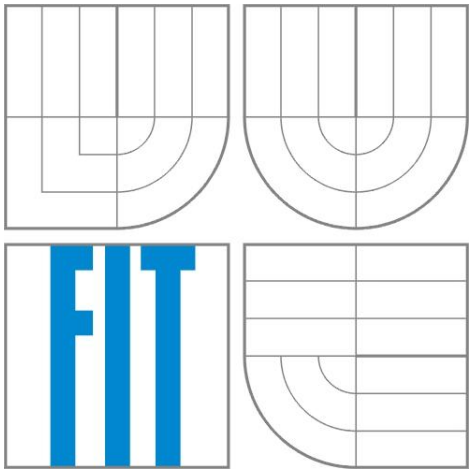
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentace k projektu do předmětů IFJ a IAL

Implementace interpretu imperativního jazyka IFJ11

Tým 9, varianta b/1/I

4. prosince 2011

Seznam autorů:
Bambušek David, xbambu02, 20%
Klvaňa Martin, xklvan00, 20%
Koňař David, xkonar07, 20%
Mareček Matěj, xmarec12, 20%
Szabo Peter, xszabo06, 20%

Výčet rozšíření:
VOIDCALL
LOCALEXPR

Obsah

1 Úvod

Tato dokumentace se zabývá vývojem a implementací interpretu pro imperativní jazyk IFJ11. Celá dokumentace je členěna do kapitol a příloh, které popisují růst a způsob kompletace dílčích části interpretu v jeho celek. Rozebrány jsou postupně všechny podstatné prvky našeho interpretu, specifické metody a algoritmy užité v našem programu.

Přidány jsou také přílohy obsahující LL-gramatiku a strukturu konečného automatu, neboť se jedná o zásadní prvky překladače. Na závěr dokumentace je obsaženo i shrnutí naší práce s hodnocením a pohledem na úspěšnost celého projektu.

2 Práce v rámci týmu

2.1 Příprava a time management

Na začátek dokumentace bychom rádi prezentovali postup práce našeho týmu při řešení projektu. Od prvních otázek, spíše formálního charakteru, až v následných fázích po ty technické, funkční ale i esterické otázky, ke kterým bylo nutno zaujmout určité stanovisko - a to na základě diskurzu. Nejlépe ke spokojenosti všech nebo, nebylo-li to možné, tak alespoň většiny.

Striktní časy ukončování jednotlivých fází našeho vývoje nebyly stanoveny a spoléhalo se na svědomitý přístup jednotlivých členů či případné napomenutí ze strany vedoucího naší skupiny. Projekt o podobném rozsahu jsme ještě doposud neměli možnost absolvovat, a tak pro nás bylo cennou zkušeností zjistit, že takovýto postup není zcela ideální - avšak stejně tak, není ani nemožný - a vývoj tedy probíhal v průběhu času různě intenzivně.

2.2 Schůzky a komunikace týmu

Prvotní ideou našeho týmu byly pravidelná skupinová setkání v plném zastoupení týmu, avšak v průběhu času se z důvodu odlišných časových dispozic členů uchýlilo k pouze několika zasedáním celého týmu a zbytek problémů se řešil jak prostřednictvím privátní internetové stránky, tak setkávání v rámci užšího počtu členů, kteří se danou problematikou zabývali. Případná rozhodnutí a další postupy se následně zveřejňovaly skrz stránky našeho týmu.

2.3 Systém správy zdrojového kódu

Nezbytným prvkem při práci v týmu je předávání a synchronizace aktuálních informací - v našem případě především zdrojových kódu či jiných poznámek. Jako médium jsme se rozhodli, po hlasování všech členů, využít tzv. *Source Code Management* (SCM) software. Konkrétně jsme zvolili Subversion a jako datové uložiště nám posloužil fakultní server. Takto vybaveni jsme následně mohli téměř kdekoli upravovat veškeré zdrojové kódy našeho programu a sdílet je mezi ostatními členy týmu.

2.4 Metodika vývoje

Třebaže tato otázka na začátku našeho projektu nebyla jasně a přesně určena, ubíral se vývoj projektu nejblíže k metodě *Rational Unified Process* (RUP). Naší snahou bylo postupné a průběžné vytváření částí komponent interpretu, které se po zpětné vazby ze strany ostatních členů týmu buď přepracovávaly či vývíjely dále v součinnosti s ostatními částmi. Častá zpětná vazba ostatních sice vedla často k četnému předělávání částí zdrojového kódu, avšak pomohla nám objevovat dílčí problémy spíše v zárodku než při kompletaci celého projektu. Rychlá syncrhonizace dat a přehled vývoje jednotlivých členů, tak zadal další pozitivní krok - přehled o již vyvinutých datových strukturách a funkcích, které tak již nemusely být vyvíjeny znovu jinými členy týmu avšak pouze upraveny pro jiná specifická využití v odlišných částech interpretu.

