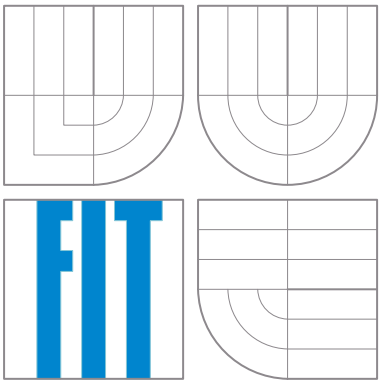


Implementace imperativního jazyka IFJ13

Tým 84, varianta: a/2/II

15. prosince 2013

Autoři:	xbrezi13	Karel Březina	20%	(vedoucí)
	xhavle03	Vojtěch Havlena	20%	
	xdobro12	Matúš Dobrotka	20%	
	xraszk03	Aleš Raszka	20%	
	xjerab18	Kamil Jeřábek	20%	

Rozšíření: FOR

Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

Obsah

1 Úvod

Tato dokumentace byla vytvořena pro účel popisu implementace interpretu imperativního jazyka IFJ13, který je podmnožinou známého jazyka PHP.

Interpret je program, který analyzuje a vykonává vstupní zdrojový kód napsaný v daném jazyce. V následujících kapitolách se dočtete o průběhu naší implementace jednotlivých částí interpretu.

2 Fáze přípravy

2.1 Tým

O tomto projektu jsme věděli s dostatečným předstihem. Na internetu koluje mnoho „zkušeností“ od studentů z vyšších ročníků, a proto nám bylo jasné, že bude třeba nepodcenit tento první týmový projekt.

Tým jsme proto začali skládat již přes prázdniny. Přestože se na počátku většina členů osobně neznala, nebyla to větší překážka ve vzájemné spolupráci.

2.2 První setkání

Cíl prvního setkání byl zřejmý. Zajistit dobře fungující systém v týmu.

Padla proto volba zvolit si vedoucího týmu, který by byl oporou ostatním členům a kontroloval postup práce na projektu. Dále zajišťoval spojení mezi lektory a pomáhal jednotlivým členům týmu upřesnit směr implementace. Na tuto pozici byl jednohlasně zvolen člen týmu, Karel Březina.

2.3 Komunikace v týmu

Komunikace v týmu je nedílnou součástí větších projektů a mnohdy tvoří tenkou hranici mezi jejich úspěšným a neúspěšným dokončením.

Dohodli jsme se proto na týdenních setkáních v knihovně. V případě, kdy nebyla možnost se setkat v daném týdnu, byla namísto toho domluvena hlasová konference. Tento způsob komunikace jsme dodržovali a jen výjimečně se stalo, že bychom vypustili tento projekt z hlavy na celý týden.

2.4 Verzovací systém

Poslední důležitou částí k úspěšnému projektu bylo zavedení verzovacího systému.

Tento systém výrazně zjednodušuje distribuci aktuálního kódu mezi jednotlivými členy. Naše rozhodnutí padlo na systém programu git a službu `bitbucket.org`, která nám poskytla prostor k vytvoření pracovního repositáře.

Vzhledem k tomu, že ne všichni členové týmu měli zkušenosti s tímto verzovacím systémem, ujal se vedoucí demonstrace základních ovládacích příkazů.

Se zavedením git repositáře se však vázalo omezení na práci jednotlivých členů. Každý mohl spravovat pouze ty soubory, které příslušely jeho části implementace. Případnou změnu v jiném souboru bylo třeba nejdříve konzultovat s pověřeným členem týmu (případně s vedoucím).

2.5 Rozdělení práce

Jakmile byly vyřešeny všechny administrativní záležitosti, došlo na rozdělení práce.

Celý tým proto prozkoumal detaily zadání a vedoucí začal na základě této skutečnosti přidělovat členům jednotlivé části projektu. Bylo třeba zohlednit požadavek,

co který člen chce dělat, jaké jsou jeho schopnosti a zároveň zachovat vyváženost pracovní zátěže na celý tým.

