UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

ABSTRAKTNÝ SYSTÉM NA GENEROVANIE VIZUALIZÁCIÍ ALGORITMOV

diplomová práca

2013 **Bc. Alena Košinárová**

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

ABSTRAKTNÝ SYSTÉM NA GENEROVANIE VIZUALIZÁCIÍ ALGORITMOV

diplomová práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Jana Katreniaková, PhD.
Evidenčné číslo:	ccc388ca-1af6-4182-84aa-0b5d881a0c84

Bratislava, 2013 **Bc. Alena Košinárová**

64403679

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Alena Košinárová
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský

Názov: Abstraktný systém na generovanie vizualizácii algoritmov

Cieľ: Vďaka rozšíreniu používania e-learningových systémov sa v učebných materiáloch čoraz častejšie namiesto dlhého popisujúceho textu využíva vizualizácia s malým množstvom komentárov alebo sprievodného textu. Špeciálne je tento trend zrejmý v oblasti algoritmov a dátových štruktúr. Vzniká teda čoraz viac vizualizácií, ktoré majú podobný základ a požiadavky. Cieľom práce je navrhnúť abstraktný systém, ktorý v danej oblasti ponúkne možnosť generovať vizualizácie istého typu algoritmov.

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

Dátum zadania: 14.10.2011

Dátum schválenia: 02.11.2011

prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Chcela by som poďakovať svojej diplomovej vedúcej, RNDr. Jane Katreniakovej, PhD., za ochotný prístup, skvelé rady a veľkú trpezlivosť pri vedení mojej práce.

Abstrakt

Hlavným cieľom tejto diplomovej práce je vytvorenie prototypu vizualizačnej aplikácie v jazyku Java, ktorá slúži ako širokospektrálny nástroj pre tvorbu vizualizácií behu algoritmov z rôznej oblasti. Aplikácia je sprístupnená na internete, aby mohla poslúžiť pedagógom na školách všetkých stupňov. Ako sekundárny prínos práca poskytuje pohľad na vizualizáciu z hľadiska učiteľov a rozoberá systematický prístup k vizualizácií programov. Prinášame tiež prehľad spoločných prvkov jednoduchých programov a algoritmov. Práca obsahuje príručku pre používateľa, ako aj metódy pre prípadné ďalšie rozširovanie jej funkcionality.

Kľúčové slová: vzdelávanie, vizualizácia, algoritmy

Abstract

ain purpose of this master thesis is creation of prototype of a visualisation Java application, which is intended as an extensive tool for creation of algorithm visualisations on different topics. Application is placed online, in order to support the learning process and be of use for teachers on any stage of study. As a secondary contribution, the thesis provides educational view of visualisations and analyzes systematic approach in code visualisation. We also add an overview of common code structures in simple algorithms and programs. The thesis includes user manual, as well as methodology for possible future improvement of application's functionality.

Keywords: education, visualisation, algorithms

Predhovor

Vizualizácia dát patrí medzi veľmi efektívne metódy vyučovania nielen na stredných a základných školách (kde je učenie sa z "ilustratívnych obrázkov" bežné), ale tiež na vysokých školách a pracovných školeniach. Jej použitie v e-learningových formátoch, ako aj pri bežnom vyučovaní, sa dostáva na významné priečky v použiteľnosti pre pedagógov aj študentov. Vizuálna interpretácia údajov dáva možnosť nahliadnuť pod formalizáciu zabezpečenú programovacími jazykmi a poskytuje náhľad do diania v programe aj bez aktívnej znalosti zápisového jazyka. Práve vďaka tomu je pre vizualizácie dôležité odbremeniť sa od striktne zvoleného programovacieho jazyka a zamerať sa na samotnú myšlienku programu, algoritmu či iného vizualizovaného formátu.

Obsah

Úvod

Prístupov k vizualizácií je mnoho. Vizualizovať sa dajú nielen existujúce známe fungujúce algoritmy, ale ľubovoľné programy, dokonca aj tie, ktoré by v skutočnosti nefungovali podľa očakávaní kvôli chybám v programe alebo pri kompilacii. Zmysel takýchto vizualizácií je nielen ozrejmenie vzniku a efektu rôznych druhov chýb, ale tiež možnosť ich odhaľovania týmto spôsobom. Ďalším významom je triviálne ozrejmovanie fungovania dátových štruktúr predvádzaním tradičných aj netradičných úkonov s dátami v nich. Vďaka takýmto možnostiam vie pedagóg predviesť napríklad chyby, ktoré vytvára výmena ostrej a neostrej nerovnosti na rôznych miestach pri implementácii haldy, alebo tiež následky neošetrenia kapacity cyklickej fronty.

Dôležitou súčasťou vizualizácie je taktiež možnosť zobraziť paralelne tú istú štruktúru či algoritmus rôznymi spôsobmi. To nám dáva možnosť porovnávať myšlienkovú reprezentáciu s implementačnou a to prípadne niekoľkými spôsobmi naraz. Dobrým príkladom využiteľnosti v tejto oblasti sú štruktúry ako halda, ktorej implementačná štruktúra môže byť napríklad pole, zatiaľčo myšlienková má tvar binárneho stromu. Táto metóda zobrazenia zjednodušuje chápanie súvislostí medzi dátovou štruktúrou a implementačnou reprezentáciou v rôznych fázach programu.

Špeciálnou, príbuznou, oblasťou k vizualizácií behu programov je ich krokovanie pri odstraňovaní chýb[1]. Aj pri tejto činnosti sledujeme vývoj hodnôt v jednotlivých premenných počas behu algoritmu, avšak zvyčajná forma nie je vizualizáciou v pravom slova zmysle. Zobrazenie je totiž prevažne jednoduché a účelové. Napriek tomu však

[1] debugging

zostáva využitie v takýchto aplikáciach možnou, i keď nie primárnou, formou použitia výsledkov našej práce.

Počas vypracovávania svojej bakalárskej práce [3] sa autorka stretla s mnohými vizualizáciami nielen v oblasti grafov. Nešlo len o jednoduché samostatné vizualizačné aplikácie, ale aj o komplexné projekty zaoberajúce sa vizualizáciou zložitých algoritmických postupov. Podrobnejšie sa im budeme venovať hneď v prvej kapitole, v ktorej poukážeme na klady aj zápory existujúcich riešení, ako aj na hlavné motivácie, ktoré nás viedli k voľbe témy tejto diplomovej práce. Keďže sa chceme zamerať na vizualizáciu využiteľnú vo vyučovaní, venujeme jednu kapitolu taktiež vizualizačným kriteriám z hľadiska pedagogického použitia. Neskôr tieto informácie využijeme tiež v kontexte určovania prioritných kritérií pre náš systém.

V ďalších kapitolách sa pozrieme na spoločné prvky bežných jednoduchších programov. Zameriame sa nielen na použitú implementáciu dátových štruktúr, ale tiež na rôzne potrebné spôsoby vizualizácie tej istej implementácie. Následne sa budeme venovať samotnému abstraktnému vizualizačnému sýstému, jeho užívateľskému jazyku, implementačným detailom, ako aj konkrétnym možnostiam jeho využitia. Na záver sa pozrieme na pokročilejšie, menej rozšírené spoločné prvky zložitejších programov a pojednáme o tom, ktoré z nich sú vhodnejšie na vizualizáciu a ktoré sú menej vhodné. V závere sa dostaneme k zhrnutiu výsledku práce a zdôrazníme niekoľko možných doplnení v rámci pokračovania v práci.

Za hlavnú cieľovú skupinu, pre ktorú je táto diplomová práca určená, považujeme učiteľov na stredných školách a vedúcich na mimoškolských krúžkoch a aktivitách. Nevylučujeme využitie na vysokej škole (nielen na našej fakulte) pri základných kurzoch programovania, aj keď štýl práce je určený skôr stredoškolským a základoškolským pedagógom, poprípade študentom. Medzi hlavné podmienky výsledného systému v našich očiach patrí práve zrozumiteľnosť pre nevedeckých pracovníkov v školstve, ako aj pre ľudí, ktorí ešte s vizualizačnými aplikáciami nepracovali.

Veríme, že výsledok tejto diplomovej práce poskytne flexibilnú a "user-friendly"[2] možnosť, vďaka ktorej dostanú do rúk alternatívnu metódu ako vytvoriť študijné pomôcky "na mieru" svojim požiadavkám. Rozvoj "vodcastového"[3] formátu s využitím vizualizačných metód z tejto diplomovej práce je viac než vítaným spôsobom podporujúcim nielen zrozumiteľnosť výkladu, ale tiež využitie technológií Web 2.0.

[2] priateľský k používateľom
[3] http://en.wikipedia.org/wiki/Video_podcast

1 Súčasný stav problematiky

Na začiatok sa pozrieme na rôzne projekty zamerané na vizualizácie algoritmov a/alebo dátových štruktúr, ktoré už existujú. Okrem ich rozsiahlosti a komplexnosti sa zameriame aj na ich jednotnosť, zrozumitelnosť a flexibilnosť. Budeme sa snažiť poukázať na faktory, ktoré nás viedli k záveru, že uvedené projekty nie sú celkom dostačujúce v oblastiach nášho záujmu. Nebudeme uvádzať viac než zopár konkrétnych projektov, ktoré nás niečím zaujali, pretože zozbierať reprezentatívnejší počet obdobných projektov a podrobne ich popísať, by ďaleko presahovalo rozsahový rámec tejto práce. Preto sa radšej zameriame na tie projekty, ktoré ilustrujú niektoré konkrétne nedostatky.

1.1 Rôzne projekty

Najčastejšie vizualizovanými algoritmami sú nepochybne triediace algoritmy, a to prevažne insertsort, bubblesort, quicksort, heapsort a treesort. V tejto oblasti je možné nájsť pravdepodobne všetky druhy vizualizácie, od jednoduchej cez vizualizáciu poľa či grafické znázornenie triedených prvkov, až po rôzne kuriozity. Zaujímavým projektom v tejto oblasti je nepochybne Gallesov projekt "Data Structure Visualizations"[8], ktorý vznikol na univerzite v San Franciscu. Napriek tomu, že názov napovedá, že ide o vizualizácie dátových štruktúr, sú v projekte obsiahnuté aj triediace algoritmy a základné grafové algoritmy. Veľkými pozitívami tohto projektu je vzhľadová jednotnosť jednotlivých vizualizácií, ako aj voľne prístupné zdrojové kódy a teda možnosť doprogramovať si vlastné vizualizácie. Za nedostatok, ktorý je prítomný aj pri ostatných projektoch, sa dá považovať fakt, že akékoľvek úpravy vyžadujú podrobnú znalosť použitého programovacieho jazyka a každú vizualizáciu je nutné vytvárať nanovo.

Medzi ďalšie vizualizačné projekty, tentoraz zamerané čisto na triediace algoritmy, patrí napríklad SortVis[9], ktorý má dizajnovo zaujímavé prevedenie, ako aj rovnomenný projekt[10] z Tuftskej univerzity, ktorý zobrazuje netradičné kritéria. V tomto projekte si vie používateľ prezrieť nielen postupný stav triedeného poľa, ale tiež porovnania či výmeny v rámci poľa. To všetko je farebne vizualizované v rámci dvojrozmerného obrázku. Veľmi prehľadným a pekným riešením je tiež internetová stránka sorting-programs.com[4], ktorá prišla s prehľadným tabuľkovým riešením. Našli sa však aj netradičné možnosti vizualizácie pomocou zvuku[11], alebo ešte netradičnejšie, pomocou ľudových tancov. Projekt AlgoRythmics[5] je skvelým príkladom, ako je možné priblížiť fungovanie algoritmov aj mladším či menej nadšeným študentom.

Zvyšným dvom velkým projektom venujeme samostatné podkapitoly, pretože sú obsiahlejšie a prepracovanejšie ako predchádzajúce uvedené projekty. Je treba poznamenať, že Gallesov projekt[8] by si pravdepodobne tiež zaslúžil miesto medzi nimi, ale rozhodli sme sa uviesť takto samostatne len pár príkladov.

1.2 Algovízia

Prvým z dvoch väčších projektov, na ktorý sa pozrieme, je Algovision[1], čiže Algovízia. Nepochybne zaujímavým faktom o tomto projekte je, že na rozdiel od väčšiny vizualizačných prác je k dispozícii aj v českom jazyku, a teda je prístupnejší aj pre tých, ktorí nemajú odbornú angličtinu v tejto oblasti na dostatočne vysokej úrovni. Ďalším z bodov, ktoré hovoria za Algovíziu je fakt, že je to aktívny projekt, ktorý sa ešte stále zdokonaľuje a upravuje, a teda je vždy šanca, že hľadané vizualizácie či funkcionalita v projekte pribudnú. Taktiež kniha, ktorá je voľne k dispozícii na webovej stránke projektu je príjemným ozrejmením cieľov a fungovania vizualizačných appletov.

Napriek tomu, že je projekt Algovízia na veľmi vysokej úrovni, môžeme vidieť aj niektoré nedostatky v oblasti nášho záujmu. Prvým, logickým, dôvodom je, že má očividne inú cieľovú skupinu než táto diplomová práca. Algovízia obsahuje mnohé pokročilé a

zložité vizualizácie, avšak jej zameranie je skôr na ľudí, ktorí sa v tejto oblasti už dlhší čas pohybujú. Chýbajú v nej jednoduché základy, vďaka ktorým by sa stávala vhodnou aj pre študentov na stredných či základných školách, a teda vhodnou pomôckou pre ich učiteľov. Na druhej strane, ako pokročilý materiál pre študentov, ktorých algoritmické problémy zaujímajú a už istý čas sa pohybujú v tejto oblasti informatiky, je Algovízia skvelým riešením, ktoré je dobre zvládnuté z hľadiska vysvetľovania vizualizovaných algoritmov.

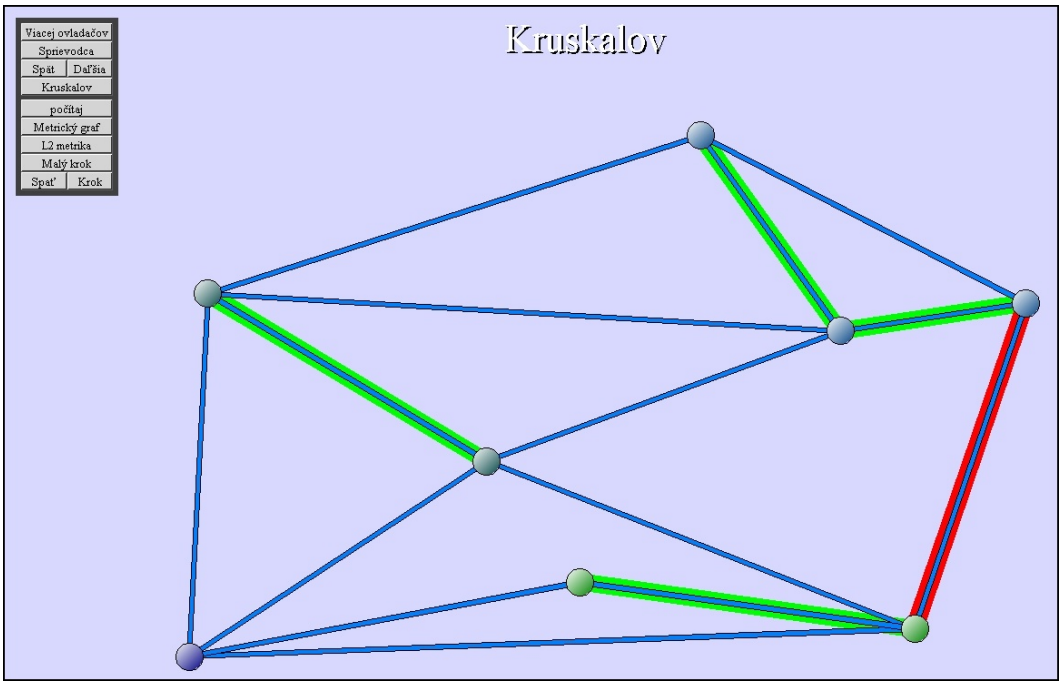
Problematickým by mohol byť aj spôsob ovládania jednotlivých vizualizačných appletov, ktorý je, podobne ako celý projekt, na pokročilej a sofistikovanej úrovni. Pre pokročilých používateľov je tento faktor veľkým plus, ale pre začínajúcich študentov, alebo tiež pre učiteľov, ktorí nie sú na podobné projekty ešte zvyknutí, môže ovládanie pôsobiť prehnane sofistikovane a mätúco. Tento pohľad na vec je dôležitý hlavne pri mladších študentoch, ktorí ešte nie sú natoľko flexibilní čo sa nových spôsobov ovládania appletov týka (kvôli menšiemu množstvu skúseností). Tu vyvstáva dôležitosť možnosti pedagóga nastaviť si možné ovládacie prvky sám, podľa pokročilosti študentov, alebo podľa vysvetľovaného učiva.

Ďalším faktorom, ktorý je prítomný pri väčšine uvedených projektov je, že sa jedná o konkrétne vizualizácie konkrétnych algoritmov, čo znemožňuje používateľom vytvárať si v jednotnom štýle svoje vlastné vizualizácie. A práve táto myšlienka je kľúčovým motívom pre voľbu témy tejto diplomovej práce.

1.3 AlgoViz Project

Druhým projektom, na ktorý sa pozrieme podrobnejšie, je projekt AlgoViz[2], ktorý vznikol na Virginia Tech-u[4] a zahŕňa veľmi širokú paletu rôznych vizualizácií, ktoré sú typovo roztriedené na dátové štruktúry a algoritmy. V rámci týchto kategorií existuje

[4]Virginia Polytechnic Institute and State University, skrátene nazývaná Virginia Tech



Obr. 1: **Ukážka projektu Algovízia.** Konkrétne ide o vizualizáciu Kruskalovho algoritmu.

aj podrobnejšie členenie na podkategórie ako "NP-úplnosť", "Vyhľadávacie algoritmy", "Analýza algoritmov" alebo tiež "Lineárne štruktúry" a "Vyhľadávacie štruktúry". Ďalšie vnútorné členenie závisí od konkrétnej podkategórie, ale čo sa celkovej prehľadnosti týka, je AlgoViz veľmi prepracovaný.

Katalóg beží formou wiki, čo naznačuje aj kooperatívny charakter projektu. Väčšina vizualizácií nemá vzájomný súvis - typový, dizajnový ani účelový - a väčšina sa nenachádza priamo na serveri AlgoVizu. Tento projekt je skôr zbierkou rôznych druhov vizualizačných aplikácií (rôzneho druhu) od rozličných autorov z celého sveta. Táto rozmanitosť však so sebou prináša výhody aj nevýhody.

Medzi neodškriepiteľné výhody patrí, že napriek tomu, že používateľovi "nesadne" niektorá aplikácia, má dobrú šancu nájsť tu inú, ktorá vizualizuje ten istý problém, ale iným spôsobom, či s iným ovládaním. Na druhej strane vytvára táto nejednotnosť problém pri snahe využiť systematicky jeho vizualizácie. Pre každú aplikáciu je totiž

pravdepodobné, že bude využívať iné metódy, iné ovládanie a iný spôsob zobrazovania dát, čo môže popliesť nielen študentov, ale tiež pedagógov, ktorí sa pokúšajú zvoliť najintuitívnejšie dostupné vizualizačné možnosti.

