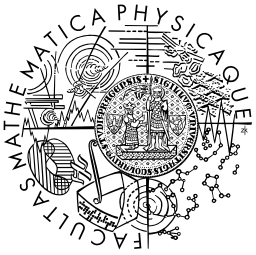


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Viktor Futó

Vizualizace vybraných algoritmů při využití XML

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.
Studijní program: Informatika
Studijní obor: Obecná informatika

Praha 2018

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Touto cestou by som sa chcel úctivo poďakovať svojmu vedúcemu práce, RNDr. Martinovi Pergelovi, Ph.D. za cenné rady, vecné pripomienky, ústretovosť a trpezlivosť pri konzultáciach a vypracovaní bakalárskej práce.

Název práce: Vizualizace vybraných algoritmů při využití XML

Autor: Viktor Futó

Katedra softwaru a výuky informatiky: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Témou tejto bakalárskej práce je návrh a implementácia nástroja na vizualizáciu vybraných numerických algoritmov s využitím jazyka XML. Nástroj ponúka vizualizačné prostredie, ktoré umožní užívateľovi zapísať numerický algoritmus pomocou vytvoreného XML jazyka a následne ho krok za krokom animovať. XML je jazyk určený pre výmenu dát a publikáciu dokumentov, ale pre jeho širokú aplikačnú využiteľnosť sa osvedčil aj mnohých iných oblastiach and pre jeho progresívnosť sa ho v našej práci pokúšame použiť ako programovací jazyk pre zápis algoritmov. Aplikácia je naprogramovaná vo frameworkoch Ruby on Rails a React a k parsovaniu XML jazyka využíva knižnicu Nokogiri. Použitie aplikácie je demonštrované na numerických algoritmoch simulovaného žíhania, násobnej iteračnej metódy a metóde gradient descentu, pri ktorej obzvlášť demonštrujeme schopnosť aplikácie vizualizovať kroky algoritmu aj na funkciách dvoch premenných.

Klíčová slova: vizualizácia XML numerická matematika algoritmy

Title: Visualisation of particular algorithms using XML

Author: Viktor Futó

Department of Software and Computer Science Education: Department of Software and Computer Science Education

Supervisor: RNDr. Martin Pergel, Ph.D., Department of Software and Computer Science Education

Abstract: The subject of this bachelor thesis is a design and implementation of a tool that visualizes particular numerical algorithms. The tool proposes a visualization environment that allows a user to write a numeric algorithm using a suggested XML language and then allowing them to animate the algorithm on the random dataset step by step. XML is a language intended for data exchange and document publishing, but as a result of its wide applicability it has established itself in many other areas. Because of its progressivity we try to use it as a programming language for writing algorithms. The application is programmed in the Ruby on Rails and React frameworks and uses the Nokogiri library to parse XML language. The usage of application is demonstrated on numerical algorithms of simulated annealing, power iteration method and method of gradient descent, in which in particular we demonstrate the application's ability to visualize steps of the algorithm also on the functions of two variables.

Keywords: visualization XML numerical mathematics algorithms

Obsah

Úvod

V súčasnom období vývoja civilizácie a mohutného a intenzívneho rozvoja informačných technológií je čoraz viac prítomnejší problém ľudského jedinca spracovávať obrovské množstvá dát, informácii a poznania [1]. Textuálny zápis nám už nedokáže v čase dostatočne rýchlo popísať poznanie, ktoré k chápaniu čoraz zložitejších vzťahov reality potrebujeme a preto je nevyhnutné hľadať nové spôsoby komunikácie poznania.

Toto je jedna z hlavných osobných motivácii výberu témy tejto práce. Nakoľko je ale náročné a rozsiahle uvažovať o nových spôsoboch prezentácie poznania ako takého, v našej práci sa zamierame na jeho úzku oblasť a to konkrétne na vizualizáciu vybraných numerických algoritmov. To môže pomôcť mnoho študentom ale i laikom k hlbšiemu, dostupnejšiemu a rýchlejšiemu porozumeniu ich fungovania. Po konzultácii s vedúcim našej práce sme sa rozhodli pokúsiť sa zefektívniť tvorbu takýchto vizualizácii vytvorením špeciálneho programovacieho jazyka, postaveného na syntaxi XML. Ten by mal byť určený k vizualizácii numerických a im príbuzných algoritmov. K jazyku vytvárame aj prostredie, ktoré algoritmy zapísané v našom programovacom jazyku interpretuje a vzápätí vizualizuje. Dôvodom výberu jazyka XML bola jeho progresivita, jednoduchá prenositeľnosť XML dokumentov a jeho čitateľnosť aj pre človeka neznalého programovania. To koreluje s našim zámerom naplňovať predovšetkým vzdelávací účel vizualizácií. Zároveň sme výberom jazyka XML chceli preskúmať nakoľko je jeho syntax vhodná k návrhu programovacieho jazyka.

Naša práca je členená do troch častí. V prvej z nich sa snažíme analyzovať požiadavky numerických algoritmov na aplikačné prostredie editora vizualizácií. V tejto časti opisujeme vybrané príklady numerických algoritmov, vizuálnu predstavu aplikácie, ktorú následne implementujeme a zjednodušený statický model vizualizácie konkrétneho algoritmu. Našim cieľom v tejto časti je mať pripravenú konkrétnu vizualizáciu algoritmu, ktorú v nasledujúcej časti popíšeme s využitím syntaxe XML za účelom vytvorenia programovacieho jazyka.

V druhej časti práce analyzujeme XML jazyk a programovací jazyk ako taký. Oboznamujeme sa so základnými abstrakciami, paradigmami a kritériami dobrého programovacieho jazyka a načrtneme problematiku jeho programového spracovania. Na základe našich praktických skúseností ale aj týchto poznatkov vytvárame návrh vlastného jazyka XML na príklade vizualizácie konkrétneho algoritmu z predošlej kapitoly. K nemu následne vytvárame tzv. parser, softvérový nástroj, ktorý náš XML zápis algoritmu interpretuje do nami vytvoreného prostredia vizualizácie.

V tretej časti našej práce rozširujeme náš XML jazyk a jeho vizualizačné prostredie o požiadavky ďalších algoritmov a to najmä o vykresľovanie funkcií jednej a dvoch premenných ako aj o nový typ okna pre zobrazovanie matematických výrazov. Nové prvky jazyka a prostredia demonštrujeme v nami vybraných nových numerických algoritmoch.

Súčasťou práce je aj užívateľská a programová dokumentácia a v prílohe je možné nájsť príklady algoritmov zapísané v nami vytvorenom XML jazyku.

1. Požiadavky na XML jazyk

Ako už bolo uvedené v úvode, cieľom našej práce je vytvoriť programovací jazyk využívajúci syntaxe jazyka XML za účelom vizualizácie vybraných numerických algoritmov. V našej práci budeme takýto jazyk postupne navrhovať, ale predtým než sa pustíme do samotného vytvárania XML jazyka, potrebujeme uskutočniť prieskum potrieb nášho jazyka aby sme dokázali poznať, čo za jazyk vlastne chceme vytvoriť. Čo náš jazyk má obsahovať? Aké kontrolné štruktúry a značky majú byť súčasťou nášho XML jazyka? Prvá časť našej analýzy je určená práve k výstavbe poznania, ktoré nám lepšie dokáže odpovedať na tieto otázku a k samotnému vytváraniu jazyka pristúpime až v jej druhej časti. Naším cieľom je nielen vytvoriť XML jazyk pre zápis vybraných algoritmov, ale vytvoriť XML jazyk, ktorý umožní taký zápis algoritmu, že priebeh algoritmu bude **vizualizovateľný v prostredí** našej aplikácie resp. interpreta nášho jazyka. Je potrebné si uvedomiť, že náš jazyk nechce byť len samoúčelná exkurzia do možností využitia jazyka XML, ale chce sa už pri tejto príležitosti pokúsiť byť aj prostriedkom k užitočnej vizualizácii algoritmov. Preto sme sa rozhodli pristúpiť najprv k tvorbe takéhoto vizualizačného prostredia a až následne samotného XML jazyka, nakoľko to nám ozrejmí jeho konkrétne požiadavky. Takto sa vyhneme vytváraniu niečoho čo by následne nemuselo byť užitočné.

1.1 Požiadavky numerických algoritmov

Už vieme, že k návrhu XML jazyka určeného pre vizualizáciu algoritmov budeme potrebovať vizualizačné prostredie, ktoré bude takýto jazyk interpretovať a daný algoritmus nejako vizualizovať. Predtým než sa pustíme do tvorby takéhoto prostredia, potrebujeme poznať samotné požiadavky algoritmov, ktoré chceme vizualizovať - potrebujeme poznať čo si pod pojmom vizualizácie máme predstaviť a to veľmi závisí od toho aké algoritmy chceme vizualizovať. Preto v tejto sekcii popisujeme náš myšlienkový proces pri tvorbe nárokov nami vybraných algoritmov na ich vizualizáciu.

Už pri výbere témy existovala predstava, že výhodou numerických algoritmov je zdieľanie práve takých vlastností, ktoré vytvárajú predpoklad pre existenciu pre nich vhodného univerzálneho vizualizačného prostredia. Jednou z týchto vlastností je ich **aproximačná povaha**, ktorá nepriamo predurčuje podobu nášho prostredia ako prostredia, v ktorom v rámci animácie, snímok po snímke je možné proces aproximácie vizualizovať. Tou druhou je, že táto aproximácia prebieha v prostredí množiny bodov, alebo funkcií, ktoré majú vďačnú vlastnosť byť jednoducho vizualizované a familiárne aj pre užívateľa, ktorý nemá dostatočné technické a matematické znalosti. Práve univerzálnosť a základné poznanie obrazu matematickej funkcie zobrazovanej na dvojrozmernom plátne karteziánskej sústavy súradníc, na rozdiel od povedzme algoritmov lineárneho programovania alebo algoritmov ktoré majú individuálne odlišné nároky na vizualizáciu i v rámci jednotlivých tried algoritmov, bola pre nás na základe našich skúseností tým rozhodujúcim faktorom pre výber práve numerických algoritmov a môže byť tým základným univerzálnym stavebným kameňom myšlienky animovania algoritmov pre ďalšie rozšírenie aplikácie.

Do predošlej časti sme vstupovali s istou predstavou o podobe numerických algoritmov, tú však potrebujeme podchytiť ich reálnou podobou a preto sme pristúpili k analýze vybraných približne dvoch desiatok rôznych druhov numerických algoritmov a ich princípov. V nasledujúcich sekciách uvádzané poznatky nevyhnutné k analýze týchto algoritmov sme získavali najmä z práce J. Solomona [2]. Naše úvodné predpoklady sa potvrdili a na príkladoch výberu algoritmov Gramovovej–Schmidtovej ortogonalizácie, násobných metód, algoritmu bisekcie, metódy zlatého rezu, metód gradient descentu, optimalizácie rojom častíc (particle-swarm algorithm) a algoritmu simulovaného žíhania sme definovali vizualizačné požiadavky našej aplikácie. Tieto algoritmy sme si vyberali v snahe vybrať reprezentatívne typy takých algoritmov, z ktorých má každý individuálne, špecifické a odlišné požiadavky na vizualizáciu. Zároveň sme sa snažili vyhnúť typovo rozsiahlejším a zložitejším algoritmom, ktorých náročnosť realizácie by mohla odkloniť našu pozornosť od snahy vytvoriť predovšetkým **univerzálne** prostredie vizualizácie algoritmov.

1.1.1 Gramova-Schmidtova ortogonalizácia

Gramova-Schmidtova ortogonalizácia je algoritmus nájdenia ortogonálnych vektorov generujúcich ten istý lineárny obal ako množina lineárne nezávislých vektorov akceptovaná na vstupe algoritmu. Princíp algoritmu spočíva vo fixovaní prvého vektora prijatého na vstupe ako toho, ku ktorému budú ostatné ortogonálne, a následnom **projektovaní** každého ďalšieho vektora **do ortogonálnej roviny** voči už existujúcim ortogonálnym vektorom. O vizualizácii algoritmu sa dá uvažovať v prípade implementácie zobrazovania vektorov v dvoj- a trojrozmernom priestore projektovanom na dvojrozmernej ploche vykresľovacieho plátna.

1.1.2 Algoritmus bisekcie

Algoritmus bisekcie slúži k nájdeniu koreňov polynómu. Algoritmus **iteratívne póli interval** a vyberá si z neho podinterval, v ktorom sa koreň musí určite nachádzať. Takmer nikdy sa mu nepodarí nájsť presnú hodnotu koreňa, ale dokážeme s ním určovať ako ďaleko od daného koreňa sa môže jeho určený odhad v danej fáze algoritmu nachádzať. Myšlienka animácie takéhoto algoritmu spočíva v animovaní vodiacich čiar vo vykreslenej funkcii, ktoré určujú aktuálne vybratý interval funkcie pre nájdenie koreňa. To sa dá realizovať aj vo finálnej verzii aplikácie vrátane postupnej transfokácie grafu do čoraz menšieho a menšieho intervalu.

1.1.3 Metóda zlatého rezu

Metóda zlatého rezu nachádza mininimum na **unimodulárnych funkciách**. To sú funkcie, ktoré sa dajú, jednoducho povedané, rozdeliť na dve rôzne monotónne časti (napr. na interval, na ktorom je funkcia klesajúca, zatiaľ čo zvyšná časť funkcia je len stúpajúca). Metóda zlatého rezu postupne osekáva interval, v ktorom sa má nachádzať minimum funkcie a to spôsobom využívajúcim konštantu zlatého rezu. Metóda funguje aj v prípade nediferencovateľných funkcií. Metóda zlatého rezu sa dá animovať podobne ako algoritmus bisekcie, zobrazova-

ním vodiacich čiar definujúcich interval, v ktorm sa nachádza minimum funkcie a voliteľnou transfokáciou grafu v okolí intervalu.

