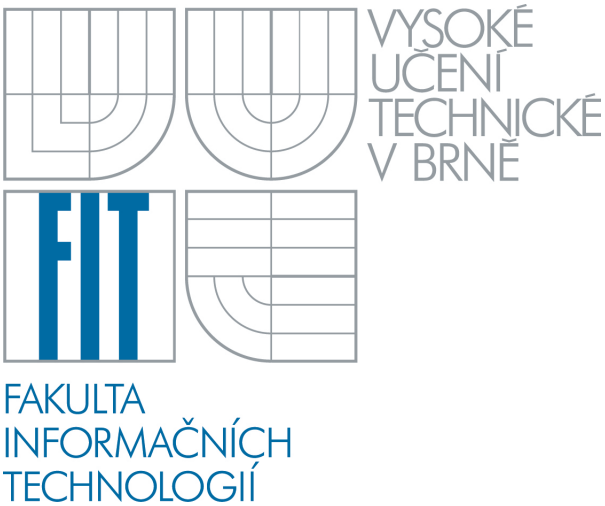
Vysoké učení technické v Brně
Fakulta informačních technológií

Implementácia interpretu imperatívneho jazyka

IFJ16

Tým 068, varianta a/2/II



Meno	Login	Rozdelenie bodov
Tomáš Strych (ved.)	xstryc05	20
Radoslav Pitoňák	xpiton00	20
Andrej Dzilský	xdzils00	20
Milan Procházka	xproch91	20
Ondřej Břínek	xbrine03	20

11. prosince 2016

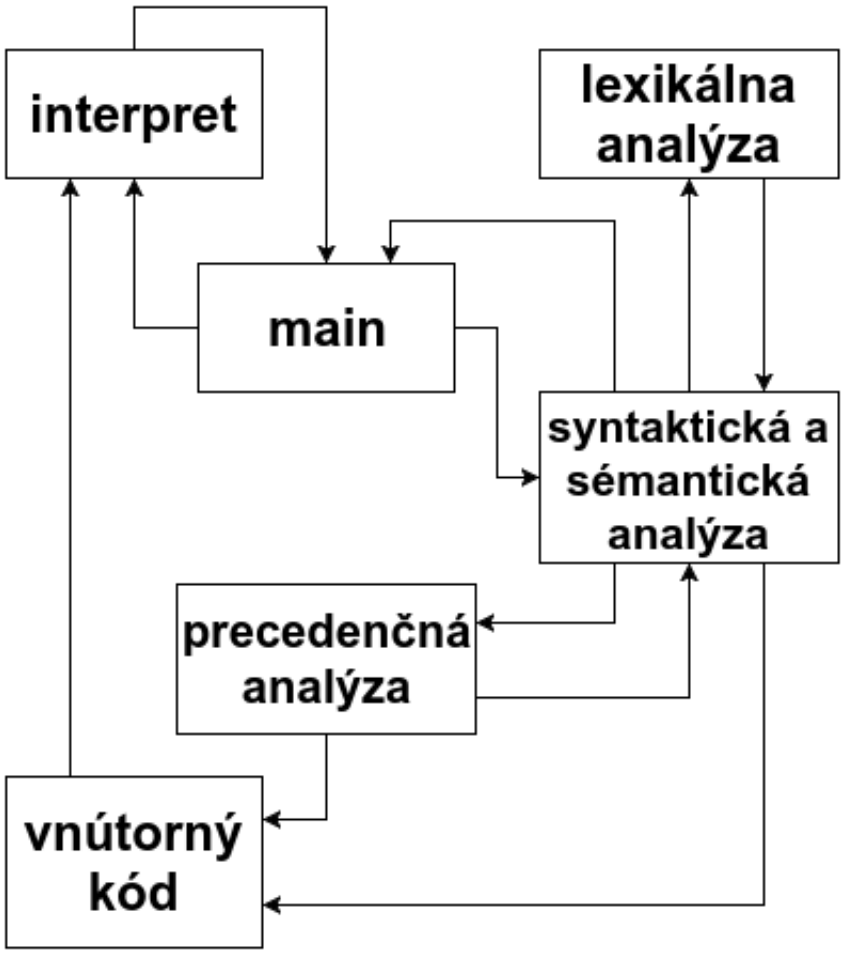
Obsah

1 Úvod

Tento dokument slúži ako dokumentácia ktorá popisuje návrh a implementáciu imperatívneho jazyka IFJ16. Dokument obsahuje spôsob riešenia z pohľadu IFJ (návrh,implementácia,spôsob práce v tíme) a z pohľadu IAL (popis riešení algoritmov a dátovej štruktúry)

Jazyk IFJ16 je veľmi zjednodušenou podmnožinou jazyka Java SE 8, čo je staticky typovaný objektovo orientovaný jazyk. Program načíta zdrojový súbor zapísaný v jazyku IFJ16 a interpretuje ho. Ak prebehne činnosť interpretu bez chýb, vracia sa návratová hodnota 0 (nula). Ak došlo k nejakej chybe,vracia sa kód detekovanej chyby.