Súčasťou projektu je okrem wiki aj rozsiahly zoznam publikácií, ktoré boli a sú používané v oblasti vizualizácií a algoritmov. Taktiež je zaujímavým doplnkom nedávno spustený[5] projekt OpenDSA Active e-book[6], ktorý prichádza so zaujímavou možnosťou využitia vo vyučovaní, čo je tiež jeho primárnym účelom. Jedná sa však o nový projekt, a teda ešte nie sú známe širšie výsledky aktívneho využívania vo vzdelávaní [7].

[5]Spustený v strede roku 2011
[6]http://algoviz.org/algoviz-wiki/index.php/Main_Page
[7]Stav zo začiatku roku 2012

2 Kritéria pre vizualizácie

Podľa Rößlinga a Napsa[13] existuje 8 primárnych kritérií pre vizualizačné aplikácie. My si ich vymenujeme, vyjadríme náš postoj k nim a následne uvedieme náš zámer ohľadne ich implementácie. Kritéria, na ktoré sme sa zamerali aj v našej práci, sú zvýraznené hrubou tlačou.

- **Dostupnosť pre široké spektrum používateľov:** *Technologická dostupnosť je pri vzdelávacích projektoch naozaj kľúčovou požiadavkou. Podobne ako autori článku, aj my sme siahli po využití Javy a Java Appletu, ktorý bude umiestnený taktiež online, pre ľahšiu dostupnosť.*

- **Širokospektrálne zameranie:** *Zameranie na úzku oblasť môže síce poskytovať možnosť vytvoriť prepracovanejšiu a podrobnejšie vypracovanú aplikáciu, avšak z hladiska všeobecnej použiteľnosti vyžadujeme aj od nášho systému čo najväčší záber ohľadom možností vizualizácie.*

- **Možnosť poskytovania vstupu:** *Variabilita, ktorú poskytuje možnosť vkladania rôznych vstupov, je v dnešnom vizualizačnom svete nutnosťou. V našej aplikácií imitujeme prácu s konzolou - v prípade vstupu aj výstupu.*

- Návrat v krokoch: *Napriek tomu, že dôležitosť spätného chodu vizualizácie je nepopierateľná z hľadiska vzdelávacieho procesu, v našom prípade sme sa rozhodli, že pomer medzi spôsobenou komplikovanosťou aplikácie a úžitkom, ktorý by táto funkcionalita priniesla nie je dostačujúci na to, aby sme sa na túto funkcionalitu zamerali. Je to prevažne vďaka nášmu cieľu, aby bola aplikácia jednoducho rozšíriteľná.*

- Štrukturálne zobrazenie algoritmu: *Napriek tomu, že zobrazovanie štruktúry programu sme do našej aplikácie začlenili, autormi navrhované pristupovanie na ľubovoľné miesta programu sme nahradili špecifickým vizualizačným príkazom, ktorý zastaví vizualizáciu na vopred určenom mieste. Preto by sme zhodnotili toto kritérium ako čiastočne splnené.*

- **Interaktívne predvídanie:** *Aplikované vo forme otázok so zadanou alebo parametrizovanou správnou odpoveďou poskytujú, podobne ako autori navrhli, možnosť korigovať hĺbku vnímania vizualizácie počas jej sledovania.*

- Spolupráca s databázou kvôli hodnoteniu: *Vzhľadom na to, že považujeme vizualizačné prostriedky len za doplňujúce pomôcky pri vyučovaní (a nie za metódu hodnotenia študentov), nepovažujeme toto kritérium za kritické a nutné pre implementáciu.*

- Plynulé prechody: *Plynulosť pohybu získava odôvodnenie v prípade, že sa v aplikácii presúvajú objekty z miesta na miesto. Avšak v prípade ako je náš, kedy zobrazujeme stavy premenných v jednotlivých okamihoch, by prelínanie stavov mohlo používateľov, naopak, miasť.*

Zamerali sme sa teda na splnenie väčšiny kritérií, ktoré boli autormi navrhnuté.

Podľa práce z Virginia Techu[15] sú v praxi najväčšími dôvodmi na nepoužívanie vizualizácií práve problémy s hľadaním vhodnej vizualizácie a tiež následné začlenenie vizualizácie do kurzu. Naša aplikácia preto považuje za kľúčové okrem vyššie uvedených kritérií tiež **jednotnosť spracovania**, **ľahké použitie**, **dobrú rozšíriteľnosť** a **prehľadnú používateľskú príručku**. Tieto kritéria majú potenciál umožniť učiteľom ľahšiu metódu začlenenia vizualizácie do kurzov.

Na záver by sme chceli ešte špecifikovať našu aplikáciu z hľadiska členenia, ktoré poskytli autori z Technickej univerzity v Helsinkách[14].

- **Pôsobnosť (scope)**: naše vizualizácie sú využiteľné v plnom spektre od lekcie

(lesson-specific) až po kurz (course-specific), pričom sú zamerané na programovanie alebo algoritmiku.

- **Integrovateľnosť (integrability)**: V tejto oblasti sme sa zamerali na väčšinu kritérií. Vybrali sme teda ľahkú inštaláciu, modifikovateľnosť (z hľadiska vstupov, programovacieho jazyka, vzhľadu, zobrazovaných premenných, atď), nezávislosť na operačnom systéme, jazykovú nastaviteľnosť, zdokumentovanosť, interaktívne predvídanie (vo forme otázok a odpovedí), ako aj nástroje pre integráciu do online materiálov (vo forme obrázkov, alebo poskytovania importných vstupov pre aplikáciu).

- **Interakcia pri tvorbe vizualizácií**: Primárne výsledky vytvárané našou aplikáciou spadajú do kategórií - konkrétny príklad, vizuálna odpoveď na otázky študentov, statické a dynamické online ilustrácie, statické knižné ilustrácie. Pre všetky tieto kategórie má mať aplikácia priamo implementované nástroje pre ich tvorbu.

- **Predpríprava potrebná na použitie**: Hlavným typom predprípravy je v našom prípade programovanie algoritmu, resp. taktiež jeho dodatočné spracovanie vizualizačným jazykom. Táto predpríprava však môže byť exportovaná vo forme súborov a neskôr opätovne využitá bez prípravy.

- **Interakcia pri sledovaní vizualizácie**: Konzument vytvorenej vizualizácie má možnosť vizualizáciu pasívne sledovať, alebo v prípade, že to typ vizualizovaného programu dovoľuje, upravovať vstupné dáta, alebo interagovať formou odpovedania na kladené otázky.

3 Základné programové schémy

Keďže sa snažíme vytvoriť abstrakný systém na generovanie čo najširšej palety vizualizácií s použitím čo najjednoduchšieho (a teda čo najľahšie osvojiteľného) aparátu, stáva sa kľúčovou voľba konkrétnych prvkov vizualizačného jazyka. Tieto prvky by mali vedieť vizualizovať väčsinu, resp. všetky prvky, ktoré sa opakujú v jednoduchých programátorských úlohách a algoritmoch. Práve hľadaním týchto spoločných čŕt a prvkov sa budeme zaoberať v tejto kapitole a jej podkapitolách. Pozrieme sa na bežné typy príkladov a ich obmeny, aby sme našli spoločné prvky.

Tieto prvky môžeme rozdeliť na niekoľko typov. Prvým sú dátové prvky. Môžeme si klásť otázku, aké formy uloženia a spracovania dát sú najčastejšie využívané v týchto úlohách, ale treba si tiež uvedomiť, že nielen sémantika dát, ale tiež jej implementácia je tým, čo nás zaujíma. Ďalej netreba zabúdať na fakt, že rôzne algoritmické dátové štruktúry sa dajú implementovať rovnakou programovou schémou, a opačne, jedna algoritmická dátová štruktúra sa dá implementovať rôznymi spôsobmi. Konkrétna algoritmická dátová štruktúra nás ovplyvní vo voľbe spôsobov vizualizácie, ktoré prepájajú implementáciou s dátovou štruktúrou - a teda spájajú syntaktický a sémantický význam danej časti programu.

Ďalším typom sú algoritmicko-implementačné prvky. Medzi tieto zarátame nielen bežné operácie s premennými, ale tiež cykly, podmienky alebo prácu so súbormi. Tieto prvky z veľkej časti zodpovedajú sami sebe aj v implementovanom algoritme, aj v samotnej implementácii. Preto nás pri nich zaujíma prevažne frekvencia ich využívania v jednoduchých príkladoch, ako aj spôsob ich využitia. Rozdiely vo vizualizácii v

tejto kategórii plynú skôr zo sémantiky využitia jednotlivých prvkov než z rozdielov v samotnej funkcionalite.

3.1 Príklady jednoduchých programov

V Prílohe A sú uvedené vybrané programy, na ktorých budeme ilustrovať zástupcov riešení bežných jednoduchých programátorských úloh. V nasledujúcej podkapitole si jednotlivé programy rozoberieme a budeme hľadať ich spoločné črty. Zameriame sme sa na tri najčastejšie vyučované jazyky, a teda Pascal, C a Java, preto sú aj naše referenčné príklady písané len v týchto jazykoch.

3.2 Spoločné schémy

V tejto podkapitole sa budeme podrobnejšie venovať jednotlivým spoločným prvkom programov a formám, v akých sa tieto prvky využívajú. Faktom totiž je, že jeden prvok (napríklad pole) sa dá využiť rôznymi spôsobmi vyžadujúcimi rôzne vizualizácie (v prípade poľa napríklad lineárne zobrazenie, cyklické, strom). Okrem iného sa pozrieme na nasledujúce prvky a vlastnosti:

- **Vizualizácia dát** - práca so vstupom a výstupom, alokovanie číselných premenných, jednorozmerné aj dvojrozmerné polia alokované jednorazovo, reťazce a znakové polia, vlastné typy (record, zložené)
- **Základné algoritmické príkazy** - matematické operácie s premennými, podmienka, vetvenie
- **Cykly** - rozdiely medzi for, repeat a while cyklom (resp. obdobách v iných programovacích jazykoch)
- **Ostatné** - knižničné funkcie

3.2.1 Vizualizácia dát

V tejto časti sa zameriame na dáta, čiže na vstupno-výstupné dáta, premenné a polia, ktoré z vizualizačného hľadiska berieme ako skupiny premenných jednotného charakteru, ktoré sú indexované. Presnejšie, pôjde o jednorázovo alokované polia (a teda nie dynamické), ktoré sú jednorozmerné alebo dvojrozmerné a obsahujú premenné ľubovoľných vizualizovateľných typov. Podrobnejšie si rozoberieme, aké rôzne využitia môže pole mať, a prípadne aké sú z toho plynúce obmedzenia.

Konkrétne ide o tieto prvky:

- **práca so vstupom**

  Príklady: *[riadky 4 a 6, KSP 23-3-2], [riadky 4, 8 a 10, KSP 28-2-2], a pod.*
  Príkazy: *read, readln, scanf, využitie InputStreamReader, ...*

- **práca s výstupom** - Dôležitým doplnením tu je, že v určitých programoch sa dá za výstup považovateľ aj príkaz "return".

  Príklady: *[riadok 17 a 34, KSP 24-2-6], [riadky 15 a 16, KSP 28-2-2], [riadky 13 a 16, Pape, Arrays - Anagrams], a pod.*
  Príkazy: *write, writeln, system.out.println, printf, ...*

- **tvorba hodnotových premenných**

  Príklady: *[riadok 1, KSP 28-2-2], [riadky 11 a 20, Pape, Arrays - Anagrams], [riadky 13, 19 a 28, KSP 24-2-6] a pod.*
  Príkazy: *var i:integer, Integer i, int i, ...*

- **jednotná pamäť** - častým využitím poľa v jednoduchých príkladoch je pohodlná implementácia viacerých sád údajov tak, aby nám zvolená metóda zabezpečila požiadavky, ktoré máme. Tieto sady chceme spracovávať rovnakým spôsobom a požadujeme možnosť iteratívne prechádzať všetky tieto sady uložené v pamäti.

  Príklady: *[riadok 14, KSP 24-2-6], a pod.*

- **fronta, cyklická fronta** - v prípade fronty potrebujeme tiež dve pridružené premenné zodpovedajúce "hlave" a "chvostu" fronty.
- **zásobník** - v prípade zásobníka potrebujeme tiež jednu pridruženú premennú zodpovedajúcu "vrchu zásobníka".
- **reťazec** - s reťazcom sa dá pracovať ako s celkom alebo ako s pseudopoľom znakov.

  Príklady: *[riadky 3 a 4, Pape, Arrays - Anagrams], a pod.*
- **dvojrozmerné polia** - pri dvojrozmerných poliach je bežnou intuitívnou reprezentáciou tabuľka.
- **pole indexujúce iné pole** - dve súvisiace polia, z ktorých hodnoty jedného sú indexami do druhého poľa.
- existujú aj **ďalšie špecifické spôsoby využitia** (napríklad pole, ktoré je indexované rôznymi hodnotami vstupu a zjednodušuje zisťovanie počtu výskytov prvkov na vstupe), avšak tieto druhy použitia sú málopočetné, a pri veľkej potrebe vizualizácie nahraditeľné niektorou z predchádzajúcich možností; vo väčšine prípadov možnosťou, kedy je pole zobrazené ako jednotná pamäť

  Príklady: *[riadok 2, KSP 28-2-2], a pod.*

3.2.2 Základné algoritmické príkazy

V tejto kategórii príkazov sú základné príkazy, ktoré sú prítomné v každom rozšírenejšom programovacom jazyku. Podrobne si vymenujeme jednotlivé príkazy:

- **operácie s premennými** - vizualizovať konkrétne operácie, ktoré sa vykonávajú s premennými, je možné do rôznej hĺbky; základnou možnosťou je vizualizovať len výsledné zmeny v jednotlivých premenných; podrobnejšou možnosťou je vizualizovanie všetkých zainteresovaných premenných; poslednou fázou je vizualizovanie operácií, ktoré sa vykonávajú a vizualizovať priebežné výsledky pri

komplikovanejších matematických úpravách; prvá možnosť je veľmi vhodná pri pomocných premenných, ktoré inkrementujeme a dekrementujeme a druhá možnosť je vhodná ako štandardná pre väčšinu ostatných premenných, u ktorých sa očakáva vzájomná interakcia

Príklady: *[napr. riadky 11, 21 a 22, KSP 24-2-6], [riadok 22, Pape, Arrays - Anagrams], a pod.*
Príkazy: *inc, dec, ++, –, i=, i:=, ...*

- **podmienkové vetvenie** - overovanie platnosti podmienky sa využíva samostatne, v rámci podmienených príkazov, ale tiež pridružene v rámci podmienených cyklov

  Príkazy: *if-then-else, ...*
  Príklady: *[riadok 4, KSP 28-1-5], [riadok 15, KSP 28-2-2], a pod.*

- **viacnásobné vetvenie** - v prípade, že sa chceme rozhodovať podľa hodnoty niektorej premennej, ktorá môže nadobúdať viac možností, prichádza do úvahy aj možnosť viacnásobného vetvenia (ktoré je samozrejme redukovateľné na viacnásobné použitie podmienkového vetvenia, avšak v danej redukovanej forme sa môže stať neprehľadným a mätúcim); je otázne, či miera využívania tejto možnosti dosahuje až takú markanciu, aby malo zmysel vytvárať pre ňu samostatné metódy

  Príkazy: *case ... of, switch, ...*

3.2.3 Cykly

Treťou dôležitou kategóriou spoločných prvkov vo väčšine programov sú cykly, vďaka ktorým vieme regulovať opakovanie niektorej časti programu. Cykly vieme rozdeliť na dve základné podskupiny - cykly končiace podmienkou a cykly s určeným počtom opakovaní. Napriek tomu, že vo svojej podstate sa dajú simulovať v istých podmienkach

tieto podskupiny navzájom, pre nás je dôležité, že práve toto je ich hlavným rozdielom. Pozrime sa na ne teda konkrétnejšie:

- **cykly s podmienkou** - v prípade, že máme podmienku, ktorá ohraničuje dĺžku behu, môžu nastať dva možné prípady: buď je daná podmienka platná počas behu cyklu a po jej porušení cyklus končí, alebo sa jedná o podmienku, na ktorú čakáme, aby sme cyklus skončili; je triviálne nahliadnuť, že sa jedná o ten istý prípad, len v znegovanom podaní

  Príkazy: *while, repeat*
  Príklady: *[riadky 5-10, transformovateľné na ďalší prípad, KSP 23-3-2], a pod.*

- **cykly s pevným počtom opakovaní** - tento druh cyklov sa zvyčajne priamo viaže na jednu premennú, ktorá sleduje koľký beh iterovanej časti programu je aktívny a jej hodnota sa mení zvyčajne len automaticky, nie umelo v rámci programu

  Príkazy: *for, for each*
  Príklady: *[riadky 5-6 a 14-15, KSP 28-2-2], [riadky 10-12, Pape, Arrays - Anagrams], a pod.*

- **Vnorené cykly** - vnorené cykly sú intuitívne zodpovedajúce viacerým súbežne pracujúcim cyklom potenciálne rôzneho typu a s rôznou dĺžkou behu

  Príklady: *[riadky 7-13, KSP 28-2-2], a pod.*

3.2.4 Ostatné

V tejto časti sa zameriame na rôzne zatiaľ nerozoberané časti programov, ktoré sa vyskytujú aj v najzákladnejších príkladoch, ale nebudeme ich vizualizovať. Pôjde o:

- **knižničné funkcie** - funkcie, ktoré programátor používa v programe ako hotové a vkladá ich z niektorej knižnice je náročné, až nemožné, vizualizovať; práve

vlastnosť, že ani samotný programátor nenahliada do presného fungovania týchto funkcií, nám bráni vizualizovať ich priebeh

Príkazy: *STL knižnica v C, hotové objektové funkcie v OO jazykoch*[8]*, ...*

Príklady: *[riadky 3, 4 a 18, Pape, Arrays - Anagrams], [riadok 26, KSP 24-2-6], a pod.*

3.2.5 Dodatok

S použitím vymenovaných kategórií prvkov vieme vizualizovať percentuálne veľmi vysokú časť bežných programátorských úloh ľahkej až strednej obtiažnosti.

[8] Objektovo-orientované programovanie

4 Vizualizácia dát

Pod dátami budeme chápať jednak jednotlivé premenné programu a ich obsah, ale tiež dáta, ktoré dostáva program na vstupe, alebo ktoré vypisuje na výstup. Tie sú viditeľné pre používateľa aj bez vizualizačných pomôcok, a teda by bolo výrazným nedostatkom opomenúť ich pri vizualizácii. Na rozdiel od ostatných vizualizovaných dát, vstup a výstup budeme vizualizovať samostatne, a to formou jednoduchého zobrazenia vstupného a výstupného textu v danom čase. V ďalšej časti tejto kapitoly už teda nebudeme brať dáta vstupu a výstupu do úvahy.

