Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Rozšírenie systému VizzA na generovanie vizualizácií algoritmov

Bakalárska práca

2014

Tomáš Trungel

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Rozšírenie systému VizzA na generovanie vizualizácií algoritmov

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Pracovisko: Katedra informatiky
Školiteľ: RNDr. Jana Katreniaková PhD.

Bratislava, 2014

Tomáš Trungel

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Tomáš Trungel
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Rozšírenie systému VizzA na generovanie vizualizácií algoritmov / *Extension of the system VizzA for creation of algorithm visualization*

Cieľ: VizzA je abstraktný systém na generovanie vizualizácií algoritmov, ktorý má pomôcť pochopiť fungovanie základných algoritmov za podpory vizualizácie. Jeho základná verzia je dobre rozšíriteľná a cieľom bakalárskej práce je obohatiť systém o novú netriviálnu funkcionalitu. Túto je potrebné zapracovať nielen z hľadiska rozpoznávania nových funkčných prvkov v algoritmoch ale aj potenciálne nových vizualizačných prvkov.

Literatúra: Alena Košinárová: Abstraktný systém na generovanie vizualizácií algoritmov, diplomová práca. FMFI UK, 2013.

Kľúčové slová: vizualizácia, algoritmy, VizzA

Vedúci: RNDr. Jana Katreniaková, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 28.10.2013

Dátum schválenia: 28.10.2013 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Pod'akovanie

Chcel by som sa pod'akovať mojej vedúcej RNDr. Jane Katreniakovej PhD. za jej pomoc, rady a dohľad pri vypracovávaní tejto bakalárskej práce.

Abstrakt

Hlavným cieľom tejto bakalárskej práce je popísať a rozšíriť systém VizzA, ktorý vznikol ako diplomová práca na Katedre informatiky, Fakulty matematiky, fyziky a informatiky Univerzity Komenského v Bratislave. Systém VizzA je abstraktný vizualizačný systém, ktorý slúži ako nástroj na vizualizáciu algoritmov. V tejto práci ho rozšírime o nové vizualizačné nástroje s hlavným zameraním na implementáciu podpory vizualizácie smerníkov počas behu programu. Práca sa zameriava na analýzu a samotnú implementáciu týchto rozšírení a medzi iným poskytuje aj praktické príklady novo implementovaných súčastí ako napríklad vizualizáciu spájaného zoznamu.

Kľúčové slová: VizzA, vizualizácia, smerník

Abstract

Main purpose of this bachelor thesis is to describe and extend system Vizza, which was developed as a master's thesis on Department of computer science, Faculty of mathematics, physics and informatics of Comenius University in Bratislava. Vizza is an abstract visualization system, which serves as a tool for algorithm visualization. In this thesis we extend this system with new tools for visualization with main purpose to support visualization of pointers during execution of a program. This paper aims on analysis and implementation of these extensions and among others provides practical examples of newly implemented parts such as linked list visualization.

Keywords: VizzA, pointer, visualization

Obsah

Úvod

Vizualizácia dát a algoritmov patrí k veľmi účinným metódam výučby informatiky na stredných a vysokých školách. Umožňuje študentom vzdelávať sa v tvorbe algoritmov a programovaní nielen na základe rôznych poučiek, ale na základe vizuálnych výstupov krok po kroku a tým veľmi silno prispievať k procesu učenia sa.

Dnes už existuje pomerne veľké množstvo vizualizačných nástrojov. Medzi jeden z nich patrí systém VizzA, ktorý vznikol ako diplomová práca v roku 2013 na Katedre informatiky, Fakulty matematiky, fyziky a informatiky Univerzity Komenského v Bratislave. Je to abstrakný systém, ktorý umožňuje načítať od používateľa načítať vstupný program v rôznych kódových jazykoch a znázorniť jeho beh na grafickej ploche. Systém môže patriť k veľmi silným nástrojom pri vizualizácií behu algoritmov a výučbe na stredných a vysokých školách.

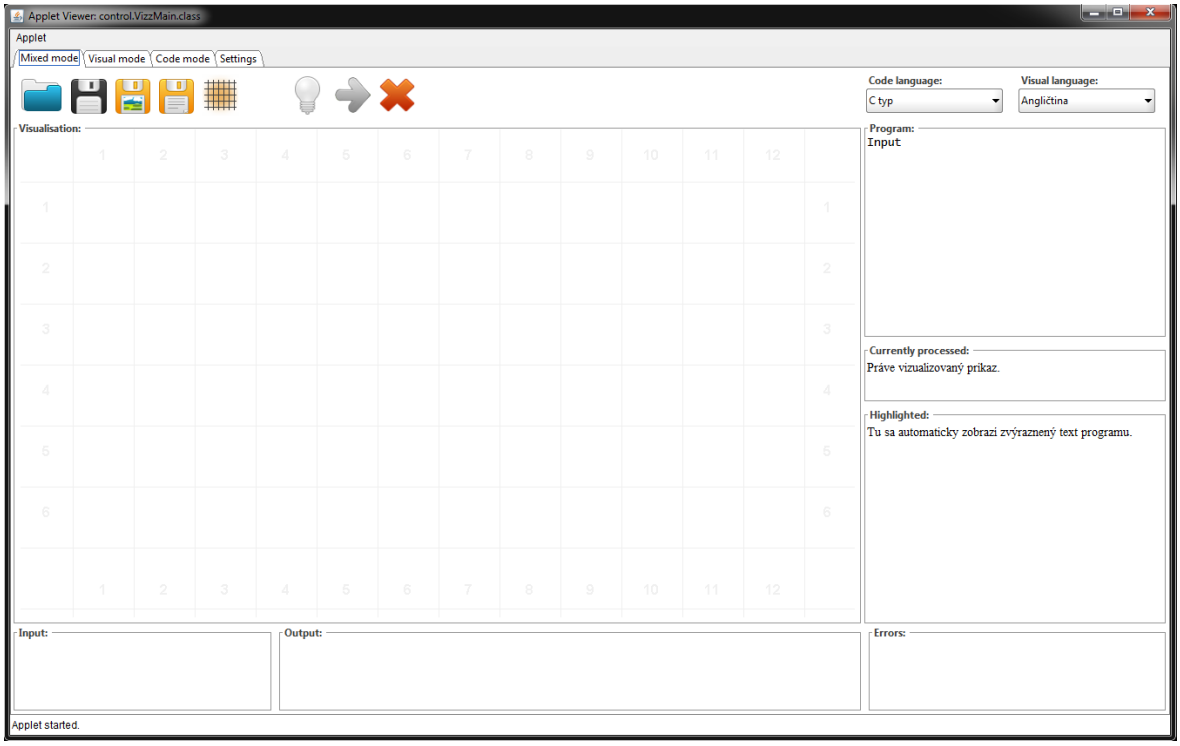
Hľavné zameranie tejto práce je popísaný systém analyzovať a rozšíriť o nové funkcionality. Medzi najdôležitejšiu však patrí rozšírenie systému o podporu smerníkov, ako ich poznáme z jazyka C alebo Pascal. Sme názoru, že takéto rozšírenie môže prispieť k tvorbe unikátneho nástroja, pretože väčšina súčasných počítačových programov zaoberajúcich sa vizualizáciou sa smerníkom nevenujú vôbec alebo iba okrajovo. Takéto riešenie preto môže priniesť do sveta niečo nové a upriamiť pozornosť na nové funkcionality.

Kapitola 1

Úvod do systému VizzA

1.1 Systém VizzA

Systém Vizza pre vizualizáciu algoritmov [4] vznikol ako diplomová práca v roku 2013 na Katedre informatiky Fakulty matematiky, fyziky a informatiky Univerzity Komenského v Bratislave. V úvode tejto práce si rozoberieme podporovanú funkcionalitu a stav systému pred implementáciou dodatočných častí. Táto kapitola nemá za cieľ zreprodukovať texty uvedené v pôvodnej diplomovej práci [4], ale prípadnému čitateľovi načrtnúť súčasný stav problematiky a stručne uviesť informácie potrebné pre rozšírenie tohto systému.



Obr. 1.1: Úvodná obrazovka systému VizzA pod operačným systémom Windows 7

Funkcionalitu systému je možné myšlienkovo rozdeliť do 3 abstraktných vrstiev: vstupná, vykonávacia a vizualizačná. Vstupná vrstva sa stará o preklad a interpretáciu vstupného jazyka do vnútorných premenných a tried systému. Jej úlohou je tiež kontrolovať správnu syntax vstupného programu a v prípade chyby ju vypísať používateľovi. Podporuje 2 vstupné kódovacie jazyky a 2 vstupné vizualizačné jazyky. Úlohou vykonávacej vrstvy je interpretovanie programu tak, ako ho zadal používateľ. Postupne vykonáva každý krok programu a v prípade, ak je potrebné prekresliť obrazovku, sa odovzdá riadenie vizualizačnej vrstve. Vstupnou a vykonávacou vrstvou sa budeme v tejto práci zaoberať viac, pretože ďalšie rozšírenia, ktoré si popíšeme, sa budú priamo na ne. Vizualizačná vrstva systému sa stará o vykreslenie priebehu vstupného algoritmu. Jej úloha je jednoduchšia, či už na pochopenie alebo implementáciu a spočíva v načítaní dát z tried a ich vykreslení používateľovi. Ako sme už povedali, zmeny, ktoré v tejto práci popíšeme, sa jej budú týkať len minimálne.

