneď po otvorení súboru. Má za úlohu nielen preveriť syntax a sémantiku zdrojového súboru, ale taktiež vkladá inštrukcie do inštrukčnej pásky. Jeho implementácia spočíva v rekurzívnom zostupe, ktorý je riadený LL gramatikou znázorňenou v prílohe 9.2.

Keďže jazyk `IFJ16` je podmnožinou jazyka *JAVA SE 8*, ktorého preklad je zložený z dvoch priechodov, je parser zložený taktiež z dvoch priechodov. Prvý priechod analyzuje program zo syntaktického a lexikálneho hľadiska, súčasne vkladá do tabuľky symbolov názvy tried, statických funkcií a premmenných, a kontroluje pokus o redefiníciu. Druhý priechod kontroluje semántiku programu, spracováva telá funkcií a vytvára inštrukčnú pásku.

3.2.1 Spracovanie jazykových inštrukcií – rekurzívny zostup

Rekurzívny zostup riadený LL(1) gramatikou, bol implementovaný tak, že každý neterminál v gramatike bol reprezentovaný funkciou, z ktorej sa postupne volali ďalšie funkcie na základe aktuálneho tokenu a pravidla v gramatike. Aktuálne načítaný token zase reprezentoval terminálne symboly gramatiky.

Druhý priechod je taktiež založený na rekurzívnom zostupe, avšak úloha jednotlivých funkcií sa od prvého priechodu značne odlišuje. Pri inicializácii statických premenných sa výraz za priradením (=) vyhodnotí precedenčnou analýzou a vložia sa inštrukcie do hlavného inštrukčného listu. Pri analýze funkcií sa postupne vložia ďalšie inštrukcie do inštrukčného listu aktuálne spracovávanej funkcie a v prípade vyhodnotenia výrazu sa predá riadenie precedenčnej analýze.

V prípade volania funkcie sa skontroluje existencia funkcie v členskej triede, počet a typ parametrov a v prípade priradenia návratový typ funkcie a premennej, do ktorej je volanie priradené. Do inštrukčného listu sa vloží inštrukcia `CALL`, ktorá uloží na hlavný zásobnik postupne odkaz na parametre volanej funkcie a inštrukcia `LAB` iba určí návratový bod, ktorý bude použitý pri skoku z volanej funkcie.

Pokiaľ nebola nájdená syntaktická alebo sémantická chyba, po narazení na token typu koniec súboru sa v tabuľke symbolov nájde trieda *Main* a funkcia *run()* zapuzdrená v tejto triede. Potom sa spojí hlavný inštrukčný list s inštrukčným listom funkcie *run()*. Pokiaľ požadovane entity boli nájdené analýza končí a riadenie je predané interpretu.

3.2.2 Precedenčná analýza výrazov

Pre spracovanie výrazov sme použili precedenčnú syntaktickú analýzu zdola nahor, riadenú vlastnou gramatikou. Parser načíta postupnosť tokenov, ktoré by mali byť súčasťou výrazu a predá riadenie precedenčnej analýze. Jej výstupom postupnosť inštrukcií, ktoré vyhodnotia tento výraz.

Precedenčná analýza je implementovaná pomocou precedenčnej tabuľky (viz príloha 9.3) a pomocného zásobníku pre ukladanie výrazov a operátorov.

Precedenčná tabulka je implementována ako statické pole, kde index prvku je vyjadrený vzťahom $[row * width + col]$. Typy tokenov reprezentujúce premenné alebo konštatnty zdieľajú jeden stĺpec.

3.3 Interpret vnútorného kódu

Interpret postupne vykonáva inštrukcie v inštrukčnom liste až kým nenarazí na chybu (napr. práca s neinicializovanou premennou) alebo na koniec listu. Každá inštrukcia má osobitnú implementáciu. Interpret aktívne využíva hlavný zásobník a zásobník rámcov. Lokálne premenné, s ktorými má pracovať sú v tabuľke, ktorá je na vrchole zásobníku rámcov. Statické premenné a konštanty sú spracovávané priamo bez vyhľadania v tabuľke.

3.3.1 Volanie a návrat z funkcií

Pri inštrukcii volania funkcie `CALL` sa podľa prvej adresy (3-adresný kód inštrukcie) zistí o akú funkciu sa jedná a koľko parametrov je na zásobníku. Vytvorí sa nový rámec s lokálnou tabuľkou skopírovanou z lokálnej

tabuľky volanej funkcie a súčasne sa naplnia parametre hodnotami premenných na zásobníku. Rámec sa uloží na vrchol zásobníku rámcov. Nasleduje skok na prvú položku inštrukčného listu danej funkcie.

