Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentace k projektu do předmětů IFJ a IAL

Implementace interpretu imperativního jazyka IFJ12

Tým 034, varianta a/1/II.

Jméno	**Login**	**Rozdělení**
Michal Duban	xduban01	20 %
Marko Fábry (vedoucí)	xfabry01	20 %
Václav Hanselka	xhanse00	20 %
Petra Heczková	xheczk04	20 %
Jan Wrona	xwrona00	20 %

9. prosince 2012

Obsah

1 Úvod

Tato dokumentace popisuje způsob vývoje a popis implementace interpretu imperativního jazyka IFJ12. Projekt je rozdělen do tří základních částí: jádrem je syntaxí řízený překlad (sekce 2.2), čtení vstupního zdrojového kódu zajišťuje lexikální analyzátor (sekce 2.1) a následné vykonání programu způsobuje interpret (sekce 2.3).

Z řady zadání jsme si vybrali variantu a/1/II. Tato nám určila použití Knuth-Moris-Prattova algoritmu pro vyhledávání. Které využívá vestavěná funkce `find()`, pro vestavěnou funkci `sort()` bylo dle zadání nutno použít algoritmu quicksort a tabulka symbolů byla implementována pomocí hashovací tabulky.

2 Popis řešení

2.1 Lexikální analyzátor

Tento modul, nazvaný scanner, je jedinou částí programu operující se vstupním souborem. Jeho úkolem je načítání lexémů a jejich předání syntaktickému analyzátoru v podobě tokenů. Důležitá vlastnost je identifikace těchto lexémů, především identifikátorů, datových typů (číselný literál, řetězcový literál, logický literál a nil), komentářů, závorek, konců řádků apod. V případě identifikátoru je součástí scanneru také rozpoznání klíčových a rezervovaných slov, které se nesmějí použít jako identifikátor.

Implementačně je tento modul tvořen především klíčovou funkcí `GetNextToken()`, která je při potřebě dalšího tokenu volána z modulu syntaktického analyzátoru a předá mu strukturu obsahující právě jeden následující token a jeho typ. Číselný literál je načten jako řetězec, zkontrolován na lexikální chyby a následně převeden na datový typ double. V případě řetězcového literálu ohraničeného uvozovkami je implementován převod víceznakových escape sekvencí na jediný znak a zařazen jako součást tokenu. Při identifikaci komentáře, ať jedno čí víceřádkového, je zajištěno jeho přeskočení. Podstatnou funkcí je také `append()`, která počítá s předem neznámou délkou načítaného lexému. Tato vlastnost je zajištěna použitím knihovní funkce `realloc()`. Dle zadání byl scanner navrhnut a implementován jako konečný automat, jehož struktura v příloze B.

2.2 Syntaxí řízený překlad

Syntaktický analytátor je velmi důležitý pro syntaxí řízený překlad. Zajišťuje překlad programu ve zdrojovém jazyce a odhalení některých druhů chyb, které se mohou při překladu objevit. Je rozdělen do dvou modulů, pracujících na odlišných částech kódu. S lexikálním analyzátorem komunikuje pomocí fuknce `GetNextToken()`.

V modulu parser je implementována metoda rekurzivního sestupu (shora dolů). Jádrem celého modulu je funkce `parser()`, která je volána ze souboru `ifj12.c`. Syntaktický analyzátor očekává na vstupu posloupnost tokenů, kontroluje jejich typ a vkládá na instruční pásku instrukce 3-adresného kódu. Vytváří uměle funkci `main()`, kontroluje předchozí definici proměnných a jejich kolize s klíčovými slovy. Pokud narazí na výraz nebo příkaz výběru řetězce zavolá modul `expression`, který ho zpracuje.

Modul `expression` obsahuje globální precedenční tabulku, ve které jsou uloženy informace o prioritě a asociativitě operátorů. Dle zadání je použita metoda precedenční syntaktické analýzy (zdola nahoru). Algoritmus porovnává typ aktuálního znaku/tokenu na vstupu s typem terminálu, uloženým nejblíže vrcholu zásobníku (zde implementováno

jako globální dynamické pole). Před vložením některých znaků je přidán ještě pomocný znak. Při rozhodování mezi jednotlivými pravidly se algoritmus řídí vzdáleností mezi pomocným znakem a vrcholem zásobníku, dále pak pro pravidlo $E \rightarrow E\ op\ E$ typem operátoru. Pro tento modul je zaveden nový typ tokenu `tExpression`, který označuje některé tokeny na zásobníku.