1.1.4 Optimalizácia rojom častíc

Optimalizácia rojom častíc (particle-swarm algorithm) je algoritmus hľadania minima, v ktorom si pamätáme zoznam možných, potenciálnych miním. Každému takémuto minimu, častici priraďujeme rýchlosť a pozíciu. Pamätáme si takisto aj globálne minimum. Na základe lokálneho optima danej „častice“ a globálneho optima všetkých častíc iteratívne meníme **rýchlosť častice** a overujeme či sa častica nachádza v novom optime, či už lokálnom alebo globálnom. Algoritmus sa dá vizualizovať vo finálnej verzii našej aplikácie vykresľovaním bodov na plátne vykreslenej funkcie dvoch premenných.

1.1.5 Metóda gradient descent

Metóda gradient descent je iteratívna metóda slúžiaca k nájdeniu minima resp. maxima funkcie skrz jednorozmerné posuny proporčné k hodnote ich príslušných gradientov. Laicky povedané, pri hľadaní minima funkcie sa pohybujeme po jej pomyselnou povrchu vždy v smere gradientu, to znamená **v smere najstrmšieho rastu** alebo poklesu a miera zmeny (rastu alebo poklesu) určuje veľkosť kroku, ktorým sa po funkcií pohneme. Takýmto spôsobom, v prípade veľkého množstva funkcií, algoritmus nájde lokálne minimum resp. maximum.

1.1.6 Násobné algoritmy

Násobné algoritmy alebo *power iteration algorithms*, označujú triedu algoritmov slúžiacich na **určenie vlastných čísel** a vlastných vektorov matíc. Násobný algoritmus, na ktorom demonštrujeme použiteľnosť nášho XML jazyka a funkčnosť našej vizualizačnej aplikácie prijíma na vstupe diagonalizovateľnú maticu A a náhodne zvolený vektor v. Následne je v cykle matica A násobená vektorom v, pričom výsledok násobenia sa znova ukladá do hodnoty vektora v. Hodnota vektora v konveruje k hodnote dominantného vlastného vektora matice A. Vektor je odporúčané pre efektívnosť násobenia v cykle normalizovať. Na základe vektora je možné dodatočne vypočítať hodnotu vlastného čísla matice.

1.1.7 Algoritmus simulovaného žíhania

Algoritmus simulovaného žíhania je spôsob nájdenia globálneho extrému. Pojem žíhanie sa odkazuje na proces v metalurgii, v ktorom je kov prvotne **nahriaty a následne ochladzovaný** aby sa jeho zložkové častice usporiadali do stavu minimálnej energie. Podobne ako pri pohybe častíc nahriatého kovu, sa v prípade algoritmu simulovaného žíhania potenciálny optimálny bod na začiatku pohybuje v rámci obrovských intervalov, ale schladzovaním sa jeho pohyb obmedzuje. Pohyb potenciálne optimálneho bodu je tiež obmedzený tým, že bod ostáva iba na tom mieste, ktoré je má extrémnejšiu hodnotu ako všetky predošle navštívené hodnoty a takto nachádza čoraz lepšie a lepšie riešenie optimalizačného problému.

To čo môžeme chcieť dosiahnuť je vedieť zobrazovať v súradnicovom systéme **množinu bodov**, **funkciu jednej premennej**, **funkciu dvoch premenných**,

vektory a prvky legendy ako rôzne **čiarové alebo bodové značky**, ktorých poloha sa v rámci rôznych snímkov môže meniť. Pretože existujú aj algoritmy, ktoré je náročné vizualizovať ako množinu bodov v karteziánskej súradnicovej sústave ako napr. násobné metódy, chceme zabezpečiť aspoň písomný animovaný opis toho čo sa v jednotlivých fázach takých algoritmov deje a preto okrem okna grafu, chceme pracovať aj s oknom, v ktorom bude vyjadrený stav algoritmu prostredníctvom **vykresleného matematického výrazu**. Je potrebné povedať, že spomenutý výber algoritmov a požiadaviek nášho interpreta bol schválne omnoho ambicióznejší než ich užší výber, ktorý sa nám nakoniec podarilo realizovať v rámci výsledneho XML jazyka a aplikácie na jeho interpretáciu. To bolo dané našou snahou zabezpečiť univerzálnosť pri návrhu XML jazyka a jeho interpreta. Širšou definíciou požiadaviek dokážeme zabezpečiť omnoho lepšiu možnosť rozšírenia XML jazyka a jeho interpreta v budúcnosti. V konečnom užšom výbere algoritmov, na ktorých demonštruujeme náš XML jazyk sa nachádza algoritmus simulovaného žíhania, násobný algoritmus a metóda gradient descentu demonštrovaná na funkciách jednej aj dvoch premenných.

1.2 Návrh vizualizačného prostredia

Oboznámili sme sa s nárokmi algoritmov, ktoré máme v záujme vizualizovať. Nakoľko je našim cieľom vytvoriť univerzálne a rozširovateľné grafické prostredie interpreta nášho XML jazyka chceme tieto jednotlivé nároky osadiť do takéhoto prostredia. K tomu si potrebujeme lepšie priblížiť predstavu toho ako také prostredie bude vyzerať. V predošlej sekcii sme už raz použili termín „okno“, ktorý odkrýva, že naša vizuálna predstava aplikácie bola silne ovplynená **okennou grafickou architektúrou**, v ktorej existujú poťahovateľné okná rôznych typov pre vizualizáciu rôznych druhov informácii. V tejto vizuálnej predstave je užívateľ pri spustení aplikácie ihneď v jej prostredí, kde jediným otvoreným oknom na pracovnej ploche je XML editor, v ktorom píšeme náš algoritmus. Ten po odoslaní a úspešnej kompilácii následne vráti podľa nášho programového XML zápisu generované okná, ktorých obsah sa môže meniť pri prechádzaní jednotlivými snímkami algoritmu. Účelom jednotlivých okien môže byť vykreslenie matematického výrazu, vykreslenie funkcie alebo množiny bodov v súradnicovej sústave a pod. Súčasťou takto vykreslenej funkcie alebo bodov, môže byť aj značka označujúca momentálnu hodnotu premennej v jedinom kroku, stave animácie algoritmu. Podobne matematický výraz môže svoju hodnotu po jednotlivých snímkoch meniť. Chápanie vizualizácie algoritmu ako vizualizáciu jeho jednotlivých stavov predurčuje nevyhnutnosť existencie kontrolného panelu, pomocou ktorého môžeme jednotlivými snímkami, stavmi prechádzať alebo si ich nechať prehrať jeden po druhom podobným spôsobom ako pri prehrávaní videa.

1.3 Algoritmus simulovaného žíhania

S prehľadovými poznatkami o nárokoch numerických algoritmov, ktoré by sme chceli vizualizovať, chceme zabezpečiť funkčnosť našej aplikácie na **jednom konkrétne vybranom algoritme**. Zároveň chceme zachovať univerzálnosť aplikácie pre ostatné algoritmy nášho výberu. Naša motivácia **demonštrovať** najprv

vizualizačné prostredie na konkrétnom algoritme pred návrhom samotného XML jazyka je určená predstavou, vyplývajúcou z našich skúseností, že k vizualizácii daného algoritmu budeme potrebovať vkladať príkazy do jeho samotného kódu. Tie budú určovať zmenu vizualizovaného stavu animácie algoritmu a to tak, že pri behu algoritmu, sa bude meniť stav jeho animácie. Štruktúra a podoba týchto vložených príkazov určí aj podobu nášho XML jazyka. Stále ale potrebujeme prvotne poznať podobu týchto príkazov v jazyku Javascript, do ktorého sa náš XML jazyk bude kompilovať. Podobu týchto príkazov a tak podobu kódu vizualizovaného algoritmu spoznáme pri demonštrácii nášho vizualizačného prostredia na konkrétnom algoritme. Pre tento účel sa nám javil ako najvhodnejší už raz popisovaný algoritmus simulovaného žíhania.

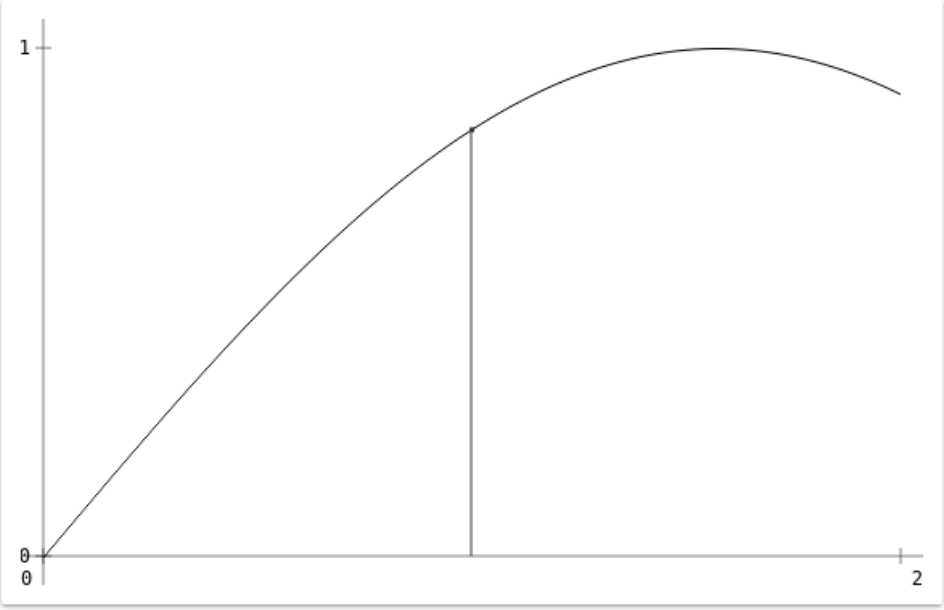
Ten je možné aplikovať, ako algoritmus hľadania extrému, tak ako na množine náhodne generovaných bodov, tak aj na spojitej funkcii. Naším cieľom je na začiatok zabezpečiť funkčnosť algoritmu v jazyku Javascript a to v jedinom statickom HTML súbore mimo nášho vizualizačného prostredia, ktorý bude zjednodušene reprezentovať obrazovku užívateľa po odoslaní XML zápisu toho istého algoritmu. Implementácia algoritmu simulovaného žíhania v jazyku Javascript na poli náhodne generovaných čísel, vyzerá takto:

```
// Generujeme náhodné dáta
var data = [];
for (var j = 0; j < 100; j++) { data[j] = Math.random(); }

// Deklarácia premenných
var t = 1.0, t_min = 0.00001, alpha = 0.9,
    y = data[Math.floor(Math.random() * (data.length - 1))], x;

// Hlavný cyklus algoritmu prebieha až kým
// teplota nedosiahne vopred určené minimum
while (t > t_min) {

  var i = 1;
  while (i <= 100) {
    new_x = Math.floor(Math.random() * (data.length - 1));
    new_y = data[new_x];

    // Pravdepodobnosť prijatia nového maxima
    p = Math.E * (new_y - y) / t;
    if (p > Math.random()) { x = new_x; y = new_y; }
    i++;
  }

  // Ochladenie teploty o koeficient alpha
  t = t * alpha;
}
```

Výpisom `console.log(Math.max(...data), y)` môžeme overiť správnosť našej implementácie algoritmu a teda či hodnota `y` sa zhoduje, alebo aspoň približuje skutočnému maximu poľa náhodne generovaných hodnôt.

1.4 Statická vizualizácia algoritmu

Našim ďalším cieľom je staticky vizualizovať priebeh algoritmu a to ideálne spôsobom zobrazenia priebežne určeného extrému na plátne, na ktorom bude vykreslená množina náhodne vygenerovaných bodov, čísel. Slovo staticky neznačí nič iné, len fakt, že zatiaľ algoritmus vizualizujeme mimo nášho univerzálne aplikačného prostredia, v **statickom HTML súbore**. Pre účely zobrazovanie extrému na plátne náhodne vygenerovaných bodov sme sa snažili nájsť knižnicu, ktorá by nám uľahčila zobrazovanie nielen polí čísel ale aj samotných matematických funkcií. Esteticky najpôsobivejšou a zároveň najdostupnejšou knižnicou na vykresľovanie funkcií písanou v jazyku Javascript bola knižnica *Function Plot*, ktorú čitateľ ľahko nájde na adrese `https://mauriciopoppe.github.io/function-plot/`. Táto knižnica nanešťastie nesplňovala naše požiadavky a to predovšetkým schopnosť **simultánne prekreslovať graf** za behu algoritmu a tiež nástroje na vykresľovanie vizuálnych znakov legendy, ako napr. vodiacej čiary určujúcej polohu nášho hľadaného extrému ilustrovaného na obrázku.



Obrázek 1.1: Vodiaca čiara určujúca priebežnú polohu algoritmom hľadaného extrému

Retrospektívne je potrebné povedať, že by sme si za istých okolností pravdepodobne vystačili s danou knižnicou avšak analýza samotnej knižnice a do nej integrácia našich potrieb vizualizácie by nám pravdepodobne trvala omnoho viac času. Preto sme sa v danej chvíli, pre naše potreby omnoho väčšej flexibility, rozhodli implementovať naše vlastné riešenie pre vykreslovanie bodov a neskôr funkcií v súradnicovom systéme pomocou elementu, alebo inak - HTML značky `<canvas></canvas>`.

1.4.1 HTML

HTML - HyperText Markup Language je značkovací jazyk určený k tvorbe webových stránok a aplikácii a popisuje štruktúru webovej stránky. Skladá sa z tzv. „tagov“ alebo značiek uzavretých v špicatých zátvorkách. Bol vytvorený ako štandardizujúci formát k písaniu webových dokumentov vymieňaných po sieti, ktorý mali zobrazovať rôzne druhy webových prehliadačov.

1.4.2 Element `<canvas></canvas>`

`<canvas></canvas>` je HTML element slúžiaci na priame vykreslenie grafických objektov a animácii v jazyku HTML. Ako uvádza D. Flanagan [3], jeho pôvodným účelom bolo predísť sťahovaniu sofistikovaných grafických objektov z webového servra a umožniť týmto objektom byť vykreslené na strane klienta. To je užitočné najmä v prípadoch, v ktorých je samotný kód slúžiaci k vykresleniu menší ako vykreslený objekt. Bol štandardizovaný vo verzi HTML jazyka 5, ničmenej bol používaný webovými prehliadačmi už dávno predtým. Dnes nachádza široké využitie pri kreslení alebo animovaní grafických objektov v rámci webových stránok.