1.2 Vstupná vrstva systému VizzA

Hlavná úloha tejto vrstvy systému je spracovať vstup od používateľa, overiť správnu syntax vstupného programu a inicializovať potrebné triedy a premenné. Termínom vstupný program myslíme počítačový program napísaný v rámci syntaktických pravidiel systému, ktorý bol zadaný ako vstup pre vizualizáciu od používateľa. Je kombináciou kódového vizualizačného jazyka z ktorých pre každý existujú 2 typy.

1.2.1 Kódové jazyky a ich spracovanie

Kódový jazyk má za úlohu riadiť tok programu. Jedná sa o klasický programovací jazyk so štandardnými konštrukciami. Je možné zvoliť si medzi jazykom odvodeným od programovacieho jazyka Pascal alebo C, pričom oba tieto jazyky sa líšia iba syntaxou. Vzhľadom k tomu, že primárnym cieľom systému VizzA je pomoc pri výučbe a pochopení algoritmov a obmedzeniam plynúcich z možnosti vizualizácie, nie sú podoporované všetky štandardné konštrukcie v týchto jazykoch. Systém pôvodne podporoval príkaz pre načítanie a vypísanie vstupu, cykly - for, while, podmienku - if, deklaráciu premennej a jej vyhodnotenie v aritmetickom, logickom alebo stringovom výraze.

1.2.2 Vizualizačné jazyky a ich spracovanie

Cieľom vizualizačného jazyka nie je riadiť tok programu ale nastaviť parametre vizualizácie. Týmto spôsobom je umožňené zvoliť si predvolený design vizualizácie, farbu a veľkosť vizualizácií, respektíve ich umiestnenie na vizualizačnej ploche. Používateľ má možnosť navoliť si medzi slovenským a anglickým vyzualizačným jazykom. Opäť však tieto jazyky

poskytujú rovnaké možnosti, líšia sa len syntaxou ich zápisu. Dôležitou poznámkou je, že vizualizačný a kódový jazyk nie sú v systéme nijak odddelené, vo vstupnom programe sa prelínajú vizualizačné a kódové príkazy.

1.2.3 Interpretovanie vstupných jazykov do príkazov v systéme

Preklad vizualizačných jazykov je jednoduchší ako pri kódových jazykoch. Stačí správne načítať a spracovať vizualizačné príkazy a tie odovzdať vizualizačnej vrstve systému. Avšak s interpretovaním vstupných kódových jazykov je to pomerne zložitejšie. V prvom kroku je potrebné zo vstupného programu odfiltrovať všetky vizualizačné príkazy. Tým sa získa program skladajúci sa výhradne z kódových jazykov. Keďže systém nepodporuje volanie funkcií, ani žiadne iné zložitejšie konštrukcie, takto očistený vstupný program je možné spracovať po riadkoch. Systém postupne pre každý riadok vytvorí jedného potomka triedy `background.commands.Command`, ktorý dokáže tento príkaz vykonať. Popíšme si teraz jednotlivé operácie a ich realizáciu.

Operácie typu vytvorenie a priradenie hodnoty do premennej sú pomerne jednoduché. K ich realizácií potrebujeme triedu spravujúcu operácie s premennými (`background.variableManager`) a triedy zabezpečujúce vyhodnocovanie aritmetických, stringových a logických výrazov (`background.expressions.MathTree`, `background.expressions.StringProcess` a `background.expressions.LogicalTree`). Vytvorenie premennej vykonáme zaregistrovaním nového názvu v spomenutej triede. Pri úprave (napríklad `x = 4 + 2;`) jej hodnoty poslúžia vyhodnocujúce stromy, ktoré nám dajú odpoveď na zadaný aritmetický výraz. Ten následne aj s názvom premennej povieme triede `background.variableManager`.

Ďalšou kategóriou sú vstupno-výstupné operácie. Systém VizzA poskytuje samostatný element pre zadanie vstupu programu, ktorý je možné načítať napríklad systémovou triedou `java.util.Scanner`. Následné priradenie hodnoty do premennej je vyriešené v predošlom odstavci. Pre výpis hodnoty daného výrazu (napríklad `write(x-y);`) je potrebné výraz vyhodnotiť stromom pre neho určeným (podľa typu: aritmetický, logický, stringový) a vypísať používateľovi.

Medzi zložitejšie príkazy patria príkazy `if`, `for`, `while`. Ich realizácia sa prevedie pomocou vyhodnotenia úvodnej podmienky (trieda `background.expressions.LogicalTree`) a v prípade jej splenia vykonaním vnorených príkazov. Potomkovia triedy `background.commands.Command` zastupujúci tieto príkazy preto obsahujú pole obsahujúce príkazy ktoré sa

majú vykonať pri splnení tejto podmienky. V prípade `for` cyklu je nutné vyhodnotiť ešte inicializačný výraz a operáciu vykonávajúcu sa pri každom prejdení cyklom.

Po vytvorení potrebných tried zo vstupného jazyka je činnosť tejto vrstvy ukončená. Následné riadenie programu preberá vykonávacia vrstva, ktorá interpretuje preložený program.

1.3 Vykonávacia vrstva programu

Vrstva začína svoju činnosť po tom, ako používateľ stlačí tlačidlo "spustiť program"(tvar zelenej šípky v menu aplikácie) a predpokladá ukončenú činnosť vstupnej vrstvy systému. O riadenie toku používateľského programu sa stará trieda `background.CommandManager`. Tá obsahuje zoznam inštancovaných potomkov triedy `background.commands.Command` pre ktorý sa postupne volá metóda `applyCommand()` pri každej inštancií. Spôsob vykonávania tejto metódy pri každom potomkovi závisí od druhu kódového príkazu, aký zastupuje.

1.4 Vizualizačná vrstva systému VizzA

Činnosť tejto vrstvy je pre systém VizzA ako vizualizačný systém najdôležitejšia. Vrstva spravuje vykresľovanie hodnôt hodnôt premenných z behu programu na vizualizačný panel. Ten je reprezentovaný triedou `visualization.VisualPanel`.

Pravdepodobne najdôležitejšia funkcionalita vizualizačnej vrstvy je schopnosť vykreslovať premenné počas behu vstupného programu. Používateľ pri písaní vstupného programu dokáže kľúčovým slovom `vizVar` pri zadefinovaní premennej prikázať systému vizualizovať jej hodnoty v priebehu programu. Ich hodnoty získava trieda `visualization.VisualPanel` z triedy určenej na správu premenných. Každá premenná sa vizualizuje ako jedna bunka na vykreslovacej ploche, kde je zapísaný jej názov, hodnota a farba, ktorá sa líši v prípade že hodnota premennej sa zmenila od posledného vykresľovania. Špeciálnym prípadom pri vykreslovaní je premenná typu jedno- alebo dvojrozmerné pole. Tie je nutné vykreslovať ako riadok jednotlivých buniek resp. tabuľku jednotlivých buniek pre každú premennú.

Priebeh vizualizácie je možné pozastaviť kľúčovým slovom z vizuálneho jazyka `vizPause`, ktoré pozastaví priebeh vizualizácie. Tiež je podporované vizualizovanie priebehu jednotlivých cyklov vo vstupnom programe použitím kľúčového slova `vizFor` a `vizWhile` vo vizualizačnom jazyku. Systém pri vstupe do takéhoto cyklu vizualizačnú plochu prekreslí, vykoná všetky vnútorné príkazy cyklu a opäť plochu prekreslí, čím bude mať používateľ možnosť uvidieť ako sa zmenili hodnoty premenných v programe. Toto sa opakuje až kým

nedôjde k prerušeniu cyklu, keď bude systém pokračovať najbližším po ňom. Systém tiež umožňuje použitie vnorených vizualizovaných cyklov.

Vizualizačný jazyk podporuje mnohé ďalšie funkcie. Medzi tie zaujímavé patria napríklad príkazy `mult#x` a `cX#X cY#Y`, ktoré umožňujú zvoliť veľkosť bunky do ktorej systém vykresluje hodnotu premennej a určiť jej pozíciu na X, respektíve Y osi. Príkazom `vizGlobal design#simple color#orange;` je zas možné zvoliť design vizualizačnej plochy a použitie farby pre vykreslenie premenných. Ďalšie konštrukcie vizualizačného jazyka je možné nájsť v pôvodnej diplomovej práci [4]. V tejto práci sa ním viac zaoberať nebudeme.

Kapitola 2

Súčasný stav problematiky

V nasledujúcej kapitole upriamime pozornosť a popíšeme si niektoré významné a zaujímavé projekty zameriavajúce sa na vizualizáciu dát alebo algoritmov. Projekty nás budú zaujímať hlavne z hľadiska zamerania tejto práce, vizualizácie smerníkov počas behu programu. Tiež tu poukážeme na nedostatky súčasných nástrojov a prípadne ich porovnáme s nástrojom VizzA, popisovaným v tejto práci.