Funkce jsou považovány za výraz, proto se detekují v modulu `expression` kombinací proměnné (identifikátor funkce) a levé závorky bez operátoru mezi nimi. Podobně se detekuje i příkaz výběru řetězce. Mají speciální označení v precedenční tabulce. Funkce se, na rozdíl od příkazu výběru řetězce, dále zpracovává v modulu parser rekurzivním sestupem. Existuje několik podfunkcí zpracovávající několik variant (vestavěných) funkcí ve zdrojovém programu, např. funkce bez parametrů nebo funkce s N parametry.

Oba moduly úzce spolupracují s tabulkou symbolů. Vkládají do ní proměnné, identifikátory funkcí a uměle vygenerované konstanty, sloužící pro uchování mezivýpočtů.

2.3 Interpret

Po bezchybném vykonáni syntaktické a lexikální analýzy je řízení programu předáno interpretu, který vykonává instrukce tříadresného kódu. Tyto instrukce jsou postupně vkládány syntaktickým/sémantickým analyzátorem na instrukční pásku, kterou reprezentujeme pomocí jednosměrně vázaného lineárního seznamu (převzat z jednoduchého interpretu)[4].

Interpret postupně prochází seznamem a vykonává příslušné instrukce. Je-li potřeba, je provedena kontrola správnosti typu u jednotlivých operandů (tato akce však není prováděna vždy, záleží na konkrétní instrukci). Seznamem procházíme do té doby, dokud není nalezena instrukce `I_STOP`, která úspěšně ukončí interpretaci, nebo dokud není odhalena nějaká chyba. V takovém případě interpret skončí s odpovídajícím chybovým kódem.

2.4 Funkce `sort()`

Pro implementaci vestavěné funkce `sort()` jsme si vybrali zadání, které obsahuje quicksort, jelikož nám tento algoritmus byl již znám. V projektu byl použit algoritmus, který využívá rekurzi. Quicksort je možné implementovat i pomocí cyklů, ale rekurzivní zápis se v našem případě ukázal jako vhodnější varianta. Dále k zamyšlení byla volba pivotu a jako ideální se jevil prostřední znak řazeného řetězce či podřetězce. Pro jednodušší pochopení algoritmu jsme využili také serveru `http://www.youtube.com`, kde jsme našli názornou videoukázku, jak quicksort pracuje.

2.5 Funkce `find()`

Pro implementaci vestavěné funkce `find()` jsme měli použít Knuth-Moris-Prattův algoritmus, který se jeví jako jednoduchý a efektivní. K pochopení KMP algoritmu stačily školní skripta. Tento algoritmus je výhodný, jelikož se od jakéhokoli obyčejného algoritmu liší tím že se není třeba vracet na znaky, které už porovnány byly a tím se proces vyhledávání zrychlí. Složitost tohoto algoritmu je $O(n + k)$.

2.6 Tabulka symbolů

Tabulku symbolů jsme měli podle zadání implementovat jako hashovací tabulku. Jelikož hashovací tabulka byla i v předmětu IAL jako domácí úloha, nebylo nutné ji znovu vymýšlet. Již napsané funkce z hashovací tabulky stačily jen mírně upravit, aby byla tabulka

použitelná pro náš projekt. Co se týče rozptylovací funkce, tak tu jsme rovněž využili z předem napsané úlohy z IAL, nebylo zapotřebí ji jinak měnit, funkce nám vyhovovala.

3 Způsob práce v týmu

Projekt jsme nebrali na lehkou váhu, tudíž první práce začaly už v rané fázi semestru. Po konečném přihlášení všech členů do týmu jsme uspořádali první schůzi. Jednotlivě jsme nastudovali zadání a pustili se do diskuze ohledně nejasností, také jsme předběžně rozdělili práci mezi jednotlivé členy tak, aby rozdělení bylo rovnoměrné. Detailní rozpis rozdělení práce je uveden v sekci 3.1. Důležitým bodem na první schůzi byla také technická stránka naší spolupráce, a to především způsob verzování, formátování a sdílení kódu, způsob vzájemné elektronické komunikace.