Pri implementácii vlastného riešenia vykresľovania množiny náhodne vygenerovaných bodov a neskôr funkcií sme sa inšpirovali existujúcim dostupným riešením M. Mighta [4]. Vykresľovanie funguje pomerne jednoducho. Máme pole náhodne generovaných čísel v rozsahu od 0 do 1 a priraďujeme im relatívne umiestnenie v rámci plátna - elementu `<canvas></canvas>`, po celej jeho dĺžke a výške. Čo je pre nás, ale dôležité, je vykresľovať tieto body počas behu algoritmu.

V prvej verzii implementácie bolo našou predstavou vloženie volania funkcie na vykreslenie všetkých bodov hneď za deklaráciou a vyplnením poľa náhodných hodnôt

```
// Generate random data
var data = [];
for (var j = 0; j < 100; j++) { data[j] = Math.random(); }

// Canvas
renderCanvas(data);
```

a následne volanie na vykreslenie vodiacej čiary určujúcej funkčnú hodnotu momentálneho maxima.

```
function sleep(delay) {
  var start = new Date().getTime();
  while (new Date().getTime() < start + delay);
}

...

if (p > Math.random()) {
  x = new_x; y = new_y;
  sleep(3000);
  renderGuideline(x, data);
}
```

Vykreslenie všetkých bodov fungovalo podľa očakávaní, horšie to bolo s animovaním vodiacej čiary. Naša predstava spočívala v tom, že pri každom volaní funkcie `sleep`, chod algoritmu zastane na dobu 3 sekúnd a až tak vykreslí novú vodiacu čiaru resp. zmenu jej polohy.

To však v praxi nefunguje, pretože webový prehliadač vykresľuje obsah statickej webovej stránky až po dokončení behu vloženého javascriptového kódu a teda nášho algoritmu simulovaného žíhania. Funkcia `sleep` beží v tom istom vlákne ako vykreslenie HTML obsahu webovej stránky a **Javascript** je jazyk, ktorý je nepozná koncept paralelného behu programov vo viacerých vláknach, inými slovami, sám beží **iba v jedinom vlákne**. Webový prehliadač teda čaká na dokončenie behu Javascriptu a až tak vykresľuje výsledný obsah a prirodzene HTML element `<canvas></canvas>` v jeho poslednom nami vykreslenom stave.

Pokiaľ chceme zabezpečiť simultánne vykresľovanie zmien v elemente `<canvas></canvas>` počas behu nášho algoritmu, musíme použiť vyhradenú metódu
`window.requestAnimationFrame(callback)`, ktorá ako argument preberá identifikátor funkcie, ktorú chceme volať pri ďalšom prekreslení obsahu okna webového prehliadača.

1.4.3 Funkcia `requestAnimationFrame(callback)`

Metóda okna webového prehliadača, ktorá požaduje od prehliadača aby pri najbližšom prekresľovaní zobrazovaného obsahu webstránky bola volaná funkcia predaná metóde ako argument. Ako úvadza D. Geary [5], pri tvorbe animácie bolo kedysi štandardom vytváranie cyklu použitím metódy `setInterval`, ktorý je uskutočňovaný v pravidelných intervaloch a volá funkcie vykreslovania stavu animácie. Pri webovom prehliadači, takýto cyklus beží sústavne aj v prípade, že webstránka nie je práve aktívna resp. zobrazovaná čo zahlcuje pamäť využívaného zariadenia. Práve k lepšej **efektívnosti využívania pamäti** slúži metóda `requestAnimationFrame(callback)`. Tá v prípade deaktivácie webovej stránky, napr. otvorením novej záložky, prestane byť vykonávaná a tak ušetrí pamäť nášho zariadenia a zlepší jeho celkový výkon. Metóda tiež uskutočňuje všetky animácie naraz a beži v predvolenej frekvencii 60 snímkov za sekundu.

Využitím metódy `requestAnimationFrame(callback)` nechávame pri statickej vizualizácii nášho algoritmu rekurzívne bežať funkciu `draw`, ktorá sústavne prekresľuje obsah plátna.

```
function draw() {
  // Vyčisti plátno
  ctx.clearRect(0, 0, w, h);

  // Vykresli súradnice
  renderAxes();

  // Vykresli dáta
  renderData();

  // Vykresli vodiacu čiaru označujúcu hľadaný extrém
  renderGuideline();
```

```
  // Pri ďalšom prekreslení požiadaj o volanie funkcie draw
  window.requestAnimationFrame(draw);
}
```

Zmena vodiacej čiary určujúcej hľadaný extrém v algoritme simulovaného žíhania tentoraz už neinvokuje opätovné prekreslenie celého plátna, ale mení iba stav zatiaľ čo plátno sa kontinuálne prekresľuje.

```
if (p > Math.random()) {
  x = new_x; y = new_y;
  this.guidelineX = x;
}
```

Problém takéhoto prístupu je, že náš algoritmus a počet iterácii je vykonaný prakticky v ráde milisekúnd okamžite po načítaní javascriptového kódu a tak naše oko nie je schopné spozorovať animovaný algoritmus a vidí iba jeho finálny stav. Potrebujeme dostať stav algoritmu sami pod kontrolu a už sme sa stihli presvedčiť, že zastavenie behu vlákna, v ktorom Javascript beží, to nepôjde. Tu prichádzame k zásadnej myšlienke, ktorá definuje spôsob animovania algoritmu pre našu aplikáciu. Miesto toho aby sme animovali a **vizualizovali** chod algoritmu **počas** jeho samotného behu, necháme algoritmus prebehnúť ihneď **po** jeho načítaní a uložíme si jeho potrebné stavy do poľa snímkov animácie algoritmu. Zapisovanie zmeny vodiacej čiary vyzerá tak nasledovne:

```
if (p > Math.random()) {
  x = new_x; y = new_y;
  frames.push({ guidelineX: x });
}
```

Takýto prístup nám umožňuje do poľa snímkov ako jeho položky ukladať prakticky neobmedzene veľké objekty popisujúce ľubovoľné premenné stavu algoritmu. Zároveň môžeme prechádzať jednotlivými snímkami algoritmu funkciami `prev()` a `next()`, ktoré nerobia nič iné len menia hodnotu indexu práve aktívneho snímku. Vo vyššie uvedenej funkcii `draw` takto vykresľujeme práve aktívny snímok na základe indexu `frame`.

```
function prev() { if (0 < frame) { frame--; } }
function next() { if (frame < frames.length - 1) { frame++; } }
```

```
<button onClick="prev()"><</button>
<button onClick="next()">></button>
```

1.5 Integrácia vizualizácie algoritmu

V predošlej sekcii sa nám podarilo vytvoriť statickú vizualizáciu konkrétneho algoritmu. Statickú preto, pretože bola vytvorená a popísaná pre konkrétny algoritmus v rámci statického `*.html` súboru. My sa ale snažíme vytvoriť XML jazyka

a jeho interpreta, v ktorom budeme môcť vytvárať a zapisovať rôzne druhy algoritmov. Potrebujeme pre to najprv vytvoriť univerzálne vizualizačné prostredie, v ktorom sa tieto algoritmy budú vizualizovať a príprava takéhoto prostredia a jeho demonštrácia na práve vytvorenej vizualizácii algoritmu simulovaného žíhania je to, čomu sa v tejto sekcii budeme venovať. Našim cieľom teda je vytvoriť základnú verziu vizualizačného prostredia, v ktorom po odoslaní predpokladaného algoritmu zapísaného v našom XML jazyku sa priebežne bude vizualizovať algoritmus simulovaného žíhania na základe v predošlej sekcii vytvorenej implementácie.

K vytvoreniu nášho prostredia resp. webovej aplikácie pre interpretáciu nami vytvoreného XML jazyka budeme používať **webový framework Rails**. Ten je bližšie popísaný v programátorskej dokumentácii, ničmenej dovoľte mi ho v krátkosti uviesť aj v tejto sekcii.

1.5.1 Ruby on Rails

Rails alebo tiež Ruby on Rails je webový framework založený na architektúre MVC - *Model-View-Controller* a napísaný v jazyku Ruby. Architektúra MVC odlišuje tri časti aplikácie a to tzv. modely, ktoré zodpovedajú za prístup a ukladanie dát, „obaľujú“ dáta o k im špecifickú funkcionalitu, tzv. kontroléry alebo radiče, ktorá prijímajú požiadavky od užívateľa a riadia zobrazenie dát na výstupe a „views“, alebo tzv. pohľady, ktorá plnia funkciu prezentačnej vrstvy zobrazovaných dát. Tak ako o tom píše S. Ruby a kol [6], Rails sa pridržiava programátorských vzorov ako napr. DRY (*Don't repeat yourself*), čo je princíp, ktorý hlása použitie abstrakcií a zamedzenie akéhokoľvek opakovania logiky kódu v rámci aplikácie alebo princíp CoC (*Convention over Configuration*). Rails takto preferuje predvolené konvenčné vzory a nastavenia a tak sa snaží ušetriť počet rozhodnutí, ktoré programátor musí vykonať poskytnutím existujúcich riešení.

Po inštalácii programovacieho jazyka Ruby a frameworku Rails na našom lokálnom počítači, inicializujeme novú aplikáciu príkazom `rails new numalgo`. Okamžite do nej pridávame balíček `haml` slúžiaci na rýchlejšie a efektívnejšie písanie HTML šablón. Významnu časť našej aplikácie tvorí **frontendový framework React**. Pod pojmom *frontend* rozumieme tú časť našej aplikácie, ktorá je zodpovedná za prezentačnú vrstvu dát určenú pre koncového užívateľa aplikácie. Okrem samotných HTML šablón ju tvorí aj štýlopis CSS definujúci vizuálne štýly HTML šablón a programová vrstva popisujúca interakciu s aplikáciou napísaná v jazyku Javascript. Práve túto interaktívnu vrstvu, pre lepšiu formu organizácie, budeme popisovať vo frameworku React. Ten inštalujeme do frameworku Rails skrz balíčky `react-rails` a `webpacker` nasledujúc postup inštalácie na oficiálnej stránke balíčka `https://github.com/reactjs/react-rails`. Framework React funguje na báze tzv. komponentov. Komponentom môžeme chápať vlastný HTML element, značku obohatenú o nami definovaný štýl a programovateľnú interakciu. Práve takéto komponenty budeme tvoriť pri tvorbe našej aplikácie - vizualizačného prostredia.

Pridaním `root 'welcome#index'` do súboru `config/routes.rb` mapujeme všetky HTTP požiadavky vyslané na koreňovú adresu `/` našej aplikácie k spracovaniu metóde `index` kontroléru `WelcomeController`. Ten musíme takisto vytvoriť v priečinku `app/controllers`, pričom bude ako trieda dediť od rodičov-

ského kontroléra `ApplicationController`. Do metódy `index` nemusíme nič písať. Metóda automaticky volá funkciu `render`, ktorá očakáva HTML šablónu na ceste `app/views/welcome/index.html.haml`. Tú tiež vytvárame a umiestňujeme do nej volanie na vykreslenie Reactovského komponentu s názvom `App - = react_component "App", {}`.

Ak chceme okamžite testovať funkčnosť našej aplikácie na koreňovej adrese aplikácie, stačí nám vytvoriť na ceste `javascripts/components/App.jsx` jednoduchý komponent `App`.

```
class App extends React.Component {
  render() {
    return <div>Hello App!</div>
  }
}
```

Komponenty sú písané v **ECMAScript 6**, čo je verzia špecifikácie pre jazyk Javascript, ktorá je postupne implementovaná naprieč webovými prehliadačmi. Tá je ešte rozšírená syntaxou frameworku React s názvom JSX, ktorá nám umožnuje písať HTML štruktúry mimo úvodzoviek textových reťazcov. K bezproblémovej funkčnosti tejto exotickej kombinácie využíva balíček `react-rails` kompilér **Babel**, ktorý špecifikáciu kompiluje do jazyka Javascript bežiaceho vo všetkých verziách a druhoch prehliadačov.