Pri inštrukcii `RET` (príkaz `return`) nasleduje skok na miesto odkiaľ bola funkcia volaná, takže na inštrukciu za inštrukciou `CALL` v predošlom liste. Miesto reprezentované inštrukciou `LAB`[1] je uložené na hlavný zásobník aby sa vedelo, ktorou inštrukciou sa má pokračovať. Návratová hodnota funkcie je uložená v špeciálnych premenných uložených v globálnej tabuľke symbolov. Jedná sa o emuláciu registrov, ktoré zjednodušujú implementáciu celého interpretu.

4 Použité algoritmy

V tejto kapitole pojednávame o popise implementovaný algoritmov potrebným pri tvorbe projektu. Na ich pochopenie a implementáciu bola použitá študijná opora z predmetu IAL (prof. Honzík).

4.1 Radiaci algoritmus – Heap Sort

Heapsort, radenie haldou, patrí medzi základné algoritmy radenia. Heap – hromada je najčastejšie štruktúra stromového typu, inak tomu nie je ani v našom prípade. Pre všetky uzly tohto stromu platí rovnaký vzťah, otcovský uzol je vždy buď vačší alebo menší ako jeho synovské uzly. Hromada je implementovaná polom o veľkosti N, kde uzol umiestnený na indexe i má svojho ľavého syna na indexe $2i + 1 - base$[2] a pravého na $2i + 2 - base$.

Dôležitou časťou algoritmu je pomocná funkcia *sift down* zabezpečujúca rekonštrukciu hromady. Ak nastane porušenie hromady v koreni, táto funkcia postupnými výmenami dostane prvok z koreňa na správne miesto a do koreňa uloží prvok tak, aby bolo splnené pravidlo hromady. Opakovaným volaním funkcie sift down sa bude v koreni nachádzať prvok s najväčšou hodnotou a tohto faktu potom využívame pri samotnom radení. Odobratý prvok z koreňa sa vždy nahradí najmenším a najpravejším uzlom hromady.

Táto radiaca metóda má linearitmickú zložitosť.

4.2 Vyhľadávací algoritmus – Knuth-Morris-Pratt

Jedná sa o algoritmus, využívaný funkciou *find*. Slúži na vyhľadanie vzoru v reťazci. K svojej činnosti využíva konečný automat, ktorý je reprezentovaný vektorom prvkov typu *int*, kde index prvku sa zhoduje s indexom znaku vo vzore a hodnota prvku reprezentuje cieľový index, na ktorý sa má vrátiť pri nezhode porovnania.

Vyhľadávanie je implementované ako jedna funkcia, ktorej parametrami je ukazovateľ na prehľadávaný reťazec a ukazovateľ na vzor. Na začiatku funkcie sa vytvorí vyššie spomínaný vektor `fail_vector`. Následne algoritmus sekvenčne prechádza prehľadávaný reťazec a vzor a porovnáva ich znaky. Pri nezhode sa index vzoru nastaví na hodnotu, ktorej odpovedá hodnota vo vektore na aktuálnom indexe. Porovnávanie prebieha, pokiaľ index reťazca a vzoru nepresiahne index ich posledných znakov. Návratovou hodnotou je index prvého vyhľadania vzoru v reťazci. V prípade neúspechu -1.

Výhodou algoritmu je, že sa nevracia späť v prehľadávanom reťazci. Nemusí porovnávať už porovnané prvky. Časová zložitosť algoritmu je $m + n$.

5 Využívané dátové štruktúry

5.1 Tabuľka s rozptýlenými položkami

Tabuľka s rozptýlenými položkami je implementovaná ako pole jednosmerne viazaných zoznamov symbolov. Symbol obsahuje mimo identifikátora aj ukazateľ na členskú triedu. Vyhľadávanie spočíva v nájdení symbolu

[1] Label – inštrukcia reprezentujúca návestie

[2] index prvej položky poľa

s rovnakým identifikátorom a súčasne rovnakou členskou triedou. Toto vyhľadávanie zabráňuje kolíziam symbolov rovnakých mien v rôznych úrovniach. Kedže lokálne premenné nemajú členskú triedu existuje varianta vyhľadávaní v tabuľke iba s jedným kľúčom.

