VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VIRTUÁLNÍ STROJ PETRIHO SÍTÍ
PETRI NETS VIRTUAL MACHINE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **TOMÁŠ LAPŠANSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK KOČÍ, Ph.D.**
SUPERVISOR

BRNO 2019

Abstrakt

Bakalárska práca formálne definuje pojem Objektovo orientované Petriho siete. Práca ďalej navrhuje koncept prekladača a virtuálneho stroja pre Objektovo orientované Petriho siete s využitím jazyk PNTalk. Popisuje implementáciu virtuálneho stroja a prekladača.

Abstract

This bachelor thesis formally defines the Object Oriented Petri Nets. Then it designs concept of compilator and virtual machine for Object Oriented Petri Nets using PNTalk language. It uses PNTalk language. It describes implementation of virtual machine and compiler.

Kľúčové slová

Petri Nets, Object Oriented Petri Nets, Virtual Machine, Compilator, PNTalk.

Keywords

Petriho síte, Objektovo orientované Petriho síte, Virtuální stroj, Prekladač, PNTalk.

Citácia

LAPŠANSKÝ, Tomáš. *Virtuální stroj Petriho sítí*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Virtuální stroj Petriho sítí

Prehlásenie

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. R. Kočího, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Tomáš Lapšanský
15. mája 2019

Obsah

1 Úvod **3**

2 Základy objektovo orientovaných petriho sietí **4**
- 2.1 Petriho siete . . . 4
- 2.2 Objektovo orientované Petriho siete . . . 5
  - 2.2.1 Primitívne a neprimitívne objekty . . . 5
  - 2.2.2 Garbage collection . . . 5
  - 2.2.3 Objekt . . . 6
  - 2.2.4 Trieda . . . 7
  - 2.2.5 Metóda . . . 7
  - 2.2.6 Miesto . . . 7
  - 2.2.7 Prechod . . . 7
  - 2.2.8 Synchronizačný kanál . . . 8

3 Smalltalk medzikod **9**
- 3.1 Smalltalk . . . 9
- 3.2 Interpretácia . . . 9
- 3.3 Návrh medzikódu . . . 10
  - 3.3.1 Zásobníkový medzikód . . . 10
  - 3.3.2 Skokový medzikód . . . 10
  - 3.3.3 Zasielací medzikód . . . 10
  - 3.3.4 Návratový medzikód . . . 10
- 3.4 Inšpirácia . . . 11

4 Tvorba prekladačov **12**
- 4.1 Lexikálna analýza . . . 12
  - 4.1.1 Regulárne výrazy . . . 12
  - 4.1.2 Konečný prevodník . . . 13
  - 4.1.3 Lexikálny analyzátor . . . 14
- 4.2 Syntaktická analýza . . . 15
  - 4.2.1 Bezkontextová gramatika . . . 15
  - 4.2.2 Zásobníkové automaty . . . 17
  - 4.2.3 Jednoduché syntaxou riadené prekladové schéma . . . 17

5 Návrh medzikódu **19**
- 5.1 Návrh . . . 19
- 5.2 Syntax . . . 19
  - 5.2.1 Kľúčové slová . . . 19

5.2.2 Výrazy 20

6 Návrh prekladača **21**
6.1 Lexikálna analýza 21
6.1.1 Kódová sada 21
6.2 Syntaktická analýza 22
6.2.1 Výrazy 23
6.2.2 Výstupný medzikód 23
6.2.3 Sémantická analýza 23

7 Návrh virtuálneho stroja **25**
7.1 Analýza vstupného kódu 25
7.2 Užívateľský vstup 27
7.3 Krokovanie 27
7.3.1 Modifikácia miest 28
7.3.2 Volania metód 28
7.4 Beh programu 29
7.5 Optimalizácia 29
7.5.1 Instance objekty 29
7.5.2 Garbage collection 29

8 Záver **31**

Literatúra **32**

A Obsah CD **33**
A.1 Parametre prekladača 33
A.2 Parametre virtuálneho stroja 33

B Syntax jazyka PNTalk **34**

Kapitola 1

Úvod

Tento dokument slúži ako technická správa k bakalárskej práci Virtuální stroj Petriho sítí v akademickom roku 2018/2019 na Fakulte informačných technológií VUT v Brne.

Dokument predpokladá základné znalosti o fungovaní a princípe Petriho sietí a objektovo orientovaného programovania. V prípade nedostatočných znalostí sa odporúča preštudovanie dizertačnej práce [3] od Doc. Ing. Vladimíra Janouška, Ph.D., v ktorej popisuje tieto princípy a jazyk, z ktorého táto bakalárska práca vychádza.

Dokument v `kapitole 2` popisuje základný princíp Petriho sietí a ich objektovo orientovanej reprezentácie. Vysvetľuje aj základné princípu jazyku PNTalk navrhnuté v dizertačnej práci. [3] V tejto kapitole je taktiež vysvetlený pojem `Garbage collector` (nástroj pre automatickú správu pamäti) a popísané rôzne prístupy k tejto metóde.

`Kapitola 3` je venovaná jazyku `smalltalk` a návrhu jeho medzikódu, ktorým bol inšpirovaný návrh medzikódu pre prekladač a virtuálny stroj pre jazyk PNTalk. Samostatná časť je vyhradená pre vysvetlenie spojenia medzi medzikódom pre jazyk smalltalk a medzikódom navrhnutým pre riešenie virtuálneho stroja pre PNTalk.

Neskôr v `kapitole 4` dokument popisuje prístupy pre tvorbu prekladačov. Zavádza pojmy ako je lexikálna alebo syntaktická analýza. Pre jednoduchšie pochopenie problematiky je uvedených niekoľko príkladov riešenia jednotlivých problematík. Kapitola čerpá poznatky z dokumentu `Opora predmetu IFJ` [1]. Pre hlbšie uvedenie do problematiky prekladu zdrojového kódu a jeho transformáciu na finálny výstup sa odporúča bližšie naštudovanie pomocou spomínanej literatúry.

`Kapitoly 5 - 7` sú venované navrhnutému riešeniu. Rozoberá sa základná syntax medzikódu pre navrhnutý virtuálny stroj a odôvodnenie použitej syntaxe. V `kapitole 6` sa detailne rozoberá návrh prekladača po vzoroch uvedených v `kapitole 4`. Rieši použité postupy pri vyhodnocovaní kľúčových prvkov vstupného kódu ako je definícia tried a vyhodnocovanie výrazov a ich transformácie pre lepšiu formu. Samostatná čast je venovaná návrhu sémantického analyzátora, jeho prístupu pre objektovo orientované jazyky a objektovo orientované Petriho siete.

Posledná `kapitola 7` venuje pozornosť samotnému návrhu virtuálneho stroja. Popisuje spôsob analýzy vstupného medzikódu a jeho transformácie vo virtuálnom stroji. Navrhuje programovú štruktúru, ktorá je využitá pre simuláciu objektovo orientovaných petriho sietí a detailne popisuje komunikáciu medzi jednotlivými prvkami tejto štruktúry. Popisuje možnosť užívateľského vstupu do programu, vysvetľuje rôzne spôsoby evolúcie siete a popisuje ich detailný návrh implementácie. V poslednej časti rieši pamäťovú optimalizáciu navrhnutého modelu, implementáciu `Garbage collectoru`, vysvetľuje nedostatky riešenia pamäťovej správy a ponúka komplexnejšie alternatívy ich vyriešenia.

Kapitola 2

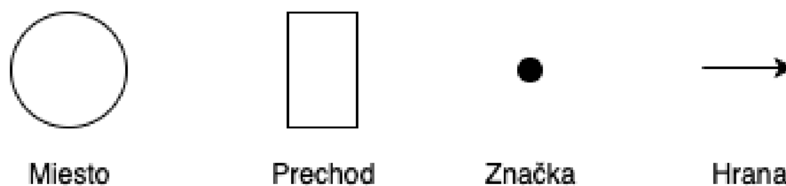
Základy objektovo orientovaných petriho sietí

V tejto kapitole bude všeobecne popísaný koncept Petriho sietí a ich objektová reprezentácia. Ďalej bude popísaný princíp práce s objektami a ich interná reprezentácia v jazyku PNTalk. [3]

2.1 Petriho siete

Petriho sieť je druh orientovaného grafu spolu s počiatočným stavom, nazývaným začiatočné značkovanie. Petriho siete v jednoduchosti definujeme ako $PN(N, M_0)$, kde N je štruktúra Petriho siete a M_0 je počiatočné značkovanie. Medzi základné prvky Petriho sietí patria [4]

- Miesta
- Prechody
- Hrany
- Značka (token)
- Váha hrany - reprezentuje násobnosť hrany
- Značkovanie - priraďuje každému miestu počet tokenov



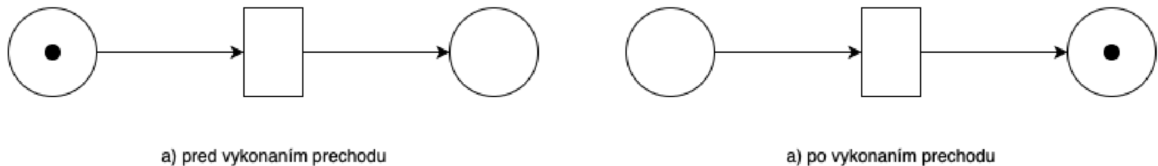
Obr. 2.1: Grafická reprezentácia základných prvkov Petriho sietí

Petriho siete sú vhodným modelom pre modelovanie diskrétnych systémov, hlavne vďaka ich jednoduchej grafickej reprezentácií, dobrými možnosťami simulácie a dobrej verifikácií a formálnej analýze.

Stavové informácie sú v Petriho sieťach ukladané vo forme značiek (tokenov). Tieto tokeny sa ukladajú v takzvaných miestach, ktoré ako celok reprezentujú stavovú informáciu, ktorú v danom čase Petriho sieť uchováva.

Ďalším základným prvkom Petriho sietí sú prechody a hrany, ktoré znázorňujú možnosť a podmienky zmeny stavu jednotlivých miest. Miesto a prechod sú spravidla spojené hranou. Typické grafické znázornenie týchto základných prvkov je viditeľné na obrázku 2.1.

Zmeny stavov na základe vykonateľnosti jednotlivých prechodov nazývame evolúciou Petriho siete. Grafické znázornenie evolúcie Petriho siete po vykonaní prechodu je viditeľné na obrázku 2.2.



Obr. 2.2: Grafická reprezentácia evolúcie Petriho siete

Petriho sieť teda definujeme ako päticu

$$PN = (P, T, F, W, M0),$$

kde

$P = \{p_1, p_2, ...p_m\}$ je množina miest
$T = \{t_1, t_2, ...t_n\}$ množina prechodov
$P \cap T = \varphi F \subseteq (P \times T) \cup (T \times P)$ množina hrán
$W : F \rightarrow \{1, 2, 3, ...\}$ váhová funkcia
$M_0 : P \rightarrow \{0, 1, 2, 3, ...\}$ začiatočné značkovanie

2.2 Objektovo orientované Petriho siete

Objektovo orientované petriho siete (ďalej len `OOPN`) spájajú základný princíp fungovania Petriho sieti a koncept objektovej orientácie, ktorý zahŕňa prvky ako sú objekty, metódy, triedy, dedičnosť, a tak ďalej.