V našom prípade sa potrebujeme zamerať na znázornenie jednak sémantického významu, ktorý je dátami reprezentovaný, ale tiež ich syntaktickej implementácie, ktorá je kľúčová na vytvorenie spojiva medzi časťami algoritmu. Z dôvodu rozlíšenia jednotlivých vrstiev a častí vizualizácie sme sa rozhodli zaviesť niekoľko nových pojmov. Najprv si stručne zhrnieme významy a rozdiely týchto pojmov a následne sa v podkapitolách budeme venovať ich konkrétnym príkladom v našej výslednej aplikácii. V poslednej podkapitole sa pozrieme na použiteľnosť vybraných vizualizovaných častí pre jednotlivé spoločné programové schémy.

Upozornenie: Pojem dátová štruktúra tu využívame v inom než v informaticky zaužívanom význame.

- **Dátová štruktúra** je v našom ponímaní syntaktický prvok, v ktorom sú dáta zaznamenané. Sama o sebe má prevažne, ak nie úplne, syntaktický význam a jej reprezentácia hovorí práve o štruktúre, akú majú zobrazované dáta. Sama o sebe nie je vizuálne zobraziteľná.

Príkladom je pole celých čísel.

- **Vizualizačná technika** je priradená k niektorej dátovej štruktúre v prípade, že chceme dátovú štruktúru zobraziť vizuálne. Tá istá vizualizačná technika môže byť aplikovaná na viac dátových štruktúr, a tá istá dátová štruktúra môže byť zobrazená viacerými vizualizačnými technikami. Vizualizačná technika priraďuje štruktúre dát sémantický zmysel a vyjadruje tiež vzťahy medzi nimi.
  Príkladmi pre štruktúru poľa celých čísel sú zoznam rovnocenných údajov, strom, pole smerníkov.

- **Dizajnový variant** určuje konkrétny výzor grafickej reprezentácie vizualizačnej techniky. Určuje doplňujúce sémantické informácie, sémantické dôrazy alebo sa tiež prispôsobuje konkrétnej cieľovej skupine či zameraniu.
  Príkladom pre vizualizačnú techniku stromu je červeno-čierny strom, alebo tiež strom so zobrazením vnútornej štruktúry vrcholov.

4.1 Dátové štruktúry

Uvedieme si zoznam dátových štruktúr, ktoré budeme rozlišovať. Vďaka častým spoločným črtám a faktorom je ich reálna implementačná štruktúra viacúrovňová a stretávame sa s niekoľkými abstraktnými dátovými štruktúrami. Konkrétne spoločné črty a vzťahy je možné vidieť na Obrázku 2. Ku každej reálnej aj abstraktnej dátovej štruktúre si stručne uvedieme, z akého dôvodu má samostatnú kategóriu, alebo čo všetko pod jej názvom budeme chápať.

Reálne dátové štruktúry:

- Premenná - *Medzi nekonkretizované premenné rátame napríklad znakové premenné. V ich prípade potrebujeme jasne vizualizovať jej obsah, avšak metódy zobrazenia sú v tomto prípade len veľmi obmedzené.*

- Reťazec - *Premenné štandardného typu String môžeme na rozdiel od ostatných*

premenných chápať nielen ako jeden celok. Môžeme s nimi taktiež pracovať ako s určitou formou poľa znakov.

- Pravdivostná hodnota - *Pri premenných štandardného typu Boolean nás často nezaujíma priamo napísaná hodnota premennej (či už vo forme "true" vs. "false" alebo 1 vs. 0), ale stačí nám akákoľvek forma, ktorá nám jasne určí jednu z dvoch možných hodnôt.*

- Číselná premenná - *Pri číselných premenných nás zaujíma konkrétna hodnota, avšak tá môže byť v prípade čísel reprezentovaná rôznou formou a preto budeme považovať takúto premennú za isté "rozšírenie" nekonkretizovanej premennej. Taktiež je možné chápať číselnú premennú ako index do poľa, kedy vyžaduje zobrazovacie techniky prislúchajúce ku smerníkom.*

- Pole - *Pole zložené z premenných, ktoré pre nás nie sú ničím špeciálne a teda zodpovedajú dátovej štruktúre Premenná.*

- Pole reťazcov - *Pole reťazcov môžeme chápať buď ako jednorozmerné pole jednotlivých reťazcov, alebo v špecifických prípadoch ako dvojrozmerné pole znakov.*

- Pole čísel - *Pole pozostávajúce z číselných premenných.*

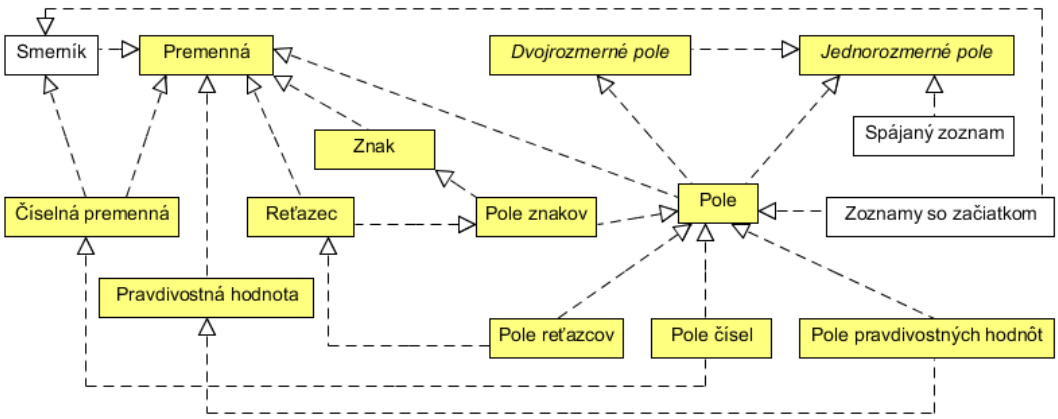
- Pole pravdivostných hodnôt - *Pole pozostávajúce z premenných typu Boolean.*

- Smerník - *Smerníky môžeme chápať ako jednoduché premenné zobrazujúce index v pamäti, avšak častejšie je smerník chápaný ako "šípka" ukazujúca na niektorú premennú.*

Zložitejšie dátové štruktúry pozostávajúce z niekoľkých premenných:

- Spájaný zoznam - *Sady údajov naviazané na seba v určitom poradí danom smerníkmi na ich následníka a príp. predchodcu.*

- Zoznamy so začiatkom - *Polia, ktoré zodpovedajú frontám, zásobníkom, cyklickým frontám a pod. majú spoločný vizualizačný základ. Existuje na nich jedno alebo*



Obr. 2: **Vzájomné vzťahy vysvetlených dátových štruktúr.** Farebne zvýraznené sú štruktúry implementované v našom prototype.

> *dve význačné miesta, na ktoré ukazujú reálne alebo sémantické (vo forme indexovacej premennej) smerníky. Tieto smerníky sú súčasťou dátovej štruktúry a sú zobrazované spoločne.*

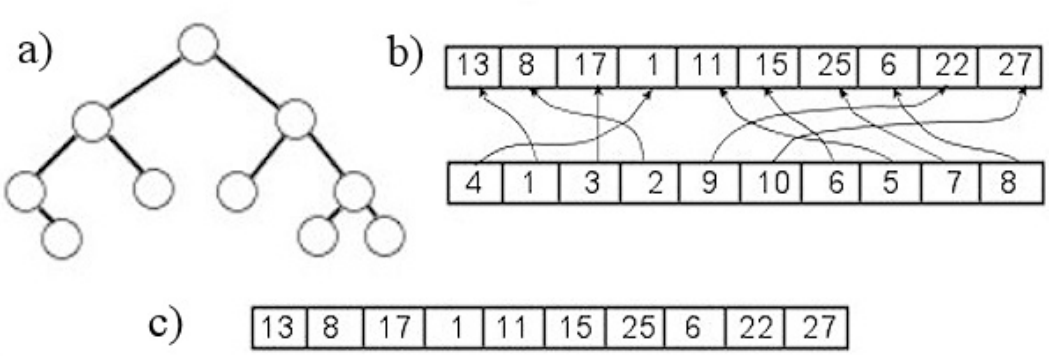
Pod abstraktnými dátovými štruktúrami chápeme súhrny spoločných vlastností niektorých dátových štruktúr, pričom tieto súrhny nezodpovedajú žiadnej samostatne sa vyskytujúcej dátovej štruktúre:

- Jednorozmerné pole - *Jednorozmerné pole statickej dĺžky má jednoznačné štandardné zobrazenie vo forme jasne oddeleného zoznamu hodnôt jednotlivých položiek.*

- Dvojrozmerné pole - *Dvojrozmerné pole statickej veľkosti je štandardne zobrazované vo forme kompletnej tabuľky alebo vo forme oddeleného zoznamu jednotlivých jednorozmerných polí.*

4.2 Vizualizačné techniky

Vizualizačné techniky majú medzi sebou vzťahy podobného druhu ako sme predstavili pri dátových štruktúrach. Avšak existencia vizualizačných techník je priamo



Obr. 3: **Ilustrácia významu pojmu Vizualizačná technika.** Na obrázku sú zobrazené vizualizačné techniky aplikované na "Pole čísel": a. Strom so Zobrazením existencie; b. Indexovacie pole; c. Blokové pole so Zobrazením hodnoty

naviazaná na dátové štruktúry a teda uvedieme taktiež diagramy hovoriace o ich vzájomných vzťahoch. Samotné vizualizačné techniky by sa dali rozdeliť do troch kategórií: zobrazenia samostatných dát, zobrazenia skupiny dát, zobrazenia vzťahov.

Na rozdiel od dátových štruktúr, ktoré nemajú vlastný ”vzhľad”, pri vizualizačných technikách sa už dá ukázať ilustratívny obrázok. Na Obrázku 3 je ukážka niekoľkých základných vizualizačných techník aplikovaných na dátovú štruktúru ’Pole čísel’. Avšak aj v tomto prípade je nutné poznamenať, že pri formálnom ponímaní našej terminológie je výsledný obrázok výsledkom aplikácie nielen vizualizačnej techniky, ale tiež určitého ”štandardného” dizajnového variantu.

Zobrazenia samostatných dát:

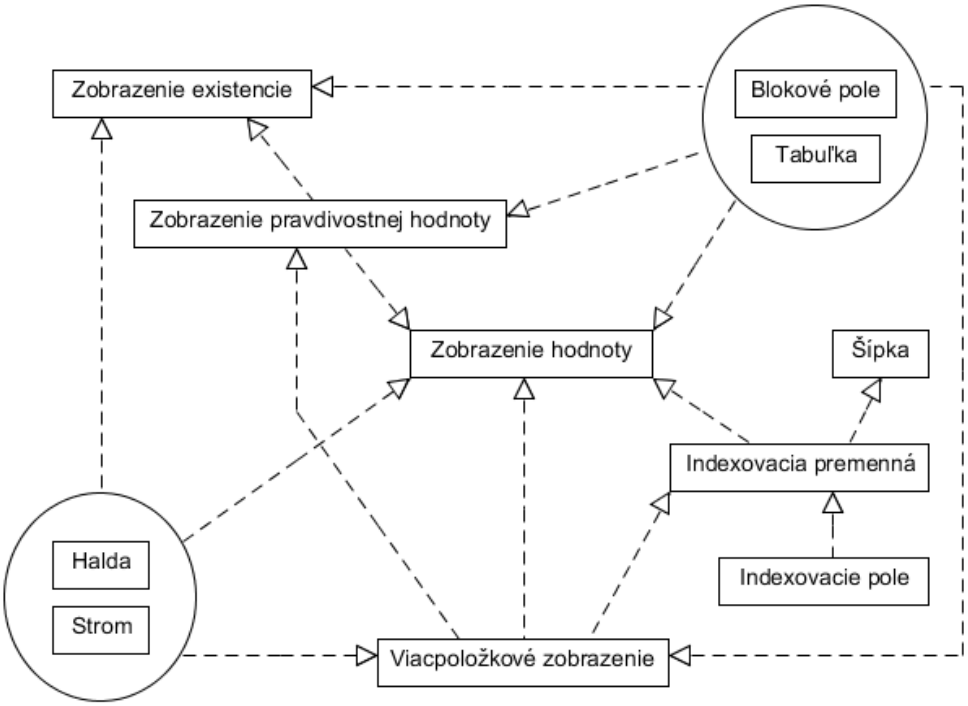
- Zobrazenie existencie - *Túto možnosť využívame v prípadoch, kedy nám stačí zamerať sa na existenciu niektorých premenných a ich samotná hodnota nie je kľúčová, na rozdiel od ich vzniku či odstránenia (napríklad pri rekurziách, pomocných premenných, atď).*

- Zobrazenie hodnoty - *Najčastejšie vyžadovanou variantou vizualizácie premenných je práve priebežné zobrazovanie ich hodnoty.*

- Zobrazenie pravdivostnej hodnoty - *Špeciálnou možnosťou zobrazenia pre pravdivostné hodnoty je možnosť, pri ktorej sa líši jej zobrazenie v prípade hodnoty*

"pravda" a v prípade hodnoty "nepravda".

- Indexovacia premenná - *Špecifickým prípadom zobrazovania číselnej premennej je prípad, keď je táto premenná využívaná na indexovanie pri prechodoch poľom.*



Obr. 4: **Vzájomné vzťahy vizualizačných techník.**

Pod zobrazovaním skupiny dát si predstavujeme zobrazenie polí, záznamov[9], spájaných zoznamov a príbuzných dátových štruktúr, pri ktorých zohrávajú dôležitú úlohu vzťahy medzi jednotlivými členmi skupiny. Zobrazenie jednotlivých členov sa dá v niektorých prípadoch prebrať priamo zo zobrazovacích techník pre samostatné dáta.

- Viacpoložkové zobrazenie - *Zobrazenie spolu súvisiacich premenných, resp. viaczložkových objektov. Takéto dáta chceme zobrazovať združene a zdôrazniť ich vzájomný súvis. Napr. record v Pascale, smerníky a hodnota prvku pri spájanom zozname, atď.*

[9]Pascal: `record`

- Blokové pole - *Jednoduché zobrazenie poľa, v ktorom sú jasne oddelené jednotlivé jeho prvky, pričom tie sú zobrazené niektorou zo zobrazovacích techník pre samostatné dáta.*

- Indexovacie pole - *Pole indexovacích premenných.*

- Strom - *Do kategórie Strom budeme zaraďovať celú sadu zobrazení, ktoré sú príbuzné svojím vzhľadom, ale líšia sa metódou vzniku. Vyžadujú totiž rozdielne dodatočné informácie o konkrétnom rozložení vrcholov v rámci stromu (napr. úplný k-árny strom, strom so smerníkmi, atď)*

- Halda - *Zobrazenie príbuzné so stromovým zobrazením, ale vzhľadom na vlastnosti haldy nevyžaduje žiadne dodatočné údaje o rozmiestnení vrcholov v rámci poľa.*

- Tabuľka - *Zobrazenie typické pre dvojrozmerné polia, ktoré je vo forme tabuľky.*

Zobrazenia vzťahov zodpovedajú rôznym prepájacím elementom, ktoré sa týkajú jednej alebo viacerých dátových štruktúr:

- Šípka - *Pri programoch sa môžeme stretnúť so smerníkmi, ale tiež s rôznymi indexovacími premennými, pri ktorých jedna premenná určuje svojou hodnotou inú premennú.*

Dôležitým faktorom pri vizualizačných technikách je ich priradenie k jednotlivým dátovým štruktúram. Zatiaľčo niektoré sú aplikovateľné univerzálne, iné sú užšie špecializované, a teda je potrebné jasne určiť účel ich využitia. Tento jav vnímame prevažne v prípade rozdielov medzi jednoduchými a zloženými (obsahujúcimi viac ako jednu premennú) dátovými štruktúrami.

4.3 Dizajnové varianty

Dizajnové varianty poskytujú variabilitu v prípade konkrétneho vzhľadu vizualizácie, ako aj zdôraznenie niektorých dodatočných vlastností vizualizácie a sémantiky

Vizualizačná technika	Implementácia
Zobrazenie existencie	áno
Zobrazenie pravdivostnej hodnoty	áno
Zobrazenie hodnoty	áno
Indexovacia premenná	nie
Šípka	nie
Viacpoložkové zobrazenie	čiastočná
Blokové pole	áno
Indexovacie pole	nie
Halda	nie
Strom	nie
Tabuľka	áno

Tabuľka 1: Tabuľka znázorňujúca implementované techniky.

programu. Najčastejším využitím však ostáva určovanie konkrétneho vzhľadu vizualizovaných premenných z hľadiska prehľadnosti, estetiky a názornosti.

Medzi najdôležitejšie vlastnosti dizajnových variantov patria:

- **Univerzálnosť:** Dizajn by mal byť aplikovateľný (resp. mať implementované potrebné nástroje) na valnú väčšinu vizualizačných techník, pretože inak by sme dospeli k rozporu medzi naším riešením a našou požiadavkou na možnosť vytvorenia jednotného vzhľadu väčšieho množstva vizualizácií. Špecifickou odchýlkou môžu byť dizajnové varianty, ktoré prinášajú výrazný sémantický význam.
- **Prispôsobiteľnosť:** Dizajn by mal aspoň v prípadoch farebných schém ostať z väčšej časti prisposobiteľný (výnimky využívajúce špecifické obrázky sa vymykajú tejto požiadavke), opäť z dôvodov jednotnosti výsledného vzhľadu vizualizácie.
- **Jednoduchosť:** Funkcionalita dizajnových variantov by mala ostať len v hladine formy vykresľovania jednotlivých častí vizualizačných variantov. Nemali by zasahovať do rozloženia jednotlivých premenných voči sebe navzájom.

4.4 Aplikácia na spoločné schémy

Teraz si uvedieme tie prvky spoločných programových schém, na ktoré budeme aplikovať pojmy a termíny zavedené v predchádzajúcich podkapitolách. Zameriame sa pri tom aj na uvedenie komentárov týkajúcich sa priamo vizualizácie týchto prvkov.

- **práca so vstupom** - načítanie vstupu môže byť vizualizované globálne, kedy vidíme celý budúci vstup a vizualizujeme jeho spracovanú časť, práve načítanú časť a ešte nespracovanú časť; druhou možnosťou vizualizácie vstupu, ktorá je vhodná najmä pri priebežných vstupoch, je simulovanie postupného vznikania vstupu, pri ktorom vidíme len práve načítanú časť vstupu na špeciálnom mieste; treťou základnou možnosťou, je možnosť interaktívneho zadávania vstupu

- **práca s výstupom** - v prípade vizualizácie výstupu závisí konkrétna voľba od toho, či je výstup generovaný priebežne (kedy je vhodné ponechať ho priebežne viditeľný a dopĺňať ho), alebo hromadne na konci programu (kedy stačí výstup ukázať jednorazovo vo chvíli, kedy už nie je treba vizualizovať ostatné prvky)

- **tvorba hodnotových premenných** - vytvorenie novej premennej zodpovedá pridaniu novej zodpovedajúcej dátovej štruktúry (*Premenná* a rozširujúce dátové štruktúry) a priradenie vhodných vizualizačných techník a dizajnových variantov. Tie odlišujú tiež rozdiel medzi lokálnymi a globálnymi premennými. Taktiež ostáva možnosť tú-ktorú premennú vôbec nevizualizovať.