Vráťme sa k nášmu cieľu integrovať do tohoto prostredia vizualizáciu algoritmu simulovaného žíhania. Komponent `App` je komponent obaľujúci celú klientsku funkcionalitu našej aplikácie a jeho úlohou je odoslať XML zápis algoritmu, obdržať jeho javascriptový kompilovaný kód, ten následne exekuovať a vygenerovať tak snímky animácie algoritmu určené k prehrávaniu. O odoslaní XML zápisu algoritmu hovoríme preto, pretože ho budeme parsovať v backendovej časti aplikácie, to znamená na strane servra - v programovacom jazyku Ruby. Uvažujeme priebežne nami uvedenú nasledujúcu základnú podobu komponentu bez odosielania XML zápisu algoritmu, ale už s obdržaným javascriptovým kódom pripraveným k exekúcii. Tú vykonávame stlačením tlačidla `Submit` XML editora.

```
class App extends React.Component {
  constructor() {
    super();

    this.state = {
      frame: 0,
      frames: [{ canvases: {} }]
    }
  }

  submit() {
    let frames = [];

    // Generate random data
    var data = [];
    for (var j = 0; j < 100; j++) { data[j] = Math.random(); }
```

```
    // INSERTED:
    frames.push({ canvases: { 1: { data: data } } });

    // Variables
    var t = 1.0, t_min = 0.00001, alpha = 0.9, x,
        y = data[Math.floor(Math.random() * (data.length - 1))];

    // Ak teplota este nedosiahla minimum
    while (t > t_min) {
      var i = 1;
      while (i <= 100) {
        new_x = Math.floor(Math.random() * (data.length - 1));
        new_y = data[new_x];

        // Acceptance probability (switch ys to change max to min)
        p = Math.E * (new_y - y) / t;
        if (p > Math.random()) {
          x = new_x; y = new_y;

          // INSERTED:
          frames.push({ canvases: { 1: { guidelineX: x } } });
        }

        i++;
      }

      // Cool the temperature
      t = t * alpha;
    }

    this.setState({ frames: frames, frame: 0 });
  }

  render() {
    const frames = this.state.frames,
          frame = this.state.frame;

    return <div>
            { Object.keys(frames[frame].canvases).map((key) =>
              <Graph key={key} {...frames[frame].canvases[key]} />) }
            <XML onSubmit={this.submit} />
           </div>
  }
}
```

V komponente `App` prichádzame prvýkrát do styku s **konceptom stavu**. Každý nami definovaný komponent framework Reacta môže mať špeciálny atribút `this.state`. Tento atribút meníme metódou `setState()` a framework React pri každej takejto zmene znova obnoví vykresľovanú HTML štruktúru komponentu, v ktorej atribúty stavu môžeme používať a ich zmeny sa takto do danej HTML štruktúry okamžite premietnu. V stave ukladáme hodnotu indexu momentálneho

snímku a pole snímku animácie. Tri bodky v metóde vykresľovania značia rozbalenie všetkých kľúčov premennej ako atribútov HTML elementu, resp. komponentu `Graph` v tomto prípade. Pre každý ďalší objekt `canvas` animácie, zobrazujeme komponent `Graph` vykresľujúci jemu predané dáta. V algoritme dosadzujeme do poľa `frames` jednotlivé snímky, ktoré po úspešnom zbehnutí algoritmu sú priradené do premennej stavu komponentu. Po tomto priradení sa znova vykreslia všetky komponenty `Graph`. Problémom je, že komponenty `Graph` si nepamätajú svoj predošlý stav a tak pri nastavení pozície vodiacej čiary nevykreslia žiadne dáta, pretože žiadne dáta ani neobdržia. To znamená, že každý snímok musí byť definovaný ako **snímok predošlý zjednotený so zmenou**. Umiestnene do kódu:

```
let frames = [{ canvases: {} }];
...
frames.push(merge(
  frames[frames.length-1],
  { canvases: { 1: { data: data } } }
));
...
frames.push(merge(
  frames[frames.length-1],
  { canvases: { 1: { guidelineX: x } } }
));
```

Pri zjednocovaní objektov predošlého snímku a nového snímku používame funkciu `merge` importovanú z knižnice `deepmerge`, ktorá slúži na hĺbkové zlučovanie objektov. Takto zapísaný algoritmus už funguje správnym spôsobom. Je potrebné spomenúť, že používame dva komponenty, ktoré sme ešte nespomenuli. Komponent `Graph` obaľuje element `canvas` použitý pri statickej vizualizácii algoritmu simulovaného žíhania a je praktickou kópiou funkcií použitých k vykresleniu grafu. O jednotlivých funkciách je možné sa dozvedieť viac v programátorskej dokumentácii. Komponent `XML` definuje prostredie XML editora. V nami definovanom komponente `XML` zobrazujeme komponent `AceEditor` balíčka `react-ace`. Ace Editor je rozhranie webového editora pre písanie a úpravu kódu napísané v jazyku Javascript a vsadzovateľné do webových stránok.

Pre dokončenie prípravy nášho vizualizačného prostredia chceme demonštrovať **úplny užívateľský proces** používania našej aplikácie aj s odoslaním HTTP požiadavku pre kompiláciu XML zápisu algoritmu. Vytvárame teda `AlgorithmsController` v priečinku `app/controllers` a v ňom podľa štandardov a konvencie frameworku Rails metódu `create` pôvodne určenú pre „vytvorenie“ algoritmu, ktorú my budeme chápať ako metódu určenú pre interpretáciu algoritmu písaného v našom XML jazyku. Radič bude vracať odpoveď vo formáte JSON, čo je Javascriptový formát pre objektový zápis, pomocou metódy `render` kde `algorithm` je textový reťazec javascriptového kódu nášho algoritmu simulovaného žíhania.

```
render json: { algorithm: algorithm }
```

Okrem toho potrebujeme tiež mapovať adresu HTTP požiadavky na danú metódu. Dosadením „railsovského“ konštruktu `resources :algorithms` do súboru `config/routes.rb` zabezpečujeme vytvorenie odpovedajúcej cesty na adrese `/algorithms`. Bližší popis fungovania tohoto konštruktu nájdeme v programátorskej dokumentácii. To čo nám stačí v tejto chvíli vedieť je, že na adresu `/algorithms` akcie `create` vytvorenej cesty môžeme posielať náš algoritmus písaný v XML ako súčasť **HTTP požiadavky** metódy `POST`. Do metódy `submit` komponenty `App` dosadíme kód, ktorý miesto XML štruktúry bude priebežne posielať prázdny reťazec, ničmenej ako odpoveď budeme v parametri `algorithm` očakávať znenie algoritmu zapísané v jazyku Javascript, pripravené k exekúcii. Exekúciu po úspešnom uskutočnení požiadavky vykonávame príkazom `eval`. K uskutočneniu HTTP požiadavky metódy `POST` používame knižnicu **axios**.

```
axios.post('/algorithms', {
  algorithm: { xml: "" }
}).then(function (response) {
  eval(response.data.algorithm)
});
```

Takto definované spracovanie odpovede požiadavky však nerobí absolútne nič. Ladením zisťujeme, že funkcia `eval` síce exekuuje daný kód ale robí to v inom kontexte. Skutočná hodnota premennej `this` pri zmene stavu komponenty `App` sa odlišuje od jej želanej hodnoty. Tento problém riešime tak, že pred **exekúciou javascriptového kódu** algoritmu definujeme premennú `algorithm` a do nej priradzujeme generovaný javascriptový kód algoritmu. Podobne v prostredí `eval` Javascript nepozná importovanú knižnicu `merge`. Priradzujeme ju teda do premennej s názvom `m`, na ktorú sa v rámci algoritmu, už môžeme odkazovať nasledovne:

```
axios.post('/algorithms', {
  algorithm: { xml: "" }
}).then(function (response) {
  let algorithm, m = merge;
  eval(response.data.algorithm)
  //response.data.algorithm == "algorithm = function() { let frames...
});
```

2. Návrh XML jazyka

Predchádzajúcu kapitolu sme ukončili demonštráciou vizualizácie vybraného algoritmu v našom aplikačnom prostredí a v tej nasledujúcej si konečne popíšeme proces tvorby vlastného XML jazyka, ktorý naša aplikácia bude kompilovať do jazyka Javascript. Predpokladom nášho XML jazyka je, že súčasťou výsledného kompilovaného kódu môžu byť aj príkazy určené k vizualizácii premenných alebo stavov algoritmu. Pojem XML jazyka bol v práci použitý niekoľko krát, ale doteraz sme si presne nepovedali čo pojem XML znamená a preto je pred návrhom XML jazyka vhodné si tento pojem v krátkosti objasniť.

2.1 XML Jazyk

XML v originále nazývaný eXtensible Markup Language je štandard vytvorený organizáciou World Wide Web Consortium, ktorej úlohou je vytvárať a upravovať webové štandardy. Prvá špecifikácia XML bola vydaná ako odporúčanie v roku 1998 [7], následne bola doupravená v roku 2000. Ako popisuje E. Castro [8], cieľom XML bolo vytvoriť **univerzálny formát pre výmenu dát** na webovej sieti ako alternatívu k HTML, doterajšiemu jazyku pre tvorbu webových dokumentov, ktorého jednoduchosť bola vnímaná ako obmedzenie a orientácia jeho značiek predovšetkým na formátovanie obsahu zťažovala možnosť nového použitia obsahu v inom kontexte. Tieto výzvy chcel XML jazyk vyriešiť rozšíriteľnosťou značiek a tak umožniť programátorom vytvárať vlastné XML jazyky pre výmenu dát.

Interpretácia značky v XML na rozdiel od HTML nemala význam formátovania obsahu ale niesla **sémantickú informáciu** o význame obsahu vnútri danej značky, popisuje tak obsah a nie jeho prezentáciu ako v prípade HTML.

2.1.1 Značka

Termínom značka označujeme **lexikálnu jednotku XML jazyka**, tradične uzavretú v špicatých zátvorkách. V práci používame aj z angličtiny prevzatý termín element označujúci to isté. Pre ilustráciu uvádzame názorný syntaktický príklad toho, čo nazývame XML značkou.

```
<znacka></znacka>
```

Technické odlišnosti jazyka XML od HTML zhrňujú S. Abiteboul a kol. v [9] do troch bodov:

1. Používateľ jazyka môže definovať nové značky
2. Značky do seba môžu byť vnorované ľubovoľne mnohokrát
3. XML dokument môže obsahovať opis jeho gramatiky

Pričom pri opise gramatiky sa autori odkazujú na tzv. formát DTD (*Document Type Definition*), ktorý definuje povolené a povinné atribúty jednotlivých XML značiek, povolené štruktúry vnorenia a pod.

2.2 Návrh programovacieho jazyka

Pri tvorbe samotného programovacieho jazyka využívajúceho syntaxe XML, budeme priamo a nepriamo používať isté pojmy a princípy využívané pri procese návrhu a implementácie programovacieho jazyka. Tieto pojmy si objasníme v nasledujúcej sekcii. Pri ich objasňovaní a snahe o lepšie pochopenie princípov programovacích jazykov sa nechávame konceptuálne viesť prácou C. K. Loudena a A. K. Lamberta [10].

Programovacím jazykom môžeme rozumieť pravidlá zápisu toho čo od počítača očakávame aby vykonal. Definícia takéhoto programovacieho jazyka môže byť rozdelená na dve časti a to jeho **syntax a sémantiku**. K popisu syntaxe využívame formálny model zápisu jazyka a syntax môžeme chápať ako gramatiku programovacieho jazyka, analogicky ku gramatike reálnych jazykov. Ide v princípe o popis ako rôzne časti jazyka môžu byť usporiadané a aké štruktúry môžu tvoriť. Formálny model jazyka je matematický model jazyka popisujúci jeho štruktúru.

Dôležitou súčasťou kompilácie alebo interpretácie programovacieho jazyka je jeho syntaktická analýza, ktorá prijíma program zapísaný v danom programovacom jazyku ako textový reťazec - sekvenciu znakov a jej výstupom je štruktúra jednotlivých prvkov daného jazyka. Tento proces tvorby takejto štruktúry sa nazýva aj **parsing**. V procese **syntaktickej analýzy** sú v textovom reťazci nášho programu nájdene lexikálne jednotky alebo slová, ktoré sú tiež nazývané *tokenmi*. V našom prípade XML jazyka budeme náš zapísaný program resp. algoritmus parsovať tzv. SAX parserom, určeným k parsovaniu XML dokumentov. Tokeny sú v našom prípade lexikálne jednotky XML jazyka a to napr. názvy elementov resp. elementy samotné, ich atribúty, ich textový obsah a pod.

Druhou časťou spracovania alebo kompilácie programu je sémantická analýza. Zatiaľ čo pri syntaktickej analýze rozpoznávame štruktúru lexikálnych jednotiek jazyka, pri sémantickej analýze dávame týmto lexikálnym jednotkám význam. Je potrebné si uvedomiť, že syntax sám o sebe netvorí význam ale význam lexikálnych jednotiek programu súvisí z jeho syntaxou. Pri sémantickej analýze si kompilátor udržuje tabuľku symbolov alebo identifikátorov, ktoré v programe boli použité a pamätá a zapisuje si ich atribúty. Atribúty takýchto lexikálnych jednotiek sú v našom prípade atribúty samotných XML značiek a miesto vytvárania tabuľky ich priradzujeme ako inštančné premenné k inštanciám jednotlivých elementov alebo teda lexikálnych jednotiek.

2.2.1 Abstrakcie programovacieho jazyka

Pre lepší popis a pochopenie programovacieho jazyka rozlišujeme dva druhy abstrakcií a to **dátovú abstrakciu** a **kontrolnú alebo riadiacu abstrakciu**. Zatiaľ čo dátová abstrakcia ma za úlohu rámcovať vlastnosti a správanie dát, kontrolná riadi a vetví činnosť programu. Abstrakcie môžeme deliť aj vertikálnym spôsobom a to na základe úrovne komplexity do základných, štrukturovaných a jednotkových abstrakcií. Základné abstrakcie sú určené pre prácu s konkrétnou individuálnou hodnotou a jej reprezentáciou v pamäti počítača, zatiaľ čo štrukturované pracujú s väčším množstvom takýchto hodnôt. Napr. v prípade individuálnej celočíselnej premennej hovoríme a základnej dátovej abstrakcii, zatiaľ

čo v prípade pola celočíselných premenných hovoríme o štruktúrovanej dátovej abstrakcii. Jednotkové dátové abstrakcie združujú súvisiace dáta, aj viacerých štrukturovaných dátových abstrakcií a operácie nad takýmito dátami. Porovnanie základných abstrakcií a jednotkových abstrakcií môžeme vnímať analogicky porovnaním dátových a abstraktných dátových typov.

Podobne v prípade riadiacich, kontrolných abstrakcií chápeme základné riadiace abstrakcie ako odkazy na individuálne strojové inštrukcie, alebo nanajvýš sekvenciu pár strojových inštrukcií. Operáciu sčítania a uloženia hodnoty do premennej môžeme chápať ako základnú riadiacu abstrakciu. Pri štrukturovaných riadiacich abstrakciách pracujeme s vetvami programu. Pod pojmom vetva programu rozumieme časť inštrukcií uložených oddelene na inom v mieste v pämati, ku ktorým sa náš program odkazuje. K štruktúrovaným riadiacim abstrakciám zaradzujeme napr. kontrolné abstrakcie cyklu, procedúry, funkcie atď. Ako jednotkovú riadiacu abstrakciu rozumieme samostatnú časť programu abstrahujúcu logicky prepojené a súvisiace operácie inej časti programu.

V našom jazyku budeme potrebovať odlíšiť XML značky určujúce dátovú abstrakciu a XML značky určujúce riadiacu abstrakciu. Toto odlíšenie je v našom jazyku vyjadrené **veľkosťou prvého písmena** XML značky nášho jazyka.