2.2.1 Primitívne a neprimitívne objekty

Základný koncept OOPN definuje primitívne a neprimitívne dátové objekty. Neprimitívnymi dátovými objektami rozumieme objekty definované užívateľom, nazývame ich triedy. Triedy sa počas evolúcie výstupného programu dynamicky menia na základe krokov jednotlivých objektov a volania ich zadefinovaným metód. Neprimitívnym dátovým objektom rozumieme konštanty zadefinované svojimi menami alebo hodnotami. Jedná sa o primitívne dátové typy ako je boolean, celé číslo, znak alebo reťazec znakov.

2.2.2 Garbage collection

Pre OOPN je rušenie objektov otvorenou otázkou. Rušenie objektov môže prebiehať buď explicitne, čo znamená, že objektu je zaslaná správa s informáciou, že má byť zrušený.

Vtedy virtuálny stroj uvoľní pamäťové zdroje pridelené tomuto objektu. Druhou možnosťou je implicítne rušenie objektu v prípade, že ku objektu už nemá prístup hlavný objekt programu. Táto metóda sa nazýva `garbage collecting`, jedná sa o programovací nástroj, ktorý sa stará o uvoľňovanie pamäte. V tomto prípade je potrebné ošetriť viacero stavov. Je potrebné uvoľniť pamäťové zdroje aj objektom, na ktoré vlastní referencie len tento daný objekt, inak by mohlo vzniknúť k vzniku cyklických referencií a zahlcovaniu systémovej pamäte. Poznáme viacero metód garbage collectingu. [5]

Mark and sweep

Všetky referencie na objekty sú uložené na hromade. Collector prechádza koreňové objekty a každý objekt označuje špeciálnym bitom, ktorý značí, že objekt je používaný. Takto prechádza aj na ďalšie objekty podľa toho, aké referencie objekt, ktorý bol označený za používaný vlastní. Všetky tieto objekty sú na konci cyklu označené ako používané. Následne collector prechádza hromadu, a uvoľňuje pamäťové zdroje všetkých objektov, ktoré nie sú označené ako používané.

Nevýhodou tejto metódy je, že počas práce collectora musí byť zastavený beh programu, pre kompletné vykonanie čistenia.

Kopírovanie kolekcií

Táto metóda rozdeľuje adresový priestor na dve polovice. Využíva však reálne iba polku adresového priestoru a v prípade, že sa tento priestor zaplní, kopíruje objekty, ktoré sa používajú do druhej polovice priestoru, prvú vyčistí a cyklus sa znova opakuje.

Nevýhodou tohto prístupu je vysoká pamäťová náročnosť a práca s adresami. Collector musí upraviť adresy všetkých premenných a v prípade, že jazyk dovoľuje ukladanie referencií do premenných, musia sa meniť aj hodnoty premenných. Tento proces taktiež zastavuje program po dobu vykonávania čistenia.

Počítanie referencií

Metóda počítania referencií ukladá každému objektu extra atribút, ktorý udáva počet miest, ktoré obsahujú jeho adresu. Tento počet sa dynamicky mení podľa zmeny hodnôt premenných. V prípade že tento atribút klesne na hodnotu 0, program okamžite uvoľňuje použitú pamäť.

Výhodou tohto prístupu je okamžité uvoľnenie pamäťových zdrojov a taktiež nie je potrebné zastaviť beh programu. V tomto prípade je ale vysoká pravdepodobnosť vzniku cyklických referencií.

Z dôvodu najlepšej efektívnosti s prácou pamäte bol zvolený tento prístup ako kľúčový pri ďalšom návrhu virtuálneho stroja pre PNTalk.

2.2.3 Objekt

Objektom v OOPN rozumieme základnú jednotku modelujúcu Petriho sieť. Objekty počas behu programu dynamicky vznikajú a zanikajú. Každý objekt je jedinečný a reprezentovaný triedou, ktorá je určená pri jeho inštancií. Objekt je špecifický stavom v danom čase behu programu, ktorý môže meniť iba on sám, žiadny iný objekt doň nemôže zasahovať. Pri zadefinovaní nového objektu je objektu priradený jednoznačný identifikátor (adresa), ktorú nadradený objekt môže využiť k zasielaniu správ novému objektu.

Objekty medzi sebou môžu komunikovať pomocou zasielania správ (volaním metód) v prípade, že vlastnia jednoznačný identifikátor objektu, ktorému správu zasielajú a objekt je schopný túto správu rozpoznať (je definovaná metóda odpovedajúca danej správe).

2.2.4 Trieda

Triedy popisujú jednotlivé objekty. Každá trieda definuje spoločné vlastnosti objektov, ktoré sú podľa tejto triedy deklarované. Medzi základné vlastnosti triedy patria jej metódy, vrátane hlavnej metódy, ktorú nazývame objekt. Jednotlivé objekty vytvorené po vzore jednej triedy sa môžu líšiť iba ich jednoznačným identifikátorom a ich aktuálnym stavom.

2.2.5 Metóda

Metódy a hlavná metóda tvoria základné vlastnosti jednotlivých tried v OOPN. Každá metóda definuje vlastnosti, ktoré definujú triedu, ktorej je metóda pridelená. Medzi základné prvky metódy patria miesta (places) a prechody medzi jednotlivými miestami (transitions). Každá metóda je jednoznačne definovaná jej názvom, ktorý z pravidla začína malým písmenom a poľom jej vstupných parametrov. Trieda obsahuje hlavnú metódu nazývanú objekt, ktorá definuje jej implicitne chovanie pri základnom prechode objektom, ktorému táto metóda patrí. Ďalšie metódy sú jednoznačne identifikované ich názvom a poľom vstupných parametrov, ktoré môže byť aj prázdne. Tieto metódy sú volané objektom samotným, alebo pomocou zasielania správ od iných objektov, ktoré vlastnia jednoznačný identifikátor objektu, ktorému je daná metóda priradená.

2.2.6 Miesto

Miesto je jednoznačne identifikované jeho názvom (z pravidla začínajúce malým písmenom). Každému miesto je pridelená hodnota vo forme jednoznačne určeného dátového typu, poľa rôznych hodnôt alebo setu, ktorý obsahuje viacero odlišných hodnôt. Miesto jednoznačne identifikuje stav, v ktorom sa jej nadradený objekt nachádza v danom čase.

2.2.7 Prechod

Prechod je jednoznačne identifikovaný jeho názvom. Prechod na základe jeho určených atribútov určuje akciu, ktorá sa má vykonať v danom čase v prípade, že sú naplnené podmienky uskutočnenia prechodu. Priamo mení hodnotu miest, ktoré im prislúchajú podľa jeho nadradenej metódy a hlavnej metódy nadradeného objektu.

Prechod je vyhodnocovaný na základe podmienok zakomponovaných v štruktúre prechodu. Medzi základné podmienky, ktoré prechod vyhodnocuje, patrí COND, PRECONDITION a POSTCOND. Ďalšími základnými vlastnosťami prechodu je GUARD a ACTION.

COND

Condition definuje existenciu hodnôt v miestach, ktoré sú určené v definícii cond. Môže overovať existenciu konkrétnych hodnôt v danom mieste, alebo len overovať ich existenciu, ktorú následne ukladá do premenných, ktoré sú zadefinované v cond. Tieto premenné sú ďalej využívané v ďalších častiach prechodu.

PRECOND

Precondition definuje existenciu hodnôt v miestach, ktoré sú určené v definícií precond. Môže overovať existenciu konkrétnych hodnôt v danom mieste, alebo len overovať ich existenciu, ktorú následne ukladá do premenných, ktoré sú zadefinované v precond. Tieto premenné sú ďalej využívané v ďalších častiach prechodu.

Po ukončení vykonávanej metódy sú na základe tejto podmienky hodnoty z daného miesta odstránené, simuluje sa tým "presun"značiek medzi jednotlivými miestami siete.

GUARD

Guard je v prechode využitý pre kontrolu vstupných podmienok s využitím premenných zadefinovaných v predošlých podmienkach. Guard podporuje základné porovnávacie operátory, ako je väčší, menší, rovný a ich kombinácie. Kontrola platnosti prebieha vždy pred spustením akcie action.

ACTION

Action vykonáva základnú operáciu, ktorá je vykonaná nad prechodom po splnení všetkých vstupných podmienok prechodu. Pracuje so zadefinovanými premennými, ktoré sú definované v predošlých krokoch.

Medzi základné možnosti vrámci volania action patrí definícia a prepísanie existujúcej premennej, vytvorenie novej inštancie zadeklarovanej triedy a uloženie jej jednoznačného identifikátora do lokálnej premennej, alebo volanie metódy objektu, ktorého jednoznačný identifikátor daný prechod vlastní na základe jeho prebratia vo vstupných podmienkach.

Action však neupravuje hodnoty uložené v miestach prislúchajúcich danému objektu. S hodnotami pracuje na úrovni lokálnych kópií, ktoré zanikajú po ukončení vykonávania prechodu.

POSTCOND

Postcondition upravuje hodnoty miest prislúchajúcich danej triede na základe lokálnych premenných, ktoré boli zadefinované v predchádzajúcich krokoch prechodu.

2.2.8 Synchronizačný kanál

Synchronizačným kanálom špecifická metóda určená na synchronizáciu rôznych objektov. Objekty sú synchronizované pomocou volania synchronizačného kanálu spôsobom totožným zasielaniu správ. Objekt vlastniaci identifikátor objektu obsahujúceho synchronizačný kanál volá tento kanál so vstupnými parametrami, ktoré následne tento objekt overuje.

Kapitola 3

Smalltalk medzikod

V tejto kapitole je v krátkosti popísaný jazyk Smalltalk a medzikód navrhnutý pre tento jazyk. Implementácia zásobníkov pre kalkuláciu výrazov, ktorá sa využíva v tomto jazyku, je taktiež využívaná pre výpočty v navrhnutom virtuálnom stroji pre PNTalk. [2]

3.1 Smalltalk

Smalltalk, na rozdiel od iných objektovo orientovaných jazykov, ako je napríklad C++, je čisto objektovo orientovaný jazyk, ktorý nepodporuje procedurálne programovanie. Každý element jazyka je braný ako samostatne fungujúci objekt.

Smalltalk na komunikáciu medzi objektami využíva metódu zasielania správ. Objekt, ktorý správu prijíma, má zadefinované telo správy, ktoré vykoná po jej prijatí. Atribúty tohto objektu môžu byť zmenené len objektom samotným pri vykonávaní metód prislúchajúcich tomuto objektu. Každý atribút objektu je teda braný ako private property, čo znamená, že nie je verejne dostupný iným objektom.

3.2 Interpretácia

Interpretácia programu napísaná v jazyku Smalltalk využíva dva odlišné prístupy k behu programu.

Častou formou interpretácie je kompilovanie do medzikódu, často nazývaný bytecode. Prekladač pre jazyk Smalltalk preloží jazyk do programových inštrukcií, ktoré sú navrhnuté pre virtuálny stroj, na ktorom je spúšťaný program. Tento medzikód je neskôr virtuálnym strojom interpretovaný a poskytuje veľké možnosti pri návrhu virtuálneho stroja.

