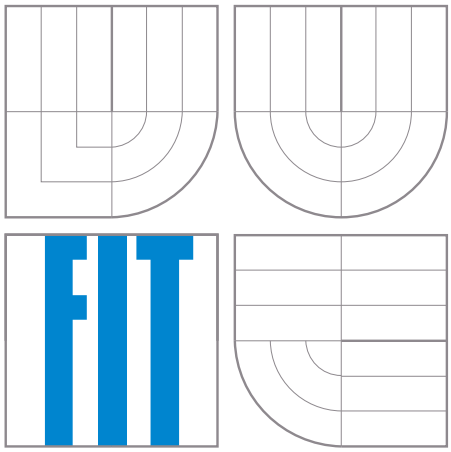
Vysoké Učení Technické v Brně
Fakulta Informačních Technologií



Dokumentace k projektu pro předměty IAL a IFJ

Implementace interpretu imperativního jazyka IFJ09

Tým 6, varianta a/4/II

13. prosince 2009

Autoři:	Bosternák Balázs	xboste00	25%
	Hlaváč Jan	xhlava18	25%
	Hrdina Pavel	xhrdin05	25%
	Chmiel Filip	xchmie02	25%
	Jelínek Tomáš	xjelin20	0%

Obsah

1 Úvod **1**

2 Organizace projektu **1**

2.1 Verzovací systém Mercurial 1

2.2 Vedení projektu 1

3 Návrh řešení **2**

3.1 Struktura programu 2

3.2 Lexikální analyzátor 2

3.3 Syntaktický analyzátor 2

3.4 Interpret 3

4 Implementace **3**

4.1 Tabulka symbolů 4

4.2 Řazení 4

4.3 Vyhledávání 4

4.4 Optimalizace 4

5 Závěr **5**

A Struktura konečného automatu, který specifikuje lexikální analyzátor **6**

B Metriky kódu **7**

Kapitola 1

Úvod

Při studiu na vysoké škole jsme se setkali už s několika programovacími jazyky jako například C nebo Pascal. Zápis nějakého algoritmu pomocí jednoho z těchto jazyků je však bez překladače, v našem případě spíše interpretu, jen pouhým textem. Právě interpret nebo překladač jej umožňují převést na skutečnou činnost. Cílem tohoto projektu je návrh a implementace interpretu jazyka IFJ09. V následujících kapitolách dokumentace se seznámíme s jednotlivými částmi interpretu a jejich návrhem tak, abychom ve výsledku byli schopní interpretovat jakýkoliv algoritmus správně zapsaný jazykem IFJ09. V dokumentaci je rovněž popsáno jak jsme při vývoji interpretu postupovali a jakých technologií jsme využívali.

Kapitola 2

Organizace projektu

V této části dokumentace se seznámíme s požitým verzovacím systémem, dále nastíníme rozvržení práce v týmu a s tím spojená úskalí.

2.1 Verzovací systém Mercurial

Rozsah projektu a možnost spolupráce jednotlivých členů týmu na různých úlohách vyžadovala použití některého SCM (Source Code Management) systému. Možností, který SCM software použít, je nespočet. Mezi nejznámější patří např.: CVS, Subversion, BitKeeper či Monotene. My jsme zvolili systém Mercurial hostovaný serverem bitbucket.org. Tento server poskytuje kvalitní webové rozhraní, které umožňuje zobrazit změny provedené na jednotlivých souborech, zobrazit seznam posledních úprav či vypsat aktivity jednotlivých členů týmu. Funkce pro soupis problémů, wiki a některé další funkce jsme vzhledem k velikosti našeho projektu nevyužili. Pro práci rozsáhlejších projektech se ale zajisté hodí.

2.2 Vedení projektu

Práce v týmu z počátku nezpůsobovala příliš mnoho problémů, jelikož tři členové spolu již úspěšně spolupracovali na projektech v minulém semestru. Velice užitečné pro nás byly prostory fakultní knihovny, konkrétně zasedací místnosti, kde se studenti mohou scházet a nikým nerušeni zde společně pracovat. V tomto ohledu se dá říci, že jsme aplikovali některé principy extrémního programování, jako například párové programování, častá revize kódu, neustálé testování a implementace jen těch části programu, které jsou v danou chvíli potřebné. S blížícím se termínem odevzdání se však vyskytl problém v rámci komunikace a spolehlivosti jednoho z nových členů týmu. Toto dospělo až do stavu, kdy jsme museli rozvržení práce přerozdělit a dohnat tak ztrátu vzniklou nezodpovědností člověka, který nás několikrát ujišťoval, že na zadaném úkolu pracuje.