2.1 Projekty bez podpory vizualizácie smerníkov

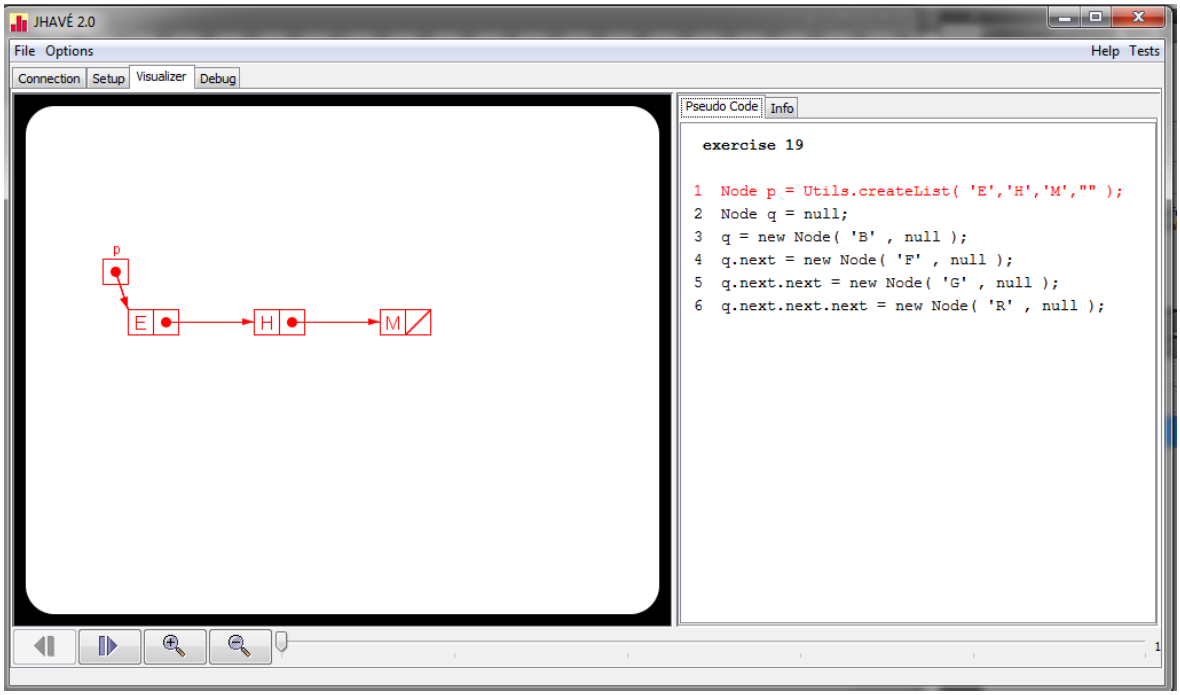
V pôvodnej diplomovej práci [4] boli spomenuté rôzne projekty zameriavajúce sa na vizualizáciu algoritmov. Boli popísané projekty zameriavajúce sa na triedenie údajov ako SortVis [3], internetová stránka sorting-programs.com [6], alebo projekt AlgoRythmics [8]. Žiaľ, aj keď sú tieto projekty pomerne dobre a použiteľne spracované, ani v jednom nie je možné vizualizovať smerníky. Ich zameranie je často úzko profilované na vizualizáciu triedení a nechávajú len malý priestor pre vstup používateľa.

Tiež tu boli spomenuté aj rozsiahlejšie projekty, ako napríklad Algovision [5]. Medzi neodškriepiteľné výhody tohto projektu patrí jeho prepracovanosť a dokonca dostupnosť lokalizácie v českom jazyku. Na druhej strane sa v projekte nenachádza ani len náznak snahy zobrazovať alebo čo i len podporovať smerníky. Skúsme preto nájsť iné projekty.

2.2 Projekt JHAVÉ

JHAVÉ [7] je aplikácia naprogramovaná v jazyku Java, ktorá umožňuje renderovanie algortimov a ich vizualizácií. Jej hlavným zameraním je výuka programovania pre študentov informatiky na univerzitách a vysokých školách. Do projektu sa zapájajú ľudia z viacerých známych univerzít, ako napríklad University of Wisconsin - Oshkosh alebo Grand Val-

ley State University. Samotný projekt je šírený pod Creative Commons licenciou [1], ktorá umožňuje voľné šírenie a publikovanie tejto práce.



Obr. 2.1: Snímok obrazovky z behu nástroja JHAVÉ - proces vizualizácie spájaného zoznamu pomocou smerníkov

V súčastnosti projekt podporuje dokonca 3 rôzne skriptovacie jazyky pre programovanie vizualizácií. Sú to GAIGS, Animal, a XAAL. Na stránke projektu je dostupná dokumentácia ku každému spomenutému jazyku, avšak programovanie v nich môže byť zložité.

Z hľadiska tejto práce je dôležité že niektoré vizualizácie v tomto projekte podporujú či už v menšom alebo väčšom rozsahu vykresľovanie smerníkov. Problémom však zostáva že aj keď je dostupné pomerne veľké množstvo predprogramovaných vstupov, vstupov podporujúcich vizualizáciu smerníkov je však len málo a smerníky nie sú ich primárnym zameraním. Medzi ďalšie nevýhody môžme započítať neprehľadné a náročné zadávanie nových vizualizácií z hľadiska používateľa. V tomto je zameranie a funkcionalita systému VizzA pokročilejšia a pri vizualizácií smerníkov používateľsky príťažlivejšia.

2.3 Ostatné projekty

Aj keď je dnes dostupných mnoho vizualizačných nástrojov, len málokedy v nich existuje snaha vizualizovať smerníky počas behu vizualizovaného programu. Napriek autorovej snahe vyhľadať a popísať ďalšie nástroje je problém vôbec nejaké nájsť. A preto sa v rámci tejto kapitoly budeme musieť uspokojiť len s jedným podrobnejšie popísaným nástrojom podporujúcim smerníky.

Veríme, že tento neúspech len zdôrazňuje dôležitosť tejto práce pre vytvorenie nástroja podporujúceho vizualizáciu smerníkov.

Kapitola 3

Možnosti implementácie smerníkov pre systém VizzA

3.1 Smerníky

Smerníky patria k dôležitým funkcionalitám mnohých programovacích jazykov. V základnom chápaní je smerník obyčajná premenná, ktorá obsahuje adresu inej premennej. Programátorovi dávajú moc manuálne spravovať vyhradenú pamäť programu a upravovať ju. Smerníky sa v programovaní používajú pomerne často, pretože často vedú k efektívnejšiemu, jednoduchšiemu a kompaktnejšiemu kódu. Samozrejme, nadmerné alebo nesprávne používanie smerníkov vedie k presnému opaku a k mnohým programátorským chybám. Aj preto niektoré mnohé moderné jazyky (ako napríklad Java, C#) smerníky nepodporujú a namiesto nich používajú automatickú správu pamäte. Predpokladáme však, že koncept smerníkov je čitateľovi známy, preto sa ním ďalej nebudeme zaoberať.

Z výučbového hladiska informatiky patria smerníky k funkcionalitám netriviálnym na pochopenie pre začínajúceho programátora. Preto by bolo zaujímavé rozšíriť systém VizzA práve o smerníky. Ako sme už popísali, systém VizzA momentálne podporuje 2 vstupné kódové jazyky. Preto si v nasledujúcej časti rozoberieme funkciu smerníkov v každom z týchto jazykov a pokúsime sa rozšíriť syntax pre každý z týchto jazykov.

3.1.1 Smerníky v jazyku C

Jazyk C patrí medzi svetovo najúspešnejšie a najpoužívanejšie programovacie jazyky. Jeho syntax je pomerne veľmi dobre známa a je východiskom pre mnohé iné úspešné programovacie jazyky. Stručne si opíšeme ako fungujú smerníky v jazyku C, aké veci sa s nimi spájajú a ako vyzerá ich využitie.

Jazyk C pozná smerníky na premenné alebo funkcie. Vzhľadom na systém VizzA nás budú zaujímať iba smerníky na premenné.

```
1 int x = 5, y = 4;
2 int *p;
3 p = &x; // p teraz obsahuje adresu premennej x
4 if (*p == 5) ; // true, na adrese na ktoru ukazuje p je ulozene cislo 5
5 if (p == &y) ; // false, p obsahuje adresu premennej x
6 if (p == y) ;  // chyba, porovnavame adresu premennej s hodnotou
7 p++;           // zvacsenie adresy p o 1
```

Listing 3.1: Vytvorenie a použitie smerníka typu int v jazyku C

Vidíme, že na získanie a priradenie adresy ľubovolnej premennej môžme použiť unárny operátor &. Unárny operátor * zas po použijeme na získanie hodnoty uloženej na adrese na ktorú ukazuje daný smerník. Ďalej vidíme že porovnávanie adresy s hodnotou je chybné a vedie k programátorským chybám. V optimálnom prípade by kompilátor systému Vizza toto nemal podporovať a vypísať chybu.

