Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentácia k spoločnému projektu IFJ a IAL

Implementácia interpretu imperatívneho jazyka IFJ11

Tím 095, varianta b/4/I

Rozšírenia: MODULO, LOCALEXPR, REPEAT

Peter Michalík xmicha47 : 20 %
Michal Lukáč xlukac05 : 20 %
Milan Seitler xseitl01 : 20 %
Jakub Sznapka xsznap01 : 20 %
Jan Hrivnák xhrivn01 : 20 %

11. decembra 2011

Obsah

1 Úvod

Tvorba prekladačov je názorným príkladom toho, keď sa naše teoretické vedomosti, zozbierané na prednáškach, stretávajú s praktickým využitím a premieňajú sa na výsledok s plnohodnotným uplatnením. Je potrebné využiť praktické skúsenosti potrebné pre implementáciu v cieľovom jazyku, ako aj teoretické vedomosti o formálnych jazykoch.

Táto dokumentácia popisuje tvorbu interpretu imperatívneho jazyka IFJ11. Zaoberá sa vývojovými metodikami použitými pri práci v tíme, implementačnými detailami, formálnymi modelmi a popisom častí vybraných algoritmov.

2 Práca v tíme

2.1 Vývojové metodiky

Pred začatím prác na projekte sme o žiadnej ucelenej vývojovej metodike nerozmýšľali. Po zadaní projektu sme sa začali stretávať raz do týždňa prevažne v priestoroch seminárnej miestosti knižnice FIT VUT. Rozhodli sme sa, že budeme určite potrebovať nejakú agilnú vývojovú metodiku, ktorá sa využíva práve v malých tímoch. Hlavnou výhodou je flexibilnosť tímu a tým pádom aj lepšie prispôsobenie sa rôznym zmenám v zadaní. To bolo výhodou najmä v prípadoch kedy sme nejakú časť zadania nepochopili správne a neskôr sme, na základe upresnenia, boli nútení spraviť určité zmeny.

Základným kameňom agilnej metodiky je komunikácia. Hlavné problémy sa riešili na tímových schôdzkach, drobnosti na tímovom chate, poprípade súkromnom chate medzi dvoma členmi tímu. Na každej schôdzke tímu sme preriešili nezrovnalosti, prípadne vzájomnú pomoc.

Ako sa chýlilo odovzdanie projektu, začala naša práca nadobúdať podobu jednej z agilných metodík a to extrémne programovanie. Postupovali sme malými zmenami a odladľovaním všetkých problémov, ktoré mohli nastať. Pracovali sme všetci spolu, zvyčajne v priestoroch CVT. Hlavným rysom bolo párové programovanie, pri ktorom sa nad určitým problémom zamýšľali dvaja ľudia.

2.2 Verzovací systém

Neoddeliteľnou súčasťou práce v tíme je verzovací systém pre udržovanie všetkých úprav a zmien v projekte. Rozhodovali sme sa medzi dvoma nástrojmi: git a svn. Nakoniec sme sa rozhodli pre svn. Hlavným dôvodom bola kompatibilita s operačným systémom Windows aj Linux, nakoľko sme ako tím neboli v tomto jednotní. Ďalším z dôvodov bola možnosť vytvorenia zdieľaného repozitára na školskom serveri, ktorú sme aj využili.

2.3 Rozdelenie úloh

K tímovému prístupu k práci súvisí aj správne prerozdelenie činnosti. Pri tomto projekte nebolo ľahké prerozdeliť prácu už na začiatku rovným dielom. Aj napriek tomu, sme si už v začiatkoch spravili akú takú predstavu o rozdelení. V tíme nás bolo päť. Tomuto počtu zodpovedalo nasledujúce rozdelenie: jeden človek pracoval na lexikálnej analýze, dvaja na parseri, jeden na interprete a vedúci tímu mal na starosti tímovú réžiu a časť interpretu. Takisto bolo potrebné prerozdeliť ešte ostatnú prácu, ktorú bolo treba spraviť. Popri tom ako sa vyvíjala lexikálna analýza, ostatní pracovali na implementácii merge sortu, Boyer-Mooreovho algoritmu a binárneho vyhľadávacieho stromu. Ku koncu sa zasa všetky voľné zdroje sústredili na testovanie, písanie dokumentácie a prípravu prezentácie.