2.2.2 Paradigmy programovacích jazykov

Samotné odlíšenie značiek alebo, ako sme už pomenovali, jednotlivých abstrakcií jazyka ale k vytvoreniu programovacieho jazyka nestačí. Existujú rôzne spôsoby usporiadania štruktúry nášho XML jazyka, čo nakoniec načrtneme aj pri postupe jeho návrhu. Tieto spôsoby môžeme charakterizovať ich dominantnými programovacími paradigmami, ktoré ich ovplyvňujú. Pod programovacou paradigmou rozumemieme princíp fungovania programovacieho jazyka, ktorý ho dominantným spôsobom **odlišuje od iných** programovacích jazykov. Paradigmy programovacích jazykov boli silno ovplyvnené počítačovou architektúrou a to von Neumannovským modelom, ktorý pracuje s jedinou procesorovou jednotkou. Tá vykonáva inštrukcie sekvenčne a operuje s hodnotami uloženými v pamäti. Toto sekvenčné spracovávanie inštrukcií vytvorilo vhodný predpoklad pre implementáciu tzv. imperatívnych programovacích jazykov. Okrem imperatívnych existujú aj iné druhy jazykov.

Imperatívny

V imperatívnom programovacom jazyku je program tvorený **sekvenciou inštrukcií**, príkazov resp. imperatívov. Imperatívny programovací jazyk charakterizujú Louden a Lambert [10] troma vlastnosťami:

1. Sekvenčné vykonávanie inštrukcií
2. Použitie premenných, ktoré sú odkazmi na miesta v pamäti
3. Operácia priradenia, ktorá mení hodnotu premenných

Tieto vlastnosti budeme schopný spozorovať aj pri tvorbe nášho XML jazyka, ktorý je silne ovplyvnený jazykom Javascriptom, do ktorého je kompilovaný. Ten sám o sebe je imperatívnym programovacím jazykom, i keď jeho nová špecifikácie ECMAScript ho posúvajú bližšie ku paradigmu funkcionálneho jazyka.

Funkcionálny

Funkcionálny programovací jazyk sa vyhýba konceptu stavu programu a teda aj deklarácie premenných. Funkcionálna paradigma programovacích jazykov chápe chod programu ako vyhodnocovanie matematických funkcií alebo funkcií vôbec. Funkcionálne programovacie jazyky boli ovplyvnené tzv. **lambda kalkulom**, čo je formálny logický systém popisujúci výpočty pomocou **abstrakcie funkcie**. Ovplyvnenie funkcionálnymi programovacími jazykmi je v našom jazyku cítiť napr. pri XML značke `Function` nášho jazyka, ktorá definuje funkciu nepodporujúcu v nej existujúce lokálne premenné.

Logický

V logickom programovaní je exekúciou programu proces dokázania teórie definovanej sadou podmienok a pravidiel. Výpočet je teda riadený ako logický **proces dokazovania teórie** realizovaný odvodzovaním dôsledkov pri vytváraní potencionálneho riešenia - dôkazu programu. Toto odvodzenie sa dá predstaviť aj ako strom vetviacich sa a navzájom vyplývajúcich logických rozhodnutí.

Objektovo-orientovaný

Jednou z najrozšírenejších programovacích paradigiem, je paradigma objektovo-orientovaného programovacieho jazyka. V nej sú imperatívne sekvencie inštrukcií triedené a organizované do tzv. objektov, ktoré môžu byť vytvárané a používané na mnoho miestach jazyka. Objekty reprezentujú **abstrakciu reálnych objektov**, s ktorými prichádzame do styku v reálnom živote. Paradigma objektovo-orientovaného programovacieho jazyka je riešením pre rozsiahle imperatívne programy, pozostávajúce z dlhých zoznamov inštrukcií. Výstupom objektovo-orientovaného programovacieho jazyka je program, ktorého jednotlivé funkcie sú oddelené v objektoch, ktoré si zachovávajú svoju myšlienkovú konzistenciu. Inými slovami, objekt pracuje s partikulárnou sadou príkazov programu, ktorých spája buď účel, dáta, s ktorými objekt pracuje alebo iný spoločný znak.

2.2.3 Kritéria návrhu programovacieho jazyka

Popísali sme si abstrakcie programovacieho jazyka a rôzne druhy programovacích paradigiem, čo ale robí programovací jazyk dobrým programovacím jazykom? Pri návrhu programovacieho jazyka potrebujeme zvážiť kritéria jeho kvality a naše ciele. Je zložité hodnotiť programovací jazyk na základe abstraktných kritérii čitateľnosti alebo kontroly komplexnosti, preto Louden a Lambert [10] uvádzajú ako 3 základné parametre úspechu programovacieho jazyka nasledovné body:

1. Programovací jazyk dosahuje **ciele** jeho tvorcov 2. Programovací jazyk má široké **využitie** v oblasti aplikácií 3. Programovací jazyk je použitý ako **model** pre iné jazyky, ktoré sú sami o sebe úspešné

V nasledujúcich sekciách sa pokúsime naznačiť na čo by sa prakticky mali prihliadať pri návrhu programovacieho jazyka.

Efektívnosť

Existujú dva pohľady na efektívnosť programovacieho jazyka. Efektívnosťou môžeme rozumieť **krátky kompilačný čas**, ktorý vedie k vytvoreniu spustiteľného programu. Pre porovnanie, pri statickej typovej kontrole sú typy premenných deklarované priamo v programe (napr. Java) a preto nemusia byť overované v behovom prostredí. Zatiaľ čo pri dynamickej typovej kontrole, pri ktorej nedefinujeme typy používaných premenných (napr. Python), potrebuje behové prostredie overiť dátový typ premennej pred vykonaním operácii. Spustenie programu je tak pri dynamickej typovej kontrole pomalšie a menej efektívne.

Iný pohľad na efektívnosť programovacieho jazyka je **programátorská efektívnosť** a to ako efektívne dokáže človek čítať a programovať v danom jazyku. Tento pohľad úzko súvisí aj so schopnosťou jazyka jednoducho vyjadriť a popísať komplexné procesy a štruktúry a je v príkrom protiklade s predošlým výkladom efektívnosti. S dostupnou výpočetnou kapacitou je práve akcent na programátorskú efektívnosť presadzovaný pri dnešných požiadavkách na programovacie jazyky a ich spôsobe využitia, ničmenej stále existujú príklady využitia programovacích jazykov a to napríklad v medicínskom, priemyselnom alebo armádnom použití kde je vyžadovaná bezpečnosť, pri ktorých efektívnosť a spoľahlivosť kompilácie je rozhodujúca a tak nadradená efektívnosti programátora.

Rovnomernosť

Princíp rovnomernosti alebo regularity opisuje ako dobre sú funkcie jazyka integrované. Pri používaní pravidelných vzorov v jazyku, obmedzujeme výskyt nezvyčajných jazykových konštruktov, ktoré môžu byť mätúce a spôsobovať neočakávané druhy interakcií medzi objektami jazyka. Cieľ rovnomernosti sa dá popísať aj ako nasledovanie **princípu tzv. najmenšieho prekvapenia** pre vývojára. Rovnomernosť programovacieho jazyka, môžeme rozdeliť do troch kategórii a to všeobecnosti, ortogonálnosti a uniformovanosti.

Všeobecnosť Dobrá všeobecnosť jazyka implikuje obmedzenie výskytu špeciálnych prípadov alebo konštruktov pri použití jazyka. Dá sa dosiahnuť **zoskupovaním súvisiacich konštruktov** do jediného všeobecne platného konštruktu. Príkladom špeciálneho prípadu, ktorým sa v programovacom jazyku chceme vyhnúť, je napr. neexistujúca možnosť porovnania dvoch polí v jazyku C inak dostupným operátorom `==`.

Ortogonálnosť Ortogonálnosť je matematický pojem a označuje napr. kolmosť dvoch priamiek, ale používa sa vo všeobecnejšom význame. V programovacom jazyku môže znamenať ako dobre sú jednotlivé **funkčné bloky jazyka** od seba **oddelené** a **nezávislé** a ako dokážu byť **zmysluplne kombinované** bez akýchkoľvek obmedzení. Takýmto obmedzením je napr. fakt, že v jazyku C nemôžeme v návratovej hodnote funkcie vracať objekt typu poľa.

Uniformovanosť Uniformovanosť je znakom, ktorý popisuje ako podobné konštrukty v jazyku fungujú podobným spôsobom. V našom prípade XML jazyka, môžeme hovoriť, že XML značky `<set />` a `<Var />` svojím spôsobom zachovávajú uniformovanosť spôsobu priradenia hodnôt do premenných. Uniformovanosť

sa zaoberá hlavne **konzistenciou** toho ako jazyk vyzerá. Od tých **konštruktov**, ktoré vyzerajú podobne sa očakáva podobná funkčnosť a naopak. Príkladom zanedbanej uniformovanosti je výskyt dvojbodky po deklarácii triedy v jazyku C++. Po deklarácii funkcie sa ale dvojbodka nesmie nachádzať.

Bezpečnosť

Po rovnomernosti je jedným z kritérií dobrého programovacieho jazyka aj bezpečnosť. Tá popisuje spoľahlivosť jazyka a jej cieľom je **znižovať pravdepodobnosť výskytu chýb** pri používaní jazyka. To dosahuje vytvorením dobrých nástrojov objavenia chýb a **ladenia programov**. Jedným z takýchto znakov bezpečného jazyka je aj tzv. **silné typovanie**. Silno typovaný jazyk neumožňuje prístup k premenným zmenou ich typov. Inými slovami, v silno typovanom jazyku, nemôžeme jednoducho pretypovať premenné a pristupovať k nim ako k iným typom premenných. Typ premennej je silno zviazaný so samotnou premennou. Nedostatkom bezpečnosti programovacieho jazyka môže byť tiež aj zanedbanie upozorniť programátora na semántické chyby v jeho programe. Jednou z takýchto chýb môže byť napríklad pristupovanie k premennej poľa prostredníctvom indexu, ktorý sa vyčleňuje z rozsahu indexov daného poľa.

Rozšíriteľnosť

Rozšíriteľnosť programovacieho jazyka popisuje ako dobre jazyk umožňuje jeho používateľovi pridávať k nemu **vlastnú definovanú funkcionalitu**. Takouto novou funkcionalitou môže byť deklarácie nových typov premenných, nových operácii ako funkcií alebo procedúr, nových objektov a pod. Väčšina moderných jazykov sa dnes prispôsobuje spôsobom ich používania a vydáva nové verzie inšpirované práve riešeniami vývojarov v danom jazyku.

2.3 Algoritmus simulovaného žíhania v XML

Po teoretickej príprave sa môžeme pustiť do tvorby XML jazyka. V predošlej kapitole sa nám podarilo napísať javascriptový kód algoritmu simulovaného žíhania, ktorého exekúcia nám pripraví snímky určené k jeho animácii. Pre pripomenutie uvádzame jeho kompletné znenie:

```
let frames = [{ canvases: {} }];

// Generate random data
var data = [];
for (var j = 0; j < 100; j++) { data[j] = Math.random(); }

// INSERTED:
frames.push(m(
  frames[frames.length-1],
  { canvases: { 1: { data: data } } }
));

// Variables
```

```
var t = 1.0, t_min = 0.00001, alpha = 0.9, x,
    y = data[Math.floor(Math.random() * (data.length - 1))];

// Ak teplota este nedosiahla minimum
while (t > t_min) {
  var i = 1;
  while (i <= 100) {
    new_x = Math.floor(Math.random() * (data.length - 1));
    new_y = data[new_x];

    // Acceptance probability (switch ys to change max to min)
    p = Math.E * (new_y - y) / t;
    if (p > Math.random()) {
      x = new_x; y = new_y;

      // INSERTED:
      frames.push(m(
        frames[frames.length-1],
        { canvases: { 1: { guidelineX: x } } }
      ));
    }

    i++;
  }

  // Cool the temperature
  t = t * alpha;
}
```

V nasledujúcej sekcii sa budeme snažiť postupne nahradiť jednotlivé časti javascriptového kódu tohoto algoritmu XML značkami, ktoré sa doň budú neskôr kompilovať. Už pri letmom pohľade na javascriptový kód nášho algoritmu, môžeme jednoducho čítať, čo náš XML jazyk bude potrebovať. Prvý riadok definície poľa snímkov bude v prípade všetkých kompilovaných algoritmov rovnaký. Následne generujeme pole zvolenej dĺžky, ktoré ihneď vypĺňame náhodnými hodnotami. K XML verzii deklarácie takéhoto poľa potrebujeme poznať identifikátor poľa, jeho dĺžku a to či ho vypĺňame náhodnými hodnotami. Prirodzene sa do našej mysle dostáva nasledujúca predstava zodpovedajúcej XML značky.

```
<Array length="200" name="data" random="true" />
```

Atribút `name`, môže byť univerzálne využívaný ako identifikátor aj pri iných značkách nášho XML jazyka. Následne vkladáme nový snimok do poľa snímkov, ktorého cieľom je vykresliť bodové dáta na plátne grafu. Tu je potrebné povedať, že hoci pri implementácii vizualizačného prostredia sme použili termín „canvas“ odvodený z HTML elementu `<canvas></canvas>`, pri definovaní nášho XML jazyka budeme používať sémanticky príznačnejší termín `Graph`. Potrebujeme sa tiež odkazovať na premennú, ktorá určuje pole hodnôt, ktoré chceme vykreslovať. Na to použijeme atribút `ref`, ktorý sa už raz používa v XML jazyku na odkazovanie na XML značky v dokumente.

```
<Graph name="1" ref="data" />
```

Pri deklarácii a definícii premenných sa budeme pridŕžať už raz nami vytvorenej konvencie atribútu `name` ako atribútu nesúceho názov **identifikátora premennej**. Jej hodnotu budeme zapisovať v atribúte `value`. Tie deklarované premenné, ktoré ostávajú bez priradenej hodnoty, atribút `value` nemajú.

```
<Var name="alpha" value="0.9" />
```

Tu je potrebné spomenúť, že pri deklarácii hodnôt matematických výrazov ako v prípade premennej `y` sa nám do hláv vtláčala myšlienka vytvorenia značkových náhrad pre Javascriptové vstavané premenné. Toto je hrubé načrtnutie jednej z neohrabaných alternatív ako by to mohlo vyzerať:

```
<Var name="y">
  <get name="data">
    <Math.floor>
      <Math.random /> * (<data.length /> - 1)
    </Math.floor>
  </get>
</Var>
```

a pozmenená, čistejšia verzia:

```
<Var name="y">
  <get name="data" type="index">
    <func>
      <get name="Math" type="prop">floor</get>
      <prod>
        <func>
          <get name="Math type="prop">random</get>
        </func>
        <sub>
          <get name="data" type="prop">length</get>
          <num val="1" />
        </sub>
      </prod>
    </func>
  </get>
</Var>
```

Prirodzene experimentálne zaujímavá verzia XML štruktúry pre zápis `var y = data[Math.floor(Math.random() * (data.length - 1))];` sa nám zdala **mierne nepraktická**. Dovoľte mi len námatkovo popísať ideu takejto verzie nášho jazyka. V štruktúre vystupuje element `get`, ktorý získava buď index alebo atribút nejakého objektu a číslo daného indexu alebo atribútu je definované obsahom elementu. Element `func` prvý dcérsky element interpretuje ako funkciu,

ktorú následne volá, pričom každý jeho ďalší dcérsky element je ďalším argumentom volanej funkcie. Značky `prod` a `sub` transformujú matematické operácie do tzv. prefixovej notácie, v ktorej vystupuje najprv operátor, až následne jeho argumenty.

