Vysoké učení technické v Brně
Fakulta informačních technológií

Elektronická kuchařka
Dokumentácia z predmetu ITU

30. listopadu 2013

Varianta - č.22

Hodnotenie:

Vojtěch Meca xmecav00 ,
Jiří Macků xmacku03
Martin Maga xmagam00
Rozšírenie: **FUNEXP**

Obsah

1 Úvod

Táto dokumentácia sa zaoberá vývojom, implementáciou a testovaním interpretu pre jazyk *IFJ12*. Je logicky rozdelená do celkov, ktoré popisujú jednotlivé fázy, ktorými musel projekt postupne prejsť. Jednotlivé kapitoly a podkapitoly popisujú podstatné problémy, algoritmy a spôsoby, ktoré sme použili pri implementácii interpretu. Dokumentácia taktiež obsahuje grafický návrh konečného automatu realizujúceho funkciu lexikálneho analyzátora a LL-gramatiku.

2 Práca v tíme

2.1 Príprava na projekt

Pred samotným začatím projektu prebiehalo základné naštudovanie zadania. V počiatočných fázach projektu nám bolo množstvo jeho častí nejasných, z dôvodu neznalosti problematiky implementácie interpretu a jeho súčastí. Preto sme sa snažili naštudovať informácie popri samotnej implementácií, čo nám niekedy prácu trochu spomaľovalo. Pre správne zdieľanie zdrojových textov a ostatných súborov sme sa rozhodli používať verziovací systém *Git*, ktorý je vzhľadom na jeho ľahké a intuitívne používanie celkom príjemný a tak isto dostatočne zdokumentovaný pre prípady, kedy sme museli za chodu riešiť nejaké problémy s verziami programu.

2.2 Komunikácia v tíme

Na prvom stretnutí sme sa dohodli, že budeme organizovať každý týždeň tímové stretnutie, kde budeme vždy prejednávať aktuálne rozpravacované časti projektu a taktiež problémy, ktoré sme mali pri vývoji jednotilvých častí interpretu. Vzhľadom na časovú náročnosť komunikácie online sme sa rozhodli použiť narýchlejšií spôsob, a tak sme využili voľne dostupný "instant messenger"[1] *Skype*.

2.3 Použité metodiky

Pri počiatočnom vývoji sme nemali celkom jasný smer akým sa budeme uberať. Na začiatku sme zvolili trend, ktorý vychádza z metódy *Extreme programming*. Rozdelili sme si teda celý interpret na fázy, z ktorých sa skladá: lexikálna analýza, syntaktická analýza, sémantická analýza a interpreter. Jednotlivé fázy boli porozdelované medzi členov vývojového tímu, kvôli snahe o najoptimálnejší výsledok zdrojového kódu a taktiež o maximálnu časovú efektivitu. Každý sa snažil naštudovať potrebné informácie o jemu pridelenej časti skôr, ako boli preberané na prednáškach, aby nám na konci zostalo dostatok času na dôkladné nájdenie a opravenie chýb. Následne po samotnej implementácií boli jednotlivé časti poriadne otestované, a to z dôvodu vysokej nadväznosti medzi nimi. Pri takomto pridelení práce slúžili aktuálne voľní členovia tímu ako pomocná sila a vytvárali parciálne časti, ktoré boli väčsinou reprezentované funkciami.

[1] Program umožňujúci posielať a čítať správy v reálnom čase.

3 Implementácia

3.1 Shell-sort

Pri implementácií vstavanej funkcie `sort` bola využitá metóda Shell-Sort, ktorá pracuje na rovnakom princípe ako *Bubble-sort*[1]. Táto metóda patrí medzi rýchle metódy, to znamená, že jednotlivé prvky sa premeistňujú k miestu kam patria, väčším krokom, ako iné algoritmy s kvadratickou zložitosťou.
Na začiatku je použitý krok N/2, kde N je dĺžka řeťazca. Vznikne tak N/2 sekvencií znakov, z ktorého každá je spracovaná jedným bublinkovým priechodom. V každom dalšom priechode sa krok zníži na polovicu, tým sa vytvorí polovičný počet čiastočne zoradených podsekvencií a každá se opäť spracuje jedným bublinkovým priechodom. V poslednom priechode sa na celú sekvenciu aplikuje bublinkový priechod s krokom jedna.
Tento algoritmus je nestabilný, a umožňuje prácu „in situ“. Jeho časová zložitost je $\Theta(n^2)$, ale napriek tomu ho môžeme považovať za jeden z najlepších radíacich algoritmov.