3 Podrobnější pohled na zadání a návrh implementace

Naším úkolem bylo naprogramovat interpret, pro tento účel vytvořeného, jazyku IFJ11. Práci na tomto projektu jsme rozdělili na několik dílčích částí (jednotlivé informace k jejich implementaci jsou v následující kapitole), které jsou spolu úzce propojeny a bez jakékoli z částí bychom nemohli zaručit funkčnost celku a výstup interpretu by nebylo možné považovat za validní. Práci jsme rozčlenili na tyto základní části:

- lexikální analyzátor
- syntaxí řízený překlad
  - syntaktický analyzátor
  - sémantický analyzátor
- generátor tříadresného kódu
- interpet
- vestavěné funkce interpretu
- datové struktury
- testování

Implementaci jednotlivých částí jsme si rozdělili a snažili se vytvořit vhodné datové typy aby nedocházelo k plýtvání paměti a redundanci dat.

4 Implementace

4.1 Věstavěné rozšíření

V našem projektu jsme implementovali tato rozšíření:

- voidcall - umožňuje volat funkce bez nutnosti přiřazení návratové hodnoty do proměnné
- localexpr - rozšíření které inicializuje proměné přiřazením výrazu.

4.2 Quick Sort

Pro vestavěnou funkci řazení - `sort` je dle specifik zadání uplatněn algoritmus Quick Sort. V porovnání s ostatními algoritmu patří mezi nejrychlejší, navíc nabízí možnost řazení tzv. *in-place* - tedy že pro přesun dat používá malou konstantní velikost paměti nezbytnou pro záměnu pozic dat. Pro jeho implementaci lze využít 2 způsoby: iterativní a rekurzivní. V našem případě jsme využili rekurzivního zápisu tohoto

algoritmu neboť nám přišel přehlednější. Nezůstali jsme však u pouhého základního algoritmu, avšak provedli jsme i optimalizaci.

Efektivnost tohoto řadícího algoritmu tkví v určení hranice mezi dvěma poli - často nazývaného jako *pivot* - kde na jedné straně pole obsahuje hodnoty menší a na straně druhé hodnoty větší než hraniční prvek. Tento prostřední prvek (pivot), může být určen náhodně, pevně či matematickým výpočtem. Právě poslední ze zmíněných možností nabízí prostor pro zrychlení metody řazení, a proto za účelem co pokud možnost největší efektivnosti jsme také implementovali tuto variantu. Tento hraniční prvek jsme tedy nedefinovali obecně u všech řazení na první či poslední prvek pole, nýbrž jako prostřední hodnotu pole vypočítanou pomocí aritmetického průměru krajních hodnot pole.

4.3 Boyer-Moore algoritmus

Věstavěná funkce `find` sloužící pro vyhledávání podřetězce v rámci řetězce je založena na algoritmu Boyer-Moore, který nabízí více než jeden typ heuristiky pro hledání shod mezi řetězcem a podřetězcem. Na základě těchto metod vyhledávání shodných, respektive neshodných znaků, je tento algoritmus schopen některé znaky naprosto přeskočit bez nutnosti jejich porovnání a tudíž i rychleji prohledat celý řetězec. Při implementaci algoritmu jsme se inspirovali informacemi z předmětu IAL (viz [1]).

4.4 Lexikální analyzátor

Lexikální analyzátor je konečný automat s epsilon přechody (ty slouží ke korektnímu rozpoznávání číselných konstant, které rozeznává pomocí mezistavů), který načítá znaky ze zdrojového programu a převádí je na tokeny. Tyto tokeny jsou reprezentovány abstraktním datovým typem, který obsahuje informace o řádcích, sloupcích a lexému. Uložené informace jsou nezbytné pro další činnost celého interpretu.

Tabulka rezervovaných a klíčových slov je realizována jako pole řetězců.