- **jednotná pamäť** - zodpovedá dátovej štruktúre *Pole*, alebo niektorej inej dátovej štruktúre, ktorá rozširuje *Pole* (čiže *Pole čísel*, *Pole pravdivostných hodnôt*, *Pole reťazcov*)

- **fronta, cyklická fronta** - zodpovedá dátovej štruktúre *Zoznam so začiatkom*, pričom k hlavnému poľu prislúchajú dva smerníky. Spoločne s vhodnými vizualizačnými technikami a dizajnovými variantami je možné zobraziť štandardnú

verziu fronty s ”hlavou” a ”chvostom”, ako aj dizajnovú variantu pre kruhové zobrazenie cyklickej fronty.

- **zásobník** - zodpovedá dátovej štruktúre *Zoznam so začiatkom*. V kombinácii s vhodnými vizualizačnými technikami a dizajnovými variantami vytvára zobrazenie primerané konvenčnému zvislému zasobníku, do ktorého pribúdajú prvky smerom nahor. Patrí k nemu jeden smerník zodpovedajúci ”vrchu” zásobníka. Samotný variant zobrazenia zodpovedá zvislému zobrazeniu poľa, na rozdiel od štandardného vodorovného.

- **reťazec** - zodpovedá dátovej štruktúre *Reťazec*.

- **strom, halda** - zodpovedajú dátovým štruktúram *Pole* (alebo rozširujúcim štruktúram k tejto štruktúre) s vizualizačnou technikou *Strom*, resp. *Halda*.

- **dvojrozmerné pole** - zodpovedá dátovej štruktúre *Pole* s využitím rozšírenia abstraktnej dátovej štruktúry *Dvojrozmerné pole* a vizualizačnej techniky *Tabuľka*.

- **pole indexujúce iné pole** - zodpovedá dátovým štruktúram *Pole* (alebo rozširujúcim štruktúram k tejto štruktúre) s vizualizačnou technikou *Indexovacie pole*.

5 Implementácia systému

V tejto kapitole sa budeme venovať reálnej implementácii systému z hľadiska programátora. Na začiatku uvedieme požiadavky, ktoré na tento systém kladieme, aj s ich odôvodnením. Ďalej si spomenieme princípy, na ktorých je založené jadro systému, v jadre využité programové schémy, ako aj prepojenia k jednotlivým vizualizačným častiam. Popíšeme taktiež princípy fungovania vizualizačnej časti systému a na záver si spomenieme niekoľko vybraných "doplnkových" funkcionalít. K problematike implementácie sa vrátime ešte v jednej z posledných kapitol, v rámci popisu potrebných postupov pri rozširovaní systému. Nebudeme sa venovať konkrétnym príkazom a syntaxi vizualizačných jazykov, či zoznamom implementovaných prvkov, pretože na tento účel slúži používateľská príručka, ktorú sa dá nájsť v Prílohe B ako súčasť tejto diplomovej práce.

5.1 Požiadavky na systém

Medzi základné vlastnosti, ktoré požadujeme od nášho systému patria:

- **rozšíriteľnosť:** Pod rozšíriteľnosťou chápeme možnosť pridávať rozumným spôsobom novú funkcionalitu bez toho, aby sme museli upravovať kompletne celý systém. Konkrétnymi postupmi v prípade rozširovania systému sa budeme zaoberať v kapitole 7, avšak tu si uvedieme, pre ktoré typy funkcionality by sme primárne chceli umožniť:
  - pridávanie nových rozpoznávaných slov (aj pre jednotlivé jazyky selektívne)
  - pridávanie (z funkčného hľadiska nových) príkazov

  - pridávanie ďalších typov systémových premenných
  - pridávanie nových vizualizačných postupov
  - variabilita grafického rozhrania

- **dobrá štruktúra:** Vhodná voľba štrúktury systému a rozdelenia kompetencií jednotlivým jeho zložkám patrí medzi kľúčové praktiky, ktorými dosahujeme možnosť jednoduchého pridávania a rozširovania funkcionality.

- **variabilita rozpoznávania jazyka:** Podstatnou časťou funkcionality systému je rozpoznávať nielen vizualizačný, ale tiež programovací jazyk. Dôležitou sa preto stáva možnosť zvoliť si ho z viacerých možností, alebo vybraný vlastný jazyk jednoducho doimplementovať. To isté platí pre varianty vizualizačného jazyka, ktoré je možné prispôsobiť.

Tieto vlastnosti spolu úzko súvisia a teda bolo dôležité aplikovať ich princípy súbežne. Preto sme sa rozhodli pre hojné využitie návrhových vzorov, ako napríklad Strategy či DesignFactory, ktoré nám poskytujú potrebnú jednotnosť z hľadiska prístupu k inštanciám, ale zároveň vysokú variabilitu a upraviteľnosť z hľadiska rozširovania systému. Taktiež treba poznamenať, že hlavný systém spĺňa prevažne úlohu koordinátora, zatiaľčo jednotlivé kompetencie sú prerozdelené medzi samostatné správcovské triedy.

5.2 Jadro systému

Po tom, ako sme si uviedli základné požiadavky, ktoré na náš systém kladieme, pozrieme sa na jeho jednotlivé súčasti a ich implementáciu. Ostaneme pochopiteľne v hladine myšlienok implementácie a nebudeme sa venovať konkrétnym použitým príkazom. Uvedieme taktiež v každej časti triedu, modul alebo skupinu tried, ktorú tú-ktorú funkcionalitu zabezpečuje. V špecifických prípadoch sa zameriame taktiež na vnútornú štruktúru a fungovanie popisovanej časti.

5.2.1 Rozpoznávanie príkazov a jazyky

Pod pojmom **"jazyk" (Language)**[10] v našom systéme rozumieme skupiny kľúčových slov s prípadnými doplňujúcimi informáciami vo vopred určenom formáte. Jazyk mapuje tieto kľúčové slová na "príkazy", ktoré môžeme následne spracovať. Rozoznávame dva základné typy použitých jazykov: vizualizačný jazyk a programovací jazyk. Pri bežnom fungovaní programu sa počíta s tým, že je naraz aktívny práve jeden vizualizačný a najviac jeden programovací jazyk, pričom len niektoré vizualizačné jazyky sú "sebestačné" na samostatné použitie. Ostatné slúžia ako nadstavba aktívneho programovacieho jazyka.

Celý proces rozpoznávania a spracovania vstupu má na starosti trieda **Prekladateľ (Translator)**, ktorá má svoj priradený vizualizačný a programovací jazyk. Sprostredkúva funkcionalitu spracovania vstupu pre hlavné jadro systému a oddeľuje ho od práce s jednotlivými jazykmi. Postup spracovania vstupného programu pozostáva z niekoľkých krokov. Prvým je spracovanie pôvodného vstupu, podľa typu programovacieho jazyka, do tzv. "hviezdičkového formátu".

"Hviezdičkový formát" programu je formát, v ktorom je hierarchia príkazov určená počtom znakov "*" na začiatku riadku a nie pomocnými znakmi ako sú množinové zátvorky[11] alebo slová `begin` a `end`[12]. Tento formát pôsobí ako pracovný formát pre náš program, pričom abstrahuje, vo formáte, od jednotlivých programovacích jazykov.

Ďalším krokom je rozpoznanie jednotlivých samostatných príkazov a vytvorenie zoznamu príkazov (pričom nemáme na mysli určovanie jednotlivých významov, ale oddeľovanie samostatných príkazov od seba). Ten je následne hierarchicky, podľa vlastností "hviezdičkového formátu", spracovaný v rámci tretieho kroku, ktorým je rozpoznávanie jednotlivých príkazov.

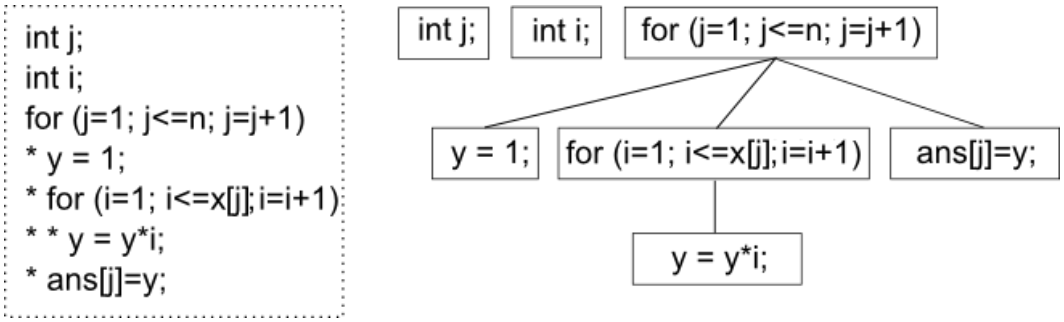
[10] text uvedený v zátvorke zodpovedá reálnemu označeniu v rámci zdrojového kódu v programe
[11] napr. pre jazyk C
[12] napr. pre jazyk Pascal

Proces prebieha podľa vizualizačného a aj podľa programovacieho jazyka, pričom prioritu má v prípade duálneho rozpoznania príkazu ten príkaz, ktorý bol určený vizualizačným jazykom. Samotná implementácia sa nachádza priamo v jednotlivých jazykoch a nie je funkčne nutné, aby dokázal jazyk rozpoznať všetky existujúce príkazy.

5.2.2 Štruktúra a typy príkazov

V rámci tejto práce budeme **"príkaz" (Command)** brať ako základný alebo zložený krok v rámci vykonávaného vizualizovaného programu. Na tvorbu príkazov využívame **Príkazového manežéra (CommandManager)**, vďaka čomu funkcionalitu týkajúcu sa práce s príkazmi oddeľujeme a združujeme mimo hlavného systému.

Zloženým príkazom budeme nazývať taký príkaz, ktorý obsahuje podmienený (alebo nepodmienený, vtedy sa jedná jednoducho o oddelenú časť programu) vnútorný sled príkazov, ktoré musia byť vykonané na úspešné a kompletné ukončenie vonkajšieho príkazu. Príkladom takýchto príkazov sú napríklad podmienky[13] a cykly[14]. Každý **jednoduchý príkaz** je len špeciálny prípad zložených príkazov, pričom nemá žiadne vnútorné príkazy a je nepodmienené. Nepodmienené príkazy sú špeciálnym prípadom podmienených príkazov s podmienkou, ktorá je vždy pravdivá. Toto zovšeobecnenie použijeme pri vysvetľovaní, v akom poradí vykonávame príkazy.



Obr. 5: **Ilustrácia hierarchie príkazov.** Jedná sa o časť programu na výpočet faktoriálu s n vstupnými dátami.

[13] if, if-else
[14] for, while, repeat-until

Príkaz	**vizualize(Rodič)**	**vizualize(Príkaz)**
true	true	true
true	false	false
false	true	false
false	false	false

Tabuľka 2: Zjednodušené znázornenie fungovania vizualizačnej funkcie.

Špeciálnym príznakom sa stáva informácia o prípadnom opakovaní príkazu až do splnenia podmienky, čo nastáva pri všetkých druhoch cyklov. Cyklus `for` aplikujeme po úprave rovnako ako cyklus `while`, pričom jeho počiatočná iniciácia sa vykoná pred jeho začiatkom a inkrementácia jeho premennej sa vykoná na konci každej sady jeho vnútorných príkazov.

Hierarchia príkazov teda vzniká zo zoznamu jednoduchých a zložených príkazov, pričom každý zo zložených príkazov má opäť svoj vnútorný zoznam jednoduchých a zložených príkazov. V prípade, že sa na hierarchický strom príkazov pozrieme ako na strom a na vykonávanie príkazov ako na jeho výpis, môže čitateľ nahliadnuť, že sa jedná o upravenú verziu preorder výpisu. V každom vrchole sa najprv pozrieme na podmienku vo vrchole, v ktorom sa nachádzame a pokiaľ je pravdivá, vykonáme príkaz a následne vykonáme všetky jeho vnútorné príkazy. Na ilustráciu pridávame taktiež Obrázok č.5, na ktorom je znázornená trojvrstvová hierarchia a stretávame sa s vnoreným cyklom.

Ďalšou dôležitou časťou je vizualizácia vykonávania jednotlivých príkazov. Aj táto sa totiž prenáša v rámci hierarchie, podľa pravidla: Ak má príkaz nastavenú vizualizáciu a jeho rodič sa vizualizuje, tak aj príkaz sa vizualizuje. Príkaz, ktorý má nastavenú nevizualizáciu sa nevizualizuje nikdy. Príkaz, ktorého rodič sa nevizualizuje, ale sám má nastavenú vizualizáciu sa taktiež nevizualizuje. Vizualizácia príkazu sa teda správa ako logické AND medzi vizualizačným príznakom príkazu a výsledkom tejto operácie pre jeho rodiča.

5.2.3 Práca so systémovými premennými

Premenné z vizualizovaných programov potrebujeme reprezentovať aj v našom systéme. Tento účel plnia **Systémové premenné (SimpleVariable a ArrayVariable)**. Systémová verzia poľa je realizovaná formou zoznamu prvkov, ktoré musia byť jednotného typu s typom poľa, ale inak sú to len ďalšie systémové premenné. Rozdiel pri nich je, že nevystupujú v systéme inak ako vo forme súčasti poľa. Vďaka tomuto prístupu je triviálne možné vytvoriť pole z ľubovoľnej jednoduchej premennej.

V rámci systému si neustále udržiavame zoznam všetkých aktívnych systémových premenných, pričom o tento zoznam sa stará **Menežér premenných (VariableManager)**, ktorý poskytuje metódy pre prácu s premennými, ich tvorbu, úpravu, ako aj získavanie informácií a zoznamov aktuálne aktivnych systémových premenných.

Taktiež je vhodné uviesť, že v tejto verzii systému nie sú implementované všetky existujúce (alebo všetky bežne používané) dátové typy. Jedná sa napríklad o desatinné čísla, Nie je však problémom pre ďalších pokračovateľov doimplementovať potrebné systémové premenné, ako si uvedieme v kapitole 7.

5.2.4 Evaluácia výrazov

Pri práci s výrazmi a operáciami s premennými je potrebné si uvedomiť, že tieto operácie môžu mať zložitú štruktúru, a tiež, že fakt, že pracujeme s logickou podmienkou neznamená, že sa v nej neobjavia matematické výrazy, alebo nebudú porovnávať textové premenné. Preto je dôležitou súčasťou nášho systému **Interpreter výrazov (ExpressionInterpreter)**, ktorý poskytuje rozhranie pre prácu s rozpoznávaním hodnoty výrazov, alebo vytváraním ich evaluačných modelov.

Taktiež je na mieste uviesť, že tieto výrazy sa nevyskytujú len pri priraďovaní hodnôt do premenných, ako by sa mohlo na prvý pohľad zdať. Vyskytujú sa celkovo v

týchto kontextoch:

- priraďovanie vypočítaných hodnôt premenným
- indexovanie poľa, alebo znaku v textovom reťazci
- podmienky v cykloch alebo podmienených príkazoch

Zatiaľčo v prípade matematických a logických výrazov sa jedná o stromovú štruktúru, pre prácu so skladaním textových premenných nám plne postačí lineárne spracovanie. V Tabuľke č.3 sa podrobnejšie pozrieme na vzťahy jednotlivých typov výrazov medzi sebou, pretože práve tieto vzťahy sú veľmi dôležitým faktorom pri vytváraní jednotlivých interpreterov. Poznamenajme ešte, že samostatne stojaca premenná niektorého z typov - číslo, textový reťazec, pravdivostná hodnota - je tiež len špeciálnym prípadom výrazu tohto typu.

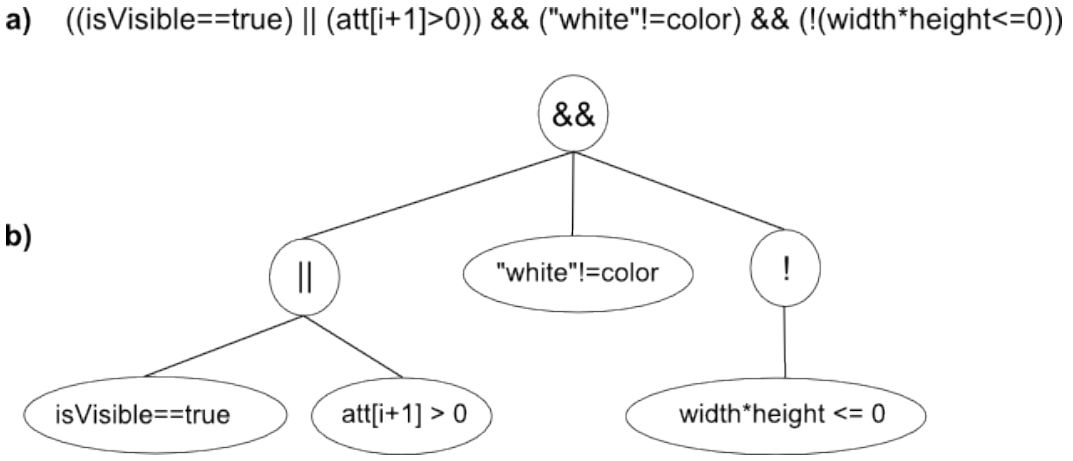
Môže obsahovať:	**Logický výraz**	**Matemat. výraz**	**Sklad. reťazcov**
Logický výraz	áno - v hlavnej časti	áno - v hlavnej časti - aj v indexovaní	áno - v hlavnej časti
Matemat. výraz	nie	áno - v hlavnej časti - aj v indexovaní	nie
Sklad. reťazcov	nie	áno - v indexovaní	áno - v hlavnej časti

Tabuľka 3: Vzájomné vzťahy medzi typmi výrazov.

Spracovanie textových reťazcov (StringProcess) prebieha, ako už bolo spomenuté, lineárne. Jednotlivé skladané časti sú najprv rozdelené podľa znaku "." značiaceho operáciu zreťazenia. Pri spracovaní sú následne testované na to, či ide o reálny text, alebo o premennú, a poľa toho sa k výsledku pripája tá správna hodnota.

Zložitejším procesom prechádza spracovanie matematických a logických výrazov cez **Evaluačné stromy (EvaluationTree)**. V oboch prípadoch musia byť výrazy

kompletne uzátvorkované s dvomi výnimkami: úplne vonkajšia zátvorka okolo celého výrazu; viac členov, pri ktorých sa vykonáva tá istá operácia (napríklad sčítanie 4 čísel môže byť uvedené formou $a + b + c + d$ namiesto napr. $(a + b) + (c + d)$). Spoločný postup spracovania výrazov evaluačným stromom je postupné prechádzanie od najvyššieho "levelu" zátvoriek až po najhlbšie vnorené zátvorky, vždy hľadajúc začiatočnú a koncovú zátvorku prvého výrazu, znak vykonávanej operácie a potom začiatočnú a koncovú zátvorku druhého výrazu (pri unárnej operácii sa vynechá prvá časť).



