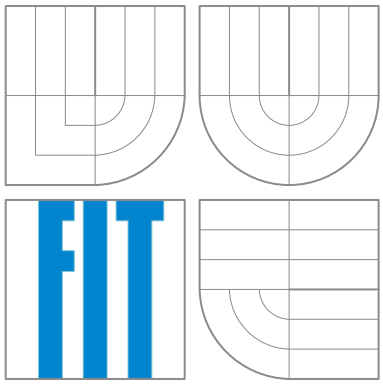
Vysoké učení technické v Brně
Fakulta informačních technologií



Dokumentace k projektu do předmetu IFJ a IAL

Interpret imperativního jazyka IFJ12

Tým 014, varianta b/3/II

9. prosince 2012

Autoři :

Tomáš Varga, xvarga10 20%

Jakub Vojvoda, xvojvo00 20%

Adam Warzel, xwarze00 20%

Jaroslav Španko, xspank00 20%

Tomáš Nesvadba, xnesva02 20%

Obsah

1 Úvod

Dokumentace se zaobírá návrhem, implementací a samotným vývojem interpretu pro imperativní jazyk IFJ12. Jazyk IFJ12 je podmnožinou jazyka Falcon, což je moderní skriptovací jazyk. Dokumentace je rozdělená do několika kapitol a příloh, které popisují způsob vývoje interpretu pro jazyk IFJ12. Postupně jsou vysvětleni jednotlivé části interpretru, implementace, použité algoritmy v projektu.
Závěr patří pohledu na funkčnost, úspěšnost a zhodnocení projektu.
Příloha obsahuje kromě metriky kódu a použité literatury také LL gramatiku a strukturu konečného automatu.

2 Implementace

2.1 Lexikálni analyzátor

2.1.1 Návrh a implementace lexikálního analyzátoru

Princip lexikálního analyzátoru spočívá v načtení zdrojového programu, v našem případe v jazyku IFJ12 po znacích a následné rozdělení na jednotlivé lexémy, které jsou reprezentováni tokeny. Jednotlivé stavy konečného automatu jsou reprezentovaní datovým typem výčet, které obsahuji atributy.
Samotná implementace lexikálního analyzátoru spočívá v navrhnutí deterministického konečného automatu a jeho implementaci.
Na chod lexikálního analyzátoru má přímý vliv syntaktický analyzátor, tj. lexikální analyzátor pracuje jenom když si to vyžádá syntaktický analyzátor. Výstupem lexikálního analyzátoru je vždy jeden token.
Výsledná struktura konečného automatu viz příloha B a implementace lexikálního analyzátoru v souborech lex.c a lex.h .

2.1.2 Popis struktury konečného automatu pro lexikální analyzátor

Lexikální analyzátor tvoří deterministický konečný automat. Stavy, z kterých pozůstává konečný automat dělíme do dvou kategorií.
První část tvoří tzv. ukončující stavy, tj. stavy, u kterých získáme token hned po přečtení prvního znaku(+,-,*,/,.).
Druhou část tvoří stavy, u kterých je pro získání tokenu potřebné přijmout víc jak jeden znak. Např. token = může reprezentovat vícero stavů, které je nutné odlišit. Je potřeba načítat další znak pomoci funkce `T_StringAppend` .

V některých případech, jako např. při zmíněném = využívame funkci `ungetc()` , která v případě, že znak = je samostatný token, tedy vyjadřuje rovnost, vrátí načtený znak do fronty. Znak musíme vrátit do fronty, protože je součásti dalšího tokenu.
V případe = platí, že daný znak může vyjadřovat jak rovnost(=), tak i přiřazení(==).
Takto postupujeme u všech přípustných stavu jazyka IFJ12.

2.2 Syntaktický analyzátor

2.2.1 Návrh a implementace syntaktického analyzátoru

Princip syntaktického analyzátoru spočívá v přijímaní tokenů z lexikálního analyzátoru. Analyzátor spolupracuje jak s lexikálním , tak i se sémantickým analyzátorem. Implementace syntaktického analyzátoru spočívala ve využití LL gramatiky, která je hlavním a nejdůležitějším prvkem syntaktické analýzy.
Pro zpracovaní tokenů existují dva způsoby. Prvním je rekurzívní způsob založený na LL gramatice, kterou si rozebereme v následující kapitole. Druhým způsobem je precedenční syntaktická analýza. Pro syntaktickou analýzu byla využitá metoda rekurzívního sestupu.

2.2.2 Popis LL-gramatika syntaktického analyzátoru

Protože výstupem lexikálního analyzátoru je právě jeden token, principem gramatiky je ověřit tento token a zjistit, co představuje.
V případě, že načtený token není je jednoznačný, tj. nemůžeme určit, co reprezentuje, syntaktický analyzátor "požádá" lexikální analyzátor o další a proces se opakuje, dokud není je jednoznačné o jaký jde.
Tento token není vrácený zpět. Jednotlivé možnosti LL-gramatiky se nacházejí v prílohe A.