Ďalšie podporované operácie so smerníkmi v jazyku C je ich inkrementácia/ dekrementácia o určitú hodnotu. Tým získame dynamické vlastnosti smerníkov, pre ktoré sú často využívané. Napríklad pokiaľ smerník p ukazuje na hodnotu v poli, zvýšením hodnoty smerníka budeme ukazovať na nasledujúcu hodnotu v danom poli a naopak. Samozrejme, týmto spôsobom je možné získať aj hodnoty mimo rozsahu daného poľa, čo je v prípade systému Vizza neželané. V nasledujúcich častiach vyberieme len požadované vlastnosti a ostatné zakážeme.

3.1.2 Smerníky v jazyku Pascal

Pri opisovaní syntaxe smerníkov v jazyku Pascal budeme vychádzať z jeho open source verzie Free Pascal [2] a jeho oficiálnej dokumentácie dostupnej na internete.

Smerníky v jazyku Pascal fungujú na podobne v jazyku Pascal ako v jazyku C. Na rovnakom príklade si predvedieme ich použitie a na základe toho neskôr vytvoríme syntax pre systém VizzA.

```
1  Var p : ^Longint;  // smernik
2      x,y : Longint; // premenne
3
4  ...
5
6  x := 5;
7  y := 4;
8  p := @x;                // p teraz obsahuje adresu premennej x
9  if p^ = 5 then ...      // true, na adrese na odkazovanej p je ulozene 5
10 if p^ = @y then ...     // false, p obsahuje adresu premennej x
11 if p = y then ...       // chyba, porovnavame adresu premennej s hodnotou
12 inc(p);                 // zvacsenie adresy p o 1
```

Listing 3.2: Vytvorenie a použitie smerníka typu Longint v jazyku Pascal

Obdobne vidíme že na získanie adresy premennej sa tentoraz používa operátor @. Operátor ŝa používa na získanie hodnoty na ktorú ukazuje smerník. Tento príklad je pre nás dostačujúci na vytvorenie syntaxe pre systém VizzA. Potrebujeme však ešte zistiť aké operácie bude systém podporovať. Obdobne ako v jazyku C, aj jazyk Pascal podporuje smerníkovú aritmetiku, ktorú budeme potrebovať pri implementácií okresať.

3.2 Operácie so smerníkmi v systéme VizzA

Ujasnime si aké operácie bude systém VizzA podporovať. Keďže jazyky Pascal a C fungujú v práci so smerníkmi rovnako, je rozumné zadefinovať rovnaké operácie pre oba jazyky. Jazyky sa budú odlišovať iba syntaxou v rámci zachovania čo najväčšej podobnosti s pôvodnými jazykmi.

Medzi základné operácie bude samozrejme patriť vytvorenie smerníka a priradenie adresy premennej do daného smerníka. Samozrejme, smerník musí byť rovnakého dátového typu ako je daná premenná. Medzi ďalšie elementárne operácie bude patriť zmena hodnoty na adrese, na ktorú smerník ukazuje. V neposlednom rade potrebujeme dokázať smerníky interpretovať v logických, aritmetických a stringových výrazoch. Táto časť je pomerne jasná, smerník budeme interpretovať buď ako adresu alebo ako hodnotu na tejto adrese, podľa syntaxe v C, resp. Pascal-type jazyku.

Ďalej sa však musíme zamyslieť nad operáciami inkrementácie, dekrementácie a priradenia nejakej adresy do daného smerníka. Jazyky C a Pascal podporujú priame priradenie

adresy do smerníka. Toto však v systéme VizzA nebude možné implementovať už len z dôvodu nedostatočnej podpory jazyka. Jazyk Java, v ktorom je systém VizzA naprogramovaný nepodporuje smerníky, dovoluje nám s pamäťou pracovať iba pomocou referencií. Vďaka tomu nemôžme niečo takéto implementovať priamo ale iba emulovať. Premenným v danom programe by sme mohli priradiť adresy podľa počtu bitov, ktoré zaberajú a tým vytvoriť virtuálny adresný priestor.

3.3 Voľba syntaxe pre systém VizzA

Už sme popísali ako fungujú smerníky v jazykoch Pascal a C. Vieme tiež akú syntax majú tieto jazyky pri práci s nimi. Zostáva nám už len navrhnúť syntax takú, aby zachovala čo najviac syntaktických vlastností pôvodných jazykov a bola dobre interpretovateľná v systéme Vizza.

V C-type syntaxi si vyhradíme zvolíme znaky * a &. Znak * budeme používať pri inicializácií premennej a znak & bude značiť nastavenie ukazovateľa na danú premennú.
Pre pascal-type syntax použijeme analogicky znaky znaky â @.
Povolené operácie budú vyzerať nasledovne:

Operácia	C-type syntax	Pascal-type syntax
Vytvorenie premennej	int *p;	p : ^Longint;
Nastavenie hodnoty smerníka	p = &x;	p := @x
Úprava hodnoty premennej	*p = 5;	p^ = 5;
Získanie hodnoty na ktorú smerujeme	if (*p == 5)	if (p^ = 5)
Zvýšenie hodnoty smerníka	p=p+1;	p:=p+1;

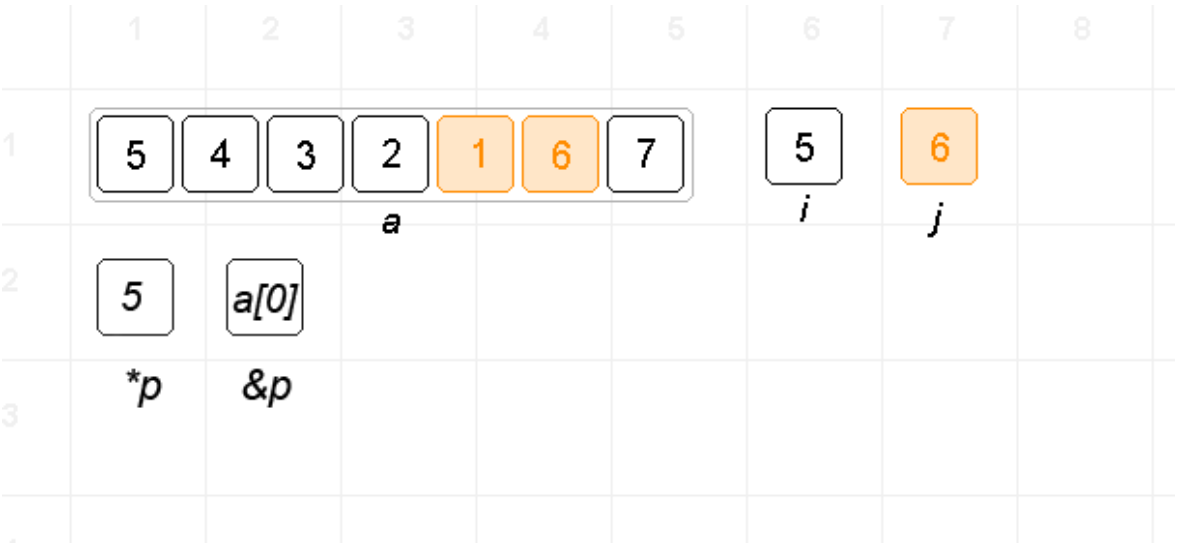
Tabuľka 3.1: Voľba syntaxe pointrovej aritmetiky v systéme Vizza

Tieto operácie budú pre prácu so smerníkmi úplne dostatočné a ďalšie rozšírenie nebude potrebné. Teraz však musíme preskúmať ako môžme dané operácie a smerníky vizualizovať.

3.4 Možnosti vizualizácie smerníkov

Asi najdôležitejšia časť systému Vizza je vizualizácia naprogramovaného algoritmu. Považujeme preto za nesmierne dôležité zvoliť vhodný a systematický prístup vizualizácie. Máme niekoľko možných prístupov ako naprogramovať vizualizáciu smerníkov v systéme. Rozoberme si niekoľko z nich.