Obr. 6: **Ilustrácia evaluačného stromu logického výrazu.** a) pôvodný výraz; b) logický evaluačný strom zodpovedajúci tomuto výrazu

Evaluačný strom je pre nás taký "nie nutne binárny" strom, ktorý má v každom vnútornom vrchole znak operácie, ktorá sa vykonáva na jeho deťoch a v každom liste má niektorý zo samostatných výrazov, ktoré už vieme spracovať nerekurzívne. Pri vyhodnocovaní celého stromu sa začína v jeho koreni a postupuje sa podľa postorder schémy, kedy sa najprv vypočítajú hodnoty jeho detí a následne sa aplikuje operácia v samotnom vrchole. V prípade listu sa vyhodnotí rovno výraz, ktorý sa vo vrchole nachádza. Príklad evaluačného stromu je zobrazený na ilustračnom obrázku č.6.

5.2.5 Vstup, výstup a rozpoznané chyby

Často sa okrem práce samotného programu stretávame aj s interakciou s externým vstupom (z konzoly), alebo s výpisom výstupu. Napriek tomu, že pre chápanie samotného behu programu tieto doplňujúce funkcie často nepotrebujeme, implementovali sme jednoduchý nástroj poskytujúci prácu s príkazmi na rozpoznávanie vstupu a tvorbu výstupu. Mohli by sme tento nástroj brať ako substitúciu za klasickú konzolu, s ktorou by sme za normálnych okolností pracovali. Z hľadiska používania platí, že vstup je nutný až pri spustení spracovaného programu, nie pri spracovaní samotnom.

Aplikácia obsahuje tiež správu niektorých rozpoznávaných chýb v programe, ktorá namiesto toho, aby sa nám program ani nespustil, prípadne aby zlyhal bez zjavnej príčiny, vypíše vybranú chybu na špeciálne, na to určené, miesto. Vo vykonávaní vizualizácie a simulácie programu z bezpečnostných príčin po odhalení chyby nepokračujeme, avšak aplikácia funguje ďalej bez problémov a používateľ môže chybu v programe opraviť a ďalej s ním pracovať. Je namieste spomenúť, že sa nejedná len o chyby, ktoré by bránili kompilácii programu, prípadne spôsobili jeho pád počas behu, ale tiež o chyby, ktoré ovplyvňujú samotnú funkcionalitu programu (napr. chýbajúci očakávaný vstup).

5.3 Implementácia vizualizácie

Pri implementácii vizualizácie bola kľúčovou schopnosť neustáleho sledovania vzájomných prepojení jednotlivých častí vizualizácie. Základ vizualizácie spočíva vo vizuálnom paneli, ktorý poskytuje priestor a funkcionalitu potrebnú pre sledovanie umiestnenia jednotlivých vizuálných prvkov. Tento panel tiež ostáva primárnym dejiskom celej vizualizácie programu. V rámci neho pracujeme s vizuálnymi komponentami, ktoré sú priradené niektorým zvoleným systémovým premenným - tým, ktoré chceme vizualizovať v priebehu programu.

5.3.1 Vizualizačný jazyk

Pojem jazyka sme zaviedli už v predchádzajúcej podkapitole, avšak vtedy sme sa zamerali hlavne na štruktúru rozpoznaných príkazov v jazyku. Teraz sa však pozrieme na špecifický prvok vizualizačných jazykov, ktorým sú vizualizačné parametre. Slúžia na pridávanie doplňujúcich voliteľných vlastností daného vizuálneho príkazu, ale ich použitie je dobrovoľné, a príkaz funguje aj bez ich zadania, v prednastavenom variante.

Tieto parametre uvádzame v programe vždy za príkazom vizualizačného jazyka a pred príkazom z programovacieho jazyka. Ich formát by sme mohli nazvať "mriežkový", kvôli znaku #. Presný formát je

```
meno-parametra#hodnota-parametra
```

pričom je žiadúce, aby meno ani hodnota neobsahovali medzery. V prípade viacerých parametrov sú parametre medzi sebou oddeľované medzerou a nezáleží na ich poradí. Parametre môžu mať taktiež svoje alternatívne jazykové formy, resp. formy prislúchajúce len niektorým vizualizačným jazykom, podobne ako pri samotných príkazoch.

Rozpoznávanie parametrov nie je vykonávané automaticky, ale vykonáva sa len pri tých príkazoch, kde chceme s parametrami pracovať. Dôvod je triviálny - pri týchto príkazoch vieme zaručiť, že nenastane neželaná interakcia s neočakávaným oddeľovacím znakom.

5.3.2 Vizuálny panel

Vizuálny panel (VisualPanel) je adaptáciou štandardného Swing komponentu v Jave. Jeho primárnym účelom je zobrazovanie vizualizovaných premenných v rámci času a behu programu. Z toho dôvodu má funkcionalitu, ktorá poskytuje možnosť pridávať a vykresľovať jednotlivé vizualizované komponenty.

Kľúčovou časťou pri vykresľovaní premenných je ich umiestnenie v rámci vizuálneho panelu. Možných prístupov je hneď niekoľko:

- **Automatické zobrazovanie:** Prvky sú umiestňované automaticky, podľa možnosti tak, aby sa neprekrývali.
- **Voľba používateľa:** Prvky sú umiestňované podľa výslovných pokynov používateľa systému a v prípade ich nedodania sú umiestňované na základnú pozíciu bez ohľadu na prekryv.
- **Zmiešaný prístup:** Prístup, pri ktorom sa prvky umiestňujú automaticky na voľné pozície, pokiaľ nie je používateľom povedané inak. Vtedy sú umiestnené podľa pokynov používateľa. Automatické umiestnenie teda predstavuje náhradu za umiestňovanie na základnú pozíciu a zabezpečuje odstránenie prekryvov prvkov.

My sme sa rozhodli zvoliť zmiešaný prístup, ktorý je z hľadiska používateľa najpohodlnejší a zároveň nestráca nič z variability poskytovanej ostatnými prístupmi. V rámci prehľadnosti pre systém aj pre používateľa sme zvolili možnosť umiestňovania do mrežových bodov, ktoré je možné zobraziť vo vizuálnom paneli.

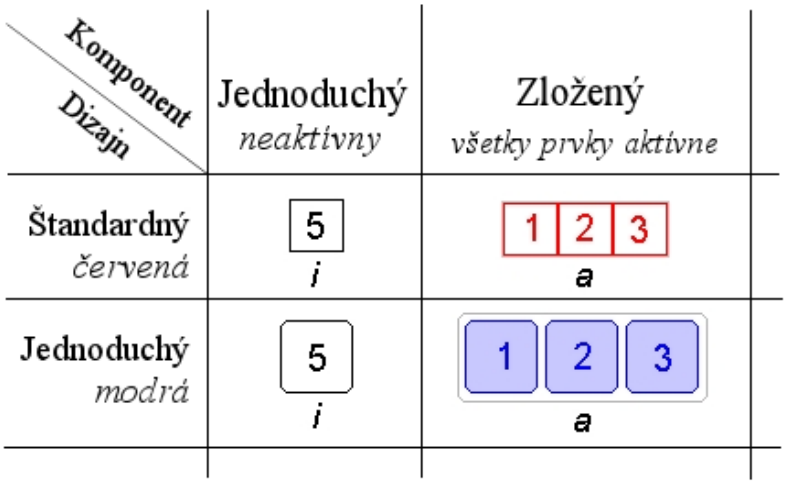
Automatické umiestňovanie volí následne prvý mrežový bod, pri "čítaní po riadkoch" v rámci mriežky, do ktorého je možné umiestniť prvok bez prekryvu s ostatnými prvkami. Pokiaľ také umiestnenie nie je možné, pristúpi k umiestneniu do základnej pozície vľavo hore. Avšak vzhľadom na bežnú potrebu vizualizácie je tento prípad veľmi nepravdepodobný a výrazne ovplyvniteľný používateľom.

5.3.3 Komponenty a dizajny

V predchádzajúcej časti sme spomenuli vizuálne komponenty, ktoré sa zobrazujú vo vizuálnom paneli. Pod pojmom **Vizuálny komponent (VisualComponent)** budeme chápať inštanciu triedy starajúcej sa o zobrazovanie jednoznačne priradenej sys-

témovej premennej. Jej hlavným účelom je starosť o zobrazovanie a rozloženie v rámci priestoru, o zobrazované a skryté súčasti (napr. hodnota, názov premennej). V prípade zobrazovania viacerých premenných, resp. premennej pozostávajúcej z viacerých hodnôt (napr. polia) taktiež umiestnenie premenných/hodnôt voči sebe navzájom.

Každému komponentu je priradený **Dizajn (Design)**, ktorý sa stará o konkrétne zobrazenie jednotlivých súčastí komponentu. Ich veľkosť, vzhľad, farebnú schému, a pod. určuje práve priradený dizajn, pričom niektoré z týchto vlastností sú variabilne nastaviteľné s pomocou parametrov. Každý dizajn má techniky na vykreslenie jednotlivých súčastí, z ktorých sa môže komponent skladať, a teda je možné ho aplikovať na ľubovoľný komponent. Je na mieste poznamenať, že prednastavený dizajn je možné, v rámci jednoduchosti používania, zmeniť aj globálne, príkazom na začiatku vizualizovaného programu.



Obr. 7: **Ilustrácia aplikácie dizajnov na komponenty.** Obrázok ilustruje aplikáciu dvoch rôznych dizajnov na dva rôzne vizualizačné komponenty, v dvoch rôznych stavoch (aktívny, neaktívny) a s dvomi farebnými schémami.

Dizajny poskytujú, ako bolo spomenuté, taktiež určitú variabilitu v interakcii s používateľom, a to nielen z hľadiska ich voľby, ale priamo z hľadiska prispôsobenia samotného dizajnu. Na tento účel slúžia **farebné schémy (DesignColorSet)**, ktoré po-

skytujú možnosť jednoduchšieho prispôsobenia jednotlivých vizualizácií podľa potreby učiteľa. Niektoré dizajny môžu zo špecifických príčin využívať len časť z poskytnutej farebnej schémy a dopĺňať ju striktne zvolenými zvyšnými farbami.

5.3.4 Používateľské rozhranie

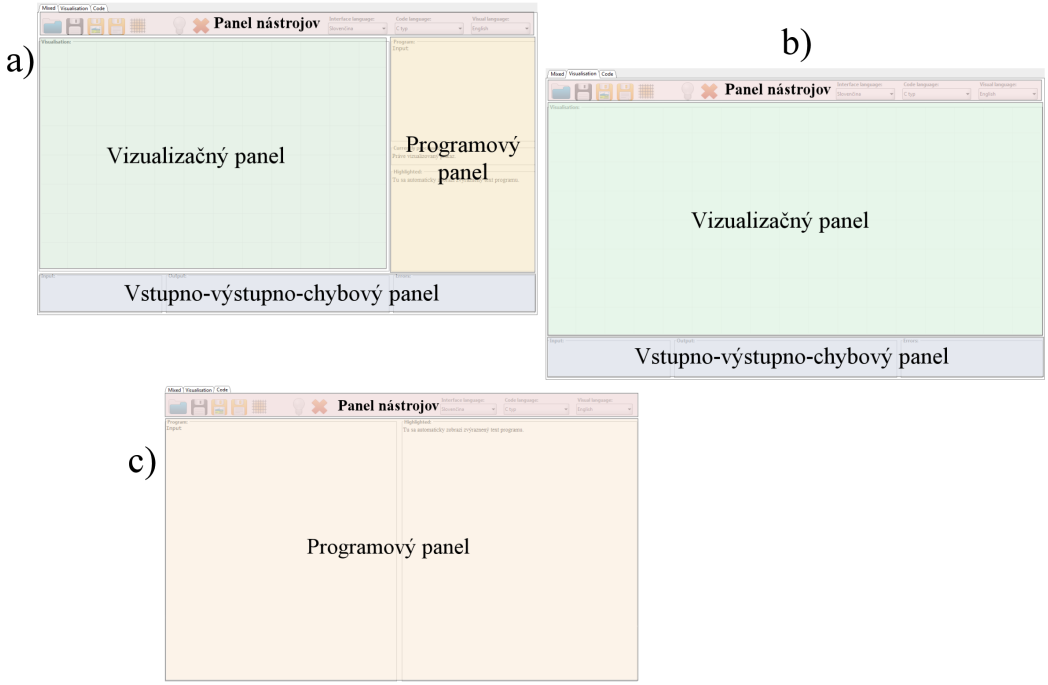
Pri návrhu používateľského rozhrania sme kládli dôraz na prehľadnosť, dostatok priestoru pre samotnú vizualizáciu a tiež možnosť pohodlnej práce so vstupným programom. Vďaka týmto kritériam sme dospeli k názoru, že jednoduché používateľské rozhranie by neposkytovalo používateľom dostatočný komfort práce, a preto sme sa rozhodli vytvoriť viac variantov rozloženia prvkov, z ktorých každý poskytuje iný dôraz pri tvorbe animácie.

Vybrali sme tieto varianty:

- **Zmiešaný variant:** Variant poskytujúci "z každého rožku trošku". Zobrazuje jednak vizuálny panel a vstupno-výstupné komponenty, ako aj programovú časť spoločne so zvýrazňovaním príkazov.
- **Vizualizačný variant:** Tento variant necháva plnú šírku vizuálnemu panelu, spoločne so vstupno-výstupnými komponentami.
- **Programovací variant:** Komplementom k vizualizačnému variantu sa stáva programovací variant, pri ktorom je celá veľkosť prenechaná panelom s pôvodným a zvýrazneným programom.

Okrem toho sme pridali ešte samostatný variant pre nastavovanie používateľského rozhrania a niekoľkých ďalších globálnych parametrov.

Z hľadiska využívania variantov je dôležité, že v prípade prepínania variantov počas behu programu, bude vizualizácia pokračovať v tých rozmeroch, v akých bola spracovaná. Inak by hrozila nekompatibilita medzi jednotlivými zobrazeniami, hlavne pri automatickom umiestňovaní prvkov, ale tiež pri ich vykresľovaní.



Obr. 8: **Rozloženie jednotlivých variantov grafického rozhrania.** a) zmiešaný variant; b) Vizualizačný variant; c) Programovací variant.

Súčasťou používateľského rozhrania je taktiež panel nástrojov, nachádzajúci sa v každom variante a poskytujúci všetku funkcionalitu potrebnú na prácu s aplikáciou. Okrem tlačidiel pre export a import súborov sa tu nachádza tiež tlačidlo pre spracovanie programu a jeho počiatočné či opätovné spustenie (po prerušení programu). Na jeho pravej strane sa nachádzajú nastavenia pre rozpoznávané programovacie a vizualizačné jazyky.

5.4 Doplňujúce funkcie

Z ďalších vybraných funkcionalít aplikácie ešte uvedieme niekoľko špecifických, ktoré sú význačné z hľadiska cieľovej skupiny a využitia. Jedná sa nielen o možnosť importu a exportu programov, ako aj možnosť exportu obrázkov, ale tiež o špeciálny príkaz, ktorý preruší vizualizáciu a sledujúcemu používateľovi položí zadanú otázku a vyčká na jeho odpoveď.

5.4.1 Import / Export

Dôležitou funkciou aplikácie je možnosť poskytovať vytvorené vizualizácie študentom, kolegom či iným ľuďom. Prvou možnosťou je poskytnúť im predpripravený vizualizačne obohatený program, ktorý si môžu sami spustiť v našej aplikácii. Druhou možnosťou je využitie obrázkov vytvorených pomocou našej aplikácie, buď priamo z obsahu vizuálneho panelu, alebo obrázok celej aplikácie.

Import / Export programov

Export a import programov funguje v štandardnom textovom formáte, pričom súbor môže, ale nemusí, obsahovať aj dáta pre vstup, prípadne špeciálne nastavenia pre aplikáciu. Jednotlivé časti programu sú oddelené tromi znakmi #. Výsledný exportovaný súbor má potom štruktúru

```
vizuálne upravený program
###
vstupné dáta
###
dodatočné nastavenia
```

Export obrázkov

Rozoznávame dve možnosti exportovania obrázkov s pomocou našej aplikácie. Jedna funguje na princípe zobrazenia celého aktuálneho variantu aplikácie, zatiaľčo druhá do obrázku vkladá len obsah vizuálneho panelu.

Primárnym cieľom exportovania obsahu vizuálneho panelu je vytváranie ilustrácií k programom či algoritmom. Zobrazuje totiž stav vybraných premenných vo zvolenom momente behu programu. Funkcionalitu získavania týchto obrázkov vieme zabezpečiť dvoma spôsobmi:

- **Použitie tlačidla na panele nástrojov:** Toto použitie je možné v ľubovoľnom momente behu programu. Vizualizácia krokov programu je pozastavená počas procesu nahrávania obrázku, takže používateľ nepremešká žiaden z krokov programu.

- **Použitie špeciálneho príkazu v rámci programu:** Takáto možnosť je ideálna v prípade, kedy už pri príprave programu vie používateľ o tom, že bude chcieť získavať z programu obrázky - a tiež, v ktorých momentoch. Tak má istotu, že žiaden z nich nezabudne vytvoriť, alebo že nepremešká správny moment na jeho získanie.

Účelom exportovania kompletného obrazového vzhľadu aplikácie je prevažne vytváranie pomocných obrázkov do návodov, prípadne inštrukcií, ktoré môžu poskytovať učitelia - napríklad svojim študentom. Na získavanie takéhoto obrázku je taktiež tlačidlo na paneli nástrojov.

5.4.2 Kladenie otázok

Špecifickou a čisto pedagogicky opodstatnenou funkcionalitou je možnosť vkladania otázok do programu. Otázky môžu mať absolútne zadanú odpoveď, alebo môže ísť o otázku smerovanú na hodnotu niektorej premennej v okamihu otázky. Takáto možnosť kladenia otázok poskytuje učiteľovi možnosť podnietiť prijímateľa vizualizácie k samostatnému premýšľaniu o nezobrazených premenných.