Nikdo z nás neměl zkušenosti s žádným VCS (version control system), ale volba nakonec padla na program git, což se ukázalo jako dobrá volba. Ve spolupráci s teamovým privátním online repozitářem poskytovaným zdarma na serveru `http://www.bitbucket.com` jsme měli k dispozici jednoduchý a spolehlivý VCS s neustálým online přístupem ke kódu, navíc s výborným webovým grafickým rozhraním.

Co se elektronické komunikace týče, využívali jsme skupinového chatu na serveru `http://www.facebook.com`, ale také komunikační služby ICQ. Naší teamovou výhodou bylo ubytování všech členů na stejné koleji, proto jsme upřednostňovali před elektronickou komunikací jednoznačně efektivnější komunikaci ústní. Také se častěji scházeli členové týmu, jejichž moduly spolu přímo spolupracovaly a díky tomu nebylo třeba tak často pořádat schůze celého týmu. Přesto jsme se ale snažili každé úterý pořádat schůzi celého týmu, kde jsme seznámili ostatní členy s důležitými změnami, které jsme během týdne provedli.

3.1 Rozdělení práce

Modul	**Autor**
`expressions`	Petra Heczková
`htable`	Marko Fábry
`ial`	Václav Hanselka, Marko Fábry
`ifj12`	Marko Fábry
`ilist`	Převzat z interpretu[4] na stránkách projektu
`interpret`	Michal Duban
`parser`	Marko Fábry
`scanner`	Jan Wrona
`stack`	Marko Fábry

4 Závěr

Program byl úspěšně otestován v prostředí operačního systému Linux a to jak na serveru `merlin`, tak i na osobních počítačích s architekturami 32 i 64 bitovými. Vstupními programy pro testy byly použity vzorové programy v zadání, naše vlastní programy, ale i kódy jiných týmu, které je zveřejnily pro testování. Program dodržuje veškeré specifikace zadání, formát vstupních i výstupních dat, používá požadované algoritmy a programovací techniky. Interpret může být použit jako součást dalších programů či skript, při dodržení vstupních dat specifikovaných jazykem IFJ12.

Reference

[1] Prof. Ing. Jan M Honzík, CSc., *Algoritmy: Studijní opora*. Verze 12-D.

[2] Alexandr Meduna, Roman Lukáš, *Formální jazyky a překladače: Studijní opora*. Verze 1.2006 + revize 2009 a 2010.

[3] *Recursive descent parser*. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2012-12-09]. Dostupné z: `http://en.wikipedia.org/wiki/Recursive_descent_parser`

[4] *Zjednodušená implementace interpretu jednoduchého jazyka*.
Dostupné z: `https://www.fit.vutbr.cz/study/courses/IFJ/private/projekt/jednoduchy_interpret.zip`