3 Dekompozícia a návrh

Už na druhom stretnutí sme začali celú problematiku rozčlenovať na jednotlivé časti. Spočiatku sme od hlavných problémov prekladača odčlenili pomocné mechanizmy. Začali sme s implementáciou vstavaných funkcií jazyka, ktorými boli radiaci algoritmus, v našom prípade merge sort a Boyer-Mooreov alogritmus na vyhľadávanie podreťazca v reťazci. Pokračovali sme implementáciou tabuľky symbolov, ktorá bola u nás reprezentovaná binárnym vyhľadávacím stromom. Pri práci sme využili zdrojové súbory `ilist.c` a `ilist.h` pre zoznam inštrukcií ako aj `str.c` a `str.h` pre špeciálny typ `string`.

Popri tom sme však nemohli odkladať ani prácu na samotnom prekladači. S každotýždňovým prílivom nových informácií sme boli schopní posunúť našu mieru pochopenia problému a tým aj hĺbku abstrakcie. Interpret sme rozdelili na tri hlavné časti:

- lexikálny analyzátor
- parser
- interpret

Lexikálny analyzátor je konečný automat, spracúvajúci lexémy a predávajúci ich vo forme tokenov parseru. Parser tvorí srdce nášho interpretu. Prijíma tokeny z lexikálneho analyzátora a spracúva ich. Skladá sa z dvoch častí: analýza na základe rekurzívneho zostupu a precedenčná analýza pre výrazy. Poslednou časťou je interpret, ktorý spracúva inštrukcie a vykonáva podľa nich zadané operácie.

Ďalšou z vecí, na ktorú sme nemohli zabudnúť, bola správna kompatibilita. Najhlavnejším bolo vytvorenie správnej inštrukčnej sady, ktorou koumikuje parser a interpret. Nemenej dôležitou tiež bola číselná reprezentácia tokenov, ktoré predáva lexikálny analyzátor.

4 Implementácia

4.1 Boyer-Mooreov alogoritmus

Boyer-Mooreov algoritmus je považovaný za efektívny vyhľadávací algoritmus pre prehľadávanie podreťazca v reťazci. Výhoda Boyer-Mooreovho algoritmu je prehľadávanie podreťazca od konca a následny posun o určitý index. Toto riešenie je efektívnejšie ako konvenčné riešenie prehľadávania znak po znaku. Naša implementácia Boyer-Moore algoritmu využíva prvú heuristiku zo skrípt do predmetu algoritmy. Algoritmus môžeme rozdeliť na dve fázy: prípravnú a prehľadávanie. Najprv si vytvoríme pole pre ASCII tabuľku. Toto pole do ktorého budeme pristupovať pomocou znakového literálu ohodnotíme veľkosťou reťazca. Následne pre znaky hľadaného podreťazca nastavíme hodnoty do pola podľa toho na akom indexe sa nachádzajú. Následne prehľadávame podreťazec s reťazcom. Porovnáme posledný znak podreťazca so znakom reťazca ktorý ma rovnaký index. Pokiaľ sa znaky zhodujú posunieme sa o jeden znak späť. Pokiaľ sa znaky nezhodujú posunieme sa o počet prvkov udávajúci tabuľka ktorú sme si na začiatku vypočítali.