3.2 Knuth-Morris-Prattov algoritmus

Tento algoritmus bol použitý pri implementácií vstavanej funkcie `find`, ktorá je súčasťou jazyka *IFJ12*. Algoritmus slúži na vyhľadávanie podreťazca v rámci reťazca. Pri implementácií sme sa inšpirovali informáciami, ktoré boli podané na predmete IAL[1].

Algoritmus vychádza z myšlienky, ktorá zamedzuje opätovné porovnávanie už porovnaných znakov v reťazci. Tento postup sa docieli vytvorením tabuľky, v ktorej budeme mať pre každú poziciu vo vyhľadávanom reťazci napísané číslo, ktoré nám bude určovať, koľký prvok vyhľadávaného reťazca mamé porovnávat s aktuálnym znakom v reťazci. Ak bolo porovnanie na tejto pozícií neúspešné nevraciame sa k začiatočnej pozícií prehľadávania reťazca[2]. Princíp: Algoritmus najprv vytvorí tabuľku, v ktorej bude mať zapísané, s ktorým znakom vzoru má porovnať aktuálny znak reťazca pri neúspešnom porovnaní. Táto hodnota sa môže líšiť pre každý znak vzoru. Znamená to, že pri nájdení nezhody nemusí prechádzať už porovnané prvky, ale číslo z tabuľky mu dá informáciu o tom, s ktorým znakom vzoru má tento aktuálny znak reťazca porovnať pred ďalším posunom v reťazci tak, aby zbytočne nemrhal časom a porovnaniami, ktoré vlastne už vykonal. KMP algoritmus prechádza reťazec sekvenčne ale v prípade nezhody pokračuje v porovnaní aktuálneho znaku so znakom vzoru zadaným v tabuľke alebo prejde na znak nasledujúci. Algoritmus porovnáva vzor a reťazec pokiaľ počet zostávajúcich znakov v reťazci je väčší alebo rovný ako potrebný počet úspešných porovnaní pre nájdenie zhody so vzorom.Algoritmus pri úspešnom nájdení vracia počiatočný index reťazca, inak vracia hodnotu -1.

Algoritmus má zložisť Θ(n+m), teda lineárnu.

3.3 Tabuľka symbolov

Tabuľka symbolov je štruktúra, ktorá uchováva informácie o všetkých premenných a funkciách, ktoré užívateľ v priebehu programu definuje. Základom jej implementácie je hashovacia tabuľka s jednosmerne viazanými zoznamami synoným. Z dôvodu zjednodušenia implementácie rekurzivného volania funkcií, je štandartná tabuľka rozšírená o údaj o počte prvkov v jednotlivých zoznamoch a takisto o skupinu funkcií slúžiacich pre vytváranie (a odstraňovanie) kópií prvkov tabuľky.

3.4 Lexikálny analyzátor

Základný princíp implementácie lexikálneho analyzátoru spočíval v korektnom návrhu konečného automatu, ktorý tento analyzátor vytvára, a ten musel vychádzať zo špecifikácie jazyka *IFJ12*. Lexikálny analyzátor slúži na rozpoznávanie lexémov zdrojového kódu. Realizuje to na základe jednotlivých stavov konečného automatu. V prípade, že lexikálny analyzátor narazí na lexikálnu chybu, vráti príslušný chybný token s informáciou o chybe. V opačnom prípade vráti správny typ tokenu a jeho hodnotu reprezentovanú reťazcom. Daný konečný automat rozpoznáva taktiež klúčové a rezervované slová jazyka *IFJ12* a escape sekvencie. Kľúčové a rezervované slová sú rozpoznávané pomocou priechodu statickým poľom obsahujúcim im zodpovedajúce reťazce a porovnávaním hodnoty tokenov s jeho položkami.

3.5 Syntaxou riadený preklad

3.5.1 Globálna syntaktická analýza

Pri implementácií syntaktického analyzátoru sme využili LL-gramatiku a syntaktickú analýzu programu sme implementovali v súlade so zadaním metódou zhora nadol pomocou rekurzívneho zostupu. Pri tejto metóde je každý neterminál reprezentovaný pomocou funkcie, pričom počiatočným neterminálom je neterminál *¡parser¿*. Výnimkou z tohoto postupu je analýza výrazov, kde sme použili inú metódu.