2 Postup pri riešení



Projekt sme implementovali z niekoľkých funkčných častí ktoré môžete vidieť na obrázku.
Main - hlavný program ktorý spracováva argumenty príkazového riadku. Z hlavného programu sa volá modul parser a interpret ktorých úlohou je preložiť a previesť vstupný program. V nasledujúcich kapitolách a podkapitolách sú komplexnejšie popísané jednotlivé moduly.

2.1 Lexikálny analyzátor

Úlohou lexikálneho analyzátora je spracovať vstupný súbor po lexikálnej stránke. Na základe lexikálnych pravidiel jazyka IFJ16, sú časti súboru rozdelené na lexémy. Základom lexikálnej analýzi je konečný deterministický automat, ktorý precháza vstupný súbor a rozpoznáva jednotlivé lexémy. viz. 5

Nastavením vstupného súboru je lexikálny analyzátor pripravený k používaniu. Na žiadosť parsera zašle token predstavujúci aktuálny lexém. Token je základným komunikačným prostriedkom lexikálnej analýzi. Token v sebe uchováva informácie o druhu lexému a zároveň jeho obsah, riadok kde bol načítaný, a veľkosť načítaných dát. Token zároveň môže signalizovať lexikálnu chybu i s riadkom výskytu. Ďalšia úloha lexikálneho analyzátora je expanzia escape sekvencie, kedy znaky so špeciálnym významom musia byť uvedené znakom "\"a expanzia je prevedená na jeden znak ktorý reprezentuje. Lexikálny analyzátor pracuje v dvoch režimoch. Počas prvého režimu "saving"je každý vyžiadaný token uložený do vnútornej pamäte typu FIFO. Pri vypnutom režime "saving"je token vrátený z vnútornej pamäte, pokiaľ sa tam nejaký nacháza, pokiaľ nie načíta sa zo vstupného súboru. Implicitne je saving vypnutý.

2.2 Syntaktický a sémantický analyzátor

Syntaktický analyzátor je hlavná časť celého interpretu. Jeho úlohou je skontrolovať správnosť syntaxu, ale aj sémantiky zdrojového súboru a tiež vygenerovať inštrukcie do inštrukčnej pásky. Je implementovaný pomocou rekurzívneho zostupu, ktorý je riadený LL gramatikou. viz. 6

V jazyku IFJ16 je preklad zložený z dvoch prechodov. V prvom prechode sa kontroluje správnosť syntaxu a zároveň ukladá do tabuľky symbolov triedy, statické premenné a funkcie prislúchajúce daným triedam. Druhý prechod kontroluje sémantickú správnosť zdrojového kódu a generuje inštrukcie na inštrukčnú pásku.

2.2.1 Rekurzívny zostup

Rekurzívny zostup je volaný na začiatku vo funkcii main. Parser si počas rekurzívneho zostupu postupne volá lexikálny analyzátor, ktorý mu vráti načítaný lexém.

V druhej fáze prechodu pri priradeniach do statických premenných sa inštrukcie generujú na globálnu pásku, ktorá sa odinterpretuje ako prvá. Pri analýze funkcii sa vložia inštrukcie do inštrukčnej pásky aktuálne spracovávanej funkcie. V prípade vyhodnotenia výrazov sa predáva riadenie precedenčnej analýze spolu s odkazom na aktuálnu inštrukčnú pásku.

V prípade volania funkcie sa skontroluje či daná funkcia existuje v tabuľke symbolov aktuálnej triedy, ak je použitý plne kvalifikovaný prístup tak skontrolujeme či daná funkcia existuje v tabuľke symbolov triedy určenej kvalifikovaným prístupom. Následne sa na pomocnú inštrukčnú pásku vygenerujú inštrukcie pre nastavenie parametrov danej funkcie a pomocou inštrukcie In_FUN_CALL sa vygeneruje inštrukcia na volanie funkcie, ktorá dostane túto pomocnú inštrukčnú pásku ako parameter. Okrem toho dostane aj odkaz na inštrukčnú pásku danej funkcie.