4.2 Merge sort

Merge sort je radiaci algoritmus, ktorý využíva priamy prístup k prvkom poľa. Vyznačuje sa rýchlosťou, avšak, kvôli zložitej implementácii nepatrí medzi využívané algoritmy. Pracuje na základe hesla rozdeľuj a panuj. V našom prípade sme použili pomocné pole rovnakej veľkosti ako vstupné. Radenie má dve časti. Funkcia `mergesort()` rekurzívne rozdeľuje vstupné pole na čiastky najskôr zľava a potom sprava. Funkcia `merge()` potom zakladá tieto zoradené čiastky medzi sebou, aby vznikla zoradená čiastka väčšej úrovne. Takto sa jednotlivé čiastky postupne zaradzujú až vznikne zoradená postupnosť.

4.3 Binárny vyhľadávací strom

Tabuľku symbolov sme implementovali ako binárny vyhľadávací strom. Je to jedna z najpoužívanejších implementácii pre dynamické vyhľadávacie tabuľky. Strom je implementovaný nerekurzívne, kvôli rýchlosti prístupu. Jediná funkcia, ktorá je implementovaná rekurzívne je funkcia `destroyTree()`, pretože sa volá len raz, preto na jej rýchlosti až tak nezáleží.

4.4 Lexikálny analyzátor

Lexikálny analyzátor sme implementovali ako konečný automat, ktorý prechádza zadaný zdrojový kód v jazyku IFJ11 po jednotlivých znakoch a svoj výstup posiela syntaktickému analyzátoru k dalšiemu spracovaniu vo forme tokenov. Vrátený token obsahuje číselnú konštantu označujúcu typ načítaného lexému. V prípade textových reťazcov, číselných literálov a identifikátorov obsahuje tiež obsah lexému. Ďalšou možnosťou je vrátenie tokenu obsahujúceho konštantu značiacu načítanie chybného lexému. V tom prípade nasleduje ohlásenie lexikálnej chyby.

Funkciou lexikálneho analyzátoru je tiež prevod číselných escape sekvencií v textových reťazcoch na znaky s odpovedajúcimi ASCII hodnotami. Pri zadaní escape sekvencie nepovolenej v zadaní je hlásená lexikálna chyba. Naopak scaner nijako nezasahuje do tabuľky symbolov a túto prácu necháva až na syntaktickú analýzu. Rozhodli sme sa tak z dôvodu, aby sa do tejto tabuľky nemuselo pristupovať dvakrát.

4.5 Parser

Syntaktická analýza sa delí do dvoch častí. Tou hlavnou je analýza pomocou rekurzívneho zostupu, druhú časť tvorí precedenčná syntaktická analýza, ktorá spracúva výrazy.

Teoretickou predlohou rekurzívneho zozstupu je LL gramatika, z ktorej vychádza LL tabuľka určujúca povolenú štruktúru vstupného zdrojového súboru. Samotná implementácia tejto tabuľky sa tiež nazýva parser. Ten načítava vstupný súbor po jednotlivých tokenoch, čo je práca lexikálnej analýzy, konkrétne funkcie `getNextToken()`. Funkcie, používané k rekurzívnemu zostupu, zodpovedajú štruktúre LL gramatiky. Súčasťou kontroly syntaxe je tiež súbežné vykonávanie sémantických akcí, teda ošetrení nepovolených situácií, ako je použitie nedefinovanej premennej, či redefinícia premenných. Vnašom prekladači sa parser s rekurzívnym zostupom stará tiež o volanie funkcií a to ako užívateľom definovaných, tak aj vstavaných. V druhom prípade sú opäť vykonávané sématické kontroly z dôvodu pevne stanovených dátových typov parametrov u vstavaných funkcií. Po syntaktickej analýze nasleduje interpretácia, ktorá pracuje so sadou trojadresných inštrukcií, preto je nutné generovať príslušné inštrukcie behom spracovania vstupného súboru. K tomu slúži funkcia `generateInstruction()`.

Syntaktický analyzátor výrazov je založený na precedenčnej analýze, ktorá nám umožňuje použitie LR metódy (metóda zhora nadol). Pre potreby tejto analýzy je využitá precedenčná tabuľka.