Vzhľadom pre motiváciu vytvoriť efektívny XML jazyk a pre naše zameranie algoritmy hlavne vizualizovať, sme ostali v bezpečných vodách prvotného návrhu:

```
<Var name="y"
     value="data[Math.floor(Math.random() * (data.length - 1))]" />
```

Z nášho algoritmu simulovaného žíhania jednoznačne vyplýva, že budeme potrebovať značku **cyklu**. Javascript pozná dve kontrolné štruktúry pre tvorbu cyklov a to `for` a `while`. My si pre jednoduchosť nášho jazyka vystačíme s cyklom `while`, ktorý je o niečo jednoduchšie zapísať v jazyku XML.

```
<while cond="t > t_min">
```

Atribút `cond` určuje podmienku behu cyklu a jeho obsah je prevzatý z javascriptovej verzie algoritmu simulovaného žíhania, ktorý sme si pripomenuli na začiatku sekcie. Hodnota atribútu `cond`, môže byť akýkoľvek javascriptový kód, ktorý vracia booleovskú hodnotu `false` alebo `true`. Tento atribút môžeme ihneď využiť aj pri definovaní značky pre podmienku. Tú zatiaľ nebudeme komplikovať alternatívou `else`, ktorú je možné definovať podmienkou opačnej pravdivostnej hodnoty.

```
<if cond="p > Math.random()">
```

Element `Var` slúžil na deklaráciu premenných a predpokladáme, že pri kompilácii takéhoto elementu bude pred identifikátorom premennej, vo výslednom javascriptovom kóde, kľúčové slovo `var` určujúce deklaráciu novej premennej. Pri priradení hodnoty do už deklarovanej premennej toto kľúčové slovo ale nepoužívame a preto potrebujeme definovať novú XML značku priradenia - značku `set`.

```
<set name="p" value="Math.E * (new_y - y) / t" />
```

XML značka `set` funguje na podobnom princípe ako značka `Var` a do premennej s názvom v atribúte `name` ukladá hodnotu z atribútu `value`. To posledné čo chceme v našom algoritme docieliť je nastaviť **vykreslenie vodiacej čiary** určujúcej momentálnu hodnota maxima. Potrebujeme nielen nastaviť jej novú hodnotu, ale tiež odkazovať na už raz definovaný graf, v ktorom túto vodiacu čiaru chceme vykreslovať. To dosiahneme využitím atribútu `name`. Referenciu na vykresľované dáta, nakoľko sme ju už raz definovali, nemusíme znova použiť.

```
<Graph name="1" markerX="x" />
```

Výsledná podoba nášho algoritmu transformovaného do jazyka XML vyzerá takto:

```
<Array length="200" name="data" random="true" />

<Graph name="1" ref="data" />

<Var name="alpha" value="0.9" />
<Var name="new_x" />
<Var name="new_y" />
<Var name="p" />
<Var name="t" value="1.0" />
<Var name="t_min" value="0.00001" />
<Var name="x" />
<Var name="y"
     value="data[Math.floor(Math.random() * (data.length - 1))]" />

<while cond="t > t_min">
  <Var name="i" value="0" />
  <while cond="i < 100">
    <set name="new_x"
         value="Math.floor(Math.random() * (data.length - 1))" />
    <set name="new_y" value="data[new_x]" />
    <set name="p" value="Math.E * (new_y - y) / t" />

    <if cond="p > Math.random()">
      <set name="x" value="new_x" />
      <set name="y" value="new_y" />

      <Graph name="1" markerX="x" />
    </if>

    <set name="i" value="i + 1" />
  </while>
  <set name="t" value="t * alpha" />
</while>
```

2.4 Vytvorenie parseru

Máme algoritmus zapísaný v jazyku XML a jeho kompilovaný kód v Javascripte. Máme vizualizačné prostredie, ktoré odosiela algoritmus písaný v jazyku XML a v odpovedi získava jeho javascriptový kód. To čo potrebujeme je vytvoriť parser, ktorý nám daný XML zápis do želaného Javascriptu bude kompilovať. Pri kompilácii jazyka XML používame tzv. SAX parser.

2.4.1 SAX Parser

SAX je aplikačné rozhranie používané k prístupu k XML dokumentom. Jeho skratka značí *Simple API for XML*, čo v preklade znamená *Jednoduché aplikačné rozhranie pre XML*. SAX je štandardom k parsovaniu XML dát. Tak ako popisuje Abiteboul a kol [9], jeho parser postupne prechádza textový reťazec XML doku-

mentu a **detekuje jednotlivé elementy** alebo značky a ich prípadný textový obsah. Na základe tejto detekcie potom uskutočňuje nami deklarované príkazy. Jeho alternatívou je tzv. DOM - *Document Object Model* alebo dokumentový objektový model, ktorý na vstupe kompiluje celý XML dokument a vytvára pre neho jeho stromovú štrukturálnu reprezentáciu, ktorá je objektovo orientovaná. To znamená, že každý uzol stromu - element XML - má svoje rozhranie, cez ktoré k nemu môžeme pristupovať. DOM definuje aj rozhrania na prístup k prípadnému textovému obsahu XML značiek a na prístup k ich atribútom.

K vytvoreniu vlastného SAX parseru písaného v jazyku Ruby použijeme balíček **Nokogiri**, ktorý slúži na prácu s XML dokumentami. Nasledujeme definíciu SAX parsera v dokumentácii knižnice Nokogiri a do akcie `create` nášho kontroléru pridávame nasledujúce riadky:

```
parser = Parser::AlgorithmDocument.new
xml = "<algorithm>"+algorithm_params[:xml]+"</algorithm>"
Nokogiri::XML::SAX::Parser.new(parser).parse(xml)
algorithm = parser.result
```

V nich vytvárame tzv. **parsovací dokument**, ktorému následne predávame textový reťazec XML programu určený ku kompilácii. SAX parser prechádza XML program sekvenčne element za elementom a volá metódy `start_element` a `end_element`, ktorými indikuje začiatok a koniec XML elementu. Na základe daného elementu a podľa jeho atribútov, tiež prístupných v argumente metódy `start_element`, následne vraciame textový reťazec, ktorý je časťou javascriptového kódu nášho kompilovaného algoritmu. Tento textový reťazec budeme ukladať do inštančnej premennej parsera s názvom `result`.

Zaiste jednou z možností implementácie nášho parseru je umiestnenie obrovského bloku `switch`, alebo teda v prípade jazyka Ruby `case`, ktorý bude sekvenčne vykonávať príkazy na základe názvu práve prechádzaného elementu. V snahe nevytvárať veľké bloky a **odčleniť programovú logiku jednotlivých elementov**, vytvárame pre každý XML element samostatnú triedu s metódami `start_print` a `end_print`. Metóda `start_element` nášho parsovacieho dokumentu vyzerá v takom prípade nasledovne:

```
def start_element(tagname, attrs = [])
  el = Parser::Elements.const_get("#{tagname.capitalize}Element").new
  attrs.each{ |attr| el.instance_variable_set("@"+attr[0], attr[1]) }
  @result << el.start_print
end
```

Inštanciám jednotlivých elementov priraďujeme atribúty ako ich inštančné premenné. V metóde `start_print` jednotlivých inštancií elementov, budeme definovať návratovú hodnotu ako textový reťazec, ktorý zodpovedá časti javascriptového kódu, do ktorého sa program písaný v jazyku XML kompiluje. V prípade elementu `<Array />` k nemu priradená trieda vyzerá nasledovne.

```
class ArrayElement < Element
  attr_accessor :length, :name, :random
```

```
  def start_print
    i = "#{name}_i"
    value = random ? "Math.random()" : 0

    "let #{name} = []; for (let #{i} = 0; #{i} < #{length}; #{i}++)
                       { #{name}[#{i}] = #{value}; };"
  end
end
```

Ako si môžeme všimnúť element `<Array />` so svojími atribútmi presne zodpovedá časti javascriptového kódu, ktorý reprezentuje kompilovaný kód algoritmu určený k animácii. Aby sme sa ubezpečili, že nepoužívame ten istý identifikátor pre iterátor, jeho názov je závislý na názve identikátoru samotného poľa. Existujú v našom prípade ale elementy, ktoré sú párové a teda majú dcérske elementy, ktoré sú do nich vnorené. Príklad takéhoto elementu je XML element riadiaceho cyklu `while`. Pri výpise javascriptového kódu potrebujeme uzatvoriť blok cyklu while zloženou zátvorkou. Inými slovami používame aj metódu `end_print`. Tú voláme z metódy `end_element` parsovacieho dokumentu. Problémom, ale je, že od začiatku otváracieho elementu `<while>` sme parsovali mnoho jeho dcérskych elementov a tak na inštanciu triedy elementu `<while>` už nemáme v metóde `end_element` prístup. To riešime zavedením **zásobníka** pre **inštancie tried prechádzaných elementov**.

```
def start_element(tagname, attrs = [])
  ...
  @stack << el
end

def end_element(tagname)
  @result << @stack.pop.end_print
end
```

Pri definícii triedy elementu `<while>`, môžeme spokojne použiť aj metódu `end_print`.

```
class WhileElement < Element
  attr_accessor :cond

  def start_print
    "while (#{cond}) {"
  end

  def end_print
    "}"
  end
end
```

Trieda elementu `<if>` vyzerá analogicky ako trieda `WhileElement` a parsovacie triedy elementov `Var` a `set` vracajú javascriptový kód, ktorý je zjednodušením

návratového kódu triedy `ArrayElement`. V ňom sa uskutočňuje iba priradenie alebo deklarácia premennej (`"var #{name}#{value.present? ? "=#{value}": ""};"`). Pri triede `GraphElement` elementu `Graph`, vracia trieda textový reťazec priradenia nového snímku do poľa snímkov.

```
class GraphElement < Element
    attr_accessor :name, :ref, :markerX

    def start_print
      "frames.push(m(
        frames[frames.length-1],
        { canvases: {
          #{name}: { data: #{ref}, markerX: #{markerX} }
        } }
      ));"
    end
  end
```

Jediné čo nám nastáva upraviť je, vkladať hodnoty `data` a `markerX` jedine v prípade, že ich príslušné atribúty sú v XML značke použité. Inak na tieto miesta do snímku bude odkaz na nulovú hodnotu čo zabezpečí, že v danom snímku sa nebudú v grafe zobrazovať žiadne dáta, alebo, že z grafu zmizne vodiaci bod. K tomu sme si vytvorili metódu `hash`, ktorá na základe kľúčov nám takýto javascriptový objekt ako textový reťazec generuje.

```
def hash
  h = {}
  h[:data] = ref if ref.present?
  h[:markerX] = ref if markerX.present?
  h.to_json.gsub('"', "")
end

def start_print
  "frames.push(m(
    frames[frames.length-1],
    { canvases: { #{name}: #{hash} } }
  ));"
end
```

Je potrebné spomenúť, že kolekcia kľúčov a hodnôt vo frameworku Rails, má tu príjemnú vlastnosť, že ju môžeme **transformovať do formátu JSON**. V ňom sa nachádzajú hodnoty kľúčov v úvodzovkách, čo naše identifikátory premenných ako napr. `data`, transformuje do textových reťazcov. Je preto potrebné metódou `gsub` tieto úvodzovky odstrániť a až tak vkladať do návratového textového reťazca.

Podarilo sa nám vytvoriť náš prvý algoritmus zapísaný v našom jazyku XML a zároveň SAX parser, ktorý tento jazyk parsuje. I keď pri popisovaní jednotlivé kroky pôsobia jasne a zrozumiteľne, pri vývoji a ladení parseru, pre tento konkrétny príklad algoritmu, sme strávili nezanedbateľný čas. Drobnosti ako odstránenie nadbytočných úvodzoviek, vhodné vypisovanie identifikátorov ale aj

realizácie štruktúry samotného parseru si vyžadovali istú dávku trpezlivosti a zamyslenia a riešenie nebolo od začiatku vôbec tak jasné. Pri jeho konceptuálnej tvorbe sme využívali princípy frameworku Rails a jazyka a Ruby, v rámci ktorých sme sa snažili pridržiavať čo najjednoduchšieho, najzrozumiteľnejšieho a tak najefektívnejšieho riešenia.

3. Rozšírenie XML jazyka

Máme vytvorenú prvotnú verziu nášho jazyka, k nemu prislúchajúci SAX parser a vizualizačné prostredie. To všetko funguje zatiaľ na demonštrácii jediného konkrétneho algoritmu a to algoritmu simulovaného žíhania. V našich požiadavkách vizualizácie numerických algoritmov v prvej kapitole sme toho chceli dosiahnúť omnoho viac preto našu aplikáciu a vytváraný programovací jazyk na báze XML syntaxe budeme doplňovať o **funkcionalitu potrebnú pre vizualizáciu ďalších algoritmov**.