Projekt jsme rozdělili na tyto části a aplikovali na nich vertikální způsob rozdělení práce. Jedná se o vytvoření jednotlivých funkčních bloků, které mají pevně stanovené rozhraní.

1. Lexikální analýza (Matúš Dobrotka)
2. Syntaktická a sémantická analýza + generátor (Vojtěch Havlena)
3. Interpret (Kamil Jeřábek)
4. Tvorba abstraktních datových struktur (Aleš Raszka)
5. Testování + dokumentace (Karel Březina)

3 Implementační část

3.1 Lexikální analýza

Lexikální analýza je základem každého překladače. Implementovali jsme ji v podobě deterministického konečného automatu podle obrázku 1.

Proces spočívá v analýze daného typu lexému (základní jednotka jazyka) ze vstupního souboru (zdrojový kód). Pro dosažení tohoto výsledku je třeba sekvenčně načítat znaky a správným průchodem dojít do koncového stavu konečného automatu.

Zde bylo třeba dát si pozor na všechny možné kombinace zápisu, především u lexémů generující typ double a string (expanze posloupnosti znaků). Odlišení názvů funkcí jsme pak vyřešili jednoduchou porovnávací rutinou.

Všechny potřebné informace (typ, název, hodnota aj.) o daném typu lexému jsou poté uloženy do datové struktury, která se obecně nazývá token. Datovou strukturu token jsme použili pro všechny typy lexémů (včetně funkcí).

Tato část byla stěžejní pro další postup v implementaci syntaktického analyzátoru.

3.2 Syntaktická analýza

O syntaktické analýze by se dalo říct, že je srdcem překladače.

Pro implementaci syntaktické analýzy jsme využili metodu rekurzivního sestupu shora dolů, která nám byla doporučena lektory. Tato metoda je z pohledu implementace jednodušší než prediktivní syntaktická analýza. Zde bylo potřeba vytvořit LL gramatiku (příloha B), která popisuje jazyk IFJ13 a všechny jeho přípustné kombinace zápisu.

Vzhledem k tomu, že pro vyhodnocení výrazů je syntaktická analýza shora dolů nevhodná, použili jsme pro ně syntaktickou analýzu zdola nahoru. Tato metoda využívá precedenční tabulku a zásobník pro dočasné uložení tokenů při zpracovávání výrazu.

Syntaktická analýza je úzce spjata se sémantickou analýzou a generátorem cílového kódu.

3.3 Sémantická analýza a generátor cílového kódu

Úloha sémantické analýzy spočívá v odhalování redefinic funkcí a špatného počtu parametrů při volání funkcí. Pokud by našla tento typ chyby, došlo by k ukončení překladu a interpretace by neproběhla.

Generátor cílového kódu je pak způsob jakým lze zprostředkovat vše z analýzy vstupního programu do praktické roviny.

My jsme zvolili generování tříadresného kódu, který je z pohledu optimalizace jeden z nejlepších. Generovaný kód je ukládán do obousměrně vázaného seznamu, kterým následně interpret postupuje sekvenčně (mimo instrukce skoku). Zde bylo

třeba vyřešit, jakým způsobem přistupovat do správné části paměti (otázka rekurze), způsob předávání parametrů do funkce a také jakým způsobem je odlišit od pomocných proměnných interpretu.

Toho jsme docílili využitím obousměrně vázaného seznamu nad datovou strukturou hashovací tabulky. Při zavolání funkce dojde k vytvoření nové hashovací tabulky v obousměrně vázaném seznamu a posunutí ukazatele na tuto tabulku. Vzhledem k tomu, že proměnné mohou být pouze lokální, má každá funkce (včetně „funkce“ <?php) svoji hashovací tabulku.

Předávání parametrů do jednotlivých funkcí probíhá pomocí datové struktury fronta a speciálních instrukcí, které provádí potřebné operace nad touto strukturou a tvoří tak obslužnou rutinu před skokem do funkce.

Pro odlišení pomocných proměnných jsme implementovali pomocný generátor názvů.

3.4 Interpret

K čemu by nám byl cílový kód bez interpretu, který by ho dovedl zpracovat.