Kapitola 3

Návrh řešení

V následující kapitole postupně probereme to, jak jsme interpret koncipovali jako celek, dále probereme také jeho základní části. Jsou jimi lexikální analyzátor založený na deterministickém konečném automatu, syntaktický analyzátor jehož jádrem je LL-gramatika a interpret.

3.1 Struktura programu

Jako jádro překladače jsme zvolili modul syntaktické analýzy, probíhá tak tedy syntaxí řízeny překlad. Syntaktická analýza využívá modul lexikální analýzy, který vždy načte nový token a případné výsledky ukládá do tabulky symbolů. Syntaktický analyzátor poté provede ještě sémantickou kontrolu a pokud se nevyskytnou žádné chyby, vygeneruje tří-adresný kód. Dále naše implementace využívá pomocné moduly pro práci s nekonečně dlouhými řetězci, operace se zásobníkem a operace nad jednosměrně vázaným seznamem, do kterého se ukládá výsledný tří-adresný kód.

3.2 Lexikální analyzátor

Základem lexikálního analyzátoru je konečný automat, který přesně definuje jeho chování. Struktura konečného automatu, který specifikuje náš lexikální analyzátor je uvedena v příloze A. Mezi dvě nejobtížnější části patří příjem čísla typu `double` a příjem řetězce včetně escape sekvencí. Pro správnou funkci analyzátoru bude však některých případech nutné použít funkci `ungetc()`, která vrátí načtený znak do vstupní fronty. Jako příklad uveďme třeba token < . Pro jeho odlišení od <= budeme muset načíst další znak, který je však potřeba vrátit zpět protože je součástí dalšího tokenu.

3.3 Syntaktický analyzátor

Dle zadání se syntaktický analyzátor dělí na dvě části. Hlavní syntaktická analýza probíhá pomocí rekurzívního sestupu pro kontext jazyka založeného na LL-gramatice. Výrazy zpracovává pomocí precedenční syntaktické analýzy. LL-gramatika na které je založen náš syntaktický analyzátor má následující tvar:

G = (N, T, P, S) ,kde:

N = { **<program>**, **<declrList>**, **<declare>**, **<dataType>**, **<stateList>**, **<stateList2>**, **<state>**, **<value>**, **<idList>**, **<idList2>**, **<exprList>**, **<exprList2>**, **<expression>**, **<strParam>** }

T = { begin, do, double, else, end, find, if, integer, readln, sort, string, then, var, while, write, (,), ., ,, :, ;, :=, id, eof, t_int, t_double, t_string, $ }

S = **<program>**

```
P =  { <program>    → <declrList> begin <stateList> end . eof
       <declrList>  → var id : <dataType> ; <declare>
       <declrList>  → ε
       <declare>    → id : <dataType> ; <declare>
       <declare>    → ε
       <dataType>   → integer
       <dataType>   → double
       <dataType>   → string
       <stateList>  → <state> <stateList2>
       <stateList>  → ε
       <stateList2> → ; <state> <stateList2>
       <stateList2> → ε
       <state>      → readln ( <idList> )
       <state>      → write ( <exprList> )
       <state>      → if <expression> then <state> else <state>
       <state>      → while <expression> do <state>
       <state>      → id := <value>
       <state>      → begin <stateList> end
       <value>      → <expression>
       <value>      → find ( <strParam> , <strParam> )
       <value>      → sort ( <strParam> )
       <idList>     → id <idList2>
       <idList>     → ε
       <idList2>    → , id <idList2>
       <idList2>    → ε
       <exprList>   → <expression> <exprList2>
       <exprList>   → ε
       <exprList2>  → , <expression> <exprList2>
       <exprList2>  → ε
       <strParam>   → id
       <strParam>   → t_string                                    }
```

3.4 Interpret

Prakticky se jedná o jednosměrně vázaný seznam který je rozšířený o adresu posledního prvku. Během tvorby tří-adresného kódu se instrukce vkládají postupně za poslední prvek. Poté až syntaktický analyzátor projde celý vstupní zdrojový kód a nenarazí na chybu, spustí se interpretace vytvořeného pomocného kódu. Princip je založen na tom, že každá instrukce obsahuje název operace a tři adresy, kterých se tato operace týká. Poté co se instrukce vykoná, překladač se přesune na další prvek seznamu. Takto pokračuje, dokud nenarazí na ukončovací operaci, v tom okamžiku interpret skončí a nahlásí, že vše proběhlo bez chyby. Trochu jinak se chová instrukce `GOTO` (instrukce pro skok), ta přesune ukazatel aktivního prvku na místo, kde je definované dané návěští.