2.2.3 Vyhodnocování a analýza výrazů

Tento oddíl tvoří podstatnou část syntaktického analyzátoru. Pracuje s jednosměrně vázaným seznamem, jehož datový obsah tvoří operand uložený v typu `T_Var`, operace která s ním bude prováděna zleva a počítadlo závorek, které se inkrementuje či dekrementuje v závislosti na závorkách, takže slouží k detekci zanoření a kontrole.
Tento list je generován funkcí `gener_list`, která načítá a zpracovává příchozí tokeny (do vnitřního typu `T_Var`), v případě funkce se použije pro její zpracování, načtení argumentů `CallFunction`. Při načítání řetězce probíhá syntaktická kontrola.
V případě, že seznam obsahuje více prvků, tudíž jde o výraz s operacemi, je volána funkce evaluate, která vypočítává hodnoty v postupně v jednotlivých zanořeních. Jedná se 4 průchody podle priorit operátorů.
Probíhá zde částečná sémantická proměnná, nejsou v ní kontrolovány datové typy proměnných, které v této fázi ještě nemusejí být známy.

2.3 Sémantický analyzátor

2.3.1 Návrh a implementace sémantického analyzátoru

Sémantický analyzátor spolupracuje se syntaktickým analyzátorem. Přijíma od syntaktického analyzátoru derivační strom a jeho výstupem je abstraktní syntaktický strom.
Poněvadž předem neznáme jednotlivé datové typy, což vyplývá z jazyka IFJ12, probíhá sémantická kontrola až při generování kódu.

2.4 Interpret

Interpret je reprezentován jednosměrně vázaný lineární seznam, který obsahuje jednotlivé adresy funkci a číselný identifikátor instrukcí(IID). Funkce interpretu je ovlivněna funkčností syntaktického analyzátoru. V případe, že syntaktický analyzátor nezaznamená chybu, spouští se interpret. Cinnost interpretu spociva v posouvani prvku seznamu, teda v pripade, kdy se instrukce vykona se posune prvek seznamu. Toto se opakuje pokym interpret nenarazi na konec.

3 Rešení vybraných algoritmů z pohledu predmetu IAL

3.1 Řadící algoritmus

Pro věstavěnou funkci Sort řadící znaky v řetězci podle ordinální hodnoty jsme se rozhodli použít Shellův řadící algoritmus založený na bublinkovém vkládacím principu. Tato metoda prochází a řadí prvky po kroku s určitou velikostí, který se postupně zmenšuje.
Jako základní krok jsme použili n/2, který se potom zmenšuje vždy o polovinu. Jedná se o nestabilní metodu.
Přesné stanovení časové náročnosti je problémem, dle skript předmětu IAL je ovšem rychlejší než Heapsort a pomalejší než Quicksort. Seřazený řetězec vracíme vnitřně pomocí druhého parametru funkce ve vnitřním datovém typu `T_Var`, zastupujícím datové typy.
Návratová hodnota naší funkce je pouze 0/false, pokud vše proběhlo bez chyb a hodnota `TYPE_COMPATIBILITY_ERROR`, pro chybu, týkající se datového typu vstupního parametru `read_str`.
Stejně jsou řešeny i další funkce. Vnitřní chybové kódy jsou uloženy v souboru codes.h.

3.2 Vyhledání podřetězce v řetězci

Zde jsme použili Boyer-Mooreuv algoritmus, konkrétně podle první heuristiky. Použitá funkce `computeJumps` funguje tak, že pro každý znak(index) vypočítá hodnotu o kolik lze skočit v případě jeho nenalezení a uloží je do pole pro pozdější použití pomocí dvou for cyklů.
Samotná funkce má potom název Find, ta prohledává v cyklu pole zleva do prava (od indexu rovnajícího se délce vzorku-1), pokud narazí na hledaný poslední znak vzorku, prochází (zprava doleva) předcházející znaky a v případě neshody se posouvá o počet znaků vypočítaný ve fci `computeJumps`.
Funkce předá hodnotu indexu prvního znaku do třetího parametru a vrátí úspěch `(EXIT_SUCCESS)` v případě nalezení, jinak vrací neúspěch.

3.3 Tabulka symbolů

Dle zadání II se jedná o hashovací tabulku. Tato tabulka je implementována jako tabulka zřetězených seznamů. Implementace hashovací tabulky byla také předmětem jednoho z úkolů předmětu IAL, což práci poměrně usnadnilo, bylo možné použít tabulku z tohoto úkolu pro naše účely.
Před každým vstupem do tabulky, je potřeba vypočítat klíč, index tabulky na který se bude přistupovat, toto má na starosti funkce `KeyCreate`. Využívá bitový shift řetězce o jeho poloviční délku, tato hodnota je potom XOR-ována konstantou násobenou poloviční délkou (z důvodu jedinečnosti klíče) k hodnotě je potom připočítána hodnota posledního znaku a bitový shift klíče 1, nebo 2 podle toho, zda je délka řetězce sudá či lichá.
Funkce vrací klíč modulo velikost tabulky, která je potřebná (podle konstanty.