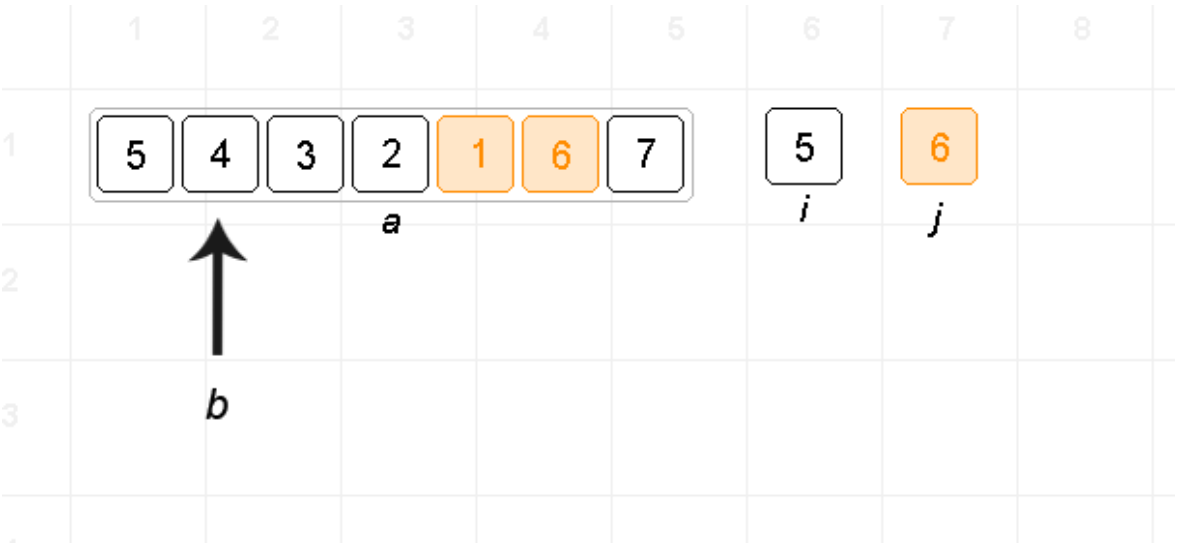
3.4.1 Vizualizácia pomocou dvoch políčok



Obr. 3.1: Náčrt implementácie smerníkov pomocou dvoch políčok

Tento spôsob zobrazenia využije pre každý smerník 2 políčka. V jednom bude zobrazovať názov premennej, na ktorú aktuálne ukazuje a v druhom políčku zobrazí jej hodnotu. Výhodou je jednoduchšia implementácia tohto riešenia v systéme VizzA, avšak nevýhoda je znížená prehladnosť. Preto bude lepšie nájsť inú možnosť.

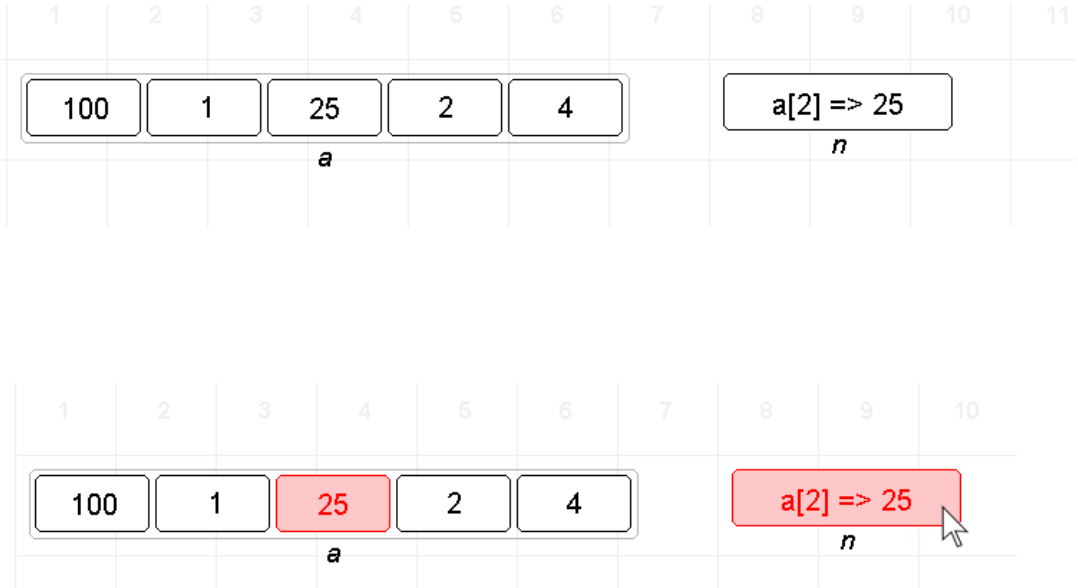
3.4.2 Vizualizácia pomocou čípky



Obr. 3.2: Náčrt implementácie smerníkov pomocou šípky

Pri tomto spôsobe zobrazovania bude šípka ukazovať na premennú, na ktorú je práve nastavený smerník. Výhodou je lepšia prehľadnosť premenných, avšak implementácia v systéme bude zložitejšia.

3.4.3 Zmiešaný prístup vizualizácie



Obr. 3.3: Náčrt implementácie smerníkov, zmiešaný prístup

Najvhodnejší spôsob vizualizovania smerníkov v systéme VizzA je zmiešaný prístup. Tento nám kombinuje výhody prvého a druhého prístupu. Používateľovi zobrazujeme aktuálnu hodnotu smerníka, ale aj hodnotu na ktorú nám ukazuje. Pri nadídení kurzorom myši nad obrázok smerníka vo vizualizačnej časti sa nám tiež rozžiari premenná na ktorú daný pointer ukazuje. To sa ponáša na výhodu šípky v druhom prístupe.

Kapitola 4

Implementácia smerníkov pre systém VizzA

V nasledujúcej kapitole sa zameriame na opis postupu implementácie smerníkov pre systém VizzA. Ako jeden z hlavných zameraní tejto práce sa budeme snažiť rozlíšiť ktoré časti pochádzajú z pôvodnej diplomovej práce [4] a ktoré sú autorovým vlastným prínosom. V prípadoch, kde to je možné, opíšeme rôzne možnosti implementácie a uvedieme prečo sa autor rozhodol práve pre zvolený spôsob. Opäť však nemáme za cieľ zreprodukovať každý jeden krok a myšlienku autora ale budeme opisovať len výrazné a významné zmeny, ktoré boli v systéme vykonané. V tejto časti uvedieme aj ukážky vytvorených zdrojových kódov s vysvetlením a popisom ich funkcionality.

4.1 Reprezentácia smerníkov

Ako už bolo opísané v úvodnej kapitole tejto práce, v jadre systému existuje trieda pre každý typ premennej zo vstupného kódového jazyka. V súčastnosti sú z primitívnych premenných (iba jednoduché hodnoty, žiadne polia, objetky, ...) podporované typy celé číslo, logická hodnota, znak a reťazec znakov. Jednou možnosťou ako reprezentovať premennú typu smerník by bolo vytvoriť pre každú túto primitívnu premennú triedu reprezentujúcu smerník na tento jeden typ premennej. Keďže však chceme implementáciu robiť univerzálne a ľahko rozšíriteľne, reprezentáciu smerníka ponecháme ponecháme jednej, univerzálnej triede. A tou bude `background.systemVariables.PointerVariable`.

V tejto triede môžme pohodlne reprezentovať smerníky za predpokladu, že v jej konštruktore zapíšeme aký typ smerníka má reprezentovať. Následne už len pri každej zmene adresy smerníka skontrolujeme či je priradzovaná premenná, resp. bunka v adrese rovnakého typu. Je to pomerne jednoduché a elegantné riešenie:

```
public void setPointer(String pointer) {
        // ziskanie objektu podla nazvu premennej
        SimpleVariable x = variableManager.getVariableByName(pointer);

        // osetrenie nespravneho nazvu
        if (x == null) {
                variableManager.errorHandler.errorHappened("Pointer");
                return;
        }

        // osetrenie rovnakeho typu smernika
        if ((x instanceof PointerVariable || !type.equals(x.getType())
           ) && type.equals("void")) {
                variableManager.errorHandler.errorHappened("
                   PointerType");
                return;
        }

        // inicializacia
        this.pointer = x;
        this.initialized = true;
        this.pointer.initialized = true;
        this.expression = pointer;
}
```

4.2 Adresný priestor

Ako ďalšiu časť systému je nutné navrhnúť a doimplementovať adresný priestor dostupný pre vstupný program. Pod pojmom (virtuálny) adresný priestor sa rozumie reprezentácia súvislých alebo nesúvislých pamäťových buniek ako súvislá pamäť s lineárnym adresovaním. Keďže je systém vytvorený v jazyku Java, ktorý neposkytuje manuálnu správu pamäte ale namiesto toho ju spravuje automaticky, je len veľmi obtiažne zistiť, kde v pamäti sú reálne uložené dáta reprezentujúce vnútorné premenné. Preto je nutné vytvoriť virtuálny priestor, na ktorom bude môcť systém pri vytvorení novej premennej alokovať 1 blok pamäte.

Keďže cieľom tejto práce je poskytnúť rozumnú podporu vizualizovania smerníkov, považujeme za celkom rozumný predpoklad, že každá primitívna premenná bude v pamäti zaberať len 1 blok. V opačnom prípade by sme mohli špecifikovať potrebné veľkosti počtu blokov (napríklad typ celočíselná premenná bude zaberať v pamäti 4 bloky), čo by bolo

výhodné, ak by sme chceli poskytnúť čo najlepšiu podobnosť jazykom C a Pascal. Avšak z hľadiska vizualizácie by to prinieslo len ďalšie nároky na prípadného používateľa.

Premenné typu pole a dvojrozmerné pole teda budú vždy v pamäti zaberať veľkosť rovnajúcu sa počtu premenných v nich obsiahnutých. Tie budú vždy radené lineárne za seba, začínajúc najnižším indexom až po najvyšší. Tým docielime rovnaké správanie ako pri jazyku C alebo Pascal. Premenné typu štruktúra (opísané až v 5 kapitole) si v pamäti vyhradia rovnaký počet blokov aký je počet ich obsiahnutých premenných, plus 1 blok potrebný na uloženie informácie o ich názvoch a štruktúre.