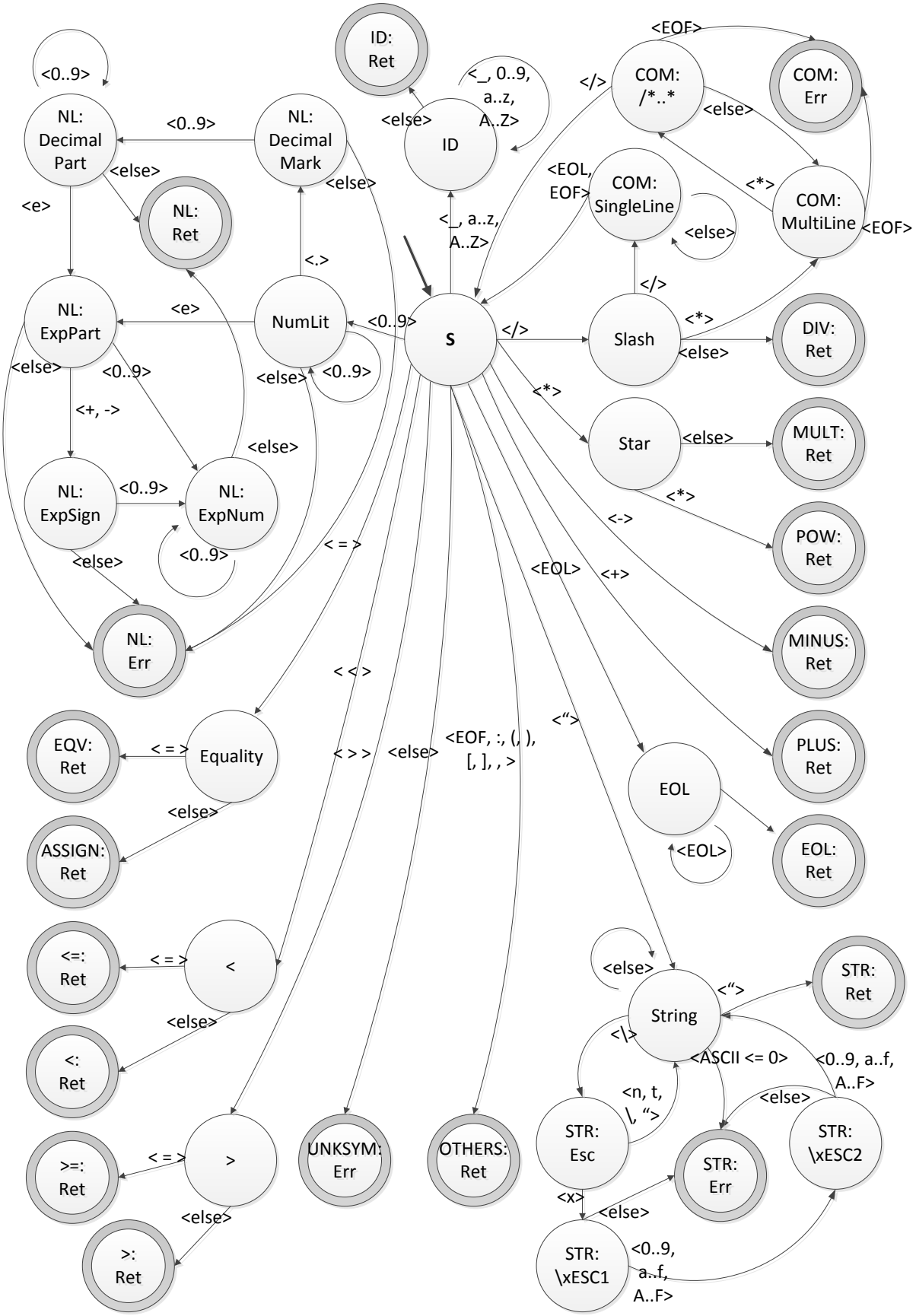
A Metriky kódu

Počet souborů: 19

Počet řádků zdrojového textu: 6135

Velikost spustitelného souboru: 63390B (systém Linux, 64 bitová architektura, při překladu bez ladicích informací

B Struktura konečného automatu



C LL-gramatika programových konstrukcí

1. $< program > \rightarrow < statementlist >$
2. $< statementlist > \rightarrow < statement >< statementlist >$
3. $< statementlist > \rightarrow < statement >$
4. $< statement > \rightarrow$ EOF
5. $< statement > \rightarrow$ IF $< e >$ EOL $< statementlist >$ ELSE EOL $< statementlist >$ END EOL
6. $< statement > \rightarrow$ IDENT $=< e >$ EOL
7. $< statement > \rightarrow$ IDENT = IDENT($< arglist >$ EOL
8. $< statement > \rightarrow$ WHILE $< e >$ EOL $< statementList >$ END EOL
9. $< statement > \rightarrow$ RETURN $< e >$ EOL
10. $< statement > \rightarrow$ FUNCTION IDENT($< arglist >$ EOL $< statementlist >$ END EOL
11. $< arglist > \rightarrow)$
12. $< arglist > \rightarrow$ IDENT, $< arglist >$
13. $< arglist > \rightarrow$ IDENT)

D Seznam pravidel pro zpracování výrazů

1. $< e > \rightarrow$ IDENT
2. $< e > \rightarrow (< e >)$
3. $< e > \rightarrow < e > ** < e >$
4. $< e > \rightarrow < e > * < e >$
5. $< e > \rightarrow < e > / < e >$
6. $< e > \rightarrow < e > + < e >$
7. $< e > \rightarrow < e > - < e >$
8. $< e > \rightarrow < e > == < e >$
9. $< e > \rightarrow < e > \neq < e >$
10. $< e > \rightarrow < e > \leq < e >$
11. $< e > \rightarrow < e > \geq < e >$
12. $< e > \rightarrow < e > << < e >$
13. $< e > \rightarrow < e > >< e >$
14. $< e > \rightarrow$ IDENT[IDENT : IDENT]