Mladšou metódou pre interpretáciu jazyka je preklad zdrojového kódu do strojového jazyka v reálnom čase pri spustení behu programu. Pri tomto prístupe je nástroj prekladača a virtuálneho stroja spojený do jedného programu, čo má za následok väčšiu dynamiku pri správe programu a dovoľuje upravovať zdrojový kód za behu aplikácie, čo ponúka širšie možnosti pri riešení problémov.

Hlavný cyklus využívaný v Smalltalku sekvenčne prechádza medzikód a vykonáva každú inštrukciu. Využíva k tomu aktívny inštrukčný ukazovateľ, ktorý inkrementuje. V prípade volania metódy objektu sa inštrukčný ukazovateľ zmení na prvú inštrukciu odpovedajúcu danej metóde.

3.3 Návrh medzikódu

Pre jazyk smalltalk je navrhnutý medzikód, ktorý je využívaný k jeho interpretácii virtuálnym strojom. Medzikód je sada inštrukcií podobných strojovému jazyku.

Inštrukcie smalltalk medzikódu sú označené číselne. Každá inštrukčná sada má dané svoje rozmedzie, ktoré definuje súbor inštrukcií podobného významu. Poznáme tieto štyri druhy medzikódu. Zásobníkový medzikód, skokový medzikód, zasielací medzikód a návratový medzikód.

3.3.1 Zásobníkový medzikód

Zásobníkový medzikód vykonáva jednoduché operácie na aktívnom zásobníku. Inštrukcie sú reprezentované celo-číselne. Príkladom zásobníkového medzikódu sú napríklad nasledujúce inštrukcie.

- inštrukcia 107 vkladá ukazovateľ na objekt na zásobník
- inštrukcia 18 ukladá ukazovateľ na objekt zo zásobníku do objektovej pamäte
- inštrukcia 1 vymazáva ukazovateľ na objekt zo zásobníku bez jeho uloženia

Zásobníkový medzikód povoľuje taktiež jednobajtové inštrukcie, ktoré na zásobník ukladajú prvých 16 inštančných premenných prijímateľa a prvých 16 lokácií dočasných rámcov.

3.3.2 Skokový medzikód

Skokový medzikód mení aktívny inštrukčný ukazovateľ na kontext podľa špecifikovanej veľkosti. Bezpodmienkové skoky menia inštrukčný ukazovateľ v každom prípade ich vykonávania. Podmienkové skoky menia inštrukčný ukazovateľ iba v prípade, že objekt na vrchu zásobníku je špecifickej boolovskej hodnoty (pravda alebo nepravda). Oba podmienkové a nepodmienkové skoky majú jednobajtovú alebo dvojbajtovú dĺžku.

Podmienkové skoky sa najčastejšie využívajú v cyklických prechodoch kódom alebo podmienených častiach kódu.

3.3.3 Zasielací medzikód

Zasielací medzikód sa používa pre zasielanie správ medzi objektami. Objektový ukazovateľ pre príjemcu a argumenty zasielanej správy sa nachádzajú na zásobníku aktívneho kontextu.

Medzikód samotný určuje výber správy a počet argumentov, ktoré príjemca vyberá zo zásobníka.

3.3.4 Návratový medzikód

Existuje šesť druhov medzikódu, ktoré vracajú kontrolu a hodnotu z aktívneho kontextu. Päť z nich vracia hodnotu správy. Invokujú sa implicitne ukončením metódy alebo explicitne špecifickým príkazom. Posledný druh medzikódu vracia hodnotu bloku. Rozdiel medzi dvoma druhmi návratu je, že prvý vracia hodnotu zasielateľovi správy do jeho domáceho kontextu, zatiaľčo druhý vracia hodnotu zasielateľovi do aktívneho kontextu.

Možné návratové hodnoty sú príjmateľ (self), pravda, nepravda, nil alebo hodnota na vrchole zásobníka. Posledný návratový druh vracia hodnotu na vrchole zásobníka vo forme bloku.

3.4 Inšpirácia

Pre návrh virtuálneho stroja pre jazyk PNTalk bola časť medzikódu inšpirovaná práve návrhom medzikódu vytvoreným pre jazyk smalltalk. Hlavnými prvkami, ktoré sú použité pre návrh medzikódu pre jazyk PNTalk sú zásobníkový a zasielací medzikód.

Zásobníkový medzikód je použitý pre interpretáciu aritmetických a logických výrazov a pre ich efektívne vyhodnocovanie v prostredí virtuálneho stroja. Transformáciou výrazu do postfixovej notácie je možné jednoducho vymedziť inštrukcie a hodnoty, ktoré sa efektívne reprezentujú vyťahovaním hodnôt z vrcholu zásobníka. V prípade, že by tento problém mal riešiť navrhovaný virtuálny stroj, nastávalo by neefektívne nakladanie s procesorovým časom, pretože by bolo potrebné analyzovať výraz ako celok. Transformáciou do zásobníkového medzikódu sa zaistí, že virtuálny stroj iba vykonáva inštrukcie, ktoré má zadefinované vo vstupnom medzikóde. Analýza aritmetického alebo logického výrazu by zabrala oveľa väčšiu časovú jednotku.

Zasielací medzikód nám v návrhu dovoľuje jednoduchú prácu s hodnotami premenných, ktoré sú zasielané pri volaniach jednotlivých metód objektov. V prípade nainicializovania globálneho zásobníka, ku ktorému má každý objekt prístup, je možné efektívne a rýchlo zasielať hodnoty prislúchajúce daným vstupných parametrom. Jednoducho sú tieto hodnoty vložené na zásobník a po zavolaní metódy objektu si objekt tieto hodnoty vytiahne a uloží do svojich lokálnych premenných. Tento prístup sa taktiež v návrhu využíva pre prideľovanie hodnôt lokálnym premenných. V prípade, že je vytvorený objekt v sieti, je jeho adresa vložená na zásobník (adresa je odlíšená kľúčovým slovom) a po vrátení kontextu späť do metódy je táto adresa odobraná zo zásobníka a pridelená konkrétnej premennej.

Ostatné dva spôsoby návrhu medzikódu neboli využité v takto značnej miere, pretože navrhovaný virtuálny stroj nevyužíva prechod medzikódu aktívnym ukazovateľom, preto by bolo neefektívne využívanie napríklad skokového medzikódu. Prvky návratového medzikódu sú však využité pri prideľovaní hodnôt, nie však v tak značnej miere ako ostatné dva návrhy.

Návrh neodráža všetky prvky medzikódu pre jazyk smalltalk. Je iba inšpiráciou pre efektivitu virtuálneho stroja metódami, ktoré boli optimalizované po dlhé roky a prispôsobený potrebám riešenia konkrétneho problému. Detailnejší pohľad na navrhnutý medzikód sa nachádza v kapitole 5.

Kapitola 4

Tvorba prekladačov

V tejto kapitole sú popísané základné postupy využívané pri tvorbe kompilátorov zo zdrojového kódu do internej reprezentácie programu. [1]

4.1 Lexikálna analýza

Zdrojový kód si vieme reprezentovať ako reťazec znakov. Tento reťazec znakov je v prvej časti prekladu čítaný lexikálnym analyzátorom, ktorý tento reťazec transformuje na reťazec lexikálnych znakov (alebo atómov). Atóm si vieme predstaviť ako symbol zdrojového kódu ako napríklad identifikátor, kľúčové slovo, celé číslo,...

Úlohou lexikálneho analyzátoru nie je len rozdelenie vstupného reťazca na súbor atómov, ale taktiež ich vnútorná interpretácia. Vnútornou reprezentáciou lexikálneho analyzátoru rozumieme zväčša celočíselnú reprezentáciu vrámci internej štruktúry prekladača. Napríklad môžeme kľúčové slová označovať číslom 1, celé čísla číslom 2, reťazce číslom 3 a tak ďalej. Činnosť lexikálneho analyzátora vieme modelovať pomocou konečného prevodníka.

4.1.1 Regulárne výrazy

Regulárnym výrazom rozumieme v jednoduchosti súbor pravidiel, podľa ktorých sú vytvorené jednotlivé lexikálne symboly daného jazyka. [1]

Buď V abecede. Regulárna množina nad V je definovaná rekurzívne:

1. ∅ je regulárna množina nad V
2. $\{\varepsilon\}$ je regulárna množina nad V
3. Ak $a \in V$, potom $\{a\}$ je regulárna množina nad V
4. Ak L_1 a L_2 sú regulárne množiny nad V, potom:
   - $L_1 \cup L_2$
   - $L_1 L_2$
   - $L_1 *$
5. Žiadna iná regulárna množina než vytvorená podľa pravidiel (1) - (4) neexistuje

Buď V abeceda. Regulárny výraz R nad V je definovaný rekurzívne:

1. øje regulárny výraz označujúci regulárnu množinu ø
2. εje regulárny výraz označujúci regulárnu množinu {ε}
3. ak $a \in V$, potom a je regulárny výraz označujúci regulárnu množinu {ε}
4. Ak R_1 a R_2 sú regulárne výrazy označujúce regulárne množiny L_1 a L_2 potom:
   (a) $(R_1 + R_2)$ je regulárny výraz označujúci regulárnu množinu $L_1 \cup L_2$
   (b) (R_1R_2) je výraz označujúci regulárnu množinu L_1L_2
   (c) R_1* je regulárny výraz označujúci regulárnu množinu L_1*
5. Žiadne iné regulárne výrazy, než vytvorené podľa (1) - (4) nad abecedou V neexistujú

Ak regulárny výraz R označuje regulárnu množinu L, potom tvrdíme, že L je reprezentovaná regulárnym výrazom R

4.1.2 Konečný prevodník

Konečným prevodníkom M nazývame šesticu

$$M = (Q, V_I, V_O, g, q0, F)$$

kde

Q je konečná množina stavov
V_I je vstupná abeceda
V_O je výstupná abeceda
g je zobrazenie $Q \times (V_I \cup \{\epsilon\})$ do množiny konečných podmnožín $Q * VO*$
$q0$ je počiatočný stav, $q_0 \in Q$
F je konečná množina koncových stavov

Konfiguráciou konečného prevodníka M rozumieme trojicu

$$(q, x, y) \in K \times V *_I \times V*_O,$$

kde

q je aktuálny stav konečného prevodníka M
x je sufix vstupného reťazca, ktorý zatiaľ nebol prečítaný
y je prefix výstupného reťazca, ktorý bol doposiaľ vytvorený

Konečné prevodníky bývajú definované nedeterministicky, čo v praxi znamená, že je možné uskutočniť prechod z jedného stavu do druhého s prázdnym vstupom. (takýto prechod nazývame ε-prechod)

Konečný prevodník M nazývame deterministický v prípade, že pre všetky $q \in Q$ platí jedna z nasledujúcich podmienok:

1. $g(q, a)$ obsahuje nanajvýš jeden prvok pre každé $a \in V_I$ a $g(q, \epsilon) =$ø
2. pre každé $a \in V_I$, $g(q, a) =$ øa $g(q, \epsilon)$ obsahuje jeden prvok

4.1.3 Lexikálny analyzátor

Úlohou lexikálneho analyzátora je rozpoznanie a zakódovanie jednotlivých lexikálnych symbolov zdrojového kódu. Lexikálne symboly programovacieho jazyka sú zväčša interpretované pomocou regulárneho výrazu. Ďalej lexikálny analyzátor využíva konečný prevodník pre rozpoznávanie a kódovanie.