Kapitola 4

Implementace

V této kapitole postupně rozebereme to, jak jsme implementovali jednotlivé části interpretu. Probereme zde konkrétně tabulku symbolů, algoritmus řazení znaků v řetězci a algoritmus pro vyhledávání podřetězce v řetězci. Závěrem se zmíníme o optimalizacích, které jsme provedli v rámci snahy dosáhnout co nejvyšší rychlosti našeho interpretu.

4.1 Tabulka symbolů

V našem řešení využíváme pro ukládání symbolů hash tabulku. Do této tabulky se ukládají proměnné, konstanty a štítky pro funkci `GOTO` používanou interpretem. Struktura jedné položky hash tabulky je následující:

- klíč
- typ uložených dat (integer, double, string popřípadě ukazatel na položku v seznamu tři-adresného kódu)
- samotná data
- ukazatel na další položku seznamu

Při volání funkce pro vložení proměnné do tabulky symbolů se používá jako klíč její název, u konstant se generuje klíč začínající znakem $, u štítku se znakem #. Klíče u konstant a štítku se využívá pouze během vkládání do tabulky, při práci s nimi se používají odkazy přímo na data v tabulce. Během interpretace vnitřního kódu se také využívají odkazy do hash tabulky, aby se nemusela neustále využívat funkce pro hledáni položky v hashi. Funkce pro hledání se používá pouze jako kontrola deklarace proměnných. Kontrolu inicializace není nutné provádět, jelikož při vkládání do tabulky se automaticky čísla integer a double inicializují na 0 a string se inicializuje na prázdný řetězec.

4.2 Řazení

Zadáním jsme měli jako řadící algoritmus předepsaný Merge sort. Jedná se o rychlý algoritmus s logaritmickou časovou složitostí. Jeho fungování je založeno na vytvoření pomocné pole o dvojnásobné velikosti než je rozměr řazeného. Do první poloviny pomocného pole se vloží vstupní data. Poté postupně odebírá z obou stran neseřazeného řetězce nejdelší neklesající posloupnosti. Tyto dvě posloupnosti během ukládání na cílové místo, čili druhou polovinu pole, seřadí. Pozice ukládání se střídavě mění z počátku cílové části pole nakonec cílového pole. Tak postupuje dokud nedojdou posloupnosti v jedné polovině, poté se úloha zdrojové části a cílové zamění. Celý tento cyklus se opakuje, dokud ve zdrojové části pole nezbude jedna neklesající posloupnost. Samotná implementace je řešená pomocí jedné funkce `strSort()`. Ta provádí řazení za pomoci cyklů pro procházení jednotlivých části pole, podmínek pro vyhodnocování směru čtení a algoritmu pro řazení neklesajících posloupností

4.3 Vyhledávání

Dle zadání jsme měli použít Knuth-Morris-Prattuv algoritmus. Principiálně se jedná o konečný automat, který využívá informaci o částečné shodě hledaného řetězce aby se nemusel pokaždé vracet. Vždy pouze vhodně nastaví posun v hledaném řetězci. V praxi se pro vyhledávání volá funkce `strFind()`, ta nejprve za pomoci funkce `calculateShift()` spočítá o kolik se bude posouvat index v případě neúplné shody (tyto informace se uloží do pomocného pole). Poté vyhledávací funkce prochází řetězec a neúplné shody přeskakuje podle pomocných informací v tabulce.

4.4 Optimalizace

Pro urychlení běhu interpretu jsme zvolili postup přímých přístupů do hash tabulky, tímto se zbavíme časové prodlevy během vyhledávání dat v tabulce. V rámci syntaktické analýzy a jejího zpracování proměnných je důležitý počet řádků hash tabulky který ovlivňuje rychlost vyhledávání proměnných podle klíče v ní. Když se zvolí dostatečně vysoký počet řádků tabulky, tak bude na každý řádek připadat menší počet položek seznamu. Právě při hledání požadované položky v seznamu dochází k největším časovým ztrátám. Řádek tabulky symbolů se při hledání určí v jenom kroku, vysoký počet řádku tedy nezpomaluje vyhledávací funkci, ale má za následek mírné zvýšení paměťových nároků hash tabulky. Z tohoto důvodu jsme zvolili hash tabulku o velikost 100 000 řádků.