3.1 Zobrazovanie funkcií jednej premennej

Jednou zo základných požiadaviek numerických algoritmov je demonštrovať daný algoritmus na konkrétnej funkcii. Našim cieľom bude zobrazovať funkciu jedinej premennej komponentom `Graph` a demonštrovať na nej algoritmus simulovaného žíhania, ktorý zatiaľ funguje iba na dátovej štruktúre poľa. Neskôr budeme chcieť takisto rozšíriť sadu algoritmov, ktoré budeme môcť vizualizovať, a napísať v nami definovanom XML jazyku, a to o metódu gradient descent. **Metódu gradient descent** budeme neskôr demonštrovať aj na funkcii dvoch premenných.

Našim cieľom je najprv zobrazovať funkciu jedinej premennej. Chceme dosiahnuť vykresľovanie funkcie jednej premennej v komponente `Graph`. Takisto je našou požiadavku možnosť zmeny veľkosti komponentu `Graph`, ktorej sa vykresľovaná funkcia prispôsobí. Inšpirovaný znova postupom Mighta [4], pri vykresľovaní funkcii v elemente `<canvas></canvas>` vnímame, že vykresľovanie funkcie nie je nič iné ako vykresľovanie množiny bodov, podobne ako v prípade poľa. Otázka je odkiaľ tie body budú prevzaté. Generovanie poľa bodov teraz prebieha v javascriptovom kóde samotného algoritmu na mieste použitia XML značky `<Array />`. Ich vykresľovanie je ale súčasťou komponentu `Graph`. Vieme, že počet bodov, ktorý potrebujeme na vykreslenie funkcie sa rovná šírke komponentu `Graph`, tá sa ale môže meniť a predsa nemôže byť kompilovaný kód algoritmu závislý od tejto šírky. V takom prípade pri každej zmene rozmerov okna grafu by sa algoritmus musel kompilovať nanovo. Jedným z riešení je generovať fixovaný počet bodov (napr. 1000) na strane generovaného javascriptového kódu, zodpovedajúceho algoritmu. Pre tieto body je následne generovaná absolútna pozícia na plátne grafu. Po dôkladnom testovaní sme ale prišli na to, že počet bodov 1000 je príliš veľký už pri zobrazovaní funkcie jedinej premennej a zjavne obmedzuje celkovú rýchlosť aplikácie a zahlcuje pamäť pri neustálom vykresľovaní. Je viac ako zrejmé, že pri zobrazovaní funkcii dvoch premenných by musel byť aj pozdĺž vertikálnej osi y-ovej súradnice počet bodov radovo v stovkách bez ohľadu na výšku grafu. To by prakticky z hľadiska pamäťovej náročnosti pri vykresľovaní značne obmedzilo rýchlosť aplikácie. Tým najlepším, čo nakoniec môžeme urobiť je **prenechať programovú logiku vykreslenia** úplne komponentu `Graph` a jeho atribútu `data` predávať priamo funkciu miesto vygenerovaného poľa. To nám značne uľahčí aj podobu kompilovaného javascriptového kódu zodpovedajúceho XML programu nášho jazyka. Dôležitou otázkou je takisto aj to ako budeme v rámci XML zápisu algoritmu predávať oknu `Graph` odkaz na funkciu a ako bude funkcia vlastne definovaná. Zatiaľ sme nedefinovali žiadnu XML značku pre definíciu funkcie. V

záujme zachovania rovnomernosti nami vytváraného jazyka a jeho uniformovanosti, prichádzame s návrhom na definíciu kombinácie XML značiek funkcie a grafu:

```
<Function formula="Math.sin(x)" name="f" />
<Graph name="1" ref="f" />
```

Podobne ako v prípade poľa pri algoritme simulovaného žíhania je komponentu `Graph` predávaný odkaz na identifikátor tentoraz funkcie k vykresleniu. Ešte v tejto nami opisovanej fáze vývoja mala značka `<Function />` za zámer deklarovať matematické výrazy určené na vykreslenie grafom. Ako ale neskôr zistíme dá sa použiť aj na deklaráciu funkcií nie len matematických výrazov. V snahe nasledovať kritérium rovnomernosti programovacieho jazyka, sme sa do hodnoty atribútu mena funkcie nesnažili vpísať aj jej argumenty čo by narušilo istú konzistenciu zápisu hodnoty do atribútu `name` naprieč XML značkami. Určenie argumentov funkcie resp. matematického výrazu sme vyriešili **skenovaním a hľadaním identifikátorov** v ňom. To prebieha **regulárnym výrazom** a metódou `scan` ako je aj definované v parsovacej triede elementu `<Function />`.

```
class FunctionElement < Element
  attr_accessor :formula, :name

  def start_print
    variables = " #{formula} "
      .scan(/[^.a-z]([a-z_]+)\s*(?![a-z.(])/i)
      .map { |i| i[0] }.uniq

    "let #{name} = function(#{variables.join(', ')})
                  { return #{formula}; };"
  end
end
```

Regulárny výraz identifikátory identifikuje ako tie slová, pred a za ktorými sa nenachádza znak bodky, a za ktorými sa nenachádza znak otvorenej oblej zátvorky. Zo zoznamu nájdených identifikátorov sú odstránené duplikáty a vytvára sa z neho metódou `join` textový reťazec argumentov funkcie oddelených čiarkou. Hodnota atribútu `formula` sa stáva návratovou hodnotou funkcie.

To čo nám ostáva je len komponentu `Graph` umožniť vykresľovanie funkcie jedinej premennej. Komponent `Graph` si stále myslí, že hodnotou jeho atribútu `data` je pole čísel určené k vykresleniu na grafe. To komponent `Graph` môže predpokladať aj naďalej pretože všetko čo nám stačí urobiť je pre funkciu v atribúte `data` vygenerovať body k vykresleniu. Komponentu `Graph` vytvárame novú metódu `generateData`, ktorú dopĺňame do volania metódy `draw` pred volaním metódy `renderData`. V metóde `generateData` generujeme pole bodov veľkosti šírky grafu nasledovne:

```
let data = this.props.data,
    w = this.props.w,
    isFunction = typeof data === "function";
```

```
if (isFunction) {
  let arr = [];
  for (let i = 0; i < w; i++) {
    arr[i] = data(i/w);
  }

  data = arr;
}

this.data = data;
```

Výsledné pole bodov ukladáme do inštančnej premennej komponentu `this.data`. Takotou implementáciou nemusíme vôbec v parseri kontrolovať, či to, na čo sa komponentom `Graph` odkazujeme je funkcia alebo pole a ostáva to plne v réžii a zodpovednosti komponentu `Graph`. Zároveň generuje sa len taký počet bodov aký je nevynutný pre vykreslenie funkcie na plátne určených rozmerov.

3.2 Zobrazovanie vybranej časti grafu funkcie

Pri zobrazovaní funkcie jednej premennej sme si uvedomili potrebu kontroly nad tým, ktorá časť funkcie sa v jej grafe zobrazuje. Doteraz sme zobrazili vždy prvý kvadrant na intervaloch $[0, 1]$ pozdĺž oboch súradníc. Pri funkcii sínus ale môžeme chcieť zobrazovať aj iný **intervalový rozsah** - napr. $[0, 2\pi]$. To nás viedlo k implementácii rozsahu vykresľovanej funkcie. Ten chceme eventuálne určovať z pozície XML kódu algoritmu. Okrem rozsahu chceme definovať aj posun funkcie. Spoločne s atribútom `markerX`, možným ďalším atribútom `markerY`, a teraz novým atribútmi pre rozsah `rangeX`, `rangeY` a pre posun `shiftX`, `shiftY` si uvedomujeme, že počet atribútov značky `<Graph />` začína byť mierne neprehľadný. Prirodzene nás to láka k implementácii vnorených XML značiek (napr. `<Marker />` alebo `<Range />`) pre značku `<Graph />`, je ale potrebné si uvedomiť, ako to bude fungovať pri implementácii parsovania. Objekt grafu predaný snímke vo svojej úplnosti vyzerá takto:

```
{ data: data, markerX: 20 }
```

To, čo by sme chceli v ideálnom prípade dosiahnuť by mohlo mať podobu

```
{
  data: data,
  marker: { x: 20 },
  range: { x: 2, y: 1 },
  shift: { x: 1, y: -1 }
}
```

Vtláča sa nám do mysle predstava otvorenia javascriptového objektu komponenty `Graph` podobným spôsobom ako sú otvorené javascriptové bloky elementov `while` a `if` pre ich vnorené elementy. Trieda `GraphElement` by následne vyzerala takto:

```
class GraphElement < Element
  attr_accessor :name, :ref, :markerX

  def start_print
    data = "data: #{ref}," unless ref.nil?
    "'#{name}': { #{data}"
  end

  def end_print
    " }),"
  end
end
```

zatiaľ čo jej dcérske elementy by definovali kľúče `marker`, `range` a `shift` ako napr.

```
class RangeElement < Element
  attr_accessor :x, :y

  def start_print
    "range: { x: #{x}, y: #{y} },"
  end
end
```

Máme predstavu XML zápisu pri definovaní zobrazovaného rozsahu grafu funkcie. Teraz potrebujeme dané atribúty akceptovať a použiť komponentom `Graph`. To čo nám stačí urobiť je upraviť projekciu bodov tak aby fungovala v rámci intervalového rozsahu a aby obsahovala posun. Nastavujeme takto počítanie funkčnej hodnoty v metóde `generateData`.

```
arr[i] = data(i*rangeX/w + shiftX);
```

Podobne implementujeme myšlienku rozsahu a posunu do metódy na vykreslovanie vodiacich bodov `renderMarkers`. V nej predpokladáme v prípade vykresľovania funkcia hodnotu `x`, ktorá určuje skutočnú hodnotu na súradnici `x` a nie index generovaného poľa bodov. Pri získavaní indexu generovaného poľa, potrebujeme počítať index aj skrz hodnotu intervalového rozsahu a posunu. **Rozsah a posun** musí byť započítaný aj v prípade vykresľovania súradníc grafu v metóde `renderAxes`. Rozsah a posun pozdĺž y-ovej súradnice je súčasťou výpočtu absolútnej polohy bodu v rámci y-ovej súradnice v metóde `projectY`. Metóda `projectX` prijíma v argumente index bodu v už generovanom poli dát, na rozdiel od metódy `projectY`, ktorá prijíma relatívnu hodnotu y-ovej súradnice.

V tejto sekcii sme dokázali implementovať nové XML značky `<Range />` a `<Shift />`, ktoré sú dcérskymi elementami elementu `<Graph></Graph>`. Značky určujú rozsah **zobrazovanej časti funkcie grafu**. Opäť je potrebné pripomenúť, že ladenie funkčnosti značiek a zobrazovania rozsahu funkcie nebolo tak priamočiare a jednou z najväčších výziev bolo nastaviť správne projektovanie vodiacich bodov, bodov samotnej funkcie ako aj súradníc. Uvádzame príklad použitia nami vytvorených značiek.

```
<Function formula="Math.sin(x)" name="f" />
<Graph name="1" ref="f">
  <Range x="7" y="2" />
  <Shift x="0" y="-1" />
</Graph>
```

3.3 Zobrazovanie funkcií dvoch premenných

V predošlej sekcii sme dokázali implementovať zobrazovanie funkcie jednej premennej. Našim cieľom je ale pokúsiť sa o implementáciu zobrazovania funkcií dvoch premenných a to z dôvodu, že pri zobrazovaní funkcií dvoch premenných vynikne vizualizácia algoritmu gradient descent omnoho výraznejšie. Prirodzenie zobrazovanie funkcií dvoch premenných môžeme využiť pri ďalších algoritmoch ako napr. pri algoritme optimalizácie rojom častíc. Naša prvotná predstava zobrazovania funkcie dvoch premenných spočíva v tom, že miesto poľa hodnôt budeme generovať v komponente `Graph` pole polí. A teda pre každý stĺpec si nebudeme pamätať konkrétnu funkčnú hodnotu ale budeme predpokladať možnosť pamätania si viacero funkčných hodnôt v jednom stĺpci. Stačí nám zistiť, ktoré funkčné hodnoty to budú. Môžeme si zvoliť, že vrstevnicu budeme vykreslovať všade tam kde sa funkčná hodnota približuje hodnote napr. 2. Čo ale ak obdržíme mriežku, v ktorej budú všetky funkčné hodnoty rovné dvom? Výsledný vykreslený graf tak bude celý čierny bez akéhokoľvek náznaku vykreslenia nejakej vrstevnice. Podstúpili sme dôkladný prieskum spôsobov vykreslovania grafov funkcií dvoch premenných a dopátrali sme sa k niekoľkým poznatkom. Pre vykreslovanie funkcií dvoch premenných potrebujeme vykreslovať niečo čo sa nazýva **izokontúrou** a ide o to isté, čo sme nazvali vrstevnicou.

3.3.1 Izokontúra

Izokontúra je definovaná ako množina n-tíc argumentov funkcie arity n, pre ktoré vracia funkcia tú istú funkčnú hodnotu.

Zároveň existuje niečo ako algoritmus tzv. pochodujúcich štvorcov (marching squares algorithm), ktorý k vykresľovaniu izokontúr funkcií dvoch premenných je práve určený.

3.3.2 Algoritmus pochodujúcich štvorcov

Algoritmus pochodujúcich štvorcov nám umožňuje vykresliť kontúrový graf matematickej funkcie dvoch premenných. Vstupom algoritmu pochodujúcich štvorcov je tzv. izohodnota, čo je zvolená funkčná hodnota danej funkcie pozdĺž ktorej chceme vykresliť izokontúru. Vstupom algoritmu môže byť aj množina izohodnôt. Algoritmus v takom prípade vykresľuje viacero izokontúr. Okrem izohodnuty je vstupom aj dvojrozmerné pole alebo mriežka funkčných hodnôt funkcie vo zvolených intervaloch. Čím väčšie rozmery dvojrozmerného pola zvolíme, tým presnejšie je algoritmus schopný vykresliť kontúry našej funkcie.