V interprete sa používaju dva typy tabuliek: lokálna a globálna. V globálnej tabuľke sú uložené symboly reprezentujúce triedy, funkcie, konštanty, statické premenné a registre. V lokálnej tabuľke, sú uložené iba lokálne premenné a parametre funkcií, ktoré majú vlastnú tabuľku symbolov.

5.2 Zásobník

Zásobník je implementovaný ako pole ukazateľov určitej kapacity. Ak je zásobník plný dôjde k automatickej realokácii a zvýšeniu kapacity zásobníku. V interprete sa používaju tri zásobníky. Jeden z nich sa používa na uloženie odkazov na miesta skokov a odkazy na parametre funkcií. Druhý zásobník je použitý na ukladanie rámcov, ktoré obsahujú lokálnu tabuľku symbolov. Posledný zásobník používa analýza výrazov, ktorá si naň ukladá operandy a operátory.

6 Rozšírenia

6.1 SIMPLE

Rozšírenie *SIMPLE* umožňuje skrátený zápis konštrukcií cyklov a podmienok vynechaním zložených zátvoriek. Keďže by terminálne symboly { } moholi byť vynechané, jednalo by sa už o kontextovú gramatiku. Tento problém bol vyriešený implementačne bez zmeny gramatiky tak, že sa v prípade absencie zložených zátvoriek namiesto volania funkcie reprezentujúcej neterminál `STLIST`, vložila funkcia reprezentujúca neterminál `STAT`, čiže sa namiesto zloženého príkazu vykonal jednoduchý príkaz.

6.2 BOOLOP

Rozšírenie umožňuje poporovať nový dátový typ – *boolean*. Interpret podporuje deklarácie premenných a funkcií typu boolean. Navyše umožnuje vyhodnocovanie zložených podmienok.

7 Testovanie

Návrh testov bol založený na metóde ekvivalenčných tried. Testy mali charakter regresných testov, takže zmena zdrojových súborov nebola možná ak všetky testy neboli úspešné. Testy sa delili do dvoch skupín – testy detekcie chýb a interpretačné testy. Pri testovaní bol použitý program `gcovr` od GCC, ktorý nám ukázal aké časti interpretu neboli otestované.