3.5.2 Syntaktická analýza výrazov

Analýza výrazov prebieha pomocou precedenčnej analýzy zdola nahor. Jej podstatou je tabuľka, na základe ktorej je riadená redukcia a samotné vyhodnocovanie správnosti výrazov. K tomuto analyzátor využíva jednoduchý zásobník, tabuľku symbolov a špeciálnu tabulku, ktorá obsahuje zoznam funkcií, ktoré neboli definované pri volaní. Syntaktický analyzátor priebežne volá lexikálny analyzátor, ktorý mu posiela jednotlivé terminály. Následne ich spracováva s využitím zásobníku a redukčných pravidiel, ktoré sú naviac doplnené o jednotlivé dátové typy a súčasne zabezpečujú aj sémantickú analýzu výrazov.

3.6 Interpret

Interpret je poslednou súčasťou projektu a jedná sa o modul, ktorý vykonáva samotný program. Keďže syntaktická analýza výrazov je prevádzaná pomocou precedenčnej analýzy zdola hore, je jednoduché prevádzať všetky výrazy do postfixovej notácie. Rozhodli sme sa využiť to a pre účely interpretu sme implementovali zásobník určený na vyčíslovanie výrazov. Tento zásobník funguje spôsobom, že ako operandy používa vrchné položky zásobníku a výsledok operácie opäť vkladá na vrchol. Výhod tohoto zásobníku je hneď niekoľko. Za prvé, úplne odpadá generovanie vnútorných premenných. Ďalšou výhodou je možnosť použitia zásobníku na predávanie parametrov funkcií (aj vstavaných, aj užívateľsky definovaných). Táto výhoda so sebou prináša možnosť jednoduchej implementácie rozšírenia FUNEXP, ktoré je popísané neskôr.
Interpret pracuje s jednoduchou sadou inštrukcií, ktoré sú v správnom poradí generované syntaktickým analyzátorom. Ten tieto inštrukcie vkladá do zoznamu inštrukcií, ktorý je implementovaný pomocou jednosmerne viazaného lineárneho zoznamu rozšíreného o možnosť skákania na ľubovoľnú inštrukciu. Vďaka tomu, že zásobník je vždy uvažovaný ako implicitný operand inštrukcie, sú tieto navrhnuté v dvojadresnom kóde, kde jedna adresa značí operand a druhá

funkciu, v ktorej sa daná operácia realizuje.
Základnými inštrukciami sú vloženie na vrchol zásobníku a vybratie vrcholu zásobníku. Ďalej existuje inštrukcia pre každú logickú a aritmetickú operáciu, ktoré ale nepracujú so žiadnymi operandmi, keďže výpočty prevádzajú na zásobníku. Dôležitými inštrukciami sú tie pre podmienený a nepodmienený skok a s nimi súvisiaca inštrukcia návesti, ktoré sa používajú pre skoky v rámci zoznamu inštrukcií.
Pre účely vstavaných funkcií existuje inštrukcia na vykonanie každej z nich s výnimkou funkcie `print`, pri ktorej je daná inštrukcia volaná pre každý parameter zvlášť. Nakoniec, funkčnosť užívateľských funckií je zaručená pomocou dvoch inštrukcií, jednou pre volanie funkcie a druhou pre návrat z nej. Všetky inštrukcie spracovávajúce funkcie počítajú s tým, že argumenty týchto funkcií sú dopredu vložené na vrchole zásobníku a takisto je tam návratová hodnota funkcie, aby s ňou bolo možné ďalej pracovať.

3.7 Rekurzia

Jedným z dôležitých problémov, ktoré sme museli riešiť, bola implementácia rekurzívneho volania funkcií. Pokiaľ je funkcia zavolaná viac než raz (prebieha rekurzia), vytvorí sa kópia všetkých premenných a parametrov danej funkcie v jej lokálnej tabuľke symbolov a zaradí sa na začiatok zoznamu, do ktorých patrí. To zaručí, že pri vyhľadávaní premennej v tabuľke je vždy nájdená jej najnovšia inštancia a zároveň, že ostatné inštancie patriace predchádzajúcim volaniam nebudú stratené. Jednotlivé bunky tabuľky teda fungujú ako zásobníky, u ktorých sa pracuje so skupinou premenných na vrchole. Pri návrate z funkcie sú kópie premenných najbližšie vrcholu "zásobníku"odstránené.