Ak pri rekurzívnom zostupe ani pri jednom z prechodov nebola nájdená syntaktická alebo sémantická chyba, tak skontrolujeme či sa v tabuľke symbolov nachádza trieda main s bezparametrickou void funkciou run. Pokial áno, riadenie sa predáva modulu interpret.

2.2.2 Precedenčná analýza výrazov

Precedenčná analýza výrazov je dôležitou súčasťou parsera,ktorá je volaná pri každom vyhodnotení výrazu. Pri vyhodnocovaní výrazov sme použili metódu precedenčnej analýzi zdola nahor. Na začiatku sme navrhli precedenčnú tabuľku viz. 7, pomocou ktorej sme schopný určovať prednosť operátorov vo výraze. Ďalej sme si navrhli zoznam pravidiel, vďaka ktorým je naša precedenčná analýza schopná vygenerovať inštrukcie nad binárnymi operáciami, ktoré sa vkladajú na inštrukčnú pásku. Informáciu o tom na ktorú inštrukčnú pásku sa generujú inštrukcie určuje parser. Pri generovaní inštrukcii sa aplikujú pravidlá, ktoré kontrolujú správnosť dátových typov u jednotlivých binárnych operácii. Po vyhodnotení výrazu sa v globálnej premennej nachádza odkaz s informáciami o výsledku vyhodnotenia výrazu. Treba podotknúť, že hodnotu tohoto výsledku na tento odkaz zapíše až interpret.

2.3 Modul interpret

Modul interpret je v podstate finálny modul celého programu. V základe prijíma obojsmerne viazaný zoznam, vnútorný kód, ktorý obsahuje postupnosť inštrukcii, vygenerovaný parserom. Tento zoznam potom prechádza a vykonáva jednu inštrukciu po druhej. Inštrukcie sú v takzvanom Trojadresonm kóde (*3AC, Triple Address Code*), teda každá inštrukcia obsahuje až tri operandy, určené adresou. Tieto operandy sú uložené v takzvanom *rámci* či v *tabuľke symbolov*. Princíp tabuľky symbolov je bližšie popísaný v kapitole 3.3. Rámec je v podstate len zjednodušená verzia tabuľky symbolov, obsahujúci iba premenné dôležité pre danú funkciu. Predpis tohoto rámca je unikátny pre každú funkciu a pre každé volánie funkcie je vygenerovaná vlastná inštancia daného rámca, ktorá je potom naplnená aktuálnymi variantami premenných a konštánt.

3 Algoritmy a tabuľka s rozptýlenými položkami

V každom zložitejšom projekte je nevyhnutné použiť rôzne dátové štruktúry a algoritmy. Táto kapitola predstaví práve niektoré z nich. Popisu implementácie algoritmov a dátovej štruktúry k predmetu IAL sú venované samostatné podkapitoly.

3.1 Heap sort

Heap sort alebo radenie haldou je jeden z najlepších obecných algoritmov radenia, založený na porovnávaní prvkov. Jeho časová náročnosť je O(N log N). Heapsort nie je stabilne radiaci algoritmus. Základnou myšlienkou je využitie haldy,hromady čo je stromová štruktúra najčastejsie binárna hromada založená na binárnom strome. Prvý krok algoritmu spočíva zostavení haldy. V druhom kroku vymeníme koreňový uzol za posledný prvok poľa. Tým je zaradený na správne miesto je vyreadený z poľa, ale porušila sa nám halda. Takže ju potrebujeme znovu zostaviť. Potom čo je halda znovy zostavená,postup sa opakuje pre stále sa zmenšujúcu haldu.

3.2 Knuth-Moris-Prattov algoritmus

Algoritmus slúži na vyhľadanie vzoru v reťazci. Tento algoritmus vychádza z myšlienky, že je zbytočné vracať sa pri neúspešnom porovnaní v reťazci späť a porovnávať znova znaky, ktoré už porovnané boli. Časová zložitosť algoritmu je O(m + n). Ku svojej činnosti využíva konečný automat reprezentovaný vektorom,kde index prvku sa zhoduje s indexom znaku vo vzore a hodnota prvku reprezentuje cieľový index, na ktorý sa má vrátiť pri nezhode porovnania. Funkcia na vstupe príma reťazec a vzor a porovnáva ich znaky. Na začiatku sa vytvorí vektor a algoritmus prechádza reťazec a porovnáva znaky. Pri nezhode sa index vzoru nastaví na hodnotu ktorej odpoveda hodnota vo vektore na aktuálnom indexe. Funkcia vracia -1 v prípade neúspechu a v prípade úspechu vracia pozíciu nájdeneho prvku počítanú od nuly.