Adresný priestor budeme reprezentovať v triede `background.AddressSpace`. Jej najdôležitejšími metódami budú `public String getVariable(int address);` ktorá sa bude starať o preklad čísla adresného bloku na názov premennej, ktorá je na tomto bloku uložená. Samozrejme, ak sa na požadovanom bloku nenachádza žiadna, je potrebné vypísať chybu. Opakom tejto metódy je metóda `public SimpleVariable getPosition(String name);`, ktorá dokáže z názvu premennej určiť číslo bloku, na ktorom je uložená. Pre krátkosť tu neuvedieme ukážku zdrojového kódu, čitateľ si ho však môže nájsť na priloženom CD.

4.3 Interpretovanie vstupného jazyka

Ďalej je nutné zmeniť funkcionalitu tried `language.code.CTypeCode` a `language.code.Pascal`. Ako už bolo spomenuté v prvej kapitole tejto práce, tieto triedy sa starajú o interpretáciu vstupných kódových jazykov systému a inicializáciu potrebných tried pre beh systému. Medzi ich dôležitú vlastnosť patrí, že dokážu odstrániť syntaktické rozdiely medzi oboma typmi jazykov a ďalej systém pracuje univerzálne.

Ďalej je nesmierne dôležité na tejto úrovni výrazne rozlíšiť o akú operáciu so smerníkom sa jedná (tabuľka 3.1). Keďže použité operátory a ich syntax pri oboch kódových jazykoch sa líšia, je nutné si zadefinovať nové symboly označujúce každú operáciu. Systém VizzA vnútorne používa nasledovné:

Operácia	Vnútorné označenie
Obsah adresy na ktorú ukazuje smerník	$smernik
Označenie fyzickej adresy smerníka	@smernik
Adresa na ktorú ukazuje smerník	smernik

Tabuľka 4.1: Vnútorné označenie operácií so smerníkmi

Pri parsovaní vstupného programu a inicializácií tried je dôležité korektne označiť a zadefinovať tieto operácie. Tabuľka 4.1 je neskôr intenzívne využívaná pri výpočtoch v triedach reprezentujúcich logické a algebraické výrazy pri operácií zmeny hodnoty premennej alebo smerníka. Tu poznamenáme, že bolo nutné upraviť triedy z balíčka `background.expressions` tak, aby s týmto označením dokázali pracovať. Jedná sa o väčšie množstvo jednoduchých úprav, preto ich ďalej nebudeme uvádzať.

4.3.1 Implementácia kódových jazykov

Implementujme jednotné označovanie operácií uvedené v predchádzajúcej kapitole pre C-type kódový jazyk:

```
/* Vstup: jeden riadok vstupneho programu
* Vystup: korektne oznacenie operacii podla syntaktickych pravidiel
    jazyka a tabulky 4.1
*/
protected String processPointers(String s)  {
        // n = 4; sa prelozi na $n=4;
        // &n sa prelozi na @n
        // *n sa prelozi na n

        for (String name : pointerNames) {
                s = s.replaceAll("(?<![a-zA-Z0-9\\*\\&])" + name + "
                    (?![a-zA-Z0-9])", "\\$" + name);
        }

        for (String name : pointerNames) {
                s = s.replaceAll("(?<![a-zA-Z0-9]) *\\*" + name, name)
                    ;
        }

        s = s.replaceAll("&(?!&)", "@");

        return s;
}
```

Medzi ďalšie úpravy tejto triedy patria obdobné úpravy pre inicializáciu pola smerníkov, ktoré predstavujú špeciálny prípad polí. Tiež je tu potrebné upraviť súčasný zdrojový kód tak, aby dokázal pracovať so smerníkmi. Obdobné zmeny je potrebné vykonať v triede `language.code.Pascal`.

4.3.2 Implementácia operácií

Mezdi ďalšie potrebné zmeny patria triedy z balíčku `background.commands`. Ako už bolo viackrát napísané, starajú sa o správne vykonávanie každého príkazu vstupného kódového jazyka. Najviac zasiahnutá zmenami bude `background.commands.Operation`, ktorá sa stará o správne vykonanie výrazu typu `*p = p + &a + 2;`, kde p je smerník a a je primitívna premenná. Problém, ktorý tu treba vyriešiť je spravovanie smerníkov na oboch stranách daného výrazu.

4.4 Vizualizácia smerníkov

V neposlednom rade je potrebné implementovať podporu vizualizovania smerníkov. Tú má na starosti trieda `visualization.VisualPanel`. Pre správne vykreslenie smerníkov na plohcu je potrebné vytvoriť triedu `visualization.components.PointerComponent`, ktorá dokáže vypočítať rozbery plochy potrebnej pre samotné vykreslenie. Tiež sa tu nachádza adaptér, ktorý správne prevedie vnútornú reprezentáciu smerníka na textovú, opísanú v predošlej kapitole. Implementácia podpory smerníkov si tu tiež vyžiadala množstvo ďalších menších zmien v triede `visualization.VisualPanel` a triedach `visualization.components.MultipleValueComponent`, `visualization.components.TurnedMultipleValueComponent`, `visualization.components.TableComponent`.

Medzi zaujímavé časti implementácie patrí zvýrazňovanie vizualizovanej premennej, na ktorú smerník práve ukazuje tak, ako to bolo popísané v obrázku 3.3. Táto funkcionalita si vyžaduje neustále zachytávanie pohybu kurzora myši nad vizualizačnou plochou a prepočítavanie na aký element sa pod ňou práve nachádza. Na to slúži metóda `public void highlighPointers(int x, int y)`, ktorá ako parameter dostane X a Y súradnicu kurzora myši nad vizualizačnou plochou a nájde element, ktorý sa pod ním nachádza. V prípade ak je daný element smerník, funkcia ho zvýrazdní na ploche a s ním aj premennú na ktorú smerník ukazuje.

V tejto časti pre ich zložitosť ukážky zdrojových kódov neuvedieme, čitateľ tejto práce si ich však môže nájsť na priloženom CD.

4.5 Rozpoznané chyby

Do systému sme pre prácu so smerníkmi doimplementovali nasledovné chybové hlášky:

Hláška	Popis
Incompatible data and pointer types.	Do premennej typu smerník na celé číslo nie je možné priradiť adresu obsahujúcu logickú hodnotu
Pointer out of memory range.	Smerník, alebo daná adresa sa nachádza mimo súčastne zaznamenanej pamäte adresného priestoru
Pointer manipulation error.	Všeobecná chyba pri práci so smerníkom. Smerník nie je inicializovaný alebo sa pokúšame použiť neexistujúci smerník

Tabuľka 4.2: Chybové hlášky pri práci so smerníkmi

4.6 Zhodnotenie

V tejto kapitole sa nám podarilo načrtnúť množstvo potrebných zmien v systéme. Avšak pre limitovaný rozsah tejto práce sme ich mohli ukázať len obmedzený počet. V rámci tejto práce bolo tiež potrebné implementovať, prepísať a hlavne otestovať množstvo ďalších, menších detailov v systéme. Pre zvedavého čitateľa dávame do pozornosti možnosť porovnať zdrojové súbory pôvodného a nami popísaného systému a analyzovať dopad tejto práce na jeho zložitosť.

V záverečnej kapitole tiež uvedieme príklady využitia funkcionalít implementovaných v tejto časti práce.

Kapitola 5

Ďalšie rozšírenia systému VizzA

V rámci rozšírenia možností využitia smerníkov pri vizualizácií behu programu je dobré rozšíriť systém VizzA aj o ďalšiu funkcionalitu. V prípade implementácie dátového typu štruktúra získame možnosť vizualizovať napríklad spájané zoznamy. V nasledujúcej kapitole preto opíšeme možnosti implementácie tohto dátového typu ale aj jeho samotnú implementáciu. Tiež sme názoru, že je tiež vhodné implementovať operátor pre alokáciu bloku pamäte, ktorý rozšíri možnosti práce so smerníkmi. Budeme sa preto zaoberať aj implementáciou variantu funkcie `malloc()`, ktorá je dostupná v jazyku C.

5.1 Dátový typ štruktúra

Dátový typ štruktúra, po anglicky struct, patrí k dôležitým súčastiam procedurálneho programovania. Je to komplexný dátový typ, ktorý definuje zoznam dostupných premenných, ktoré sú uložené pod spoločným názvom v rovnakom bloku pamäti. Na rozdiel od dátového typu pole umožňuje pod rovnakým názvom zoskupiť premenné rôzneho typu a každej tejto premennej dokonca priradiť vlastný názov. Tento dátový typ je pomerne často zastúpený medzi procedurálnymi programovacími jazykmi, vďaka čomu jeho podporu nájdeme aj v jazykoch C a Pascal. Načrtnime teraz ako rozšírime príslušné kódové jazyky v systéme VizzA.