Pre detailnejšie ozrejmenie lexikálneho analyzátora je použitý príklad zo Štúdijnej opory predmetu IFJ. [1]

$$< IDENTIFIKATOR >= P(C+P)*$$

kde C je číslica a P je písmeno.

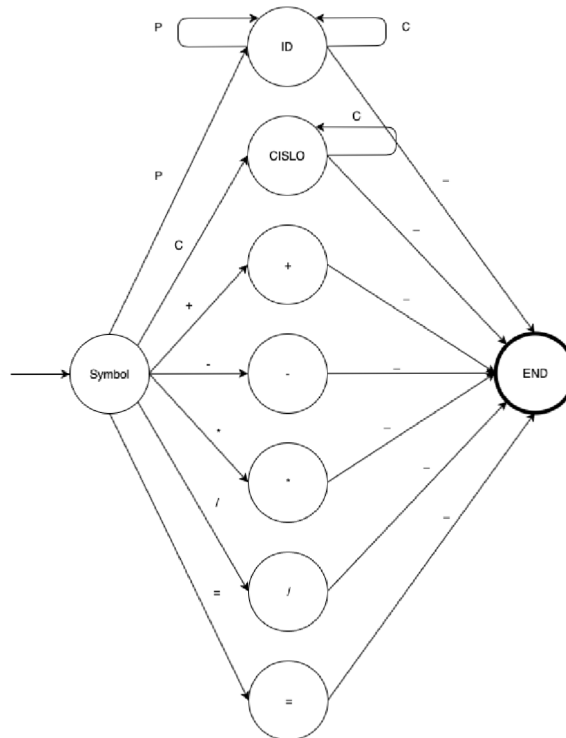
Lexikálnym symbolom celé číslo teda rozumieme:

$$< CELE - CISLO >= C(C) * .$$

Ďalšie lexikálne symboly, ktoré sú v príklade použité sú

$$+, -, *, /, = .$$

Budeme predpokladať, že každý lexikálny symbol vo vstupnom kóde je ukončený špeciálnym znakom medzera '_'. Sme teda schopní zostaviť konečný automat, ktorý bude tieto lexikálne symboly akceptovať. (obr 4.1)



Obr. 4.1: Konečný automat na akceptáciu lexikálnych symbolov

Lexikálny analyzátor definuje kódovú sadu, podľa ktorej identifikuje jednotlivé rozpoznané stavy konečného automatu. Pre uvedený príklad môžeme definovať nasledujúcu kódovú sadu.

Lexikálny symbol	Kód
identifikátor	1
celé číslo	2
+	3
-	4
*	5
/	6
=	7

Akceptačný lexikálny automat neposkytuje dostatočnú funkčnosť pre funkčný lexikálny analyzátor. Tento analyzátor by bol schopný detekcie lexikálnych chýb, no nedokázal by rozlíšiť druh symbolu, ktorému by priradil hodnotu z kódovej sady. Preto je potrebné vytvoriť konečný automat, ktorý okamžite po detekcií symbolu vykoná výstup príslušného kódu podľa kódovej sady.

Nový konečný automat bude analogický k akceptačnému konečnému automatu. Bude však okrem ohodnotenia vstupných hrán obsahovať aj ohodnotenie výstupných hrán automatu v miestach, kde bude zadetekovaný lexikálny symbol. (obr 4.2)

Uvedený návrh lexikálneho analyzátora na obrázku 4.2 je zjednodušený koncept. V praktickom svete často býva viacero druhov delimetrov, nielen '_' ako v tomto uvedenom prípade. Analyzátor prechádza taktiež oveľa viac stavmi konečného automatu, kde má viacero druhov výstupu.

4.2 Syntaktická analýza

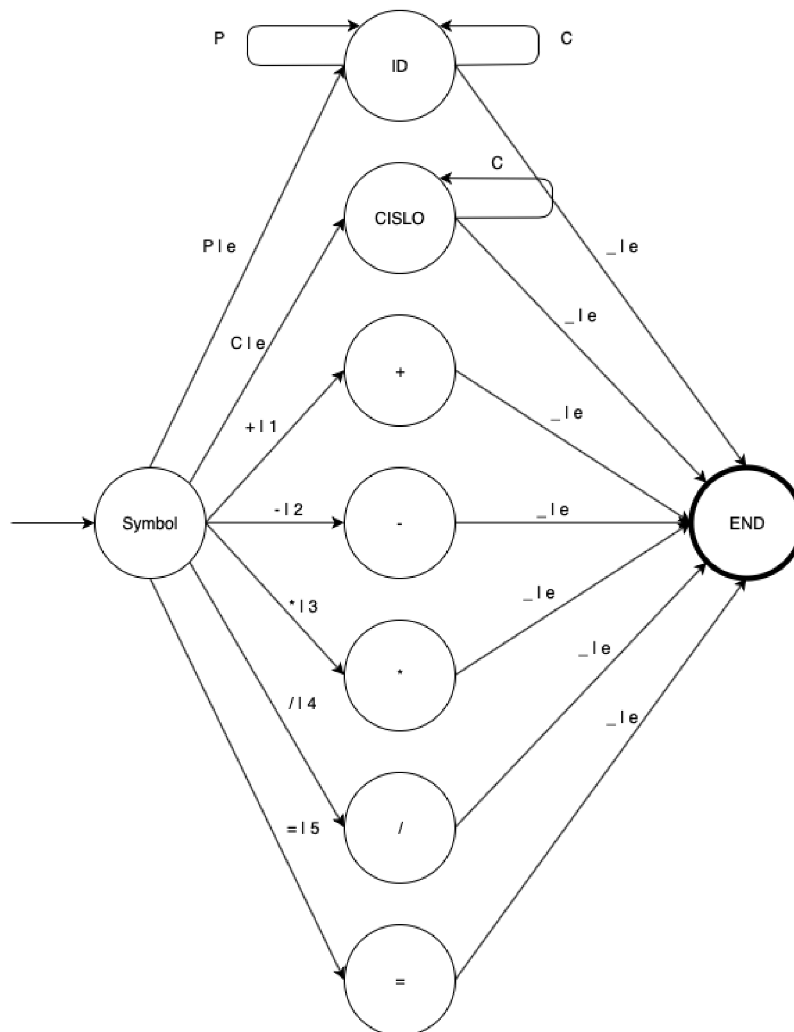
Hlavnou úlohou syntaktickej analýzy je určenie syntaktickej štruktúry zdrojového kódu. Túto činnosť syntaktická analýza realizuje pomocou skonštruovania takzvaného derivačného stromu. Preto zavádzame pojem syntaktická analýza zhora-nadol (respektíve z dola na hor). [1]

Buď $G = (N, T, P, S)$ bezkontextovaná gramatika, ktorá má n pravidiel očíslovaných 1 - n a nech $V \in L(G)$. Syntaktická analýza metódou zhora dolu je proces, ktorý vedie k nájdeniu postupnosti čísel, pravidiel použitých pri ľavej derivácií vety w.

4.2.1 Bezkontextová gramatika

Definícia syntaxe programovacích jazykov sa najčastejšie definuje pomocou bezkontextových gramatík. Tieto gramatiky úzko súvisia s takzvanými zásobníkovými automatmi. Zásobníkové automaty sú popísané v neskoršej sekcií. Tento koncept tvorí teoretický model syntaktického analyzátora.

Nejednoznačnosť bezkontextovej gramatiky môže spôsobovať pri syntaktickej analýze značné komplikácie. Existuje totiž možnosť odvodenia danej vety niekoľkými rôznymi spôsobmi, a teda nejednoznačne. Musíme si preto zaviesť dva pojmy, derivačný strom a kanonická derivácia.



Obr. 4.2: Konečný automat na detekciu lexikálnych symbolov

Derivačné stromy

Buď $G = (N, T, P, S)$ bezkontextová gramatika. Strom je derivačný strom v G ak

1. Každý uzol je ohodnotený symbolom z $N \cup T$
2. Koreň je ohodnotený symbolom S
3. Ak má uzol aspoň jedného nasledovníka, tak je ohodnotený symbolom z N
4. Ak $b_1, b_2, ...b_k$ sú priamy nasledovníci uzlu a, ktorý je ohodnotený symbolom A, v poradí zľava do prava s ohodnotením $B_1, B_2, ...B_k$ potom $A \rightarrow B_1 B_2 ... B_k \in P$

Kanonické derivácie

Deriváciu nazývame ľavou (respektíve pravou), ak v každom jej kroku nahradzujeme najľavejší (respektíve najpravejší) neterminál ostávajúcej vetnej formy. Ak teda

$$x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow ... \rightarrow x_n$$

je ľavá (respektíve pravá) derivácia v gramatike $G = (N, T, P, S)$, potom pre $1 \leq i < n$ je možné napísať x_i ako $y_i A_i z_i$ (respektíve $z_i A_i y_i$), kde $y_i \in T*$, $z_i \in (N \cup T)*$, $A_i \rightarrow w_i \in P$, $x_i + 1 = y_i w_i z_i$ (respektíve $x_i + 1 = z_i w_i y_i$).

V danej bezkontextovej gramatike existuje pre každú vetu $x \in L(G)$ ľavá derivácia. Každej ľavej derivácií vety x odpovedá v G jediný derivačný strom, ktorého je x výsledkom a naopak (to jest každému derivačnému stromu odpovedá v G jediná ľavá derivácia)

Bezkontextová gramatika G sa nazýva nejednoznačná práve vtedy, ak existuje veta $x \in L(G)$, ktorá je výsledkom najmenej dvoch rôznych derivačných stromov, inak sa G nazýva jednoznačná.

4.2.2 Zásobníkové automaty

Zásobníkový automat tvorí sedmica

$$A = (Q, T, Z, \delta, q_0, Z_0, F)$$

kde

Q je neprázdna množina stavov
T je neprázdna vstupná abeceda
Z je neprázdna zásobníková abeceda
$q_0 \in Q$ je počiatočný stav
$Z_0 \in Z$ je počiatočný zásobníkový symbol
$F \subseteq Q$ je množina koncových stavov
δ je zobrazenie $Q \times (T \cup \{\epsilon\}) \times Z$ do množiny všetkých konečných podmnožín $Q \times Z*$
δ nazývame aj prechodová funkcia

Konfiguráciu zásobníkového automatu A nazývame trojicu

$$(q, w, z)$$

kde

$q \in Q$
$w \in T*$
$z \in Z*$

4.2.3 Jednoduché syntaxou riadené prekladové schéma

Nech $T = (N, V_I, V_O, R, S)$ je syntaxou riadené prekladové schéma a nech v každom pravidle $a \rightarrow x, y \in R$ sú pridružené neterminálne symboly v rovnakom poradí v slove x a v slove y. V tom prípade G nazývame jednoduché syntaxou riadené prekladové schéma. Preklad $P(T)$ v tom prípade nazývame jednoduchý syntaxou riadený preklad.