Vkládání do tabulky:

1. položka s klíčem ještě nepoužitým je zařazena na začátek seznamu příslušného indexu
2. položka se stejným klíčem a stejným řetězcem - obsah původní položky je aktualizován
3. položka se stejným klíčem, ale jiným řetězcem - zařazena na začátek seznamu

Kromě výše zmíněných algoritmů jsme znalosti z předmětu IAL využili ještě při tvorbě vnitřních abstraktních datových typů pro překladač jako jsou seznam, zásobník nebo binární strom a hlavně při návrhu a implementaci funkcí pro práci s nimi.

4 Práce v týmu

Hned po vytvoření naší skupiny jsme se dohodli, že získané body si rozdělíme rovnoměrně a všichni se budeme snažit podílet na projektu stejně. Rozdělení práce práce bylo hierarchické, vůdce naší skupiny Tomáš Varga, někdy s pomocí Jakuba Vojvodu, navrhl strukturu jednotlivých částí projektu, napsal kostru a rozdělil funkce mezi členy týmu, kteří je následně implementovali.
Například při vytváření modulu obsahujícího naše vnitřní typy každý člen pracoval na funkcích vázaných k jednomu typu. Zdrojové kódy a ostatní soubory jsme sdíleli přes systém SVN.
Toto se týká hlavně výše zmíněného modulu s vnitřními typy types.c, builtin.c v němž se nacházejí vestavěné funkce a ještě synan.c obsahující syntaktický analyzátor. Na těchto částech se podíleli poměrně rovnoměrně všichni členové. Na části ial.c pracoval Jakub Vojvoda, lexikální analyzátor (lex.c) implementoval podle návrhu Tomáš Nesvadba, který také pracoval na části syntaktického analyzátoru zabývající se výrazy. interpret.c vytvořili Tomáš Varga, Jakub Vojvoda a Adam Warzel.
Zbytek programová části, jako například použití a sladění modulů, je práce vedoucího skupiny Tomáše Vargy s pomocí Jakuba Vojvodu.
Dokumentaci projektu napsali Jaroslav Španko a Adam Warzel.

5 Závěr

Projekt byl díky dobré organizaci rozdělený velmi dobře a pracovat se začalo hned po zveřejnění zadání.
Postupem času se měnil nejen pohled na projekt ale nastaly i změny oproti původnímu návrhu. Zajímavostí je počet revizí na svn - 160.
V rámci projektu existovala možnost pokusných odevzdání které jsme využili. Tato možnost nám pomohla rychleji odhalit a detekovat chyby. I přes všechny problémy tykající se časové náročnosti se nám podařilo implementovat funkční interpret jazyka IFJ12. Řešení projektu bylo otestováno na obou architekturach Linuxu (32 a 64 bit).
Projekt byl zajímavou a přínosnou zkušeností pro cely tým do budoucna.

A Metriky kódu

Počet souborů: 20

Počet řádků zdrojového textu: 5113 řádků

Velikost statických dat: 140kB

Velikost spustitelného souboru: 450kB

B LL gramatika

Implementace LL gramatiky

$< synan >$	$->$	<stat> <synan>
$< synan >$	$->$	EOF
$< stat >$	$->$	<keyword>
$< stat >$	$->$	id = <expr>
$< stat >$	$->$	func_id(<item> <it-list>
$< keyword >$	$->$	if <expr> <if>
$< keyword >$	$->$	while <expr> <end>
$< keyword >$	$->$	function id(id <it-list>
$< keyword >$	$->$	function id() EOL
$< expr >$	$->$	<item> + id|number|string|func_id(<item> <it-list>
$< expr >$	$->$	<item> -|*|/|** id|number|func_id(<item> <it-list>
$< expr >$	$->$	<item> EOL
$< expr >$	$->$	(<expr>
$< expr >$	$->$	) + |id|number|string|func_id(<item>|id <it-list>
$< expr >$	$->$	) -|*|/|** id|number|func_id(<item>|id <it-list>
$< expr >$	$->$	) EOL
$< item >$	$->$	id
$< item >$	$->$	number
$< item >$	$->$	bool
$< item >$	$->$	string
$< item >$	$->$	nil
$< it - list >$	$->$	, <item> <it-list>
$< it - list >$	$->$	) EOL
$< if >$	$->$	<stat2> <if>
$< if >$	$->$	else <end>
$< if >$	$->$	end EOL
$< end >$	$->$	<stat2> <end>
$< end >$	$->$	end EOL

C Konečný automat

Konečný automat pro lexikálni analyzátor