Hlavnou funkciou starajúcou sa o spracovanie je `syntAnalExpr()`. Táto funkcia je volaná v prípade, že všeobecný syntaktický analyzátor očakáva výraz. Funkcia spracuje výraz, vygeneruje potrebné trojadresné inštrukcie pre výpočet a vráti štruktúru obsahujúcu ukazateľ na miesto, kde sa nachádza výsledok. Pre potrebu analyzátoru boli vytvorené nové dátové typy. Typ `tElement` je štrukúra, ktorá obsahuje ukazateľ na položku v binárnom strome, označenie typu literálu, prečíslovaný token a chybový kód. Táto štruktúra je pri skončení vrátená všeobecnému syntaktickému analyzátoru, ktorý na jej základe pokračuje ďalej. Typ `tStack` je zásobník, do ktorého sa podľa pravidiel precenenčnej tabuľky ukladajú tokeny a sú ďalej spracovávané. Precedenčná tabuľka sa stará o správne poradie vyhodnocovania jednotlivých operátorov a zároveň kontroluje niektoré syntaktické chyby. Je implementovaná ako dvojrozmerné pole `int`. Riadky a stĺpce tu tvoria tokeny, ktoré sú načítané pomocou lexikálneho analyzátoru a tieto tokeny sú ďalej prečíslovávané pre potrebu indexácie tabuľky. Podľa pravidiel určených tabuľkou sú potom spracovávané tokenmi a to tak, že sú buďto uložené na zásobník, alebo sú redukované pomocou pravidiel. Na zásobník sú ukladané bez zarážok, pretože zásobník umožňuje pohodlné prehľadávanie vrcholu bez potreby jeho odstránenia. Po redukcií, ktorá v prípade operácií generuje trojadresnú inštrukciu, je vždy vytvorená unikátna položka v binárnom strome, v ktorom sa bude pri interpretácii nachádzať výsledná hodnota. Behom týchto operácii je

kontrolovaná syntax a sémantika a v prípade chyby je činnosť ukončená s patričným chybovým kódom. V prípade finálnej redukcie je položka s ukazateľom do stromu pripojená do štruktúry `tElement` spolu s chybovým kódom a je vrátená všeobecnému syntaktickému analyzátoru. Syntaktický analyzátor výrazov tiež spracováva funkciu `write()`, ktorú spracováva po jednotlivých parametroch a generuje potrebné trojadresné inštrukcie.

4.6 Interpret

Ako hovorí názov, interpret sa stará o interpretáciu jednotlivých inštrukcií, ktoré generuje parser. Jedinou funkciou je funkcia `interpret()` z dôvodu rýchlosti programu. Vstupom je zoznam inštrukcií a tabuľka symbolov. Funkcia postupne prechádza zoznamom inštrukcií. Program sa začína vykonávať od inštrukcie `I_LAB_MAIN`. Aktuálna inštrukcia sa vykoná podľa mena z inštrukčnej sady a trojadresného kódu, ktorý vygeneroval parser. Inštrukčná sada je uložená v súbore `interpret.h`. V prípade chyby vráti interpret interpretačnú chybu. Interpretácia končí, keď intepret narazí na náveštie `I_END`. Interpret využíva dve pomocné datové štruktúry nachádzajúce sa v súbore `jumpstack.h`. Prvou z nich je zásobnik pre skoky, v ktorom je navrchu vždy posledná adresa inštrukcie, kam má program skočiť pri `I_JMP` a `I_JMPN`. Druhou je dvojsmerný lineárny zoznam, ktorý slúži pre ukladanie náveští a jednotlivých premenných z tabulky symbolov. Týmto sme v podstate vyriešili obor platnosti premenných, keďže sa berú vždy položky v rámci danej funkcie a neprepisujeme si hodnoty v binárnom strome. Do zoznamu sa pridávajú lokálne premenné z deklaračnej časti a parametre funkcie.