Pre detailnejšie ozrejmenie tejto problematiky je použitý príklad zo Štúdijnej opory predmetu IFJ. [1]

V tomto príklade bude ukázané zostrojenie jednoduchého syntaxou riadeného prekladového schéma na zjednodušené aritmetické výrazy z jazyka $L(G)$ bez zbytočných zátvoriek.

Bezkontextová gramatika G je definovaná takto

$$G = (\{E, T, A\}, \{+, *, (,)\}, Q, E)$$

A takto je definovaná množina pravidiel

1. $E \to (E)$
2. $E \to E + E$
3. $E \to T$
4. $T \to (T)$
5. $T \to A * A$
6. $T \to a$
7. $A \to (E + E)$
8. $A \to T$

Úlohu je možné vyriešiť jednoducho pomocou jednoduchého syntaxou riadeného prekladového schéma

$$T = (\{E, T, A\}, \{+, *, (,)\}, \{+, *, (,)\}, R, E)$$

Kde R bude obsahovať nasledujúce pravidlá

1. $E \to (E), E$
2. $E \to E + E, E + E$
3. $E \to T, T$
4. $T \to (T), T$
5. $T \to A * A, A * A$
6. $T \to a, a$
7. $A \to (E + E), (E + E)$
8. $A \to T, T$

Na uvedenom príklade sa dá jednoducho presvedčiť, že napríklad vstupná veta $((a + (a * a)) * a)$ bude schémou T pretransformovaná na $(a + a * a) * a$, kde sú efektívne eliminované nadbytočné zátvorky.

Schéma T nemá jednoznačne danú vstupnú gramatiku, no obsahuje pravidlo, ktoré nám preloží ľubovoľnú vstupnú vetu na výstupnú. Takýto preklad sa nazýva jednoznačný.

Kapitola 5

Návrh medzikódu

V tejto kapitole je popísaný návrh medzikódu pre vyvíjaný virtuálny stroj. Medzikód je prispôsobený potrebám navrhnutého virtuálneho stroja.

5.1 Návrh

Navrhnutý medzikód priamo referuje vstupný jazyk PNTalk využívaný na reprezentáciu objektovo orientovaných Petriho sietí. Objekty vo vstupnom kóde prepisuje do strojovo jednoduchšej čitateľnej formy. Využíva taktiež koncept zásobníkového medzikódu, ktorý používajú virtuálne stroje navrhnuté pre smalltalk.

Medzikód je navrhnutý s ohľadom na jazykové konštrukcie jazyka PNTalk, ktorý využíva kľúčové aspekty objektov, s ktorými pracuje. Vrámci návrhu medzikódu bolo potrebné efektívne odlíšiť jednotlivé objekty v medzikóde kľúčovými slovami a následne rozlíšiť každý atribút prislúchajúci danému objektu.

5.2 Syntax

Medzikód pre virtuálny stroj využíva pre identifikáciu prvkov kľúčové slová. Každý prvok ako je objekt alebo miesto je vyznačený svojim kľúčovým slovom v jednoznačnom znení, názvom a v prípade potreby aj ďalšími parametrami, ktoré boli definované vo vstupnom kóde.

5.2.1 Kľúčové slová

Pre definíciu komplexnejších objektov ako sú triedy, prechody a metódy je začiatok riadku označený kľúčovým slovom a jednoznačným identifikátorom daného objektu. Nasleduje vnorená interná reprezentácia daného objektu rekurzívnou formou. Tento prístup nám dovoľuje jednoducho do objektov vnášať viacero kľúčových prvkov, napríklad dokážeme jednoznačne identifikovať triedu a jej názov a zároveň každú metódu, ktorá danej triede prislúcha.

V prípade elementárnych prvkov kódu, ako sú miesta, je kód označený kľúčovým slovom $PLACE$ a následne názvom a jednotlivými parametrami daného miesta.

Ak sa jedná o prvky kódu, ktoré neobsahujú jednoznačný identifikátor, ale sú identifikované zanorením do iného nadradeného objektu, tak pre ich definíciu využívame iba kľúčové slovo, za ktorým nasleduje pole. Prvky poľa sú reprezentované každým novým riadkom medzikódu.

5.2.2 Výrazy

Návrh medzikódu pre vyhodnocovanie logických a aritmetických výrazov je inšpirovaný zásobníkovým medzikódom pre jazyk smalltalk.

Aritmetické výrazy

Pre vyhodnocovanie aritmetických výrazov reprezentovaných v štandardnej infixovej notácii je využitý prístup zmeny výrazu na postfixovú notáciu. Postfixovou notáciou rozumieme zápis matematického výrazu, kde operátor nasleduje svoje operandy, pričom je odstránená nutnosť používania zátvoriek. Táto metóda je veľmi rozšírená pre implementáciu vyhodnocovania výrazov v prekladačoch a interpretoch. [6]

Po zmene výrazu na postfixovú notáciu sú jednotlivým operátorom a operandom výrazu pridelené kľúčové slová, ktoré tento výraz virtuálny stroj implementuje ako prácu s takzvaným dočasným zásobníkom (`TEMPSTACK`). Pre túto reprezentáciu sa využíva kľúčové slovo `PUSHTEMP`. Následne je do medzikódu umiestnený kód s kľúčovým slovom `POPTEMP`, ktorý je vyhodnocovaný ako vytiahnutie z dočasného zásobníka a uloženie jeho hodnoty do premennej. Pre vyhodnocovanie výrazov je vytvorené špeciálne kľúčové slovo `CALLTEMP`, za ktorým nasleduje názov operátora taktiež vo forme kľúčového slova.

Logické výrazy

Pre vyhodnocovanie logických výrazov sa rovnako ako pri aritmetických výrazoch využíva zmena na postfixovú notáciu a využitie dočasného zásobníka. V prípade logických výrazov ale rozlišujeme niekoľko možností volania `CALLTEMP`, ktoré sú jednotlivo reprezentované číselne.

- 1 = menší ako
- 2 = väčší ako
- 3 = menší rovný ako
- 4 = väčší rovný ako

Volania funkcií

V prípade volania funkcií definujeme kľúčové slovo `CALL`. V prípade vstupných parametrov funkcie a jeho návratovej hodnoty je využívaný dočasný zásobník, takže už vyššie spomínané kľúčové slová `PUSHTEMP` a `POPTEMP`. Volanie je reprezentované premennou, ktorá nesie jednoznačný identifikátor daného objektu a názov volanej metódy oddelených bodkou.

Kapitola 6

Návrh prekladača

V tejto kapitole sú rozobraté jednotlivé fázy prekladu vstupného zdrojového kódu. Budú popísané návrhy týchto jednotlivých fáz prekladu a navrhnutý návrh ich implementácie pre zdrojový kód.

6.1 Lexikálna analýza

Návrh lexikálneho analyzátora je postavený na báze rozpoznávania príkazov regulárnymi výrazmi. Základom lexikálneho analyzátora bude funkcia, nazývame ju `getToken`, ktorá sa stará o načítanie prvého rozpoznateľného výrazu vo vstupnom kóde. Tento výraz následne pretransformuje na takzvané `tokeny`, ktoré zasiela svojmu volajúcemu.

Keďže lexikálny analyzátor rozpoznáva celé príkazy a transformuje ich na sled jednotlivých tokenov, je potrebné využívať `first in first out (FIFO) frontu`. Po zavolaní príslušnej funkcie je spracovaný celý jeden riadok vstupného zdrojového kódu, ktorý je rozdelený na jednotlivé tokeny, ktoré sú vložené do fronty. Z tejto fronty následne lexikálny analyzátor odoberá jednotlivé prvky a vracia ich svojmu volajúcemu.

6.1.1 Kódová sada

Lexikálny analyzátor definuje vytvorenú kódovú sadu pre prvky vstupného zdrojového kódu PNTalk. Rozlišuje podľa nich základné prvky vstupného kódu.

Kódová sada taktiež definuje špeciálne druhy tokenov označené číselnou reprezentáciou 0-9 a 99. V prípade prvých číselných reprezentácií sa jedná o špeciálne prvky vstupného kódu, ako je napríklad `nový riadok`. V prípade prvku s kódovým označením 0 `EOF` (End of file) sa jedná o špeciálny prvok, ktorý v prípade, že sa objaví na konci fronty, z tejto fronty nie je odstránený, ale v nej ostáva. Tým sa zabezpečí stále vrátenie prvku konca súboru.

Špecifickým druhom tokenu je token s číselným označením 99. Tento token lexikálna analýza vráti v prípade, že nastane chyba pri rozpoznávaní výrazu a lexikálna analýza nie je schopná zaradiť token do svojej kódovej sady. Je následne na volajúcom, ako sa s týmto nezaradeným tokenom vysporiada.

Kód	Lexikálny symbol
0	EOF
1	keyword
2	new line
3	delimeter (,)
11	number
12	string
13	char
14	symbol
15	bool
16	nil
17	pseudo-variable
18	function
19	variable/function name
20	object
30	term
31	empty-term
32	list
33	multiset
99	undefined

Kódová sada na rozmedzí 11-20 vymedzuje označenia primitívnych dátových typov a špeciálnych tokenov určených pre názvy objektov, metód a premenných. Tieto objekty lexikálny analyzátor jednoducho rozlišuje, pretože každý z nich má jasne dané lexikálne pravidlá. Napríklad reťazec vždy začína znakom `#` a je súvislý, takže `#totoJeRetazec` ale `# toto uz nie je retazec 123`. Tak isto majú aj ostatné primitívne dátové typy jasne danú lexikálnu stavbu.

Vrámci návrhu sú zadefinované `nil` a `pseudo premenná` ako samostatné jednotky s vlastnými kódmi, aj napriek tomu, že by mohli byť zaradené medzi kľúčové slová. Avšak kvôli konzistencii rozmedzia kódovej sady sú tieto prvky zaradené medzi "primitívne dátové typy".

Poslednou časťou kódovej sady je kódové rozmedzie 30+. Do tejto časti kódovej sady zaraďujeme `výrazy`. Pre výraz samotný je vymedzené kódové označenie 30, a následne ide o špeciálne modifikácie. Ako je vidieť, pre prázdny výraz je vymedzený samotný kód. Dopomáha to pri ďalšej fáze prekladu pre rýchlejšiu analýzu, keďže vie okamžite odlíšiť prázdny výraz.

Lexikálny analyzátor však výraz netransformuje. Výraz ostáva uzavretý v zátvorkách a je identifikovaný ako výraz, prípadne je vyhodnotený do niektorej zo subkategórií navrhnutých v kódovej sade. Jeho ďalšie spracovanie a vyhodnotenie má na starosti ďalšia fáza prekladu.

6.2 Syntaktická analýza

Navrhnutá syntaktická analýza je založená na báze bezkontextovej gramatiky a derivačných stromov. Prakticky sa teda syntaktická analýza riadi preddefinovanými pravidlami, kde vždy v daný moment vyberá možnosti zo sérií pravidiel tak, ako bolo uvedené v sekcii 4.2.3.