4.5 Syntaxí řízený překlad

Při implementaci syntaktického analyzátoru jsme využili LL gramatiku (viz. Příloha B), postupovali jsme metodou rekurzivního sestupu. Pro ukládání dat jse využili binární strom do kterého jsme ukládali veškerá data, se kterými později pracuje i interpret. Pro rozlišení rozdílných dat jsme využili prefixů pro lokální proměnné a pro pomocné proměnné interpretu. Jednotlivé typy identifikátorů jsou také dále specifikovány pomocí metadat uložených v rámci datové struktury uzlu.

Pro uchování parametrů předávaných u funkcí jsme vytvořili v rámci binárního stromu i jednosměrný lineární seznam. Abychom předešli kolizím v rámci binárního stromu, vytvořili jsme vlastní ověřovací funkce, které zajistí unikátnost jednotlivých názvů uzlů.

Pro zpracování výrazů využíváme i syntaktickou analýza zdola nahoru neboť samotná LL gramatika by byla nevhodná. Oba dva druhy analýz vzájemně kooperují avšak hlavní řízení je v režii syntaktické analýzy zhora dolů a v případě, že narazí na výraz zavolá se analýza zdola nahora.

Podstatou syntaktické analýzy zdola nahoru je precedenční tabulka, na jejímž základě jsou řízeny veškeré redukce a vyhodnocování výrazů. Ke správnému fungování syntaktické analýzy podle precedenční tabulky bylo potřeba využít 2 dynamických zásobníků.

4.6 Interpret a generátor kódu

Pro náš projekt jsme využili genetáro tříadresného kódu, používající, pro naše účely námi vytvořenou, instrukční sadu. Pro zachování přehlednosti kódu generátor rozlišuje mezi vytvářením plnohodnotných instrukcí a instrukcí skoku, resp. návěstí. Tyto instrukce se pro následné využití interpretu ukládájí do tabulky instrukcí, vedle které stojí, jako doplnění, také tabulka obsahující návěští pro skoky. Interpret zpracovává instrukce lineráně s výjimkou u instrukcí skoku či při návratů z funkcí, kde se dle daných parametrů mění číslo práv zadané instrukce. Při aritmetických operacích interpret plní funkci také sémantického analyzátoru, abychom předešli duplicitní kontrole dat.

4.7 Testování

Testování našeho interpretu probíhalo jak na základě automatizovaných skriptů, které měly porovnávat výstupní data s hodnotami výstupu interpretu jazuky Lua, které se snažily odhalit především logické chyby v naší interpretaci vstupního zdrojového kódu. Následné odchylky pak byly jednoduše vypsány do logovacího souboru pro jejich manuální kontrolu, zhodnoceny a delegovány k opravě členovi, který měl daný kus programu na starost. Pro specifické chybové výstupy a další potenciálně problémové části byly cíleně vytvořeny specifická data pro verifikaci naši implementace.

