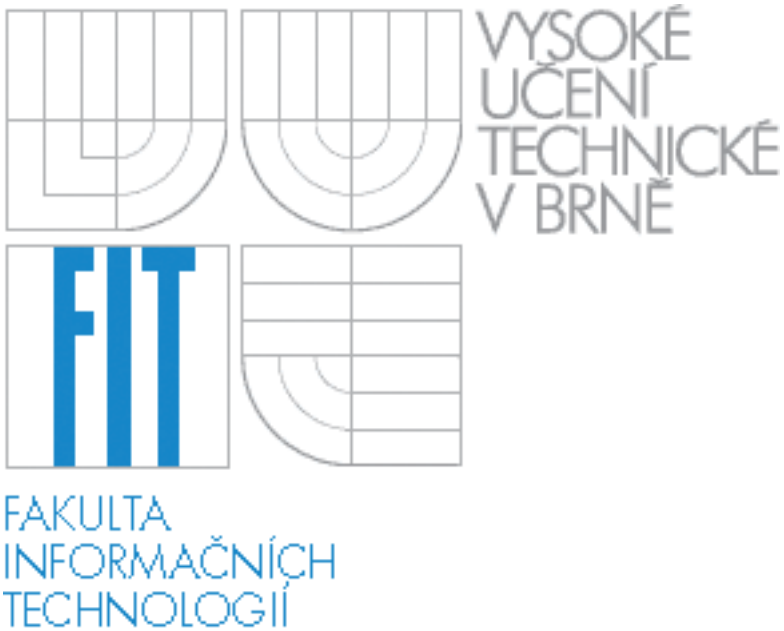

Interpret jazyka IFJ2007
Tým 3, varianta: a/2/II

Seznam autorů:
xfilom00 - Martin Filo - 25%
xkalab00 - Jan Kaláb - 25%
xkobli00 - Ondřej Koblížek - 25%
xnovyo01 - Ondřej Nový - 25%
Datum: 16. prosince 2007
Fakulta informačních technologií
Vysoké učení technické v Brně

Obsah

Část I
Úvod

Dokument, který teď právě čtete se zabývá implementací interpretu imperativního jazyka IFJ2007. Postupně se dočtete o přípravě na projekt a jeho implementaci. Také budou zmíněna některá úskalí, kterým jednotliví členové čelili.

Část II
Příprava, rozdělení a implementace všech částí

1 Příprava

Při práci v týmu bylo potřeba nejprve zvolit vedoucího, člověka, který bude vývoj řídit a dohlížet nad správnou implementací, popřípadě ukazovat na chyby, kterých se daný člen týmu dopustil. Zde byl jednohlasně zvolen Ondřej Nový. Ten se postaral o prvotní stanovení pravidelných schůzek, vytvoření pracovního repositáře a základní konstrukci projektu.

1.1 Týmové porady

Aby byla práce efektivní, zvolili jsme pravidelné schůzky týmu alespoň jednou za čtrnáct dní. Na schůzkách se kontroloval kód jednotlivých členů týmu, rozdávaly se termíny dalších prací a kárali se opozdilci.

1.2 Správa kódu

Vedoucí týmu rozhodl, že o správu kódu se bude starat systém zvaný Subversion, zkráceně svn. Ačkoliv polovina týmu neměla osobní zkušenosti s svn, pochopení a uvedení do praxe bylo celkem snadné. Díky tomuto systému jsme byli schopni si navzájem efektivně vyměňovat zdrojové kódy. Byly stanoveny podmínky pro vkládání obsahu do svn, které se ve většině případů dodrželi.

2 Rozdělení práce

Formou svobodné volby se na první schůzce rozdělila první část kódu mezi všechny zúčastněné a zapsal se termín, do kdy každý musí předložit finální a bezchybný kód. Po jeho předložení se rozdělila práce na hlavních částech programu: lexikálním, sémantickém a syntaktickém analyzátoru a interpretu. Práce byla opět provedena svobodnou volbou.

3 Implementace všech částí

3.1 Quicksort

Metoda quicksort patří mezi nejrychlejší a nejčastěji používané řadící algoritmy. Mnoho vyšších programovacích jazyků ji využívá pro své příkazy pro řazení. Norma POSIX také obsahuje specifikaci tohoto algoritmu a této normy jsme se rozhodli držet.