Keďže je základnou jednotkou vstupného zdrojového kódu trieda, tak je trieda koreňovým uzlom derivačného stromu. Od nej sa odvíjajú jednotlivé metódy a následne sa analyzátor vnára do detailov miest a prechodov. Každý objekt musí obsahovať svoju hlavnú triedu, ktorá sa vykonáva pri základnom behu programu.

Syntaktický analyzátor využíva lexikálny analyzátor na získavanie tokenov, ktoré obsahujú dvojicu `celé číslo, reťazec`. Celým číslom je rozpoznávaný token podľa vyššie spomínanej kódovej sady a reťazec určuje presnú hodnotu tokenu. V prípade, že ide o kľúčové slová, syntaktický analyzátor overuje hodnotu reťazca, keďže by bolo neefektívne každému kľúčovému slovu priraďovať jednoznačný kód. V prípade iných častí kódu analyzátor rozpoznáva iba číselnú hodnotu tokenu, pretože nie je podstatná presná reťazcová reprezentácia.

6.2.1 Výrazy

Syntaktický analyzátor vie rozpoznať viacero druhov základných výrazov. Vytvorenie nového objektu, aritmetický výraz, logický výraz a volanie funkcie objektu.

Každý výraz, či už aritmetický alebo logický, je syntaktickým analyzátorom spracovávaný konvertovaním do postfixovej notácie. Prvok, do ktorého má byť nová hodnota uložená, je vytiahnutý z výrazu pri prvotnom rozbore výrazu a následne sa pracuje so zvyšnou časťou. Tento výraz sa po konvertovaní transformuje na výstupný medzikód programu.

Výrazy, kde je vytváraný nový objekt, sú regulárnymi výrazmi detekované a následne pretransformované na výstupný medzikód.

Posledné z možností výrazov sú volania funkcií. Tie sú podobným spôsobom ako vytváranie nových objektov taktiež pretransformované na výstupný medzikód.

Pri vyhodnocovaní jednotlivých výrazov nastáva vždy priradenie do takzvanej sémantickej analýzy pri inicializácií premennej. Sémantická analýza je prekladačom vykonávaná v dvoch behoch súbežne so syntaktickou analýzou. Detaily tejto analýzy budú popísané neskôr.

6.2.2 Výstupný medzikód

Výstupom zo syntaktickej analýzy je finálny medzikód, ktorý prekladač vytvára. Pri každom vynáraní z vetvy derivačného stromu nastáva volanie funkcie `outputCode`, ktorý pridáva novú časť medzikódu do cieľového výstupného média. Výstupný medzikód je však z dôvodu redukcie redundancie generovaný iba pri prvom behu prekladača. Bol zvolený prvý beh, pretože v prvom behu je detailnejšie skúmaná syntax jazyka, ako pri druhom behu. Tieto behy budú rozobrané v časti venovanej sémantickej analýze.

6.2.3 Sémantická analýza

`Sémantickou analýzou` rozumieme proces overovania existencie prvkov výrazu. V prípade navrhnutého prekladača prebieha sémantická analýza v dvoch krokoch, v prvom a v druhom behu.

Pre potreby sémantickej analýzy vstupného zdrojového kódu PNTalk je potrebné riešiť komplexnejšiu sieť objektov, čo v praxi znamená, že každá trieda má meno a obsahuje metódy. Každá z týchto metód obsahuje svoje vlastné miesta a prechody, ktoré s hodnotami v miestach pracujú a každá metóda je schopná pristupovať na miesta svojej hlavnej metódy.

Preto bola navrhnutá nasledujúca konštrukcia pre jazyk C++, ktorá vytvára danú sieť a je jednoduché sa v nej orientovať.

```
#include <unordered_map>
#include <unordered_set>

typedef std::unordered_set<std::string> functionSet;
typedef std::unordered_map<std::string, functionSet> classMap;
typedef std::unordered_map<std::string, classMap> semanticMap;
```

Prvok `semanticMap` deklaruje hašovaciu mapu, v ktorej je kľúčom reťazec s názvom triedy, ktorú tento prvok reprezentuje. Uložená hodnota je `classMap`, ktorá detailnejšie špecifikuje danú triedu.

Prvok `classMap` deklaruje hašovaciu mapu, ktorá prislúcha každej zadefinovanej triede. V tejto mape sú uložené kľúče vo forme názvov funkcií. Každému tomuto kľúču prislúcha vo forme hodnoty vlastný `functionSet`.

Prvok `functionSet` deklaruje set, do ktorého sú uložené názvy premenných, ktoré boli v danej metóde zadefinované. Tieto hodnoty sú potom najdôležitejším prvkom tejto štruktúry, pretože od nich závisí výsledok celej sémantickej analýzy.

Ďalším dôležitým prvkom, ktorý obsahuje navrhovaný sémantický analyzátor, je zoznam miest v daných metódach a počet vstupných parametrov pre metódy. Konštrukcia v C++ vyzerá následovne.

```
std::unordered_map<std::string, int> functions;
std::unordered_multimap<std::string, std::string> places;
```

Prvok `functions` obsahuje hašovaciu mapu, do ktorej sú ako kľúče ukladané názvy metód v tvare `className.methodName`, čím je možné v globálnom poli identifikovať každú metódu v danej triede. Hodnota pridelená tomuto kľúču je počet vstupných parametrov pri volaní funkcie, keďže tento počet je potrebné pri volaniach overovať. Hlavným metódam je pridelený počet nula.

Prvok `places` obsahuje multi-hašovaciu mapu, čo v praxi znamená, že viacerým hodnotám môže byť pridelený jeden kľúč. Kľúče sú ukladané v rovnakom tvare ako pri `functions`, a tým, že je dovolený väčší počet miest, hodnoty tvoria miesta v daných metódach, ktorých názvy sú potom overované pri prechodoch.

Sémantická analýza prebieha v dvoch behoch a súbežne so syntaktickou analýzou. Vrámci prvého behu sa do vyššie definovaných štruktúr sémantickej analýzy ukladajú definované názvy tried a názvy ich jednotlivých metód. Vstupné parametre sú taktiež uložené do poľa použiteľných premenných. Overuje sa existencia premenných, ktoré sa nachádzajú v logických a aritmetických výrazoch. Ďalej sa do vyššie definovaného poľa ukladajú názvy jednotlivých miest.

V druhom behu urobí navrhovaný program niekoľko obmedzení. Vypína sa opätovné zapisovanie medzikódu do výstupného média a celý zdrojový vstupný kód sa skenuje odznova. Tentokrát sa overuje reálna existencia premenných v podmienkach jednotlivých prechodov a existencia tried, ktoré majú byť vytvorené a inicializované.

Kapitola 7

Návrh virtuálneho stroja

V tejto kapitole sa rozoberá návrh a detaily návrhu virtuálneho stroja potrebné pre jeho implementáciu. Je tu znázornený relačný databázový model štruktúry, ktorá sa stará o samotné vykonávanie behu virtuálneho stroja a simuláciu vstupného medzikódu.

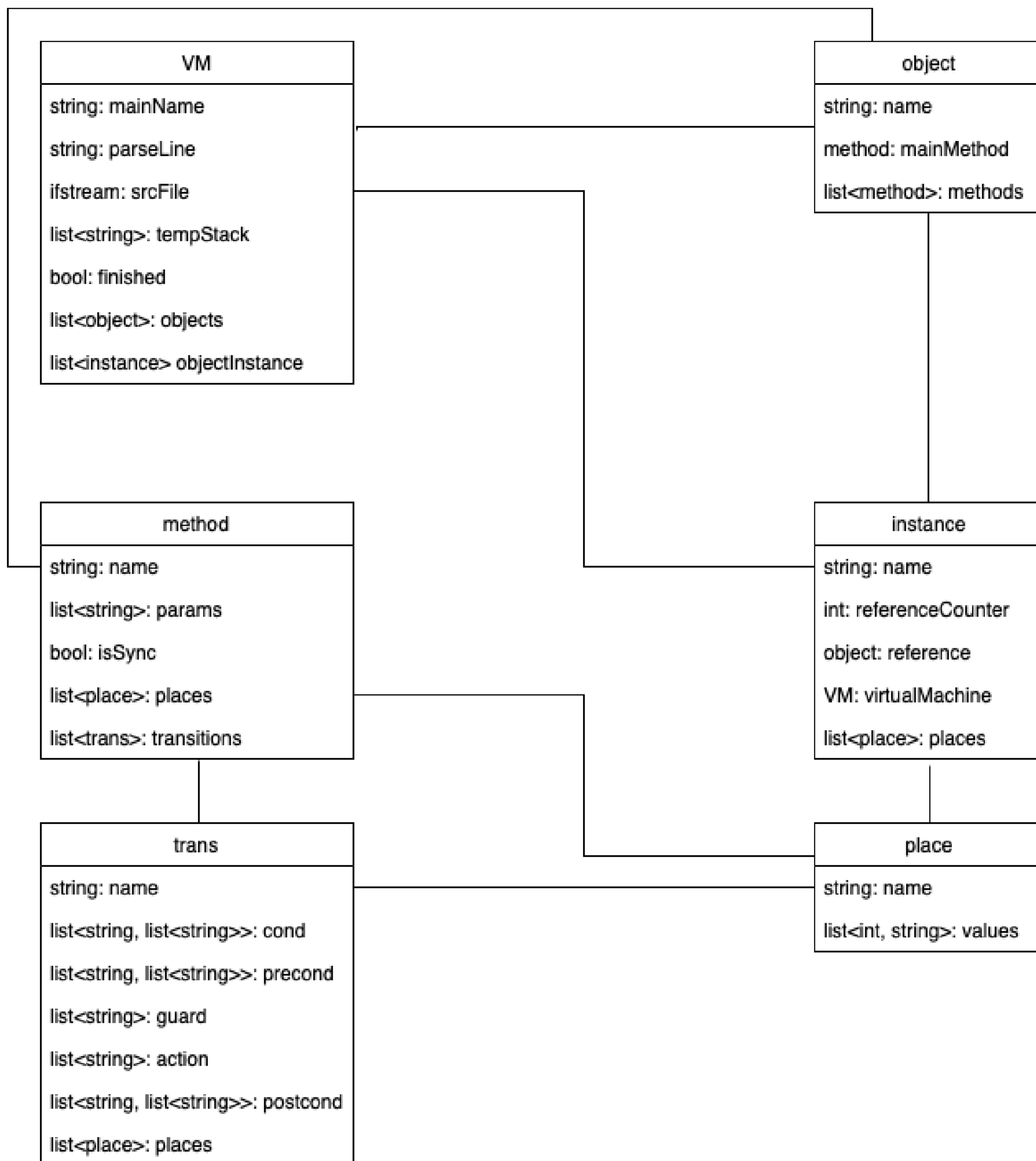
7.1 Analýza vstupného kódu

Návrh virtuálneho stroja obsahuje jednoduchý analyzátor medzikódu, ktorý do stroja vstupuje. Analyzátor funguje taktiež na báze bezkontextovej gramatiky a derivačných stromov. Zásadným rozdielom medzi vstupným zdrojovým kódom do prekladača a vstupným medzikódom do virtuálneho stroja je ten, že v prípade virtuálneho stroja nepripúšťame syntaktické a lexikálne chyby. Prekladač je navrhnutý tak, že všetky tieto chyby odchytí, a teda nie je potrebné ich analyzovať vo virtuálnom stroji. Zefektívni sa tým práca so vstupnými dátami a chod virtuálneho stroja.