5 Testovanie

S testovaním sme začali už v čase, kedy boli aspon trocha funkčné jednotlivé časti. Každá súčasť musela byť najskôr otestovaná sériami testov predtým ako bola pripojená. S kompletným testovaním sme začali asi tri týždne pred odovzdaním, keď už bol projekt aspoň čiastočne funkčný. Postupne sme vytvárali testovaciu sadu podľa toho, čo bol náš interpret schopný spracovať. Začali sme rôznymi chybami, ktoré mohli nastať. Výstupy sme porovnávali s výstupmi interpretu jazyka lua. Potom sme prešli k testovaniu od jednoduchých programov až po zložitejšie zdrojové kódy. Tak sme vytvárali testovaciu sadu, ktorou sme neustále kontrolovali náš interpret a podľa potreby opravovali chyby.

6 Záver

V tomto projekte sme si mohli vyskúšať viacero vecí. Najskôr išlo o samotnú podstatu projektu, ktorý bol zadaný ako tímový. Pri riešení sme sa museli potýkať s nástrahami, ktoré sú časté aj v praxi. Pre príklad spomeniem syndróm 90 % hotovo, kedy sme sa predčasne radovali, ajkeď sme mali pred sebou ešte dlhú cestu k úspešnému cieľu.

Ďalej to bola problematika veľkých projektov, ktoré vyvíjajú viacerí vývojári a tým pádom je nutné zabezpečiť maximálnu kompatibilitu medzi modulmi. Takisto treba spomenúť rôzne alogritmy, ktoré nezostali len na papieri, ale ich implementáciu sme si vyskúšali sami na sebe.

Kľúčom k úspechu bol fakt, že sme nič nenechali na náhodu a do projektu sme sa pustili s dostatočným predstihom. Nemuseli sme riešiť otázku tímu, nakoľko ten sme poskladali ešte pred zahájením semestra. Skorým začiatkom a zodpovednou prácou každého z nás sme si privodili kľudné pracovné prostredie bez stresových situácií a úspešné odovzdanie projektu ešte v predtermíne.