5 Závěr

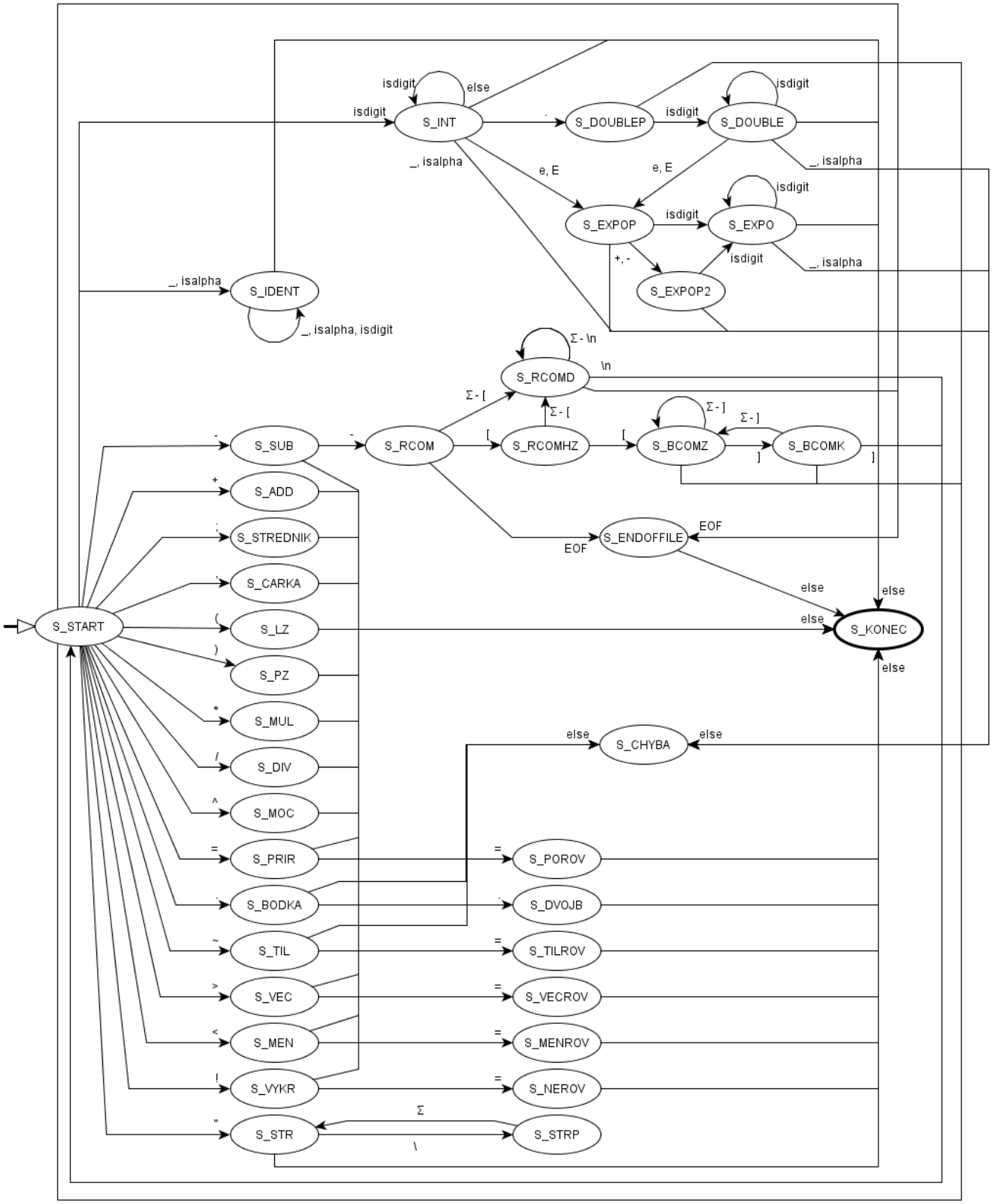
Implementace vlastního interpretu jazyka IFJ11 byl pro nás všechny výzvou, jejíhož rozsahu jsme se v rámci školních projektů ještě nikdo nesetkali. Aplikace teoreticky nabytých vědomostí, komunikaci mezi členy a dalších technických náležitosti jako sdílení dat v rámci týmu týče nás obohatil o nemalé zkušenosti a jako výsledek nelehké práce předkládáme funkční a námi otestovaný interpret jazyku.

Naše řešení bylo testováno na více operačních systémech a to jak Microsoft Windows 8 (Developer Preview), 7 i XP, tak i na Linux Mint 12, Xubuntu 11.10, Ubuntu 11.04. Testování bylo provedeno na 32 i 64-bitových architektuách.

6 Použité zdroje

[1] Skripta k předmětu IAL

A Konečný automat lexikálního analyzátoru



B LL gramatika

PROGRAM → FUNC ; eof
FUNC→ eps
FUNC→ function id (PARAMS) VARDEC LIST end FUNC

PARAMS→ PARAMSFIR PARAMSNEXT
PARAMSFIR→ id
PARAMSFIR→ eps
PARAMSNEXT→ eps
PARAMSNEXT→ , id PARAMSNEXT

VARDEC→ eps
VARDEC→ local id IDT VARDEC
IDT→ PRIRAZENI
IDT→ ;

LIST→ eps
LIST→ id PRIRAZENI LIST
LIST→ write WRITE LIST
LIST→ if EXPR then LIST else LIST end ; LIST
LIST→ while EXPR do LIST end ; LIST
LIST→ return EXPR ; LIST

VEST→ substr (ARGS) ;
VEST→ find (ARGS) ;
VEST→ type (ARGS) ;
VEST→ sort (ARGS) ;

PRIRAZENI→ = LITEXPR

LITEXPR→ EXPR ;
LITEXPR→ VEST
LITEXPR→ READ
LITEXPR→ id (ARGS) ;

ARGS→ LITFIR ARGSNEXT
LITFIR→ eps

LITFIR→ LIT
LIT→ id
LIT→ numb
LIT→ str
LIT→ bool
ARGSNEXT→ eps
ARGSNEXT→ , LIT ARGSNEXT

READ→ read (str) ;
WRITE→ write (EXPR EXPRNEXT) ;
EXPR→ eps
EXPR→ expresion
EXPRNEXT→ eps
EXPRNEXT→ , expresion EXPRNEXT