Interpret dostane na začátku vykonávání seznam instrukcí, který nejdříve analyzuje a zmapuje všechna místa (tzv. labely) pro skoky v programu. Poté se vrátí na začátek kódu (obousměrně vázaného seznamu) a začne ho sekvenčně provádět. Pro tento účel byla navrhnuta instrukční sada, která zpracovává všechny základní operace a také několik vestavěných funkcí (např. funkce pro explicitní přetypování) jazyka IFJ13.

Vzhledem k tomu, že je jazyk IFJ13 dynamicky typovaný, interpret tak hraje částečně roli sémantického analyzátoru, kdy kontroluje, zda se operace provádí nad přípustnými kombinacemi tohoto jazyka.

3.5 Tabulka symbolů

Abstraktní datový typ, který je použit pro ukládání všech tokenů, které reprezentují proměnné a funkce. Ukládané tokeny jsou potřeba při interpretování zdrojového kódu.

Vzhledem k naší variantě zadání jsme použili pro implementaci tabulky symbolů hashovací tabulku. Návrh hashovací tabulky byl částečně převzat z projektu z kurzu Jazyk C [1] a upraven do takové podoby, která vyhovuje specifikacím pro tento projekt.

3.6 Ostatní abstraktní datové typy

V tomto projektu jsme využili znalostí z předmětu IAL [2] i doporučené literatury [3] a prakticky je zakomponovali (včetně všech potřebných operací) do jednotlivých částí interpretu.

Mezi implementované ADT patří obousměrně vázaný seznam (pro instrukce a tabulky symbolů), fronta (pro nahrávání parametrů pro volané funkce), zásobník (syntaktická analýza zdola nahoru).

3.7 Požadavky předmětu Algoritmy

Součástí tohoto projektu byla mj. povinná implementace 2 algoritmů pro předmět IAL. První pro řazení řetězce a druhá pro vyhledávání podřetězce v řetězci.

3.7.1 Řazení: Heap Sort

Pro implementaci řadící funkce jsme si zvolili metodu heap sort. Využili jsme přitom studijní opory k předmětu IAL [2]. Tato metoda se vyznačuje asymptotickou časovou složitostí $O(N \log N)$ a využívá datové struktury halda. Algoritmus pracuje ve dvou krocích.

V prvním kroku dojde k přeskupení prvků řetězce, a tím k vytvoření haldy. Haldu si můžeme představit jako binární strom, který musí splňovat tyto pravidla. Pro otce platí, že má syny na indexech 2n+1 a 2n+2 (přičemž n je index otce), které mají menší hodnotu prvků než hodnota prvku otce.

V druhém kroku je z haldy opakovaně prováděno prohození mezi prvkem na indexu 0 a aktuálně posledním prvkem neseřazené části řetězce. Vzhledem k porušení haldy na indexu 0, je třeba provést opětovnou rekonstrukci haldy, která bude kratší o právě seřazený prvek a bude opět splňovat všechny podmínky haldy.

Funkci heap sort lze zavolat pomocí vestavěné funkce `sort_string`.

3.7.2 Vyhledávání podřetězce: Knuth-Moris-Prattův algoritmus

Pro implementaci vyhledávací funkce podřetězce v řetězci jsme si zvolili metodu Knuth-Moris-Prattův algoritmus. Při implementaci jsme postupovali podle studijní opory k předmětu IAL [2].

Tento algoritmus pracuje mnohem efektivněji než klasické porovnávání. Tato efektivita je zajištěna absencí porovnávání znaků, které již byly porovnány. Z pohledu implementace je zde zajímavé využití konečného automatu (zde pojmenovaného jako vektor).

Vektor obsahuje informace, se kterým znakem vzoru (podřetězce) se má porovnávat aktuální znak řetězce při neúspěšném porovnání.

V průběhu vyhledávání mohou nastat tyto 3 stavy:

- Pokud došlo ke shodě znaků a zároveň jsme na konci vzoru, tak bylo vyhledávání úspěšné. Funkce vrací index na první shodný znak v řetězci.
- Pokud jsme na konci řetězce a zároveň nejsme na konci vzoru, tak bylo vyhledávání neúspěšné. Funkce vrací hodnotu -1.
- Pokud nejsme na konci vzoru a zároveň nejsme na konci řetězce, vyhledávání pokračuje.