6 Príklady použitia systému

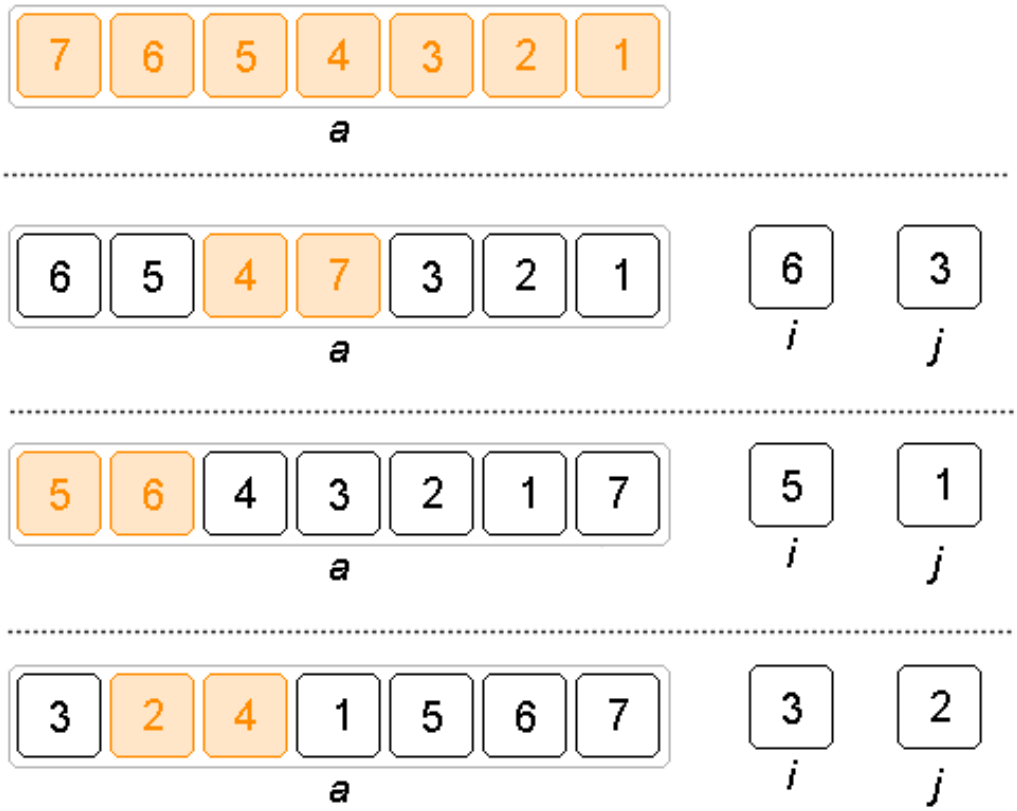
V tejto kapitole sa zameriame na praktické príklady použitia nášho systému na triviálnych aj netriviálnych programoch. Listingy k jednotlivým programom sú uvedené v Prílohe C a taktiež v importnom formáte na priloženom CD, kde slúžia primárne na preukázanie a preskúšanie možností aplikácie. Hneď v úvode tejto kapitoly uvedieme opätovne do pozornosti, že v našej aplikácií je pri For cykle inkrementačný príkaz vykonaný aj na konci posledného behu cyklu, a teda sa hodnota premennej po skončení cyklu javí väčšia než je zvolená horná hranica. Fungovanie cyklu je však korektné.

6.1 BubbleSort

Veľké množstvo vizualizačných programov sa zaoberá problematikou triedenia. Preto sme sa rozhodli uviesť aj jeden z triediacich algoritmov ako príklad toho, že aj náš systém je možné aplikovať na túto problematiku.

6.2 Jeden z našich vzorových programov

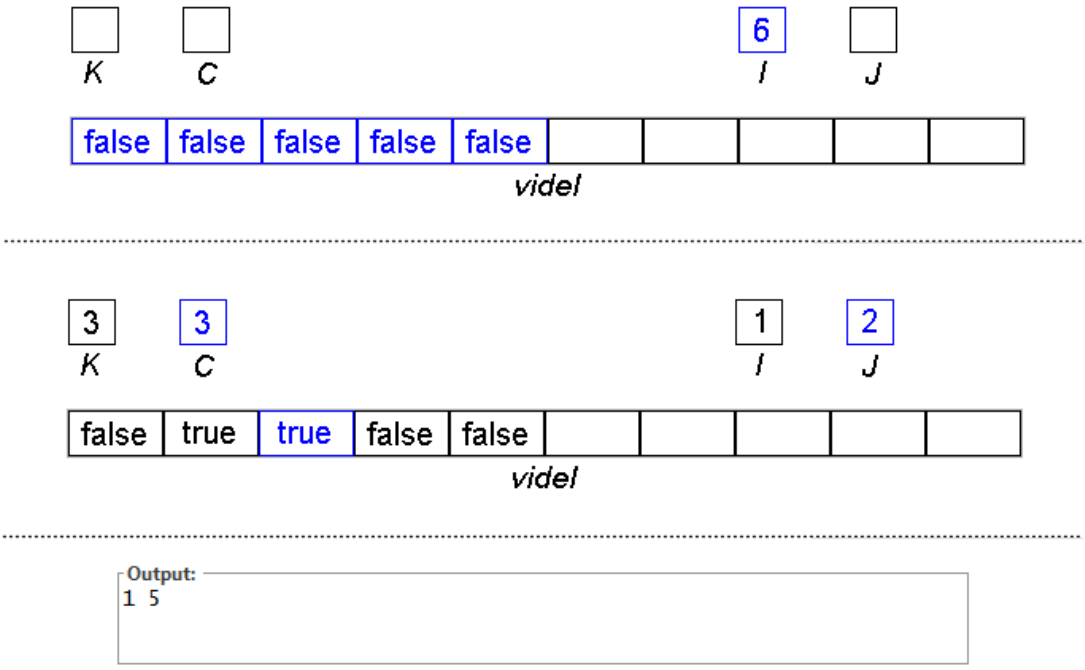
Rozhodli sme sa taktiež vizualizovať jeden z programov, ktoré sme spomínali pri základných programových schémach. Presnejšie, pôjde o program s označením KSP 28-2-2], ktorý môžeme nájsť v Prílohe A na konci tejto diplomovej práce. V tomto programe pracujeme aktívnejšie so vstupom a výstupom aj priebežne, počas behu programu.



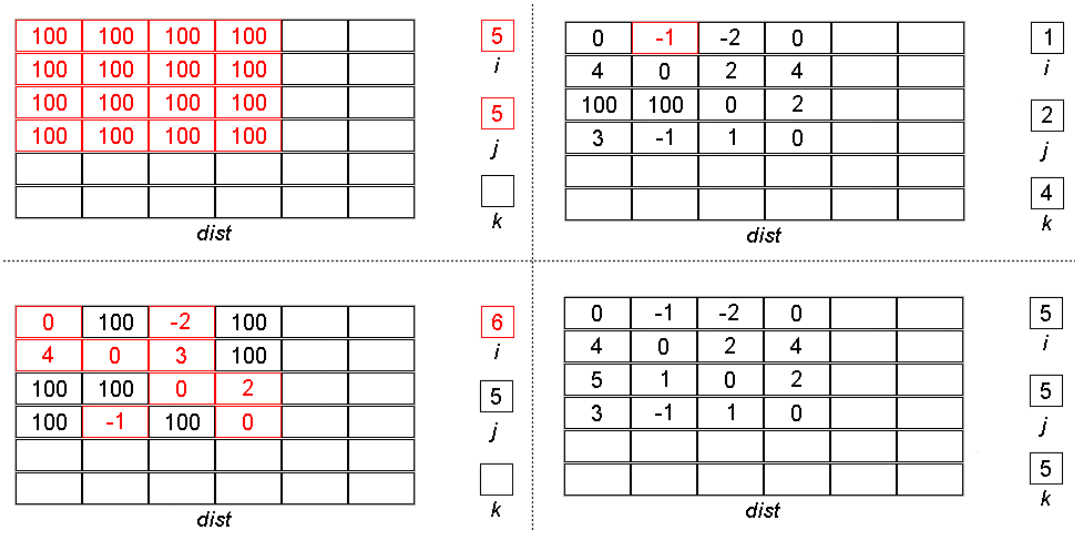
Obr. 9: **Bubblesort - ukážky.** Ukážky z rôznych krokov behu programu.

6.3 Floyd-Warshallov algoritmus

Napriek tomu, že sa jedná o grafový algoritmus a náš systém v tejto verzii neimplementuje grafové zobrazenia, zvolili sme ako tretí príklad práve Floyd-Warshallov algoritmus. Poskytuje totiž komplexnú prácu s cyklami, dvojrozmernými poľami aj vstupom.



Obr. 10: **Ukážky z behu programu KSP 28-2-2** Po skončení inicializácie programu, uprostred behu programu a výstup po skončení programu. Horná hranica veľkosti poľa bola zmenšená kvôli vizualizačným dôvodom.



Obr. 11: **Ukážky z behu Floyd-Warshallovho algoritmu.** Namiesto nekonečna bola priradená hodnota 100, keďže je dostatočne odlišná od vstupných vzdialeností.

7 Rozširovanie systému

Téma tejto diplomovej práce poskytuje množstvo priestoru na neustále zdokonaľovanie a rozširovanie systému. Preto sme zvolili možnosť implementovať v našom prototype čo najširšiu paletu typov funkcionality, čím ilustrujeme jeho široké možnosti. Vždy je však možné systém ďalej a ďalej obohacovať, a preto jedným z hlavných kritérií, ktoré sme kládli na náš systém, je jeho pestrá rozšíriteľnosť. V tejto kapitole sa najprv pozrieme na súhrn návrhov pokračovania tejto práce z hľadiska nových technológií. Následne sa budeme venovať postupom pri rozširovaní skôr rutinnejšieho charakteru, ako je pridávanie nových dizajnov, komponentov alebo nových vizualizačných jazykov. K tomuto typu rozširovania systému je systém predpripravený a konkrétne postupy poskytujú osnovu procesu rozširovania.

Z hľadiska implementácie nových technológií padá do úvahy implementovanie vynechaných vizualizačných techník, akými sú strom, halda či smerníky spolu s indexovacími poľami. V súvislosti s týmito technikami sa naskytá možnosť implementovania rozličných dizajnových variantov pre stromy, ako napríklad červeno-čierne stromy alebo stromy zadané systémom "prvý syn, ďalší súrodenec"[15]. Taktiež je možnosť vytvárania/pridávania nových vizuálnych a/alebo programovacích jazkov. Z hľadiska doplnkových funkcií je možnosť pridávania exportných možností, ako napríklad exportovanie špecializovaných appletov so zabudovaným programom, vytváranie videí kompletnej vizualizácie či vytvorenie špecializovaného formátu pre zadávanie programu rovno v rozpoznaných príkazoch.

Z hľadiska širokospektrálneho rozširovania systému, kedy sa jedná o pridávanie väčšieho množstva menších doplnkov sa naskytá možnosť pracovať na rozšíreniach ako: technika zobrazovania reťazca ako poľa znakov (pričom jeho dátová štruktúra už implementovaná v systéme je), pridávanie nových dizajnov, typov systémových premen-

[15]preklad termínu: first-child next-sibling

ných (ako napríklad reálne čísla), implementácia rozpoznávania rôznych syntaktických rozdielov oproti pôvodným jazykom, syntaktických skratiek (ako napríklad `i++`, resp. `inc(i)` pre číselné premenné), pridávanie nových rozpoznávaných chýb, alebo pridávanie ich vizualizácie a tak ďalej.

7.1 Pridávanie príkazov

Všetky príkazy sú odvodené od triedy Command, ktorá poskytuje základnú osnovu práce s príkazmi, ako aj funkcionalitu pre prácu s vnorenými príkazmi. Príkazy sú spúšťané systémom zásobníka, kedy sa zo zásobníka vezme najvrchnejší príkaz, ktorý sa aplikuje. V prípade, že obsahuje vnútorné príkazy, môže (ale nemusí) tento príkaz svoje vnútorné príkazy pridať do zásobníka na ďalšie spracovanie. Po aplikovaní príkazu pokračuje vykonávanie ďalšieho príkazu.

Každý príkaz obsahuje nástroje na spracovanie výrazov, príkazov a prácu s premennými, ako aj potenciálny zoznam vnútorných príkazov. Z vizualizačného hľadiska obsahuje tiež príznak na určenie vizualizovanosti, rozpoznané vizualizačné parametre, ako aj plné znenie príkazu, podľa ktorého bol rozpoznaný. Okrem týchto štandardných častí môže každý príkaz obsahovať ľubovoľné dodatočné dáta, ktoré pre svoj beh potrebuje, a ktoré budú rozpoznané podľa použitého programovacieho jazyka.

Každý príkaz musí, pre korektnú funkcionalitu, obsahovať implementáciu týchto funkcií:

- `getType` - funkcia, ktorá vracia interné označenie typu príkazu
- `applyCommand` - funkcia, ktorá zabezpečuje činnosť, ktorú má príkaz vykonať

Po pridaní nového príkazu je nutné, pokiaľ chceme, aby ho mohli používatelia reálne využívať, implementovať ho ešte do:

- Príkazového menežéra (CommandManager) - *do funkcie* `getCommand`*, vďaka čomu bude môcť príkazový menežér vytvárať tento typ príkazov.*

- Všeobecného vizualizačného jazyka (VizLanguage) (príp. aj do všeobecného programovacieho jazyka (CodeLanguage)) - *vďaka tomu bude vedieť podľa preloženého rozpoznaného slova určiť, aký príkaz bude treba vytvoriť; v prípade čisto vizualizačných príkazov nie je nutné pridávať tento príkaz aj do všeobecného programovacieho jazyka.*

- Niektorého implementovaného jazyka - *vďaka čomu bude možné príkaz naozaj použiť, pričom je nutné pridať jeho kľúčové slová do jeho slovníka, ako aj implementovať rozpoznávanie jeho súčastí a dát v rozpoznávacej časti jazyka.*

7.2 Pridávanie jazykov

Jazyky sú v našej práci odvodené od spoločného základu - triedy Language - avšak za hlavné dva základy pokladáme triedy, ktoré zastrešuju vizualizačné jazyky a programovacie jazyky - triedy VizLanguage a CodeLanguage. Pri pridávaní nového jazyka treba v oboch prípadoch vyplniť jeho slovník kľúčovými slovami identifikujúcimi príkazy, ako aj uviesť, aký znak je znakom konca príkazu. Vo väčšine prípadov je takýmto znakom bodkočiarka. Taktiež je nutná implementácia procedúry `recognizeData`, ktorá zabezpečí načítanie jednotlivých dát (pre už rozpoznaný príkaz) s ohľadom na syntax jazyka.

Už hotový jazyk, pokiaľ má byť umožnené používateľom ho využiť, je treba pridať do:

- vytváracej funkcie `getLangByName` v triede VizLanguage, resp. CodeLanguage.

- grafického používateľského rozhrania v hlavnom applete.

7.2.1 Vizualizačný jazyk

V prípade pridávania vizualizačného jazyka je možné využívať tiež možnosť spracovania prípadných parametrov pre daný príkaz, použitím funkcie `processParam`. Tieto

parametre sú prekladané podľa lokálneho prekladového slovníka parametrov, ktorý je taktiež možné dodefinovať aj špecializovane pre jednotlivé vizualizačné jazyky.

7.2.2 Programovací jazyk

Pri pridávaní programovacieho jazyka patrí medzi špecifiká funkcia `changeFormat` pretvárajúca pôvodný vstup na nami využívaný "hviezdičkový formát". V prípade jazykov, ktoré majú podobnú štruktúru ako napríklad Pascal či C, je možné využiť predpripravenú funkciu `formatProcess`, ktorá pracuje na princípe zadania reťazca určujúceho začiatok vnorenej časti programu a reťazca určujúceho koniec vnorenej časti programu. Taktiež je nutné zadať spôsob, akým je v jazyku určované priraďovanie do premenných.

7.3 Pridávanie systémových premenných

Pri práci s premennými rozlišujeme tri hlavné implementované triedy, a to

- `SimpleVariable`, zodpovedajúcu jednoduchým premenným rôznych typov, ktoré následne dedia jej štruktúru; pôsobí tiež ako zjednocovací prvok pri práci so zložitejšími premennými
- `ArrayVariable`, zodpovedajúcu jednorozmernému poľu, pričom konkrétna aplikácia a typ poľa závisí na type premenných, ktorými je naplnené
- `TableVariable`, zodpovedajúcu dvojrozmernému poľu, pričom podobne ako pre jednorozmerné pole, aj tu závisí na type premenných, ktorými je naplnené

V prípade pridávania nového dátového typu je nutné pracovať s dedením od SimpleVariable, ktoré zároveň zabezpečuje korektné spracovanie premennej poľami. Taktiež treba v takom prípade pridať nástroj, ktorý bude evaluovať hodnoty výrazov týkajúcich sa tejto premennej (napríklad v prípade reálnych čísel je nutné rozšíriť matematický aparát triedy MathTree).

Každá premenná musí obsahovať označenie svojho dátového typu, svoje interné dáta, ako aj implementáciu získavania a nastavovania hodnoty premennej. V rámci jednotnej práce s premennými funguje získavanie hodnoty v určitom dátovom type (String, číslo, znak,...) cez funkciu `getValue`, ktorá dostáva ako parameter ľubovoľnú hodnotu rovnakého dátového typu ako má byť jej výstup.

Po úspešnom vytvorení novej systémovej premennej treba pred jej zavedením do aplikácie pridať túto premennú do kompetencií a funkcionalít meneżéra premenných (VariableManager):

- jej vytváranie a typ pridať do funkcie `getNewVariableOfType`
- do funkcie `getDefaultValueForType` pridať jej prednastavenú (v okamžiku vzniku) hodnotu
- a v prípade, že jej formát má svoje špecifiká, je nutné upraviť tiež jej rozpoznávanie pri vyhľadávaní premennej podľa mena, resp. pri úprave jej hodnoty.

Taktiež treba odsledovať rozpoznávanie prípadných špecifík jednotlivých jazykov pri tvorbe tejto premennej, alebo práce s ňou.

7.4 Pridávanie rozpoznávaných chýb

O prácu s rozpoznávanými chybami sa stará správca chýb ErrorHandler), ktorý rozpoznané chyby vypíše na obrazovku, alebo prípadne pri ich spustení vykoná iné potrebné úkony. Zavolať ošetrenie odhalenej chyby je možné kdekoľvek v programe, pričom je nutné vykonať tak cez správcu chýb. Pri pridávaní nových rozpoznaných chýb je treba ich rozpoznanie pridať na korektné miesto v programe, kde sa môže chyba odohrávať, a následne pridať požadovanú reakciu do správcu chýb.

7.5 Pridávanie vizuálnych prvkov

Každý vizuálny komponent je založený na triede VisualComponent, ktorá mu poskytuje množstvo spoločnej funkcionality. Jednotlivé konkrétne vizuálne komponenty, pokiaľ nemajú svoje špecifické podmienky, musia implementovať len tri konkrétne funkcie, ktorými definujú rozloženie základných súčastí vizualizácie premenných, s využitím priradeného dizajnu:

- `getBorder`, ktorá vytvára ohraničenia okolo celého komponentu alebo jeho hodnoty a vizuálne zvýrazňuje jeho časti
- `getValueShape`, ktorá určuje umiestnenie hodnoty priradenej premennej
- `getNameShape`, ktorá určuje umiestnenie názvu priradenej premennej

Pri vytváraní nových dizajnov je potrebné sa držať dedenia od triedy `Design`, ktorá zastrešuje spoločnú funkcionalitu, ako aj spoločný prístup k jednotlivým designom. Samotné vytvorené reálne dizajny už potom implementujú len špecifické funkcie pre aplikovanie dizajnového zobrazenia na mieste určenom vizuálnym komponentom, ku ktorému je dizajn priradený. Taktiež sa starajú o reálne vykresľovanie spomínaných súčastí do vizuálneho panelu.