POSIX prototyp je velmi obecný a umožňuje seřadit prakticky jakékoliv myslitelné posloupnosti čehokoliv. Je to dáno tím, že funkce qsort (a tudíž i námi implementovaný quicksort) v sobě neobsahuje žádnou porovnávací funkci, ani neočekává žádný předdefinovaný datový typ. Namísto toho očekává tyto informace jako parametry. Funkci tedy předáte odkaz na sekvenci kterou chcete seřadit, dále počet prvků v této posloupnosti a velikost jednoho prvku. Z těchto informací již není problém předávat porovnávací funkci, která je dalším a posledním parametrem funkce, jednotlivé prvky, aniž by funkce měla tušení co se řadí. Možná se Vám to zdá jako zbytečné zesložitění, když jsme ze zadání věděli, že budeme řadit pouze posloupnosti znaků, a máte pravdu. Ale jako procvičení práce s ukazateli a pamětí to bylo ideální.

Efektivita algoritmu quicksort je velice závislá na volbě vhodného pivotu. I na toto jsme mysleli a výběr pivotu se pokusili optimalizovat. Není to nic složitého. U krátkých posloupností (do 3 prvků včetně) prostě zvolíme prostřední prvek. Pokud je posloupnost delší, vybere se první, prostřední a poslední prvek, ze kterých se vybere medián a ten se prohlásí za pivot.

Samotné řazení jsme se rozhodli provádět in-place, čili nepotřebujeme další paměťový prostor. Také jsme se rozhodli použít rekurzivní variantu, která nám přišla zřejmější a snazší.

3.2 Obecný syntaktický analyzátor

Obecný syntaktický analyzátor byl překvapivě jednoduchý, ačkoliv s ním kolegové z předchozích ročníků strašili a považovali jej za nejnáročnější část projektu.

Pro jeho implementaci jsme měli použít metodu rekurzivního sestupu. Na přednáškách nám bylo ukázáno, jak jí napsat pomocí LL tabulky. Tato metoda se mi zdála zbytečně složitá, a tak jsem se rozhodl porozhlédnout se po internetu po jiné, snad snazší, variantě. Internet mě velmi často odkazoval na různé automatické nástroje pro vytvoření syntaktické analýzy metodou rekurzivního sestupu. Ty jsme však neměli povoleno použít, a tak jsem je ignoroval. Ovšem velmi často jsem si ve spojení s nimi všiml zmínek o tzv. Bakus-Naurově formě. Rozhodl jsem se tedy pátrat dál, protože pokud z ní

nějak dokáže rekurzivní sestup sestavit stroj, snad by to nemusel být problém ani pro člověka. A bylo tomu tak. Je to perfektně ilustrováno v článku o rekurzivním sestupu na Wikipedii. Backus-Naurova forma (dále jen BNF) velmi věrně kopíruje strukturu výsledného kódu v jazyce C, s použitím dvou pomocných funkcí pro kontrolu toho, že právě přijatý token od lexikální analýzy je ten který jsme očekávali. Sepsání BNF pro jazyk IFJ netrvalo déle než dvě hodiny a ihned jsem se mohl pustit do její implementace v C.

Během implementace se vyskytly jen dva větší problémy, a to složené závorky a s nimi spojené blokové příkazy. Pro správnou činnost složených závorek bylo nutné velmi dobře pochopit rekurzivní zanořování jednotlivých příkazů, aby nedocházelo k ukončení blokového příkazu dříve, než k němu ve zdrojovém kódu skutečně dojde. Druhým oříškem byly složené příkazu v příkazech podmínky a cyklu. Za normálních okolností totiž příkaz po svém zakončení znakem ; očekává další klíčové slovo (další příkaz), toto ovšem právě u podmínek a cyklů neplatí. Po zamyšlení se a probrání problému s kolegy se jako nejsnazší a až trapné řešení ukázalo vložení dalšího parametru funkce který říká, zda se má provést příkaz pouze jeden nebo více. Triviální, ale funkční.

3.3 Sémantický analyzér

Sémantická kontrola provádí kontrolu typů, existenci proměnných a vkládá podmíněné a nepodmíněné skoky. S tím souvisí obrovské problémy s implementací skoků. Bylo nutné najít formu jejich zapsání jak jednoduchou pro programátora, tak efektivní v rámci programu. Díky dobře navrženým strukturám položek avl stromu, byla práce s proměnnými vcelku jednoduchá. Každá položka nese informaci o datovém typu, jaký obsahuje a to ji provází po celou dobu konání kódu.