Tento algoritmus je možné použít zavoláním vestavěné funkce `find_string`.

3.8 Testování

Testování probíhalo na základě námi vytvořených automatických testů, které kombinovaly výpisy interpretu, kód chyby a statistické údaje z programu valgrind (počet úniků v paměti a počet chyb).

Nalezené odchylky ve výpisu bylo třeba manuálně prozkoumat a opravit. Tento způsob testování nám pomáhal v rámci konečných úprav projektu, kdy bylo potřeba nejdříve otestovat opravenou verzi na automatických testech. Teprve, kdy proběhlo vše v pořádku, mohl být upravený kód odeslán do pracovního repositáře.

Testování probíhalo na 64-bitových systémech Ubuntu 13.10 a Fedora 18(19).

4 Závěr

Tento projekt byl pro nás novou a jedinečnou zkušeností. Hodnotíme ho velmi pozitivně, především kvůli vyzkoušení si „praxe“ v oblasti týmového vývoje a komunikace.

Ovšem i nová látka z oblasti formálních jazyků a překladačů byla nejenom přínosná, ale také zajímavá, a některé členy týmu zaujala dokonce natolik, že se mu hodlají věnovat i v budoucím studiu.

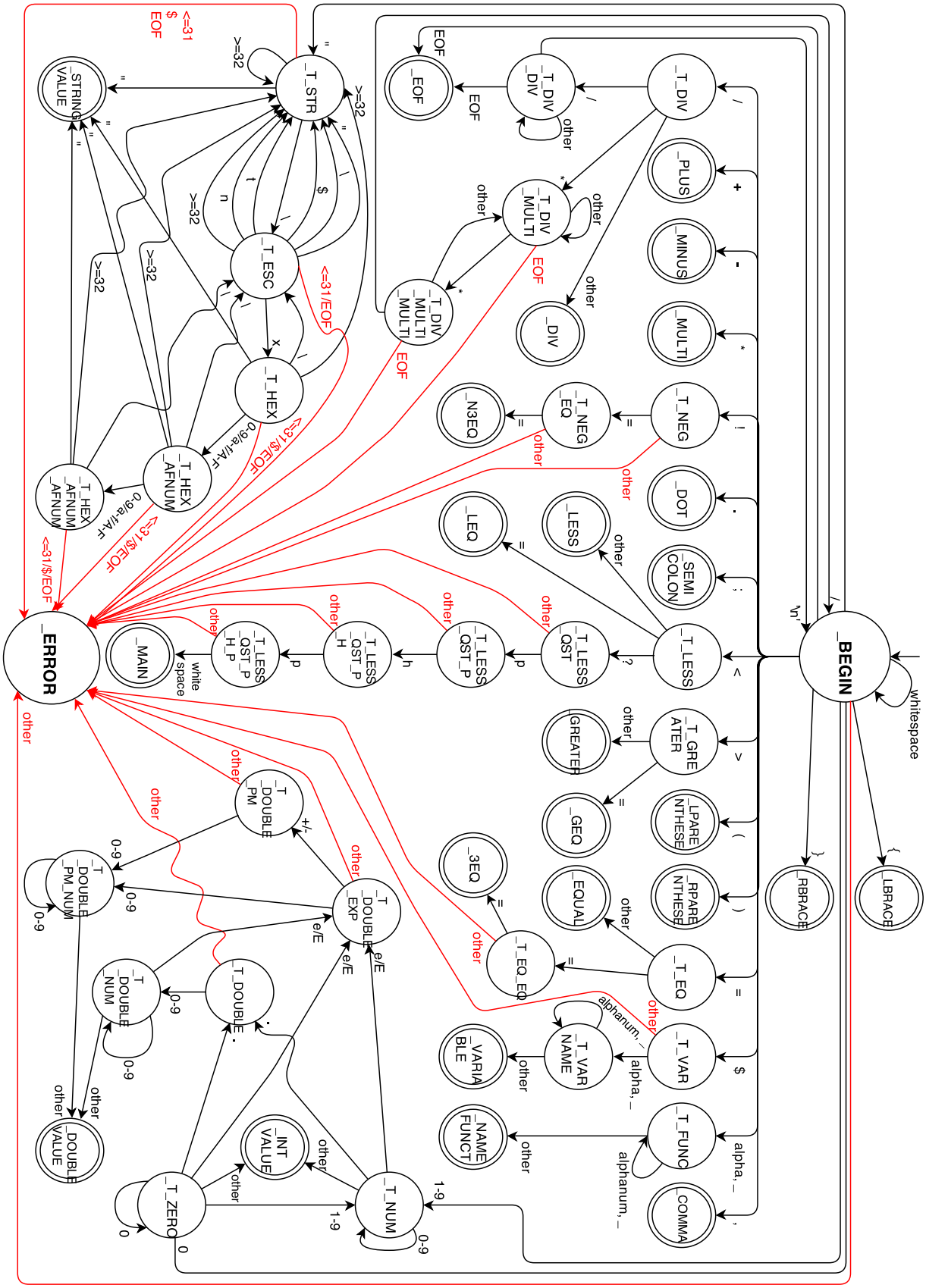
Reference

[1] Projekt č. 2, Předmět Jazyk C.

[2] Skripta k předmětu Algoritmy.

[3] SEDGEWICK, Robert. *Algoritmy v C*, Překlad Jiří Gree. Praha: SoftPress, 2003, 688s. ISBN: 80-86497-56-9.

A Konečný automat lexikálního analyzátoru