5.2 Návrh implementácie v C-type kódovom jazyku

V jazyku C môžme tento dátový typ definovať nasledovne:

```
typedef struct Point {
   char *c;
   int a;
   int b;
```

```
} Point;
Point p;
```

Kvôli obmedzeniam vo vstupnej vrstve systému VizzA by bolo náročné implementovať definovanie štruktúr podobným spôsobom. Je nutné poznamenať, že systém spracováva vstupný program postupne po riadkoch a na ich základe inštancuje a naplní príslušného potomka triedy `background.commands.Command`. Zmena tohto obmädzenia je možná, avšak vyžadovala by si neprimerané množstvo úsilia. Keďže je však cieľom tejto práce pridať do systému Vizza nové súčasti a nie jeho znovuvytvorenie, rozhodli sme sa tomuto obmedzeniu prispôsobiť syntax kódových jazykov. Nakoniec, nejedná sa o extrémne obmädzenie, aj zápis v jazyku C je možné umiestniť do jedného riadku, len s menšou stratou prehľadnosti. Nie je to však nič kritické.

Predchádzajúci zápis teda umiestníme do jedného riadku a zjednodušíme:

```
define Point: int a, int b, char *c;
Point p;
```

Takto zvolená syntax spĺňa pôvodné obmädzenia na systém VizzA, je relatívne jednoduchá a využiteľná. Môže byť preto doimplementovaná do systému.

5.3 Návrh implementácie v kódovom jazyku Pascal

V jazyku Pascal sa na definovanie dátového typu štruktúra používa kľúčové slovo `record`. Jeho použitie vyzerá nasledovne:

```
Type
    Point = Record
                X, Y : integer;
                C : char;
            End;
Var
    point : Point;
```

Syntax môžme zjednodušiť s použitím rovnakých obmädzení ako v predchádzajúcej kapitole:

```
TYPE Point = X:integer, Y:integer, C:char;
var point : Point;
```

Týmto spôsobom odvodená syntax je vhodným kandidátom pre implementáciu do systému VizzA. Spĺňa všetky uvedené požiadavky a je ľahko zapamätateľná a spracovateľná.

Je však nutné sa zamyslieť nad možnosťou vizualizácie tohto dátového typu. Treba si uvedomiť, či je nutné, aby sme do štruktúr vedeli vkladať jedno a dvojrozmerné polia. Z hľadiska možnosti písania vstupných programov pre systém je dobré, ak podporuje čo najviac rozšírení. Ak sa však na systém VizzA pozrieme ako na vizualizačný a nie programovací nástroj, uvedomíme si, že podpora polí v štruktúrach nie je potrebná a len veľmi ťažko vizualizovateľná. V prípade, ak by štruktúra obsahovala pole, nebolo by možné alebo bolo by len veľmi obtiažne vizualizovať polia dátových štruktúr.

V procese implementácie je teda nutné zvoliť si medzi podporou dátových štruktúr obsahujúcich polia a podporou polí dátových štruktúr. Sme názoru, že z vizualizačného hľadiska je vhodnejšie podporovať druhú spomenutú možnosť. Získame tým možnosť tento dátový typ na vizualizačnom paneli reprezentovať ako akýsi string, čo má za následok zjednodušenie procesu vizualizácie.

5.4 Proces implementácie

Teraz sa pokúsime načrtnúť postup implementácie. Ako prvé je potrebné vytvoriť triedu `background.systemVariables.StructVariable` pre reprezentáciu dátového typu štruktúry. Tá bude obsahovať zoznam premenných a využijeme v nej hlavne metódu prevodu tohto zoznamu do textového reťazca. V rámci zachovania spätnej kompatibility s pôvodným zdrojovým kódom systému musíme vnútorne systéme namiesto operátora `->` používať `.c`. Prevedenie celej štruktúry na reťazec môže vyzerať napríklad nasledovne:

```
public String getValue(String s) {
        String ret = new String();

        // preiterujeme zoznam vsetkych premennych
        for (String name : this.names) {
                String parts[] = name.split(".s");
                SimpleVariable x = this.vm.getVariableByName(name);

                if (x.initialized) ret += parts[1] + ":"+ x.getValue("
                    ") + " | ";
        }
        return ret.substring(0, ret.length() - 2);
}
```

K tejto triede musíme vytvoriť prislúchajúci vizualizačný adaptér `visualization.components.StructComponent`, ktorý potrebujeme pri vykresľovaní dátových štruktúr na vizualizačnú plochu. Tá obsahuje metódy na výpočet potrebného obsahu a veľkosti políčka na ploche. Jej funkcionalita je však značne rovnaká ako pri ostatným vnútorných premenných systému.

Ďalej je potrebné upraviť príslušné triedy dediace od triedy `background.commands.Command`. Tieto úpravy sú pomerne nezaujímavé a ich jediným účelom je zaistiť správne spracovanie práce so štruktúrami, preto ich nebudeme uvádzať. Zaujímavé však môže byť vytvorenie triedy `background.commands.Define`, ktorá predstavuje definíciu vnútorných premenných a názvu štruktúry vo vstupnom programe.

V neposlednom rade je potrebné rozšíriť kódové jazyky systému VizzA o nové syntaktické vlastnosti. Zamerajme sa teraz na C-type jazyk. Tu je potrebné zadefinovať kľúčové slovo `define` do slovníka. V tom prípade systém pri parsovaní vstupného programu a výskyte tohto slova automaticky predpokladá vytvorenie `background.commands.Define` príkazu.

Pri parsovaní vstupného programu sa môže vyskytnúť relatívne veľa prípadov, z ktorých každý musí byť ošetrený zvlášť. Napríklad pre polia štruktúr je potrebné vyfiltrovať znaky `[, ]` a nahradiť ich znakmi `.a, .b` pre zachovanie kompatibility s pôvodným zdrojovým kódom. Tiež je potrebné ošetriť výpočtové operácie, vytváranie štruktúr, ich výpis na štandardný výstup programu a načítanie zo štandardného vstupu. Tieto úpravy sú zložitejšie, ale všetky ich možno nájsť v triede `language.code.CTypeCode`

Pre kódový jazyk Pascal sú úpravy analogické, len s rozdielnymi kľúčovými slovami, a preto ich nebudeme uvádzať.

5.5 Funkcia alloc()

Funkcia `malloc()` patrí do štandardnej knižnice `stdlib.h` jazyka C a C++. Používa sa na manuálnu správu blokov pamäte dynamicky, počas behu programu. Patrí k pomerne často využívaným funkciám jazyka C, avšak v jazyku C++ jej úlohu často preberá operátor `new`. Vzhľadom k súčastnému stavu systému VizzA sme sa ho však tento operátor rozhodli neimplementovať.

Samotná implementácia tejto funkcie sa môže na rôznych systémoch líšiť, jej správanie je však rovnaké. Pri volaní `malloc(10)` v jazyku C dynamicky alokuje súvislý blok pamäti

o veľkosti 10 bytov. Jej návratová hodnota je smerník typu `void`, ktorý obsahuje adresu začiatku alokovaného bloku. O pridelenie tejto pamäte sa zvyčajne stará operačný systém, ktorý zabezpečí, aby sa tento blok označil ako alokovaný a nebol pridelený inej aplikácií. Programátor po ukončení potreby používať tento blok musí zavolať funkciu `free(address_ptr)`, so smerníkom na začiatok bloku, ktorý chce uvolniť. Funkcia následne zabezpečí uvoľnenie tohto bloku. Ak programátor takto neučiní, tento blok je pridelený programu až do ukončenia jeho behu. V tom prípade hovoríme o tzv. úniku pamäte. Vzhľadom k povahe systému VizzA a relatívnej jednoduchosti jeho vstupných programov budeme implementovať len funkciu `alloc()`.

Keďže mnohé funkcie na správu pamäti v jazyku Pascal sú odvodené práve z knižnice jazyka C, funkcia `malloc()` je sa v tomto jazyku používa rovnakým spôsobom ako sme popísali. Ďalej budeme preto implementovať rovnakú syntax pre oba kódové jazyky.

Pri implementácií je dôležité si uvedomiť spôsob využitia tejto funkcie. Vieme, že ju bude zmysluplne možné použiť iba pri operáciach priradenia hodnoty do smerníka. Vytvoríme preto triedu `background.systemVariables.EmptyVariable`, ktorá bude slúžiť ako "prázdna premenná", ktorej hodnotu nie je možné čítať, ani nijakým spôsobom zmeniť. Bude slúžiť iba ako nejaká zarážka, ktorá v pamäti zaberie X blokov a tie následne budú vrátené používateľovi pri volaní funkcie `alloc`.

Teraz už len stačí v triede `background.commands.Operation`, spracujúcej priradenia hodnôt do premenných vyhľadať reťazec `alloc(x)`, namiesto ktorého vytvoríme X premenných typu `background.systemVariables.EmptyVariable` a tento výraz nahradíme adresou prvej z nich. To je možné zabezpečiť týmto kódom:

```
if (s.contains("alloc")) {
        String regexString = Pattern.quote("alloc(") + "(.*?)" +
           Pattern.quote(")");
        Pattern pattern = Pattern.compile(regexString);
        Matcher matcher = pattern.matcher(s);

        while (matcher.find()) {
                Integer bytes = Integer.parseInt(matcher.group(1));
                // vytvori (bytes) krat prazdnu premennu
                int pos = alloc(bytes);

                s = s.replaceFirst("alloc\\("+bytes+"\\)", Integer.
                   toString(pos));
```

```
        }

        // vyraz nanovo rozlozime do tried pre spracovanie
            aritmetickych vyrazov
        decomposeValue(s, null);
}
```

Týmto je proces implementácie dátového typu štruktúra a funkcie `alloc()` úspešne ukončený. Príklady využitia tejto funkcionality v procese vizualizácie je možné nájsť v poslednej kapitole tejto práce.