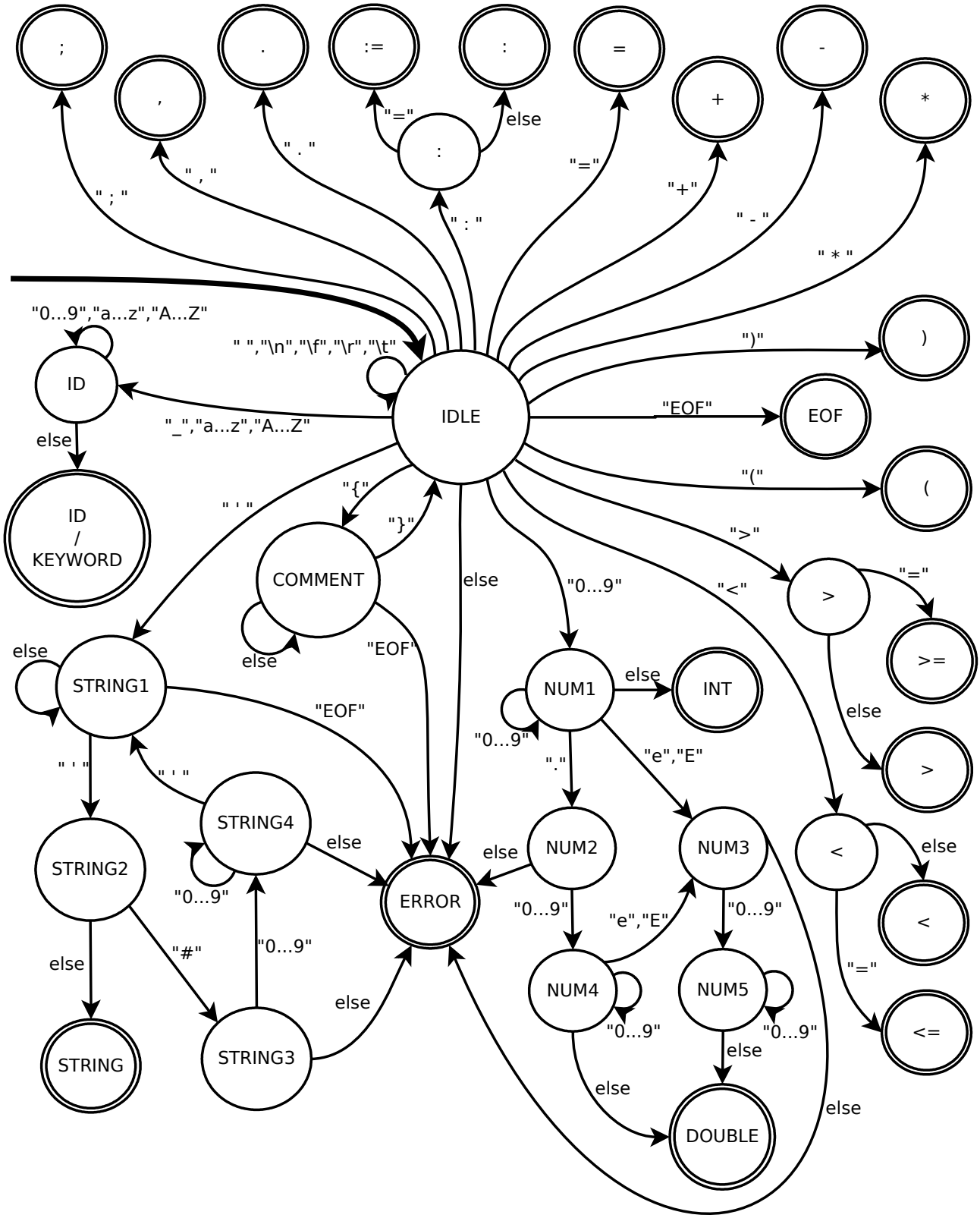
Kapitola 5

Závěr

Tento projekt, i přesto že se jednalo o značně zjednodušenou verzi překladače, skrýval problematiku navrhování softwaru ve velkých firmách. Během vývoje jsme se naučili řešit problémy s komunikací v týmu, rozvržení práce a následovné sjednocování výsledků návrhu. Těžko se ovšem dalo předejít problému s odmítavým postojem k práci, kterého se nám dostalo od nového člena týmu. Ale i přes tuto komplikaci se dá říci, že se nám podařilo zdárně implementovat plně funkční interpret jazyka IFJ09. Bohužel už nám nezbylo mnoho času na implementaci různých rozšíření, na druhou stranu jsme tohoto však využili a pokusili se interpret optimalizovat pro co nejvyšší rychlost.

Příloha A

Struktura konečného automatu, který specifikuje lexikální analyzátor



Příloha B

Metriky kódu

Počet souborů: 15 souborů

Počet řádků zdrojového textu: 3509 řádků

Velikost statických dat: 1908 B

Velikost spustitelného souboru: 39018 B (systém Linux, 64 bitová architektura, při překladu pomocí Makefile)