Pri potrebe upraviť len farebné zloženie niektorého dizajnu netreba siahať po vytváraní jeho klonu, ktorý by sa následne upravil. Stačí vytvoriť novú farebnú schému, podľa triedy `DesignColorSet`, ktorá určuje využitú farebnú škálu pre ľubovoľný dizajn (ak nie je priamo v dizajne povedané inak). V týchto schémach rozoznávame stav premennej ako "neaktívny" alebo "aktívny", pričom "aktívny" značí, že premenná bola od posledného vykreslenia vizuálneho panelu upravená. Tieto dva stavy je možné farebne odlíšiť voľbou hodnôt premenných v schéme.

7.6 Rozširovanie používateľského rozhrania

Pri rozširovaní používateľského rozhrania je okrem pridávania nových prvkov alebo prvkov k novým funkcionalitám možné vytvárať alternatívne verzie pre jazyk rozhrania, alebo pre zobrazované ikonky tlačidiel. Jazykové verzie textov v rozhraní sú zabezpečené pomocou triedy `GUIText`, pričom prípadnú novú možnosť je nutné pridať taktiež do príslušného komponentu určeného na výber jazyka rozhrania.

Podobne, konkrétne obrázky ikoniek sú zabezpečené s pomocou triedy `GUIIcons` a prípadnú novú možnosť je taktiež nutné pridať medzi zobrazené voľby pre používateľa. Pri práci s obrázkami ikoniek je na mieste ešte poznamenať, že ich načítavanie funguje striktne cez triedu SourceBound, ktorá poskytuje možnosť korektného načítania z adresára iconSets v rámci súhrnného .jar archívu pri tvorbe appletu.

Záver

V tejto práci sme najprv zhrnuli súčasný stav a používanosť vizualizácií v pedagogickom procese, aj s ukážkami jednotlivých existujúcich projektov v tejto oblasti. Zjednotili sme materiály týkajúce sa pedagogických požiadaviek na vizualizácie a zhodnotili ich implementovateľnosť a aktuálnosť, s ohľadom na praktické programovanie, ako aj na tvorbu softwaru z hľadiska informatiky.

Popísali sme teóriu nášho prístupu k vizualizácii dát a rozlíšili sme základné pojmy dátovej štruktúry, vizualizačnej techniky a dizajnového variantu. Vyabstrahovali sme najpoužívanejšie potrebné štruktúry aj techniky a uviedli sme princípy tvorby dizajnových variantov.

Vytvorili sme taktiež pokročilý prototyp aplikácie, ktorá poskytuje nový a inovatívny prístup k variabilnej vizualizácii programov, s dôrazom na najširší záber implementovaných technológií, aj za cenu nižšej implementovanej variability. Popísali sme postup, akým bola aplikácia vytvorená a tiež princípy z oblasti algoritmiky a navrhovania softwaru, ktoré boli pri nej využité. Uviedli sme konkrétne možnosti aj spôsoby ďalšieho rozširovania aplikácie, aj s podrobným popisom fungovania jednotlivých rozširovaných tried. S ohľadom na fakt, že ďalšie rozširovanie aplikácie v tejto oblasti je skôr charakteru mechanického a pracného než charakteru inovatívneho, uviedli sme aj možnosti rozšírenia celkovej funkcionality, ako príklady možností ako na prácu ďalej nadviazať.

Prácu sme obohatili o príklady použitia systému a taktiež sme vytvorili a priložili kompletnú používateľskú príručku určenú pedagógom alebo iným používateľom, ktorí

by chceli pracovať s našou existujúcou aplikáciou, ktorú sme okrem priloženého CD sprístupnili taktiež na internete, na webovej adrese: `http://people.ksp.sk/~allie/diplomovka`

Pevne veríme, že táto práca pomôže pedagógom a študentom, ktorí sa stretávajú s vyučovaním programovania a uprednostňujú efektívne a moderné metódy vzdelávania nielen v škole, ale tiež vo voľnom čase.

Literatúra

[1] KUČERA, L., *Algovision*[online], [cit. 21.10.2012],
Dostupné na http://www.algovision.org/

[2] *AlgoViz Project* [online], 2013 [cit. 20.2.2013],
Dostupné na http://algoviz.org/avcatalog

[3] KOŠINÁROVÁ, A., *Vizualizácia základných grafových algoritmov*, bakalárska práca pod vedením J. Katreniakovej, Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Bratislava, 2011

[4] MARTIN, D. R., *Sorting Algorithm Animations*, 2007 [cit. 3.1.2012]
Dostupné na http://www.sorting-algorithms.com/

[5] *AlgoRythmics project* [videosúbor], 2012, [cit. 4.1.2012],
Dostupné na http://www.youtube.com/user/AlgoRythmics/videos

[6] Oracle, *Java^{TM} Platform Standard Ed. 7 API Specification* [online], © 1993 - 2013,
Dostupné na http://download.oracle.com/javase/7/docs/api/

[7] KSP, *Archív zadaní* [online], 2013, [cit. 14.10.2012]
Dostupné na http://www.ksp.sk/wiki/Zadania/Archiv

[8] GALLES, D., *Data Structure Visualization* [online], 2011 [cit. 19.3.2012]
Dostupné na http://www.cs.usfca.edu/ galles/visualization/

[9] CORTESI, A., *SortVis* [online], 2010 [cit. 13.1.2012],
Dostupné na http://sortvis.org/

[10] *Projekt SortVis* [online], [cit. 12.3.2012],
Dostupné na http://www.cs.tufts.edu/ ablumer/sortvis/SortVis.html

[11] ANDRUT R., *What different sorting algorithms sound like* [videosúbor], 2012 [cit. 3.1.2012],
Dostupné na http://www.youtube.com/watch?v=t8g-iYGHpEA

[12] PAPE, Ch., *Aufgaben* [online], [cit. 12.1.2012],
Dostupné na http://www.home.hs-karlsruhe.de/ pach0003/
informatik_1/aufgaben/en/java.html

[13] RÖSSLING, G. - NAPS, T. L., *A Testbed for Pedagogical Requirements in Algorithm Visualizations*, ITiCSE'02, Aarhus, Denmark, 2002

[14] IHANTOLA, P. - KARAVIRTA, V. - KORHONEN, A. - NIKANDER, J., *Taxonomy of Effortless Creation of Algorithm Visualizations*, ICER'05, Seattle, Washington, USA, 2005

[15] SHAFFER, C. A. - AKBAR, M. - ALON, A. J. D. - STEWART, M. - EDWARDS, S. H., *Getting Algorithm Visualizations into the Classroom*, SIGCSE'11, Dallas, Texas, USA, 2011

Príloha A

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000
#define REP(i,n) for (int i=0; i<(n); ++i)

struct bod { int x, y; };

int cmp (const void *a, const void *b) {
  bod A = *(bod*)a, B = *(bod*)b;
  return A.y*B.x - B.y*A.x;
}
int main() {
  int N, max=0;
  bod p[MAX], d[MAX];
  scanf (''%d'', &N);
  REP(i, N) scanf (''%d %d'', &p[i].x, &p[i].y);
  if (N <= 2) { printf (''%d'', N); return 0; }
  REP(i, N) {
    int M = 0;
    REP(j, N) {
      d[j].x = p[j].x - p[i].x;
      d[j].y = p[j].y - p[i].y;
      if (d[j].x > 0 || (d[j].x == 0 && d[j].y > 0)) ++M;
    }

```

```
26      qsort (d, M, sizeof(bod), cmp);
27
28      int r=1;
29      REP(j,M-1)
30        if (!cmp(&d[j], &d[j+1])) ++r;
31        else { if (r>max) max = r; r = 1; }
32      if (r>max) max = r;
33    }
34    printf (''%d\n'', max+1);
35    return 0;
36  }
```

Listing 1: Archív KSP, 24. ročník, 2. séria, 6. príklad "O hrdinnom Micovi", Vzorové riešenie[7]

```
1  var NAJVIAC, TAHOV, N, I: Integer;
2  begin
3    NAJVIAC := 0; TAHOV := 0; N := 0; I := 0;
4    read(N);
5    while N > 0 do begin
6      read(I);
7      inc(NAJVIAC, I-1);
8      inc(TAHOV, abs(NAJVIAC));
9      dec(N);
10   end;
11   writeln(TAHOV);
12 end.
```

Listing 2: Archív KSP, 23. ročník, 3. séria, 2. príklad "Zrovnoprávnenie guličiek", Vzorové riešenie[7]

```
1  public class Anagram {
2    public boolean isAnagram(String phrase, String anagram) {
```

```
3       char [] s1 = phrase.toLowerCase().toCharArray();
4       char [] s2 = anagram.toLowerCase().toCharArray();
5       int [] h1 = new int[256];
6       int [] h2 = new int[256];
7
8       createHistogramm(s1, h1);
9       createHistogramm(s2, h2);
10      for (int i=0; i < 256; i++) {
11        if (h1[i] != h2[i]) return false;
12      }
13      return true;
14    }
15
16    private void createHistogramm(char[] s1, int[] h1) {
17      for (char c : s1) {
18        if (! Character.isWhitespace(c)) h1[c]++;
19      }
20    }
21  }
```

Listing 3: Christian Pape, Arrays, ”Check for anagrams”[12]

```
var N, K, C, I, J: integer;
videl: array [1..10000] of boolean;
begin
  readln(N);
  for I := 1 to N do
    videl[I] := false;
  for I := 1 to N do begin
    read(K);
    for J := 1 to K do begin
      read(C);
      videl[C] := true;
    end;
  end;
  for I := 1 to N do
    if not videl[I] then write(I, ' ');
  writeln;
end.
```

Listing 4: Archív KSP, 28. ročník, 2. séria, 2. príklad "Zase tie závislosti", Vzorové riešenie[7]

```
var N, K, L: integer;
begin
  readln(N, K, L);
  if N mod (L + 1) = 1 then
    writeln('Vyhra lord Berkley')
  else writeln('Vyhra lord Norton');
end.
```

Listing 5: Archív KSP, 28. ročník, 1. séria, 5. príklad "Otrávený duel", Vzorové riešenie[7]

Príloha B - Príručka k použitiu systému VizzA

Obsah

1 Úvod

Hneď na začiatku vám chceme poďakovať, že ste sa rozhodli vyskúšať systém VizzA na vizualizáciu programov. VizzA vznikol v rámci diplomovej práce autorky v rokoch 2012-2013 a v rovnakom čase vznikla aj táto príručka pre používateľa.

Primárnym účelom tejto aplikácie je pomôcť pedagógom a študentom pri vyučovaní programovania a algoritmiky modernými metódami, ako aj predviesť nástroj, ktorý poskytuje možnosť vytvárania ilustrácií k textovým publikáciam v súvisiacej oblasti. Ako hlavnú cieľovú skupinu sme brali pedagógov na základných, stredných aj vysokých školách, ale ak aj nespadáte do tejto kategórie, nezúfajte. Našu aplikáciu môžete použiť samozrejme tiež, v plnom rozsahu.

2 Špeciality pre učiteľov

Medzi špeciality určené čisto pre účely vzdelávania, ktoré VizzA inovatívne prináša, patrí Skrytý mód a Kladenie otázok. V stručnosti si spomenieme ich popis a pedagogický význam.

Skrytý mód poskytuje po nahratí programu do aplikácie možnosť zamedzenia zobrazovania zdrojového kódu vizualizovaného programu. Táto možnosť poskytuje nielen novú variantu výučbovej techniky, ale tiež poskytuje viac priestoru pre využitie Kladenia otázok.

Kladenie otázok je bonusová funkcionalita, ktorá pri aplikovaní priradeného príkazu zastaví vizualizáciu a položí používateľovi otázku. Odpoveď používateľa následne aplikácia porovná so zadanou odpoveďou a používateľovi zobrazí výsledok a správnu odpoveď. Týmto spôsobom je možné klásť otázky s absolútne zadanou odpoveďou, alebo tiež otázky na aktuálny stav niektorej premennej. Možností využitia je nespočetne, ale najdôležitejšia je, že dáva vyučujúcemu prostriedky, ako docieliť, že sa konzument vizualizácie samostatne zamyslí nad behom programu.

3 Ako na to

Základný proces tvorby vizualizácie by sa dal zhrnúť do niekoľkých bodov:

- voľba použitých jazykov - na začiatku (aj keď je možné tento krok vykonať aj neskôr, najvhodnejšie je učiniť ho ešte pred začatím programovania) je treba uzrejmiť si, aký programovací a aký vizualizačný jazyk hodláte pri príprave použiť; tieto jazyky je možné nastaviť vpravo hore na paneli nástrojov.

- vloženie programu, ktorý chceme vizualizovať - či už s pomocou importnej funkcie, alebo priameho písania či kopírovania do programového panelu, potrebujete vytvoriť želaný program. V tejto fáze je treba dávať pozor na jemné odchýlky oproti štandardnej syntaxi programovacích jazykov. Priebežne rozpoznané kľúčove slová je možné sledovať v paneli zvýrazneného programu už počas vytvárania.

- vloženie vizualizačných príkazov, alebo upravenie existujúcich príkazov do vizualizačnej formy - na to, aby bol program vizualizovaný, musí obsahovať aj vizualizačné príkazy. Premenné, ktoré majú byť zobrazené, musia byť pri svojom vzniku označené vizualizačným príkazom. Podobne aj príkazy, ktoré majú byť zobrazené, musia byť vizualizačne označené. Okrem úprav (na pôvodných príkazoch) je možné pridať tiež čisto vizualizačné príkazy, ktoré poskytujú doplňujúce možnosti.

- spracovanie programu - s pomocou tlačidla na hornej lište. V prednastavenej sade ikoniek má zobrazenú žiarovku, ktorá je šedivá, pokiaľ aktuálny program v programovom paneli spracovaný nie je, a farebná, pokiaľ už spracovaný je.

- spustenie vizualizácie - s pomocou susedného tlačidla na hornej lište. V prednastavenej sade ikoniek má zobrazenie zelenej šípky, pokiaľ je možné spustiť vizualizáciu. Červený krížik namiesto šipky sa objavuje v prípade, že vizualizácia už beží, alebo ju nie je možné spustiť z iných príčin.

- v prípade, že bola vizualizácia prerušená niektorým z vizualizačných príkazov spôsobujúcich prerušenie - pauza, otázka - tak je vo vizualizácii možné pokračovať stlačením tlačidla pre spustenie programu.

Pozn.: V prípade potreby opätovného spustenia vizualizácie je nutné vizualizáciu znovu spracovať.

3.1 Poznámky k syntaxi jazykov

Vo **všetkých jazykoch** nastali tieto zmeny:

- **definície premenných:** v jednom riadku a príkaze definujte len jednu premennú. Nie je možné spájať definíciu a prvé priradenie hodnoty

- **záporné čísla:** zapisujte, prosím, záporné čísla vo forme 0-10 namiesto -10

- **vynechávanie zátvoriek** alebo begin/end za if alebo cyklami, pokiaľ sa jedná len o jediný vnorený prikaz: nefunguje, treba vždy použiť zátvorky, resp. iné označenie začiatku a konca vnorenej časti programu

- `i++, i-, inc(i), dec(i)`: tieto syntaktické skratky nefungujú, aplikujte, prosím, `i=i+1`, resp. `i=i-1`

- výpis a načítanie **zložených formátov:** nefunguje. Vždy načítavajte v jednom príkaze len jednu premennú. Vždy vypisujte buď reťazec, alebo hodnotu premennej. Nie oboje naraz, a nie zložený reťazcový výraz.
- pre operáciu **zreťazenia** znakov/reťazcov sa používa namiesto znaku `+` znak `.`
- **skracovanie** testovania **pravdivostných hodnôt** na pravdivosť nie je možné, je nutné používať plný zápis `bool==true` namiesto `tool`
- pri **for-cykloch** bude inkrementačný príkaz vždy vykonaný aj na konci poslednej iterácie podprogramu

V **jazyku C** nastali tieto zmeny, a vznikol teda C-typ jazyk:

- for cyklus: v zátvorke sa namiesto bodkočiarok využívajú čiarky; definícia cyklovej premennej sa musí odohrať pred príkazom cyklu, nie v ňom

V **jazyku Pascal** nastali tieto zmeny:

- premenné: je nutné uvádzať `var` na začiatku každej definície premennej.
- repeat cyklus: zmena formátu. Presný formát sa nachádza v tabuľke na konci tejto príručky.
- porovnanie premenných: namiesto `=` využívame jednotné `==`.

4 Nastavovanie vizualizácie

Vzhľad vizualizácie je možné upraviť rôznymi spôsobmi, avšak najdôležitejšou cestou je využitie vizualizačných parametrov. Parametre sa píšu medzi vizualizačný príkaz a telo programového príkazu, pokiaľ nie je povedané inak. Meno a hodnota parametra sú oddelené mriežkou a nesmú obsahovať medzery ani okolo mriežky. Presný formát teda je (pričom pri viacerých parametroch sú jednoducho písané vedľa seba, oddelené od seba medzerou a nezáleží na ich poradí):