Keďže pri analýze vstupného kódu nerozlišujeme syntaktickú a sémantickú analýzu, princípy týchto dvoch analyzátorov sú spojené do jedného nástroja. V navrhnutom medzikóde tvorí každý riadok kódu samostatný príkaz, preto analyzátor overuje pomocou regulárnych výrazov celú stavbu príkazu (riadku kódu). Medzikód je navrhnutý s ohľadom na reprezentáciu kľúčových slov v každom príkaze, preto je postačujúce jednoducho zistiť, či sa dané kľúčové slovo nachádza v danom príkaze, a následne spustiť vetvu, ktorá s týmto príkazom dokáže manipulovať.

Analyzátor vstupného medzikódu nevykonáva príkazy súvisiace s behom programu. V priebehu analýzy je vstupný medzikód pretransformovaný do navrhnutej štruktúry (obrázok 7.1), ktorá reprezentuje jednotlivé prvky OOPN. Keďže je na základe kľúčových slov v medzikóde navrhnutý systém vnárania (napríklad $objekt \rightarrow metda \rightarrow miesto$), analyzátor dokáže automaticky rozpoznať a vytvoriť väzby vrámci vytvorenej štruktúry.

Po načítaní celého súboru s medzikódom a pretransformovaní jeho obsahu do navrhnutej štruktúry fáza analýzy končí a virtuálny stroj prechádza do ďalšej fázy. V ďalšej fáze nastáva slučka udalostí, to znamená, užívateľský vstup a reakcia naň. Nasledujúce fázy virtuálneho stroja budú prebraté v ďalších sekciách.



Obr. 7.1: Diagram tried pre navrhnutý virtuálny stroj

7.2 Užívateľský vstup

Virtuálny stroj by mal v jeho základnej implementácií podporovať 4 druhy užívateľských vstupov, `step`, `run`, `detail` a `quit`.

`Step` je základnou zložkou evolúcie Petriho siete a simuluje jeden krok vrámci evolúcie siete. Návrh tejto metódy bude detailnejšie rozvedený v ďalšej sekcii. Je však dôležité spomenúť, že táto akcia môže byť vyvolaná pomocou užívateľského vstupu.

`Run` simuluje súvislý beh programu od začiatku až po jeho definitívne zastavenie, čo v praxi znamená, že už žiadna evolúcia v Petriho sieti neprebieha. Spôsob návrhu implementácie tejto metódy bude taktiež popísaný v ďalších sekciách.

`Detail` reprezentuje detailne vyobrazenie vytvorených objektov v systéme virtuálneho stroja a ich detaily spojené s miestami a hodnotami v daných miestach. V prostredí navrhovaného `CLI` by daný základ interfaceu mohol vyzerať nasledovne.

```
className1 - main
    place p1 = 451
    place p2 = #true
    place p3 = 0x00007f9585502340
className2 - 0x00007f9585502340
    place b1 = #false
```

Na demonštrácii je jasne vidieť jednotlivé vytvorené objekty v sieti s ich priradenými menami tried. Každá hlavička objektu obsahuje svoj jednoznačný identifikátor vo forme buď adresy alebo označenia `main`, čo značí hlavný objekt spusteného programu. Každý objekt má taktiež jasne dané svoje miesta identifikované kľúčovým slovom `place` a názvom daného miesta vrámci definície triedy objektu. Každému miestu je pridelená jeho jasná hodnota v čase, keď užívateľ požiada o výpis z aktuálneho stavu programu. V prípade, že je v danom mieste uložená hodnota nesúca jednoznačný identifikátor iného objektu, je jeho vyobrazená hodnota adresou daného objektu. Pre túto možnosť je pripravený v návrhu atribút v triede `instance`. Jedná sa o atribút `name`, ktorý v prípade rozšírenia virtuálneho stroja bez problémov dokáže identifikovať daný objekt a vrámci výstupu hlásiť jeho meno a nie adresu v pamäti.

`Quit` má základnú úlohu pri ukončení behu programu. Keďže virtuálny stroj vytvára väčšiu štruktúru pre reprezentáciu dát a evolúciu Petriho siete, je potrebné uvoľniť pamäťové zdroje programu. Po zavolaní príkazu sú všetky zdroje uvoľnené a program je ukončený korektnou formou. Vďaka navrhnutej štruktúre je možné jednoducho a efektívne uvoľniť zdroje, ktoré virtuálny stroj využíval.

7.3 Krokovanie

Trieda virtuálneho stroja `VM` vlastní metódu `step`. Táto metóda sa stará o volanie hlavnej metódy `object` v každom objekte vo virtuálnom stroji. Virtuálny stroj vlastní jednoznačné identifikátory pre každý objekt v sieti, ktoré má uložené v homogénnom poli. Toto pole prechádza jeden po druhom a pre každý objekt volá metódu `step` patriacu objektu.

Trieda `instance` vlastní metódu `step`. Táto metóda prechádza pole pridelených prechodov `trans`, na ktoré sa odkazuje pomocou odkazu na svoju triedu, ktorý má uložený v atribúte `reference`. Každý z týchto prechodov vykonáva vo vzťahu k svojím miestam atomicky, čo v praxi znamená, že ak nie je splnená niektorá z podmienok `precond`, nie sú vykonané ani tie overené pred overením neplatnej.

Každý "krok"vytvára samostatnú premennú `variables`, ktorá je návrhom definovaná nasledovne

```
std::unordered_map<std::string, std::pair<int, std::string>> variables
```

Táto premenná je posúvaná do každej podmienky a akcie, ktorú vlastní daný prechod. V prípade, že je prechod definovaný tak, že inicializuje lokálnu premennú, je táto premenná pridaná do hašovacej mapy v premennej `variables` a jej hodnota sa kopíruje z hodnoty miesta v danom čase. S takto nainicializovanými lokálnymi premennými neskôr pracuje `guard` a `action`. Na základe nich následne môžu vznikať nové hodnoty v miestach po vykonaní `postcondition`. Premenná `variables` zaniká po ukončení overenia, prípadného vykonania daného prechodu. Hodnoty premenných sa strácajú, ak neboli uložené do iných miest.

Pre overenie podmienky `guard` a vykonanie akcie `action` je pre virtuálny stroj navrhnutý dočasný zásobník `tempStack` inšpirovaný zásobníkom využívaným v zásobníkovom medzikóde jazyka `smalltalk`, ktorý bol popísaný v časti `3.1.1`. Základné akcie, ktoré dokáže vykonávať virtuálny stroj nad dočasným zásobníkom sú vkladanie hodnoty na zásobník, odobratie hodnoty z vrcholu zásobníka a volania aritmetických a logických operátorom nad prvkami zásobníka pomocou kľúčového slova `CALLTEMP` a hodnoty operátora, ktorý je potrebné zavolať. Presnejší popis bol zadefinovaný v časti `5.2.2` pri návrhu aktuálneho medzikódu.

7.3.1 Modifikácia miest

V systéme OOPN sú zadefinované dva prípady, kedy sa modifikuje príslušné miesto v objekte. V prípade nastavenia podmienok `precondition` a `postcondition`.

V prípade podmienky precondition nastáva modifikácia miest vymazaním príslušnej hodnoty z daného miesta.

V prípade podmienky postcondition nastáva modifikácia miest vo forme nahradenia aktuálnej hodnoty v mieste novou hodnotou, ktorá je definovaná v podmienke. V prípade neexistencie hodnoty (prázdnej hodnoty) v danom mieste nastáva nahratie novej hodnoty do daného miesta.

7.3.2 Volania metód

Systém OOPN dovoľuje volania metód iného objektu v prípade, že objekt, ktorý metódu volá, vlastní jednoznačný identifikátor objektu, ktorý vlastní metódu. Keďže podľa návrhu virtuálneho stroja ukladá program každú hodnotu do miesta vo formáte reťazca, je každému reťazcu pridelené celé číslo v páre. Toto celé číslo nesie hodnotu 0 v prípade, že sa jedná o primitívny dátový typ, alebo hodnotu 1 v prípade, že je v reťazci uložená adresa iného objektu. Túto adresu dokáže virtuálny stroj z formátu reťazca zmeniť na ukazovateľ na objekt `instance`, ktorého metódy vie následne volať rovnako ako sa vykonáva metóda `step`.

Pre odovzdanie vstupných parametrov do volanej funkcie využíva virtuálny stroj svoj `tempStack`. Objekt, ktorý volá metódu, vloží hodnoty vstupných parametrov na zásobník. Objekt, ktorého metóda je volaná, si tieto premenné vloží do novo-inicializovaného poľa `variables` popísaného vyššie. Následne s nimi pracuje rovnakým spôsobom ako popisovaný implicitný krok.

7.4 Beh programu

Objekt virtuálne stroja v sebe obsahuje špeciálny atribút nazvaný `finished`, ktorý je boolovskej hodnoty. V prípade spustenia programu ako súvislého behu sa začne v slučke volať skôr vysvetlená metóda `step`, ktorej úlohou je vykonať krok v každom objekte.

Vždy na začiatku tejto slučky je nastavená hodnota `finished` na `true`. Keďže je táto hodnota atribútom objektu `VM`, každý z vytvorených `instance` objektov k nemu má prístup. V prípade, že sa podarí počas celého behu kroku vykonať aspoň jedna `transition`, táto `transition` nastaví hodnotu `finished` na `false`. Tento proces bude mať následne za následok, že sa slučka znova zopakuje až do okamihu, kedy počas celého behu kroku nedôjde k evolúcii, a teda simulácia Petriho siete je na konci.

7.5 Optimalizácia

Základným optimalizačným problémom vrámci návrhu virtuálneho stroja je pamäťová náročnosť, ktorá by v prípade rozsiahlej siete, ktorá vytvára veľké množstvo nových objektov, mohla spôsobiť veľmi jednoduché zahltenie systému. Preto boli pre zníženie pamäťových nárokov použité viaceré rôzne kroky, ktoré budú popísané nižšie.

7.5.1 Instance objekty

Pre optimalizáciu pamäťovej náročnosti boli do pôvodného návrhu virtuálneho stroja pridané objekty `instance`. Slúžia ako odľahčené objekty so spätnou referenciou na svoje triedy. Teda je v nich potrebné uchovávať iba hodnoty v miestach, ktoré obsahujú a nie kopírovať celý pamäťový priestor, ktorý patrí vzoru objektu tak, ako to bolo v pôvodnom návrhu.

Čo v praxi znamená nasledovné. Majme objekt, ktorý vlastní iba hlavnú metódu, ktorá obsahuje dva prechody a dve miesta. Tento objekt je schopný vytvoriť novú inštanciu svojej triedy, takže v prípade vytvorenia tisícich objektov nemusí virtuálny stroj uchovávať miesto pre dvetisíc prechodov a dvetisíc miest, ale prechody ostávajú vo svojom pôvodnom stave.