Kapitola 6

Príklady využitia novej funkcionality

V záverečnej kapitole tejto práce predvedieme niekoľko ukážok vstupných programov. Každý z nich využíva a predvádza novo implementované súčasti systému VizzA. Počet možných vizualizácií sa však ani zďaleka neobmädzuje na uvedené príklady, tie slúžia len pre ilustráciu.

6.1 Príklad 1: Postupné iterovanie adresného priestoru

Nasledujúci program vytvorí niekoľko premenných a do každej z nich uloží jej adresu v adresnom priestore. Celý výpočet v systéme prebieha formou animácie po krokoch

Syntax v jazyku C-type:

```
vizVar int a;
vizVar int b;
vizVar int c[3];
vizVar int d[3][2];
vizVar int *n;

int i;

vizFor (i=2, i<13, i=i+1) {
        n=i;
        *n=i;
}
```

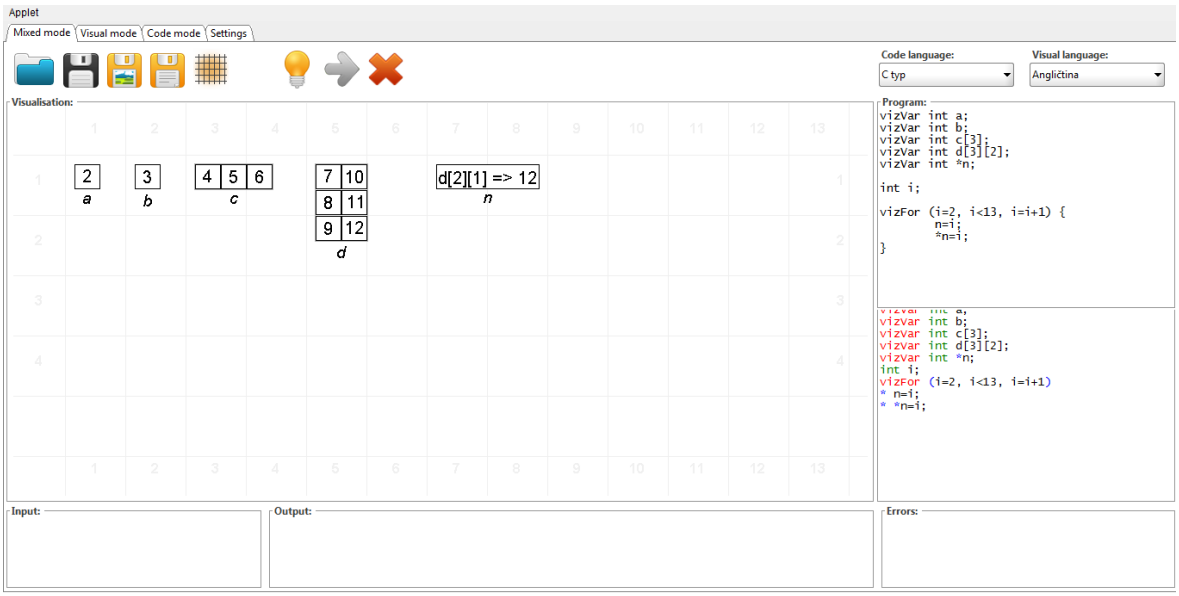
Ekvivalentný zápis v jazyku Pascal:

```
vizVar a:integer;
vizVar b:integer;
vizVar c:array[1..3] of integer;
```

```
vizVar d:array[1..3][1..2] of integer;
vizVar n:^integer;
var i:integer;
begin

vizFor i:=2 to 12 do begin
   n:=i;
   n^:=i;
end;

end.
```



Obr. 6.1: Výsledok vizualizácie príkladu 1

6.2 Príklad 2: Využitie funkcie alloc()

V nasledujúcom príklade využijeme funkciu `alloc()` popísanú a implementovanú v tejto práci. Vidíme, že pri volaní `alloc(20);` sa v adresnom priestore systému alokovala pamäť o veľkosti 20 blokov začínajúca na pozícií 5. Vytvorenie ďalšej premennej (after) má za následok jej umiestnenie na koniec adresného priestoru, adresu 25.

Uvádzame vstupný program v C-type kódovom jazyku:

```
int *p;
vizVar int start;
vizVar int end;
```

```
p = alloc(20);
start = p;
end = p + 19;

vizVar int after;
p = &after;
after = p;
```

Obr. 6.2: Výsledok vizualizácie príkladu 2

6.3 Príklad 3: spájaný zoznam

V poslednom príklade uvedieme možnosť využitia dátového typu štruktúra pri tvorbe spájaných zoznamov. V príklade 4 krát inicializujeme dátovú štruktúru "node"v ktorej každý element obsahuje smerník na ten ďalší. Posledný element obsaju smerník na prvý, čím sa nám podarilo vytvoriť akýsi cyklický spájaný zoznam. Pri použití cyklu for vzniká zaujímavá animácia.

Zdrojový kód tejto vizualizácie:

```
define node: node *p, int a;
vizVar cX#1 cY#1 node w;
vizVar cX#1 cY#2 node x;
```

```
vizVar cX#1 cY#3 node y;
vizVar cX#1 cY#4 node z;

w->p = &x;
x->p = &y;
y->p = &z;
z->p = &w;

node *p;
p = &w;

int i;
vizFor (i=0, i<30, i=i+1) {
        *p->a = i;
        p = p->p;
}
```

Obr. 6.3: Priebeh vizualizácie príkladu 3

Záver

V prvej časti tejto práce sme popísali systém VizzA a analyzovali potrebu implementácie smerníkov do tohto systému. Podľa analýzy dostupných riešení sme prišli na to, že súčasné riešenia sú nevyhovujúce, zaoberajúce sa vizualizáciou smerníkov len okrajovo alebo len veľmi ťažko použiteľné pre bežného používateľa. Zdôrazdnili sme preto potrebu vytvorenia takého systému, ktorý by dokázal smerníky vizualizovať a tak prispieť k vzdelávaciemu procesu programovania na stredných a vysokých školách.

V druhej časti tejto práce sme analyzovali zložitosť a možnosti implementácie samotných smerníkov do systému VizzA. Medzi iným sme si uvedomili, že častokrát bolo potrebné preprogramovať alebo obísť vnútornú implementáciu systému VizzA, pretože na takéto rozšírenia bol nie vždy úplne navrhnutý. Pre každú vrstvu systému sme popísali viacero možností ako systém rozšíriť a následne sme najvhodnejšie riešenia implementovali. Náš postup skončil úspešne.

V tretej časti sme si uvedomili, že samotná implementácia smerníkov nemusí byť pre množstvo vizualizácií dostatočná. Pridaním niektorých funkcií a dátového typu štruktúra, získame možnosť rozšíriť okruh možných vizualizácií.

V poslednej časti sme uviedli niekoľko príkladov, ako novo pridané súčasti používať. Medzi tie najzaujímavejšie patrila napríklad vizualizácia spájaného zoznamu alebo práca s virtuálnym adresným priestorom systému.

Veríme, že výsledok tejto práce pomôže nielen pri pri pochopení základov a vyučovaní informatiky na stredných a vysokých školách, ale aj ľuďom neprogramátorom, snažiacim sa pochopiť základné princípy tvorby algoritmov a programovania.

Literatúra

[1] Attribution-noncommercial-sharealike 3.0 united states. http://creativecommons.org/licenses/by-nc-sa/3.0/us/. [Online; cit. 11.5.2014].

[2] M. A. Azeem. *Start programming using Object Pascal*. 2013. Voľne dostupné na internete pod otvorenou licenciou.

[3] A. Cortesi. Sortvis. http://www.sortvis.org/. [Online; cit. 12.5.2014].

[4] A. Košinárová. *Abstraktný systém na generovanie vizualizácii algoritmov*. Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Bratislava, 2013. Diplomová práca pod vedením J. Katreniakovej.

[5] L. Kucera. Algovision. http://www.algovision.org/. [Online; cit. 12.5.2014].

[6] D. R. Martin. Sorting algorithm animations. http://www.sorting-algorithms.com/. [Online; cit. 9.5.2014].

[7] T. Naps. Java-hosted algorithm visualization environment. http://jhave.org/. [Online; cit. 2.5.2014].

[8] AlgoRythmics project. http://www.youtube.com/user/AlgoRythmics/. [Online; cit. 8.5.2014].

Dodatok A

CD médium

Na priloženom CD sú k dispozícií všetky upravené zdrojové súbory systému VizzA ako aj skomplovaný applet a jednoduchá webová stránka pre jeho spustenie. Tiež sa tu nachádza PDF verzia tejto práce.