```
viz-príkaz meno-parametra#hodnota-parametra telo-príkazu
```

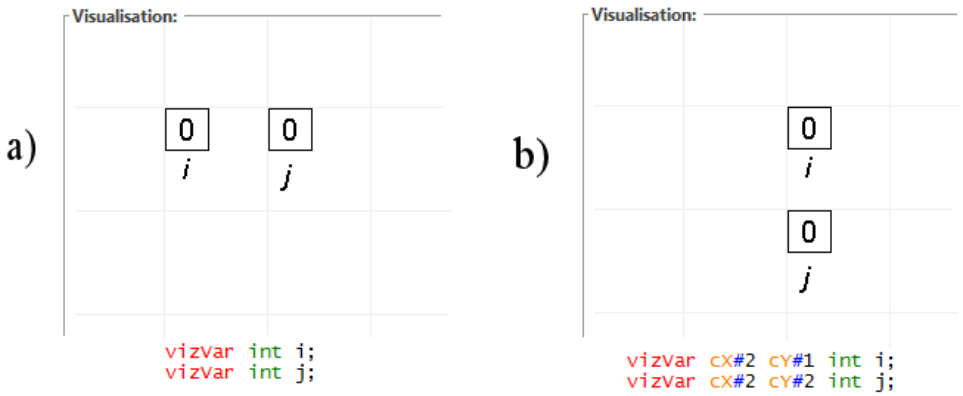
S použitím parametrov dokážeme ovplyvňovať veľa vlastností vizualizácie, a teda sa pozrieme na jednotlivé možnosti samostatne a podrobnejšie. Ešte spomeniem, že všetky parametre majú svoje štandardné označenie, ale niektoré môžu mať okrem toho aj korektné alternatívne označenia podľa použitého vizualizačného jazyka. Kompletný zoznam implementovaných parametrov k jednotlivým príkazom nájdete v tabuľke na konci tejto príručky.

4.1 Umiestňovanie premenných

Premenné, ktoré sa majú vizualizovať, majú svoj zobrazený ekvivalent umiestňovaný do mriežky, ktorej zobrazovanie je možné si zapnúť a vypnúť v paneli nástrojov. Do každého políčka v tejto mriežke je možné umiestniť len jeden prvok, pričom jeden prvok môže zaberať aj viac políčok. Vtedy sa jedná o umiestňovanie jeho ľavej hornej časti.

Pokiaľ nie je nastavené parametrami inak, prvok bude zobrazený v prvom voľnom políčku, pri "čítaní po riadkoch", do ktorého sa zmestí tak, aby sa neprekrýval so žiadnym iným prvkom a tiež netrčal z obrazovky. Pokiaľ sa také miesto nenájde, bude umiestnený do ľavého horného rohu, už bez ohľadu na prekryv. Výskyt tohto javu napovedá, že vizualizovaný panel je zúfalo preplnený a bolo by vhodné zredukovať počet vizualizovaných premenných, alebo ich preusporiadať.

Vždy je však možné určiť umiestnenie premennej podľa vlastného uváženia s pomocou parametrov pre zvislú a vodorovnú polohu - parametre `cX` a `cY`. Využívajú sa pri vytváraní premennej a ich hodnoty majú formu prirodzených čísel, ktoré zodpovedajú poradovému číslu políčka v rámci stĺpcov a riadkov mriežky.



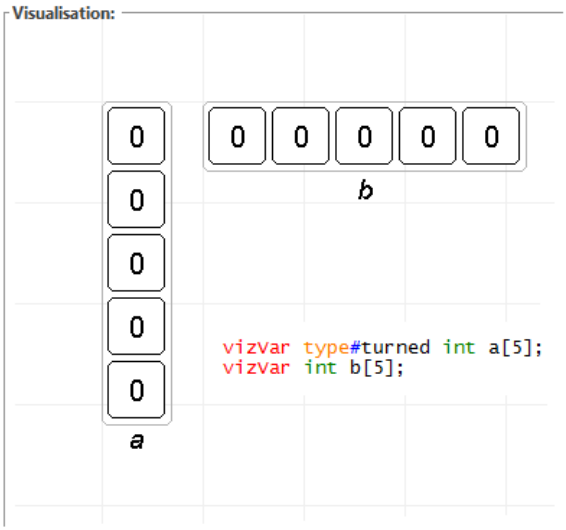
Obr. 1: **Ukážka využitia parametrov pri určovaní pozície.**
a) automaticky umiestnené premenné; b) premenné umiestnené parametrami.

4.2 Typ zobrazenia

S pomocou parametrov je možné nastaviť typ zobrazenia jednotlivých vizualizovaných premenných v prípade, že je k danému typu premennej implementovaných viac ako jedno možné zobrazenie. Meno parametra pre určovanie typu zobrazenia je `type` a uvádza sa hneď pri vytváraní premennej. Typ zobrazenia zodpovedá rozloženiu jed-

notlivých častí premennej voči sebe navzájom. Pokiaľ chcete meniť samotný výzor vizualizácie, prejdite na nasledujúcu časť, ktorou je Dizajn vizualizácie.

Aktuálne sú implementované: jednoduché zobrazenie pre premenné (`simple`), vodorovné (`multiple`) a zvislé (`turned`) zobrazenia pre jednorozmerné polia a tabuľkové zobrazenie pre dvojrozmerné polia (`table`).



Obr. 2: **Ukážka využitia nastavovania typu zobrazenia.**
pole `a` s nastavením zobrazenia na zvislé, pole `b` bez nastavenia zobrazenia a teda s aplikovaným prednastaveným vodorovným zobrazovaním.

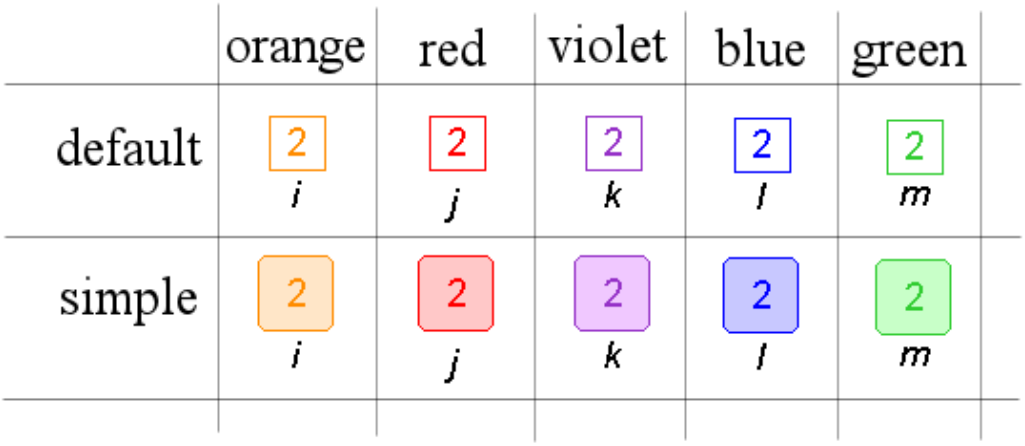
4.3 Dizajn vizualizácie

Okrem základného rozloženia prvkov premennej je nutné určiť, ako presne má premenná pri vykresľovaní vyzerať. Na tento účel slúžia dizajny spoločne s ich parametrami, na ktoré sa pozrieme. S pomocou dizajnu určujeme veľkosť jednotlivých častí vizualizácie, farebné zloženie, typ a veľkosť písma, ako aj mnoho iného. Všetky dizajnové parametre sa majú udávať pri príkaze určenom na vytvorenie premennej.

S pomocou parametra `design` sa dá nastaviť úvodná sada nastavení zhrnutá v niektorom implementovanom dizajne. Takáto sada poskytuje tvary, písma, umiestnenie hodnoty a názvu premennej, ako aj prednastavenú veľkosť a farby.

Vzhľadom na to, že často je možné očakávať, že hodnoty v niektorej premennej budú pri výpise širšie než je prednastavená veľkosť vizualizovaného prvku, je možné upraviť niekoľkonásobne šírku priestoru určeného pre hodnotu premennej. Koeficient rozšírenia je možné nastaviť parametrom `multiplicator`, ktorého hodnota vo forme prirodzeného čísla určuje koľkonásobok prednastavenej šírky má prvok zaberať.

Taktiež je možné samostatne nastaviť farebnú schému, ktorú dizajn používa, aj keď je nutné mať na pamäti, že nie každý dizajn musí využívať striktne len farby z tejto schémy. Farebné schémy sú nastavované s pomocou parametra `color`.



Obr. 3: **Ukážka implementovaných farebných schém.** Aplikované na implementovaných dizajnoch. Označenia riadkov a stĺpcov zodpovedajú hodnotám príslušných parametrov.

4.4 Globálne nastavenia

Keďže očakávame, že od vizualizácií bude aspoň v určitých smeroch žiadaná jednotnosť, pridali sme taktiež možnosť zavedenia globálneho nastavenia prednastavených hodnôt parametrov. Inými slovami, špecifickým príkazom je možné nastaviť vizuálne parametre pre všetky nasledujúce vizualizované premenné, pokiaľ pri nich nebude povedané inak.

Vlastnosť	**Globálny parameter**
Umiestnenie	nedá sa
Typ zobrazenia premennej	`varVisTech`
Typ zobrazenia jednorozmerného poľa	`arrayVisTech`
Typ zobrazenia dvojrozmerného poľa	`duoArrayVisTech`
Dizajn	`design`
Multiplikátor šírky	`multiplicator`
Farebná schéma	`color`

Tabuľka 1: Parametre pre vizualizáciu premenných.

5 Ďalšie funkcie

5.1 Vstup a výstup

Pod vizualizačným panelom sa nachádza miesto pre zadávanie vstupu a čítanie výstupu programu. Tieto dve oblasti fungujú ako náhrada klasického vstupu a výstupu s konzolou. Implementácia využívajúca tieto nástroje je jemne zjednodušená, keďže sa nejedná o vizualizačne kľúčovú vlastnosť. Vstupová konzola však spríjemňuje zadávanie vstupných dát do programu, takže sme sa rozhodli ju predsa len začleniť do VizzA. V implementácii sú medzery považované za oddeľovače vstupných dát, rovnako ako konce riadkov.

5.2 Rozpoznávanie chýb

Vedľa vstupno-výstupných panelov sa nachádza panel chýb, v ktorom sa, v prípade, že VizzA odhalí niektorú chybu, zobrazí popis nájdenej chyby. Pod chybami myslíme chybu v behu vizualizovaného programu. V aktuálnej verzii rozpoznáva VizzA tieto chyby:

- **Index mimo hraníc poľa** - v prípade, že sa snažíme pristupovať do poľa s indexom, ktorý ukazuje mimo zadaných hraníc poľa
- **Premenná nenájdená** - chyba sa objaví, pokiaľ sa pokúšame pracovať s premennou, ktorá v systéme neexistuje
- **Premenná už existuje** - ak sa snažíme vytvoriť viac rovnomenných premenných
- **Chýbajúci vstup** - pokiaľ by sme mali čítať neexistujúci vstup
- **Neinicializovaná premenná** - pokiaľ sa pokúšame pracovať s premennou, do ktorej sme predtým nepriradili hodnotu.
- **Neznámy príkaz** - ako jediná (z implementovaných chýb) sa zobrazuje už počas spracovania programu

5.3 Import a export programov

V rámci pohodlia a praktického použitia je možné zapísať pripravený program aj so vstupnými dátami a ďalšími nastaveniami do exportného textového súboru a ten si nahrať na disk. Podobne je možné z disku získaný súbor spätne vložiť do aplikácie a tým rýchlo a bez problémov posunúť svoj projekt ďalším používateľom. Na účel importu slúži prvá ikonka na paneli nástrojov (v prednastavenej sade je to modrá otvorená

zložka) a na účel exportu slúži štvrté tlačidlo na paneli nástrojov (v prednastavenej sade je to žltá disketa s textovým popisom).

Pozn.: Importný súbor má formát, ktorý je z väčšej časti čitateľný voľným okom, preto pokiaľ ho študenti nemajú vidieť, je nutné ho nahrať do aplikácie a následne odstrániť z ich dosahu.

5.4 Export obrázkov

VizzA ako jednu zo svojich hlavných funkcionalít poskytuje možnosť vytvárania vizualizačných ilustrácií k pedagogickým textom alebo prednáškam. Obrázky je možné exportovať samostatne z vizualizačného panelu, alebo z celej aplikácie. Zatiaľčo export celej aplikácie prebieha zásadne vo formáte png, pre export vizualizačného panelu ponúkame tiež možnosť formátu jpg, v prípade použitia exportného príkazu namiesto tlačidla.

Z tlačidiel slúžia na tento účel druhé a tretie tlačidlo panela, (čo je pri prednastavenej sade ikoniek čierna disketa a žltá disketa s obrázkom), pričom prvé menované vytvára obraz celej aplikácie, zatiaľčo druhé menované nahráva obrázok vizualizačného panelu. Okrem týchto variantov je možné na export využiť tiež príkaz, ktorý vo chvíli, keď sa bude mať vykonať, spustí nahrávanie obrázku aktuálneho stavu vizualizačného panelu. Po úspešnom ukončení nahrávania pokračuje beh programu ďalej. K tomuto príkazu patrí tiež voliteľný parameter `type` resp. `file`, ktorý určuje svojou hodnotou formát výstupného obrázku. Na výber sú hodnoty `jpg` a `png`.

5.5 Používateľské rozhranie

Používateľské rozhranie VizzA sa skladá z viacerých záložiek, z ktorých všetky, okrem poslednej, zodpovedajú rôznym rozloženiam prvkov aplikácie, aby mohol každý používateľ pracovať v takej forme, aká je mu prirodzená. Posledná záložka obsahuje nastavenia týkajúce sa používateľského rozhrania. Takýmito nastaveniami sú napríklad jazyk, alebo tiež povolené záložky. Tieto nastavenia sa pri exporte programov, môžu ale, nemusia tiež exportovať.

6 Tabuľky a schémy

Na nasledujúcich stranách sú uvedené tabuľký s kompletným zoznamom príkazov vizualizačného aj programového charakteru vrátane syntaktického použitia, ako aj kompletný zoznam implementovaných parametrov.

Príkaz	**Slovenčina**	**English**	**Namiesto prog. príkazu**
Pridanie premennej	`vizPremenna`	`vizVar`	áno, ak existuje
Priradenie hodnoty	`vizOperacia`	`vizOperation`	nie, píše sa pred neho
If	`vizIf`	`vizIf`	áno
For	`vizFor`	`vizFor`	áno
While	`vizWhile`	`vizWhile`	áno
Repeat	`vizRepeat`	`vizRepeat`	áno
Načítanie	`vizVstup`	`vizRead`	nie
Výpis	`vizVystup`	`vizWrite`	nie
Prerušenie	`vizVstup`	`vizRead`	čisto vizualizačný
Nahrať obrázok	`vizObrazok`	`vizImage`	čisto vizualizačný
Kladenie otázky	`vizOtazka`	`vizQuestion`	čisto vizualizačný
Globálne nastavenia	`vizNastavenia`	`vizGlobal`	čisto vizualizačný

Tabuľka 2: Vizualizačné príkazy a ich syntax vo VizzA.

Parameter	**Alternatívy**	**Príkaz**	**Hodnoty**
`multiplicator`	`mult` (slovenčina, english)	Premenná Glob. nastavenia	prirodzené čísla
`design`	`dizajn` (slovenčina)	Premenná Glob. nastavenia	`default, simple`
`color`	`farba` (slovenčina)	Premenná Glob. nastavenia	`red, blue, green violet, orange`
`cX, cY`	–	Premenná	prirodzené čísla
`type`	`visual` (english) `typ` (slovenčina)	Premenná Jednorozmerné pole Dvojrozmerné pole	`simple multiple, turned table`
`type`	`file` (všade), `typ` (slovenčina)	Nahratie obrázku	`jpg, png`

`varVisTech`
`arrayVisTech`
`duoArrayVisTech`
	–	Glob. nastavenia	`simple multiple, turned table`

Tabuľka 3: Existujúce vizualizačné parametre.

Príkaz	**C-typ**	**Pascal**

Tvorba premennej	`int i;`
	`boolean b;`
	`string s;`
	`char c;`
	`int a[5];`
	`int a[5][5];`
		`var i : integer;`
		`var b : boolean;`
		`var s : string;`
		`var c: char;`
		`var a: array [1..5] of integer;`
		`var t: array [1..5,1..5] of integer;`

Priradenie	`a[3]=5;`	`a[3]:=5;`

Podmienka	`if (`*`podmienka`*`) {`
	nutne v nových riadkoch...
	`}`
		`if` *`podmienka`* `then begin`
		nutne v nových riadkoch...
		`end;`

Začiatok a koniec tela programu	–	*premenné*
		`begin`
		program...
		`end.`

For cyklus	`int i;`
	`for (i=0,` *`podmienka`*`,`
	`i=i+1) {`
	nutne v nových riadkoch
	`}`
		`var i: integer;`
		`for i:=0 to` *`maximum`* `do`
		`begin`
		nutne v nových riadkoch
		`end;`

While cyklus	`while (`*`podmienka`*`) {`
	nutne v nových riadkoch
	`}`
		`while` *`podmienka`* `do begin`
		nutne v nových riadkoch...
		`end;`

Repeat cyklus	–	`repeat until` *`podmienka`* `begin`
		nutne v nových riadkoch
		`end;`

Načítanie	`readf(%`*`skratkaTypu`*`, i)`	`read(i);`
Výpis	`printf(i);` `printf("reťazec");`	`write(i);` `writeln("reťazec");`
Logické spojky	`&& || !`	`AND OR NOT`
Zreťazenie	`.`	`.`
Porovnanie rovnosti	`==`	`==`
Porovnanie nerovnosti	`!=`	`<>`
Porovnania	`>= <= > <`	`>= <= > <`

Tabuľka 4: Príkazy v programovacích jazykoch a ich syntax vo VizzA. Obsahuje aj zmeny oproti klasickej syntaxi.

Príloha C

```
vizGlobal design#simple color#orange;
int n;
scanf(%d,n);
vizVar int a[n];
int k;
for (k=0,k<n,k=k+1) {
  scanf(%d,a[k]);
}
vizVar int i;
vizVar int j;
int pom;
vizFor (i=n-1, i>0, i=i-1) {
  vizFor (j=1, j<=i, j=j+1) {
    if (a[j-1]>a[j]) {
      pom = a[j-1];
      a[j-1] = a[j];
      a[j] = pom;
    }
  }
}
###
7
7 6 5 4 3 2 1
```

Listing 6: Bubblesort.

```
vizGlobal color#blue;
var N : integer;
vizVar K : integer;
vizVar C : integer;
vizVar cX#7 cY#1 I : integer;
vizVar cX#8 cY#1 J : integer;
vizVar mult#2 videl:array[1..10] of boolean;
begin
read(N);
for I:=1 to N do begin
  videl[I]:=false;
end;

vizFor I:=1 to N do begin
  read(K);
  vizFor J:=1 to K do begin
    read(C);
    vizOperation videl[C]:=true;
  end;
end;
vizFor I:=1 to N do begin
  if videl[I]==false then begin
    write(I);
    write(" ");
  end;
end;
end.
###
5
3 2 3 4
```

```
1 4
1 4
0
1 3
```

Listing 7: Jeden z našich programov z programových schém.

```
var V:integer;
var E:integer;
var u:integer;
var x:integer;
var w:integer;
vizVar mult#2 dist:array[1..10,1..10] of integer;
vizVar i:integer;
vizVar cX#7 cY#2 j:integer;
vizVar cX#7 cY#3 k:integer;

begin
read(V);
read(E);

for i:=1 to V do begin
  for j:=1 to V do begin
    dist[i][j]:=100;
  end;
end;
vizPause;

for i:=1 to V do begin
   dist[i][i]:=0;
end;

```

```
26 for i:=1 to E do begin
27   read(u);
28   read(x);
29   read(w);
30   dist[u][x]:=w;
31 end;
32 vizPause;
33 vizFor k:=1 to V do begin
34   vizFor i:=1 to V do begin
35     vizFor j:=1 to V do begin
36       vizIf dist[i][k]+dist[k][j]<dist[i][j] then begin
37         vizOperation dist[i][j]:=dist[i][k]+dist[k][j];
38         vizPause;
39       end;
40     end;
41   end;
42 end;
43 vizPause;
44 end.
45 ###
46 4
47 5
48 1 3 -2
49 2 1 4
50 2 3 3
51 3 4 2
52 4 2 -1
```

Listing 8: Floyd-Warshallov algoritmus.

Príloha D

K práci je priložené DVD, na ktorom sa nachádza zdrojový kód aplikácie, použité časti GUI knižníc JGoodies, ukážkové vstupy z kapitoly Príklady použitia systému, ako aj skompilovaný JAVA-applet a jednoduchá webová stránka na jeho spustenie.