3.8 Rozšírenie *FUNEXP*

Vzhľadom na koncepciu našeho interpretu sme sa rozhodli implementovať rozšírenie *FUNEXP*. Vďaka nemu je interpret schopný spracovávať funkcie ako súčasť výrazov a tak isto výrazy použité, ako parametre funkcií. Jeho implementácia bola možná vďaka tomu, že parametre funkcií a tiež ich návratová hodnota sú predávané pomocou zásobníku interpretu. Preto je možné s týmito hodnotami pracovať pred a po volaní funkcie a počítať s nimi ako so súčasťou výrazov.

3.9 Testovanie

Testovanie nášho projektu prebiehalo na architektúrach Windows a Linux. Bolo založené na vopred napísaných testoch, ktoré porovnávali jednotlivé výsledky testovanej časti s referenčnými. Testovanie spočiatku prebiehalo po častiach, tak ako boli postupne implementované jednotlivé časti intepretu. V konečnej fáze boli vykonané komplexné testy, ktoré overili funkčnost nášho interpretu jazyka *IFJ12* podľa špecifikácie uvedenej v zadaní. V prípade, že bola objavená chyba počas testovania, táto chyba bola ihneď odstránená a interpret bol opäť dôkladne otestovaný.

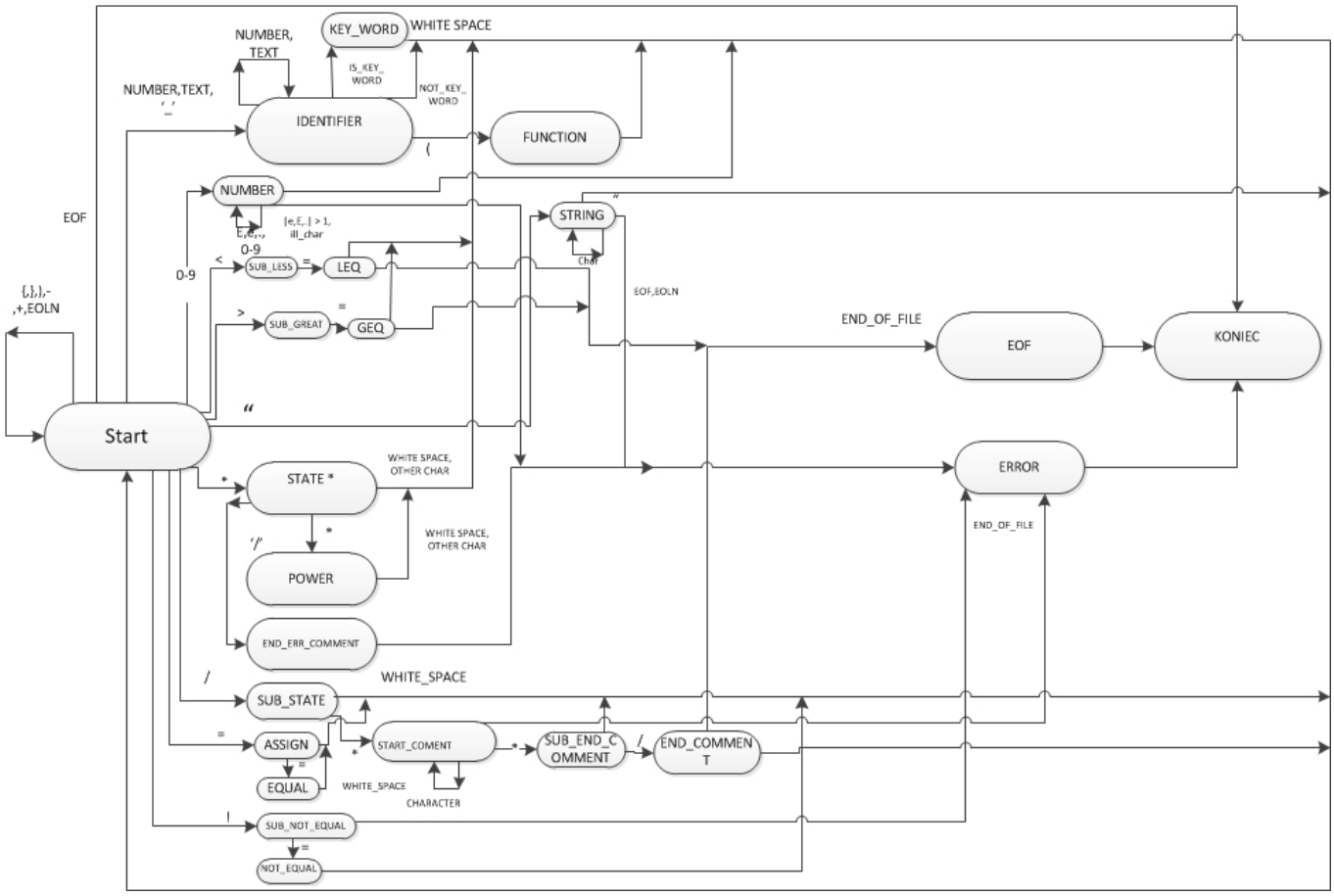
4 Záver

Práca na projekte, ktorého cieľom bola implementácia interpretu jazyka IFJ12 nám priniesla bohaté skúsenosti s rozsiahlymi projektami. Naučili sme sa, ako si správne rozdeliť prácu v tíme, a tak isto aké efektívne spôsoby komunikácie treba zvoliť pri riešení problematiky. V neposlednom rade sme sa naučili používať pokročilé nástroje pri správe verzií programu. Tento projekt nám priniesol množstvo nových informácií a skúseností pri vývoji interpretov jazykov. Funkčnosť nášho projektu bola otestovaná na platformách *GNU Linux* a *Microsoft Windows*©a výsledky boli porovnané s nami vytvorenými referenčnými výsledkami.