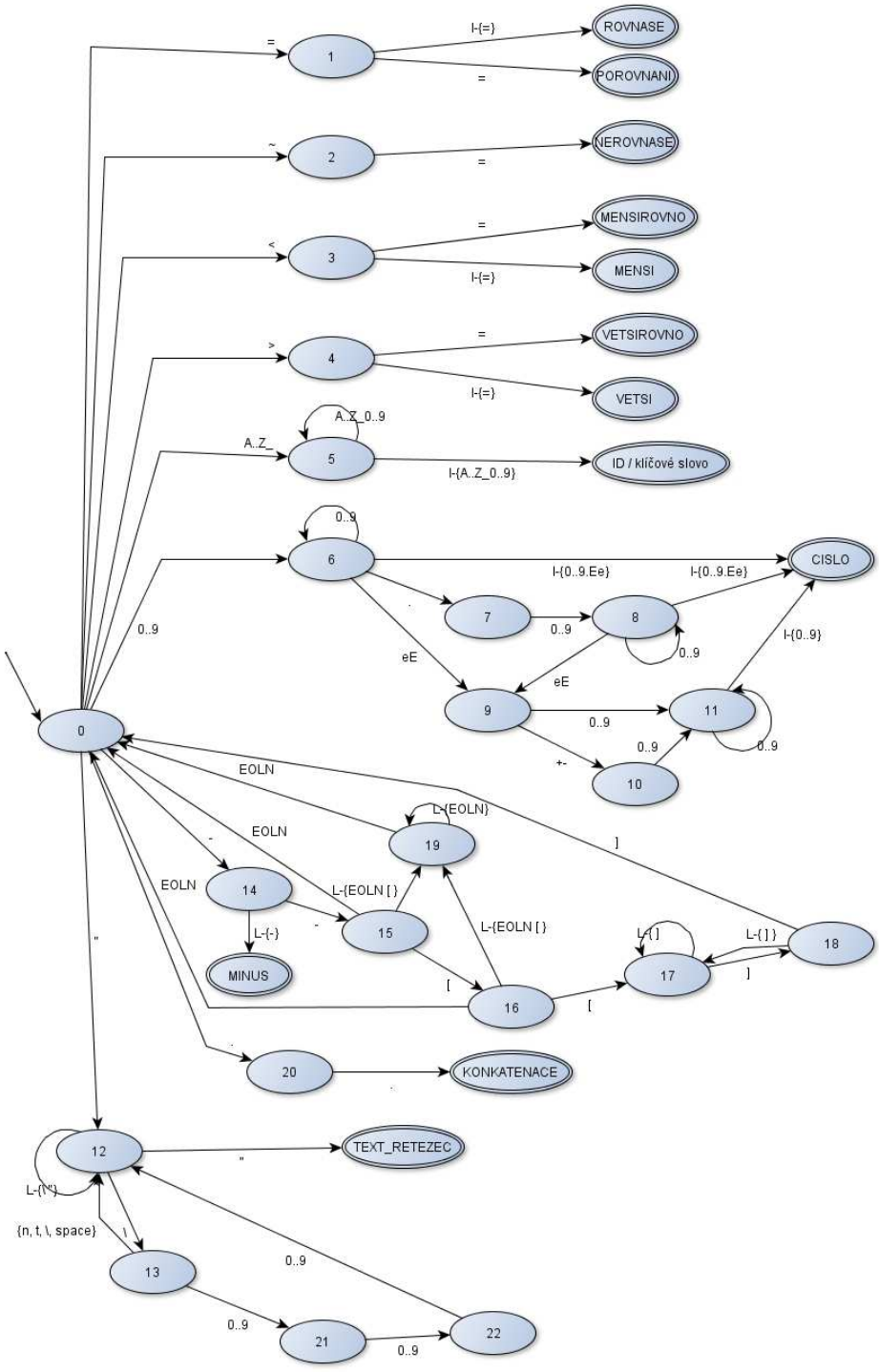
A Metriky kódu

Počet súborov: 18 súborov

Počet riadkov zdrojového kódu: 5817 riadkov

Veľkosť spustiteľného súboru: 54939 B

B Štruktúra konečného automatu



Pozn. Pre zjednodušenie niesú v grafe uvedené všetky koncové stavy. Vynechané sú len tie, jednoznakové, ktoré sa hneď vracajú ako tokeny (zátvorky, bodkočiarky a podobne).

C LL gramatika

<program>	->	function	<o_func>	<m_func>	$				
<o_func>	->	f_id	(<params>)	<body>	end	func	<o_func>		
<o_func>	->	ε							
<m_func>	->	main	(<params>)	<body>	end	;			
<params>	->	param	<params_n>						
<params>	->	ε							
<params_n>	->	param	<params_n>						
<params_n>	->	ε							
<body>	->	<var_decl>	<stat_list>						
<var_decl>	->	local	id	<decl>	<var_decl>				
<var_decl>	->	ε							
<decl>	->	;							
<decl>	->	=	<expr>	;					
<stat_list>	->	<state>	<stat_list>						
<stat_list>	->	ε							
<state>	->	write	(<expr>	<exprs>)	;				
<state>	->	while	<expr>	do	<stat_list>	end	;		
<state>	->	repeat	<stat_list>	until	<expr>	;			
<state>	->	if	<expr>	then	<stat_list>	else	<stat_list>	end	;
<state>	->	return	<expr>	;					
<state>	->	ε							
<state>	->	id	=	<assign>	;				
<assign>	->	<expr>							
<assign>	->	f_id	(<params>)						
<assign>	->	read	(<read>)						
<read>	->	number							
<read>	->	string							
<exprs>	->	<expr>	<exprs>						
<exprs>	->	ε							