3.3 Tabuľka s rozptýlenými položkami

Tabuľku symbolov sme implementovali ako hašovaciu tabuľku(tabuľku s rozptýlenými položkami) s explicitne zreťazenými synonymami. V projekte sme použili globálnu tabuľku a lokálne tabuľky. V globálnej tabuľke sú uložené triedy a konštanty. Každá funkcia a trieda má svoju vlastnú lokálnu tabuľku symbolov v ktorej sú uložene návratové hodnoty, lokálne premenné, a parametre funkcie. Lokálna a globálna tabuľka majú rovnakú štruktúru. Záznam tabuľky obsahuje kľúč, dáta a ukazovateľ na další záznam. Dáta majú svoj typ a svoju hodnotu, na základe typu vieme rozpoznať ktoré dáta v štruktúre typu union vybrať. Štruktúra union sa skladá z funkcie, premennej, konštanty a triedy.

Pre mapovanie položiek sme použili jednoduchú a rovnakú hašovaciu funkciu ako v predmete IAL.

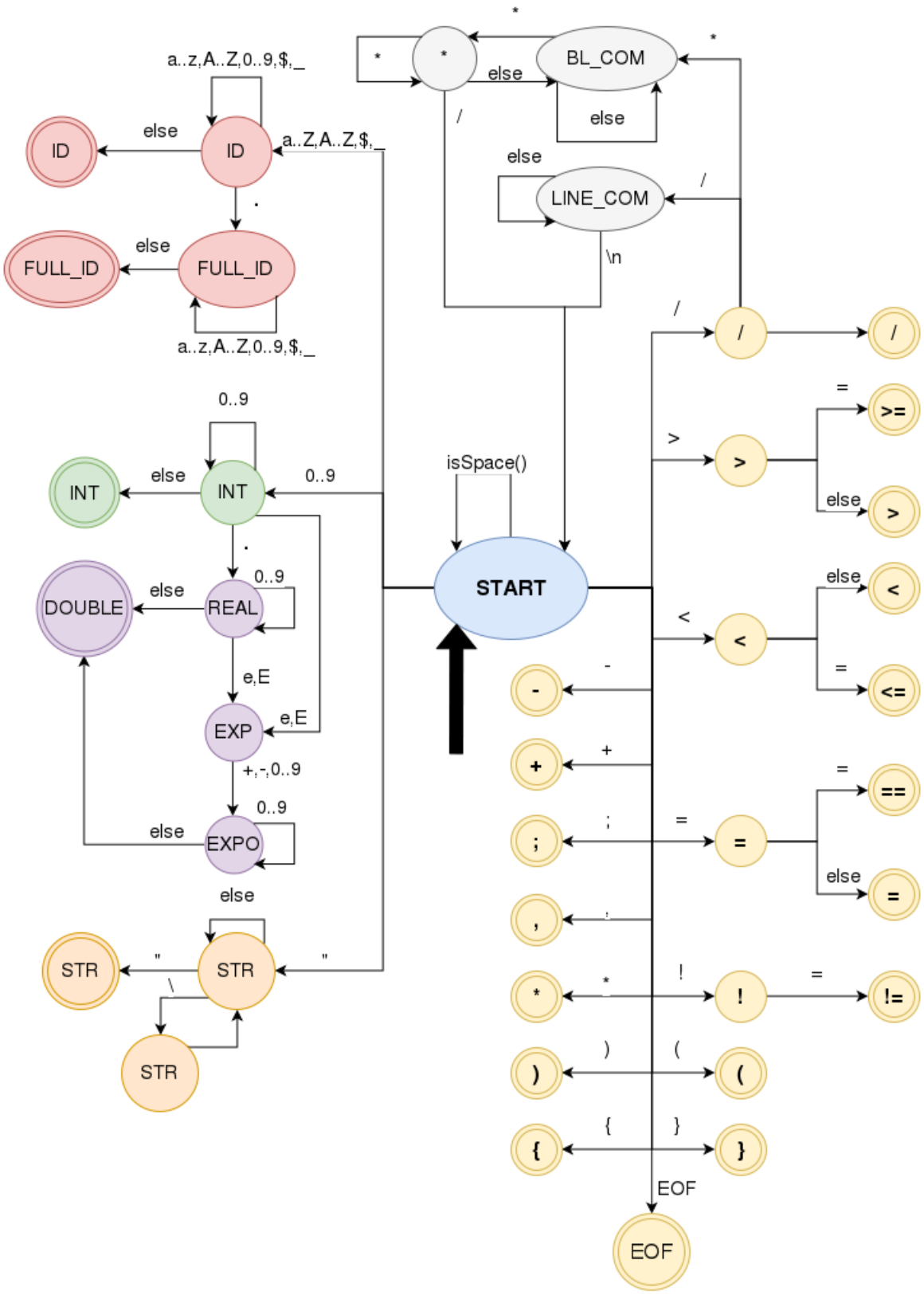
4 Práca v tíme

Na začiatku sme si rozdelili úlohy, rozhodli sme sa že budeme pracovať s verziovacím systémom git, na privátnom repozitáry na webovej službe *bitbucket.org*. Počas vývoja sme sa pravidelne stretávali, diskutovali o jednotlivých fázach implementácie aby sme predišli možným chybám. Na komunikáciu mimo stretnutí sme používaly privátny kanál na *slack.com* ktorý slúži na posielanie správ v týme.

4.1 Rozdelenie úloh

Tomáš Strych - Algoritmy a tabuľka s rozptýlenými položkami
Radoslav Pitoňák - syntaktická analýza
Andrej Dzilský - precedenčná analýza
Milan Procházka - lexikálna analýza
Ondřej Břínek - modul interpret
Každý z nás sa podielal na písaní dokumentácie a testovaní.

5 Konečný automat



6 LL-gramatika

<file>	→	<class_list> EOF
<class_list>	→	<class> <class_list>
<class_list>	→	ϵ