Obrázek 1: Konečný automat lexikální analýzy

B LL gramatika

$< PROGRAM >$ ⇒ _MAIN $< LST >$ _EOF

$< LST >$ ⇒ $< BODY >< LST >$
$< LST >$ ⇒ ϵ
$< LST >$ ⇒ FUNCTION ID ($< PARAMS >$){$< BODY >$} $< LST >$

$< BODY >$ ⇒ IF ($< EXPR >$){$< BODY >$} $< BODY >$
$< BODY >$ ⇒ WHILE ($< EXPR >$){$BODY >$} $< BODY >$
$< BODY >$ ⇒ RETURN $< EXPR >;< BODY >$
$< BODY >$ ⇒ ϵ
$< BODY >$ ⇒ ID = $< ASSIGN >;< BODY >$
$< BODY >$ ⇒ _FUNCNAME ($< ARGS >$);$< BODY >$
$< BODY >$ ⇒ CONTINUE;$< BODY >$
$< BODY >$ ⇒ BREAK;$< BODY >$
$< BODY >$ ⇒ FOR ($< FORASSIGN >;< EXPR >;< FORASSIGN >$)
{$< BODY >$} $< BODY >$

$< ASSIGN >$ ⇒ $< EXPR >$
$< ASSIGN >$ ⇒ _FUNCNAME ($< ARGS >$)

$< PARAMS >$ ⇒ $< PARFIRST >< PARNEXT >$
$< PARFIRST >$ ⇒ ϵ
$< PARFIRST >$ ⇒ ID
$< PARNEXT >$ ⇒ ϵ
$< PARNEXT >$ ⇒ , ID $< PARNEXT >$

$< ARGS >$ ⇒ $< ARGFIRST >< ARGNEXT >$
$< ARGS >$ ⇒ ϵ
$< ARGSFIRST >$ ⇒ ID
$< ARGSFIRST >$ ⇒ VALUE
$< ARGSNEXT >$ ⇒ ϵ
$< ARGSNEXT >$ ⇒ , ID $< ARGNEXT >$
$< ARGSNEXT >$ ⇒ , VALUE $< ARGNEXT >$

$< VALUE >$ ⇒ INT | DOUBLE | STRING | BOOL | NULL
$< FORASSIGN >$ ⇒ ID =$< EXPR >$

C Metriky kódu

Počet souborů: 28
Počet řádků zdrojového textu: 7182 řádků
Velikost statických dat: 126 038 B
Velikost spustitelného souboru: 81 310 B (systém Linux, 64 bitová architektura, při překladu bez ladících informací)