7.5.2 Garbage collection

Ďalší z možných problémov z pohľadu pamäťovej zložitosti sú objekty, na ktoré nemá dosah žiadny iný objekt, takzvané `ghost procesy`. Takýto objekt môže vzniknúť jednoducho tak, že sa stratí referencia na tento objekt, ktorá bola niekde uložená a objekt ostane visieť v pamäti. V prípade navrhnutého virtuálneho stroja môže nastať ešte ďalší problém a to ten, že objekt sa bude stále vyvíjať.

Keďže virtuálny stroj drží referencie uložené vo svojom atribúte, reálne referencie na tieto objekty vo virtuálnom stroji nemiznú. Stratia sa iba zo simulovanej siete, čo má v konečnom dôsledku za následok, že sa predĺži aj čas a rezervovaný procesorový čas pre jednotlivé kroky vrámci virtuálneho stroja. Tak isto môže nastať zahltenie systému v prípade,

že tieto objekty budú vytvárať stále nové a nové objekty, ktoré nebudú reálne spojené so simulovanou sieťou.

Implementácia Garbage Collectoru v navrhovanom virtuálnom stroji je založená na princípe počítania uložených referencií. Každý `instance` objekt obsahuje celočíselný atribút `referenceCounter`, ktorému je pri inicializácií objektu priradená hodnota 0.

Pri každom priradení jednoznačného identifikátora objektu do hodnoty miesta vo virtuálnom stroji nastane inkrement hodnoty `referenceCounter`, čo naznačuje že počet referencií v sieti sa zvýšil o jedna. K tomuto stavu dochádza pri ukladaní hodnôt do miest v príkaze `postcondition`.

Atribút `referenceCounter` je dekrementovaný vždy pri odstránení jednoznačného identifikátora z hodnoty miesta. Tento stav nastáva vždy iba pri príkaze `precondition`, ktorý odstraňuje hodnoty z miest.

Virtuálny stroj po započatí metódy `step` prechádza celé pole `objectInstance`, ktoré obsahuje referencie na vytvorené objekty. Pred spustením metódy `step` nad referenčným objektom virtuálny stroj overuje hodnotu atribútu `referenceCounter`. V prípade, že hodnota tohto atribútu je rovná 0, virtuálny stroj automaticky dealokuje miesto vymedzené pre objekt a pre jeho miesta. Tak isto je tento objekt odstránený zo zoznamu `instance` objektov.

Nedostatky

Jedným z nedostatkov navrhnutého Gabrage Collectoru je možnosť vzniku cyklických referencií. V praxi to znamená nasledovné.

Majme objekt A, ktorý je hlavným objektom siete a majme objekty B a C. Objekt A vlastní referenciu na objekt B. Objekt B vlastní referenciu na objekt C a objekt C vlastní referenciu na objekt B. V prípade, že nastane stav, kedy objekt A stratí referenciu na objekt B. To znamená, že atribút `referenceCounter` objektu B dekrementuje. Avšak `referenceCounter` objektov B a C ostal stále na hodnote jedna aj napriek tomu, že už neexistuje prepojenie s hlavným objektom. Objekty teda ostávajú v pamäti virtuálneho stroja a vykonáva sa aj ich evolúcia.

Kapitola 8

Záver

Táto práca popisuje implementáciu prekladača a virtuálneho stroja pre objektovo orientované Petriho siete (ďalej len OOPN). Virtuálny stroj vychádza z popisu OOPN a návrhu jazyka PNTalk pre tieto siete na základe dizertačnej práce Doc. Ing. Vladimíra Janouška, Ph.D [3]. Návrh práce bol implementovaný v jazyku C++ pre jeho rýchlosť a jednoduchú prenositeľnosť práce medzi rôznymi platformami. Implementácia tejto práce sa nachádza na CD ktoré je v tejto práci označené ako `Príloha A`.

Keďže rozsah a čas vymedzený pre bakalársku prácu nie je postačujúci pre pokrytie všetkých aspektov tak širokej problematiky, ako je komplexný prekladač a virtuálny stroj simulujúci OOPN, je riešenie navrhnuté s ohľadom na jeho možnú úpravu a rozšíriteľnosť o už existujúce alebo nové riešenia.

Vypracované riešenie dokáže načítať vstupný zdrojový kód PNTalk a pretransformovať ho na navrhnutý medzikód. Medzikód je navrhnutý s ohľadom na kľúčové prvky OOPN a je užívateľsky prívetivý a čitateľný, pretože neobsahuje žiadne špeciálne znaky alebo jazykové konštrukcie. Vypracovaný virtuálny stroj je schopný načítať tento medzikód a simulovať beh Petriho siete, či už krok po kroku alebo súvisle až do ukončenia evolúcie siete.

Pri návrhu riešenia je dbané na pamäťovú náročnosť simulácie, preto boli navrhnuté viaceré riešenia, ktoré pomohli znížiť pamäťové nároky výsledného programu. Medzi tieto riešenia napríklad patrí redukcia veľkosti objektov určených na prácu v simulácií alebo Garbage Collector, pre ktorý bolo vybrané najvyhovujúcejšie riešenie z pohľadu plynulosti chodu virtuálneho stroja. Garbage Collector je však sám o sebe rozsiahli projekt, preto bola zvolená iba jeho jednoduchšia forma, ktorá nezabezpečuje čistenie pamäte za behu programu v každej situácií.

Vypracované riešenie nie je dostačujúce v každom smere. Aktuálne riešenie nepodporuje dedičnosť v triedach OOPN, a taktiež neobsahuje plnú škálu práce s premennými, ako je napríklad konkatenácia reťazcov. Na všetky tieto nedostatky je však návrh pripravený a každé z týchto rozšírení sa dá jednoducho doimplementovať.

Virtuálny stroj poskytuje iba terminálové užívateľské rozhranie, no podporuje plnú interakciu užívateľa so simuláciou. Implementácia je pripravená na napojenie na jednoduché grafické užívateľské rozhranie, pretože podporuje celú škálu metód s jednotlivými časťami simulovanej siete. Existuje však aj jednoduchšia možnosť, a to je rozšírenie funkcionality rozhrania pre terminál. Niektoré z možností sprehľadnenia boli spomenuté v tejto práci.

Literatúra

[1] Alexander Meduna, R. L.: Formální jazyky a překladače Studijní opora. online.
URL https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIFJ-IT%2Ftexts%2FOporaIFJ.pdf&cid=12153

[2] Goldberg, A.; Robson, D.: *Smalltalk-80: The Language and its Implementation*. Xerox Palo Alto Research Center, 1983, ISBN 0-201-11371-6.

[3] Janoušek, I. V.: *Modelování objektů Petriho sítěmi*. VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ, 1998.

[4] Jelemenska: Petriho siete. online.
URL http://www2.fiit.stuba.sk/~jelemenska/PETRI_1_slide.pdf

[5] Wikibooks: Introduction to Programming Languages/Garbage Collection. online.
URL https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Garbage_Collection

[6] Wikipedia: Postfixová notace. online.
URL https://cs.wikipedia.org/wiki/Postfixová_notace

Príloha A

Obsah CD

- `Translator/` - adresár pre prekladač
  - `lib/` - knižnice pre prekladač
  - `src/` - zdrojový kód súborov prekladača
- `VM/` - adresár pre virtuálny stroj
  - `lib/` - knižnice pre virtuálny stroj
  - `src/` - zdrojový kód súborov virtuálneho stroja
- `CMakeLists.txt`

A.1 Parametre prekladača

`-f <file>` načítava zdrojový kód PNTalk zo súboru `file`

A.2 Parametre virtuálneho stroja

`-f <file>` načítava vstupný medzikód zo súboru `file`

Príloha B

Syntax jazyka PNTalk

Príloha bola prebraná z [3].

Textová verzia jazyka PNTalk definuje okrem syntaxe inskripčného jazyka aj syntax popisu štruktúry OOPN. Syntax popisu sietí, štruktúry sietí a systému tried formálne definuje rozšírenú Backus-Naurovu formu. Zápis `[...]` znamená, že sa `"..."` môže vyskytnúť nepovinne. `[...]*` znamená ľubovoľný výskyt `"..."`, `[...]+` znamená výskyt aspoň raz. Zápis `"... | ..."` znamená výber z možností. Reťazec v úvodzovkách odpovedá lexikálnemu symbolu. Východzí symbol je `...` .

```
classes: [class]* "main" id [class]*
class: "class" classhead [objectnet]
    [methodnet|constructor|sync]* classhead: id "is a" id
objectnet: "object" net
methodnet: "method" message net
constructor: "constructor" message net
sync: "sync" message [cond] [precond] [guard]
    [postcond] message: id | binsel id | [keysel id]+
net: [place|transition]*
place: "place" id"(" [initmarking] ")" [init] init:
    "init" "{" initaction "}"
initmarking: multiset
initaction: [temps] exprs
transition: "trans" id [cond] [precond] [guard]
    [action] [postcond] cond: "cond" id"(" arcexpr ")"
    ["," id"(" arcexpr ")"]*
precond: "precond" id"(" arcexpr ")" [","
    id"(" arcexpr ")"]* postcond: "postcond"
    id"(" arcexpr ")" ["," id"(" arcexpr ")"]* guard:
    "guard" "{" expr3 "}"
action: "action" "{" [temps] exprs "}" arcexpr:multiset
multiset: [n "’"] c ["," [n "’"] c]*
n: [dig]+ | id
c: literal | id | list
list: "(" [c ["," c]* ["|" [id | list]
temps: "|" [id]* "|"
unit: id | literal | "(" expr ")"
```

```
unaryexpr: unit [ id ]+
primary: unit | unaryexpr
exprs: [expr "."]* [expr]
expr: [id ":="]* expr2
expr2: primary | msgexpr [ ";" cascade
expr3: primary | msgexpr
msgexpr: unaryexpr | binexpr | keyexpr
cascade: id | binmsg | keymsg
binexpr: primary binmsg [ binmsg ]*
binmsg: binsel primary
binsel: selchar[selchar]
keyexpr: keyexpr2 keymsg
keyexpr2: binexpr | primary
keymsg: [keysel expr2]+
keysel: id":"
literal: number | string | charconst |
arrayconst: "#" array
array: "(" [number | string | symbol |
number: [-][[dig]+ "r"] [hexDig]+ ["."
string: "'"[char]*"'"
charconst: "$"char
symconst: "#"symbol
symbol: id | binsel | keysel[keysel]* | string
id: letter[letter|dig]*
selchar: "+" | "-" | "*" | "/" | "~" | "|" | "," | "<"
    | ">" | selchar2 selchar2: "=" | "&" | "\" | "@" |
    "%" | "?" | "!"
hexDig: "0".."9" | "A".."F"
dig: "0".."9"
letter: "A".."Z" | "a".."z"
char: letter | dig | selchar | "[" | "]" | "{" | "}" |
    "(" | ")" | char2 char2: " " | "^" | ";" | "$" |
    "#" | ":" | "." | "-" | "'"
```