<class>	→	class id { <def_list> }
<def_list>	→	<def_prefix> <def> <def_list>
<def_list>	→	ϵ
<def_prefix>	→	static <type> id
<def>	→	(<param_list>) { <stat_list> }
<def>	→	<init> ;
<init>	→	= <term>
<init>	→	ϵ
<param_list>	→	ϵ
<param_list>	→	<param> <param_list2>
<param_list2>	→	ϵ
<param_list2>	→	, <param> <param_list2>
<param>	→	<type> id
<call_param_list>	→	ϵ
<call_param_list>	→	<term> <call_param_list2>
<call_param_list2>	→	ϵ
<call_param_list2>	→	, <term> <call_param_list2>
<stat_list>	→	<stat> <stat_list>
<stat_list>	→	ϵ
<complex_stat>	→	{ <stat_list> }
<stat>	→	<id> <stat2>
<stat2>	→	= <stat3>
<stat2>	→	(<call_param_list>) ;
<stat3>	→	<term> ;
<stat3>	→	<id> (<call_param_list>) ;
<stat>	→	<type> id <init> ;
<stat>	→	return <term> ;
<stat>	→	if (<term>) <complex_stat> else <complex_stat>
<stat>	→	while (<term>) <complex_stat>
<stat>	→	;
<term>	→	int_term
<term>	→	double_term
<term>	→	string_term
<term>	→	<id>
<term>	→	ϵ
<id>	→	full-id
<id>	→	id
<type>	→	double
<type>	→	int
<type>	→	String
<type>	→	void

7 Precedenčná tabuľka

	(	)	+	-	*	/	<	>	<=	>=	==	!=	$	Identifier	Int,Double	String
(	<	=	<	<	<	<	<	<	<	<	<	<		<	<	<
)		>	>	>	>	>	>	>	>	>	>	>	>			
+	<	>	>	>	<	<	>	>	>	>	>	>	>	<	<	<
-	<	>	>	>	<	<	>	>	>	>	>	>	>	<	<	<
*	<	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<
/	<	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<
<	<	>	<	<	<	<							>	<	<	<
>	<	>	<	<	<	<							>	<	<	<
<=	<	>	<	<	<	<							>	<	<	<
>=	<	>	<	<	<	<							>	<	<	<
==	<	>	<	<	<	<							>	<	<	<
!=	<	>	<	<	<	<							>	<	<	<
$	<		<	<	<	<	<	<	<	<	<	<		<	<	<
Identifier		>	>	>	>	>	>	>	>	>	>	>	>			
Int,Double		>	>	>	>	>	>	>	>	>	>	>	>			
String		>	>	>	>	>	>	>	>	>	>	>	>			