5 LL gramatika

ID – terminál
⟨ ⟩ – neterminál
⟨PARSE⟩ → ⟨COMMAND⟩ ⟨PARSE⟩
⟨PARSE⟩ → ⟨FCE DEF⟩ ⟨PARSE⟩
⟨COMMAND⟩ → ⟨IF⟩ ⟨COMMAND⟩
⟨COMMAND⟩ → ⟨WHILE⟩ ⟨COMMAND⟩
⟨COMMAND⟩ → ID = ⟨STATEMENT⟩
⟨COMMAND⟩ → ID = ⟨SUBSTRING⟩
⟨COMMAND⟩ → EOL
⟨SUBSTRING⟩ → STRING [⟨SUBSTRING PARAMS⟩] EOL
⟨SUBSTRING⟩ → ID [⟨SUBSTRING PARAMS⟩] EOL
⟨SUBSTRING PARAMS⟩ → ⟨REAL E⟩ : ⟨REAL E⟩
⟨REAL E⟩ → ID
⟨REAL E⟩ → ε
⟨REAL E⟩ → REAL
⟨PARAMS⟩ → ⟨ID⟩
⟨PARAMS⟩ → ⟨ID⟩ , ⟨PARAMS⟩
⟨WHILE⟩ → while ⟨STATEMENT⟩ EOL ⟨COMMAND⟩ end EOL
⟨IF⟩ → if ⟨STATEMENT⟩ EOL ⟨COMMAND⟩ else EOL ⟨COMMAND⟩ end EOL
⟨FCE CALL⟩ → FCE-ID (⟨CALL PARAMS⟩)
⟨CALL PARAMS⟩ → ⟨STATEMENT⟩
⟨CALL PARAMS⟩ → ⟨STATEMENT⟩ , ⟨CALL PARAMS⟩
⟨STATEMENT⟩ Je řešen pomocí precedenční syntaktické nalýzy zdola nahoru. Zahrnuje veškeré výrazy, včetně volání funkcí.

6 Konečný automat lexikálneho analyzátoru



7 Referencie

Reference

[1] HONZÍK, J. M.: *Algoritmy Studijní opora*. Brno:FIT VUT, 2012.

[2] PROKOP, J.: *Algoritmov jazyku C a C++ :praktický průvodce*. Brno:Grada Publishing, 2009, ISBN 978-80-247-2751-6.

8 Metriky kódu

Počet funkcií: 97 funkcií

Počet súborov: 19 súborov

Počet riadkov:

- samotný kód:3240
- kód a komentáre:501
- samostatné komentáre:1248
- celkom:4989

Počet riadkov zdrojového textu: 982 riadkov

Veľkost statických dát: 7696B

Veľkosť spustiteľného suboru: 96.7 kB (systém Fedora, 64 bitová architektúra, pri preklade bez ladiaciach informácií)