3.4 Tabulka symbolů

Tabulka symbolů je řešena pomocí avl stromu, což je výškově vyvážený binární vyhledávací strom. Stručně lze říci, že každý uzel má v sobě proměnnou `balance`, která může mít pět stavů: -2,-1,0,1,2. Když obsahuje 2, je strom nevyvážený doprava, když -2, tak opačně. Pokud je hodnota 1, je těžký vpravo, ale vyvážený. Pro hodnotu -1 obdobně. Hodnota 0 udává, že je strom absolutně vyvážený. V avl stromu se při každém vložení nového uzlu najde pomocí klíče pozice, kam vložit uzel. Pokud je přitom narušena rovnováha, tedy jakýkoliv uzel ve stromě nabude hodnoty -2 či 2, tak se strom přebalancuje pomocí pravé rotace, levé rotace, dvojité pravé rotace nebo dvojité levé rotace.

K největším úskalím v této části kódu lze přirovnat přebalancování stromu pomocí rotací.

3.5 Syntakticá analýza zdola nahoru

Byla použita pro analýzu výrazů uvozených zepředu a zezadu znaky $. Dle zadání jsme ji implementovali precedenční tabulkou, dále (PT), která je vhodná právě na ošetření výrazů. Tato metoda si vyžádala zásobník v němž se dočasně ukládaly jednotlivé tokeny z lexikální analýzy. Precedenční tabulka obsahuje 4 stavy: posuv, redukci, chybu a ukončení výrazu. Ještě před prvním porovnáním s PT se do zásobníku vloží $. Pak se zásobník průběžně zaplňuje novými tokeny a zároveň se redukuje dle gramatických pravidel. Pokud je výraz správně zadán, tak se nakonec zásobním vyprázdní a pokračuje analýza shora dolů. Po vyhodnocení většiny pravidel se vytvoří pomocná proměnná v tabulce symbolů, kde bude uložen mezivýsledek či výsledek výrazu. V praxi se při redukování provádí i sémantická kontrola, přetypování int na double a generování tří adresného kódu.

3.6 Lexikální analýza

Lexikální analýza je tvořena konečným stavovým automatem. Ten načítá data ze zdrojového kódu jazyka IFJ2007 a načtené údaje posílá ve formě tokenů syntaktickému analyzéru. Token definuje jednotlivý prvek a říká, jak se k němu má při zpracování přistupovat. Analyzér prochází tabulku rezervovaných klíčových slov, ve které hledá, zda-li nejde o rezervované slovo se speciální funkcí. Vyhledávání je řešeno pomocí zarážky, tudíž se zajistí, že hledané slovo se vždy najde. Potom se však musí jen rozhodnou, zda-li šlo o zarážku nebo opravdu o klíčové slovo.

3.7 Interpret

Interpret se stará o vykonání kódu. Zde byl zvolen přístup vygenerování tří adresného kódu s definicí instrukcí, které interpretu řeknou, jakým způsobem se k předaným parametrům má chovat. S návrhem interpretu úzce souvisí i řešení tří adresného kódu podporujícího funkce skoků. V návrhu a implementaci kódu se zavedla hodnota prováděné instrukce. Pokud je hodnota menší než nula, jde o běžnou instrukci, pokud větší nebo rovna nule, jde o instrukci skoku nebo návěští identifikující skok.

Samotný interpret nejprve provede analýzu tří adresného kódu a uloží si do pole umístění návěští pro instrukce skoků. Tímto je zaručeno jednoduché

provedení skoku, kdy se přistoupí na index pole, které odpovídá identifikátoru návěští a jeho obsahem je výkonný řádek kódu po návěští.

3.8 Chybové stavy

Poněvadž v jazyce C neexistuje možnost vyvolání výjimky, bylo potřeba tuto možnost naimplementovat. Byla vytvořena knihovna `error.c`, která obsahuje globální proměnnou, jejíž obsahem je výčet možných chybových stavů. To zajistí možné vyvolání výjimky a všechny části kódu si výjimku mohou přečíst a podle ní po sobě regulérně uklidit a vrátit příslušný návratový kód.

3.9 Nekonečné řetězce

Pro práci s řetězci byla na implementována knihovna `str.c`, která usnadňuje práci s řetězci. Jejím kladem je dynamická alokace paměti, teoreticky nekonečné množství. Další nespornou výhodou je uvolnění nevyužitého na alokovaného místa, tudíž je snížena paměťová náročnost projektu.

3.10 Automatizované testy

Aby se snížil výskyt možných chyb, byly vytvořeny automatizované testy, které využívají referenčních výsledků a nastroje GNU diff. Než se část kódu prohlásí za finální, je třeba zajistit, aby výstup z něj byl zcela v pořádku a pozdější změna či úprava struktury neměla nežádoucí vliv na funkčnost celého projektu.

Část III
Zdroje

1. Slajdy k předmětu IFJ.
2. Wikipedia - hesla: quicksort, recursive descent parser, avl tree, simple precedence parser
3. CSTUG
   http://www.cstug.cz