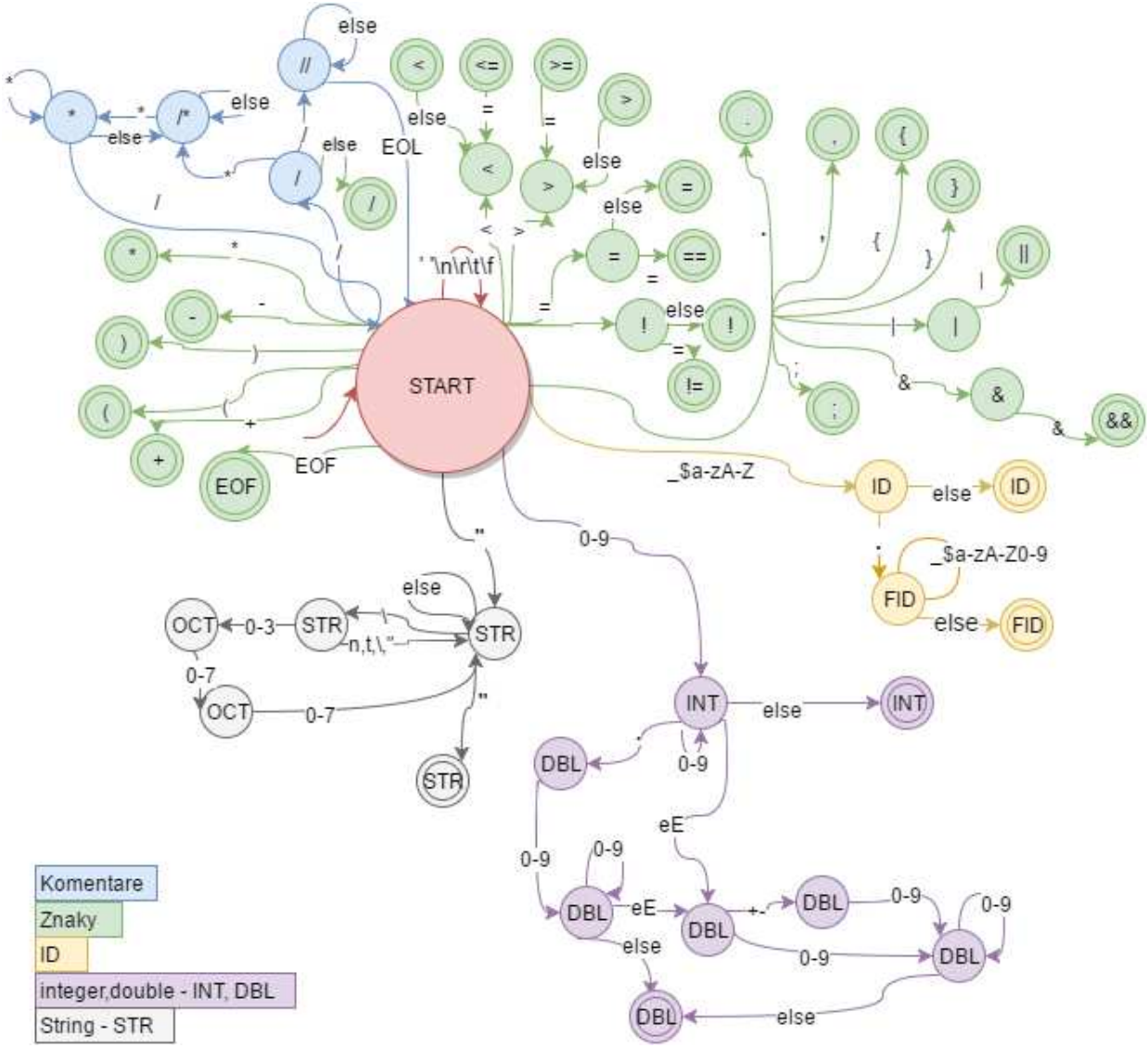
8 Použité zdroje

GCC. gcovr. Dostupné z: http://gcovr.com/.

HONZÍK, J. M. ALGORITMY IAL: Studijní opora. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php?file=%2Fcourse%2FIAL-IT%2Ftexts%2FOpora-IAL-2016-verze-16+D.pdf&cid=11418.

9 Prílohy

9.1 Konečný automat



9.2 LL(1) gramatika

PROG	→	BODY eof
BODY	→	CLASS BODY
BODY	→	ϵ
CLASS	→	class id { CBODY }
CBODY	→	static TYPE id CBODY2 CBODY
CBODY	→	ϵ
CBODY2	→	INIT ;
CBODY2	→	FUNC
FUNC	→	(PAR) FBODY
FBODY	→	{ STLIST }
PAR	→	TYPE PAR2
PAR	→	ϵ
PAR2	→	ϵ
PAR2	→	id PAR3
PAR3	→	, TYPE id PAR3
PAR3	→	ϵ
STLIST	→	STAT STLIST
STLIST	→	ϵ
STAT	→	while (EXPR) { STLIST }
STAT	→	if (EXPR) { STLIST } ELSE
STAT	→	return RET ;
STAT	→	id = CALLEXPR ;
STAT	→	fullid = CALLEXPR ;
STAT	→	static TYPE id INIT ;
STAT	→	TYPE id INIT ;
CALLEXPR	→	fullid (ARG)
CALLEXPR	→	id (ARG)
CALLEXPR	→	EXPR
ARG	→	ϵ
ARG	→	id ARG2
ARG	→	fullid ARG2
ARG2	→	ϵ
ARG2	→	, ARG3 ARG2
ARG3	→	fullid
ARG3	→	id
INIT	→	= CALLEXPR
INIT	→	ϵ
RET	→	EXPR
RET	→	ϵ
ELSE	→	ϵ
ELSE	→	else ELSE2
ELSE2	→	if (EXPR) { STLIST } ELSE
ELSE2	→	{ STLIST }
TYPE	→	void
TYPE	→	int
TYPE	→	string
TYPE	→	double
TYPE	→	boolean

9.3 Precedenčná tabuľka

	+	-	/	*	==	!=	<	>	<=	>=	&&	||	!	(	)	i	$
+	>	>	<	<	>	>	>	>	>	>	>	>	<	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	>	>	<	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	>
==	<	<	<	<			<	<	<	<	>	>	<	<	>	<	>
!=	<	<	<	<			<	<	<	<	>	>	<	<	>	<	>
<	<	<	<	<	>	>					>	>	<	<	>	<	>
>	<	<	<	<	>	>					>	>	<	<	>	<	>
<=	<	<	<	<	>	>					>	>	<	<	>	<	>
>=	<	<	<	<	>	>					>	>	<	<	>	<	>
&&	<	<	<	<	<	<	<	<	<	<	>	>	<	<	>	<	>
||	<	<	<	<	<	<	<	<	<	<	<	>	<	<	>	<	>
!	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>	>	>		>		>
i	>	>	>	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<	