Algoritmus, ako popisuje R. Wenger [11], prechádza týmto dvojrozmerným polom po individuálnych **štvorcových konfiguráciach** - podmnožinách poľa

o rozmeroch 2x2. Porovnáva položky konfigurácie s vybranou izohodnotou. Následne klasifikuje podľa 4 hodnôt štvorcovej konfigurácie, prechádzanú konfiguráciu do nového pola hodnôt. Individuálna klasifikácia konfigurácií určuje líniové segmenty izokontúr ako vidíme na obrázku.

V našej verzii implementácie algoritmu, miesto určovania štvorcových konfigurácii overujeme priamych susedov políčka dvojrozmerného poľa. Priamými susedmi alebo priamymi susednými políčkami vybraného políčka označujeme políčka po jeho ľavej a pravej strane a políčka nad a pod vybraným políčkom. Inými slovami, priamy susedia políčka sú tie políčka, ktoré majú s políčkom spoločnú hranu. Máme určenú izohodnotu a našim cieľom je určiť či dané políčko, ktoré práve prechádzame nie je hraničným políčkom potenciálnej izokontúry. To určujeme porovnaním hodnoty políčka a jeho priamych susedov s izohodnotou. V komponente `Graph` je to vyjadrené podmienkou:

```
if (
    arr[i][j] >= isocontours[l] &&
    neighbours.filter((x) => x < isocontours[l]).length > 0
   ) point = 1;
```

Podmienka určuje vykreslenie bodov izokontúry na daných súradniciach mriežky funkčných hodnôt. Izohodnotu v podmienke vyberáme z poľa všetkých izohodnôt s názvom `isocontours`. Podmienka môže byť slovami popísaná nasledovne: *Ak funkčná hodnota prechádzaného políčka je väčšia ako vybraná izohodnota a funkčná hodnota aspoň jedného z priamych susedov prechádzaného políčka je menšia, na mieste políčka vykresľujeme bod izokontúry.* Dvojrozmerné pole alebo mriežku funkčných hodnôt takto transformujeme na **dvojrozmerné pole indikátorov vykreslenia bodu**.

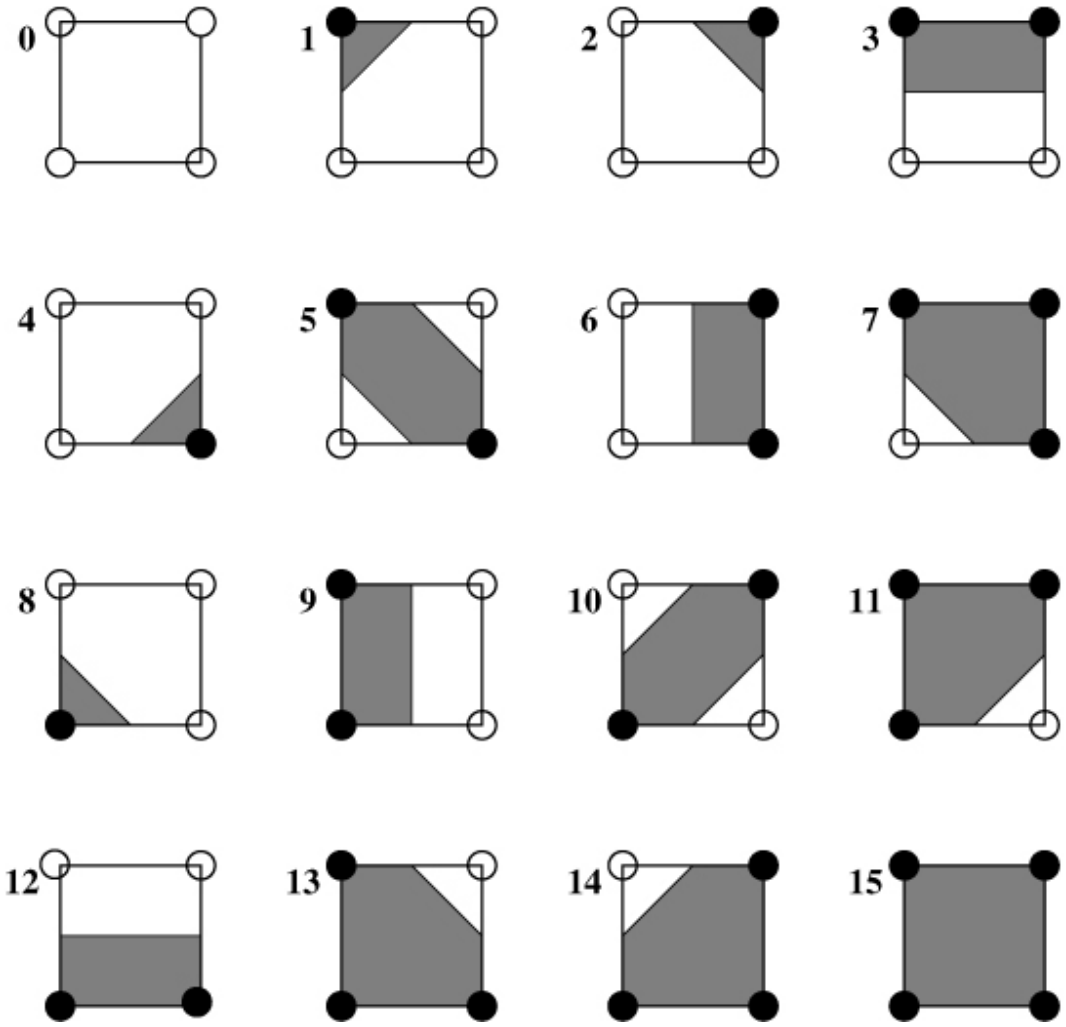
Algoritmus pochodujúcich štvorcov implementujeme v metóde `generateData` komponentu `Graph`, v ktorej ukladáme výsledné dvojrozmerné pole bodov izokontúr do inštančnej premennej komponentu `this.data`. K nej, v prípade vykresľovania funkcie dvoch premenných, pristupujeme špeciálne v metóde `renderData`.

Jednotlivé izohodnoty určujeme na základe pravidelného delenia intervalu medzi najmenšou a najväčšou funkčnou hodnotou vykresľovanej funkcie. Ten delíme na $n + 1$ častí, kde n je počet izokontúr, ktoré chceme vykresľovať. Pre počet izokontúr vytvárame v značke graf vyhradený atribút, ktorý sa následne kompiluje do javascriptového objektu grafu ako jeho ďalší kľúč.

3.4 Implementácia metódy gradient descent

Vykresľovaním funkcií jednej a dvoch premenných, implementáciou novej XML značky `Function` a implementáciou kontroly zobrazovaných intervalov grafu máme implementované takmer všetko pre vizualizáciu metódy gradient descent. To posledné čo potrebujeme je **zobrazovanie vodiacich bodov** v grafe **funkcií dvoch premenných**.

Doteraz sme zobrazovali vodiaci bod grafu na základe hodnoty jeho x-ovej súradnice (`<Marker x="20"/>`). Jeho y-ová súradnica bola určená funkčnou hodnotou vykresľovanej funkcie alebo poľa. Takýto postup nechávame značke ako predvolený spôsob jej interpretácie a doplňujeme ho o prípad výskytu atribútu



Obrázek 3.1: Možné štvorcové konfigurácie v algoritme pochodujúcich štvorcov

`y`. V prípade, že značka obsahuje aj atribút `y`, je funkčná hodnota určujúca polohu y-ovej súradnice vodiaceho bodu nahradená hodnotou atribútu `y` značky `<Marker/>`.

Metódu gradient descent, ktorá slúži k nájdeniu extrému vybranej funkcie, môžeme s využitím nových značiek nášho XML jazyka a novej funkcionality jeho vizualizačného prostredia implementovať takto:

```
<Function formula="x*x + y*y" name="f" />
<Function formula="2*x" name="dfx" />
<Function formula="2*y" name="dfy" />

<Var name="x" value="Math.random() * 100" />
<Var name="y" value="Math.random() * 100" />
<Var name="lrate" value="0.1" />
<Var name="gradientx" />
<Var name="gradienty" />
<Var name="e" value="0" />

<while cond="e < 100">
  <set name="gradientx" value="dfx(x)" />
  <set name="gradienty" value="dfy(y)" />
  <set name="x" value="x - lrate * gradientx" />
  <set name="y" value="y - lrate * gradienty" />

  <Graph name="1" ref="f" isocontours="10">
    <Range x="200" y="200" />
    <Shift x="-100" y="-100" />
    <Marker x="x" y="y" />
  </Graph>

  <set name="e" value="e + 1" />
</while>
```

Značka `<Range />` značí, že intervaly oboch súradníc grafu majú zhodne veľkosť 200. Značka `<Shift />` zase posúva počiatok vykreslenia grafu na bod $[-100, -100]$. **Počiatkom vykreslenia grafu** rozumieme absolútny ľavý dolný roh plátna grafu. Je potrebné spomenúť, že daná verzia nášho jazyka XML použitá pri zápise metódy gradient descent nie je finálnou verziou nášho XML jazyka, ale je ilustráciou verzie nášho XML jazyka v dobe jeho vývoja. Preto po vložení daného XML kódu do našej aplikácie a po jeho kompilovaní tento kód nebude fungovať.

3.5 Popisky grafu

Keď sme vizualizovali algoritmus simulovaného žíhania, vizualizovali sme ho na náhodne generovaných hodnotách. Našim cieľom bolo demonštrovať princíp algoritmu a skutočné hodnoty poľa náhodne generovaných hodnôt pre nás vo vizualizácii neboli zaujimavé. Pri zobrazovaní funkcií, či už jednej alebo dvoch premenných, a hlavne pri zobrazovaní funkcií, z ktorých vykresľujeme v grafe

špecifickú časť, ich špecifický interval, začína byť pre nás celkom nevyhnutné sa v takomto grafe aj orientovať. Preto v tejto sekcii budeme navrhovať a implementovať zobrazovanie popisných alebo **orientačných čísel** súradníc grafu.

Je potrebné si uvedomiť, že okno, v ktorom sa vykresľuje graf zvolenej funkcie je relatívne malé a jeho základom je predovšetkým línia alebo kontúry grafu, ktoré popisujú hodnoty vykresľovanej funkcie. Našim cieľom teda nie je zaplniť priestor vykresľovania grafu na jeho plátne hŕbou orientačných čísel alebo k nim vodiacich čiarok. Preto, aby mal užívateľ prehľad o tom aká časť funkcie je v grafe zobrazovaná, mu stačí poznať minimálnu a maximálnu hodnotu zobrazovanej časti oboch súradníc, čo sú presne 4 orientačné čísla. Pri implementácii tohoto návrhu sme ale zistili, že takéto čísla môžu byť číslami desatinného rádu a aj napriek našej snahe zaokrúhľovať, zmenšiť veľkosť písma a zobrazovať iba 1 desatinné miesto, sa nám pozdávalo, že orientačné čísla zaberajú v grafe veľa priestoru. Rozhodli sme sa miesto toho nájsť v grafe k ním vždy najbližšie celé čísla. Tým sme následne projekciou funkcií `projectXNum` a `projectYNum` našli ich absolútnu polohu na plátne grafu. Tú sme ešte upravili posunom vo funkciách `drawNumberWLine` a `drawNumberHLine`, tak aby popisné čísla **neprekrývali súradnice** a boli **čitateľné** napr. vedľa alebo pod súradnicou. Neskôr sme ale zistili, že problém nastáva pokiaľ je interval zobrazovanej časti grafu pozdĺž vybranej súradnice podintervalom intervalu $(-1, 1)$. V takom prípade je najbližším celým popisným číslom, číslo 0 a popisné čísla tak postráda svoj účel. V takomto prípade sme nútení zobraziť desatinné číslo ako číslo popisné. Pôvodne sme uvažovali zaokrúhlením čísla na 1 desatinné miesto, ale v prípade, ak užívateľ určí interval zobrazenia o niekoľko rádov menší, bude sa popisné číslo až veľmi výrazne odlišovať od reálnych hodnôt súradnice. Ak nezaokrúhľujeme vôbec, tak prirodzene riskujeme zobrazenie aj užívateľom neželaných dlhých desatinných čísel, k vzniku ktorých dochádza pri prevode desatinných čísel z binárnej do desiatkovej sústavy. Rozhodli sme sa preto vo funkciách `drawNumberWLine` a `drawNumberHLine` zaokrúhľovať na 4 desatinné miesta. Konštrukciou `parseFloat(...).toString()` vymazávame koncové nuly zobrazovaného čísla. Zarovnanie popisných čísel x-ovej súradnice sme mali pôvodne nastavené na hodnotu `center`, tak aby stred popisného čísla bol presne pod k nemu určenou vodiacou čiarkou. V prípade dlhších desatinných čísel pri centrovanom zarovnaní ale hrozí, že časť čísla bude (ne)vykresľovaná už mimo plochu plátna. Preto zarovnávame ľavé popisné číslo x-ovej súradnice doľava, tak aby pri pridávaní desatinných miest sa číslo nedostalo mimo plátno. Pravé popisné číslo z toho istého dôvodou zarovnávame doprava.

Jedným z dôsledkov snahy zobrazovať orientačné hodnoty súradníc, bolo aj vytvorenie pomyselného **bieleho** rámčeka alebo **odsadenia** okolo vykresľovanej funkcie. Odsadenie je stále súčasťou plátna elementu `<canvas></canvas>` a je určené pre zobrazovanie práve súradníc a ich orientačných, popisných čísel. Prvá myšlienka bola vytvoriť biele miesto iba pozdĺž spodného a ľavého okraja plátna, predpokladajúc, že najčastejší prípad umiestnenia súradníc je práve pozdĺž týchto okrajov. Neskôr sme ale zistili, že pre popisné čísla umiestnené pri pravom a hornom okraji je toto odsadenie rovnako užitočné až nevyhnutné. V tomto odsadení sa funkcia a jej funkčné hodnoty už nevykresľujú. Pri implementácii bieleho odsadenia, ktorého veľkosť definujeme v konštruktore, sme museli upraviť projekciu jednotlivých bodov a prvkov grafu.

3.6 Zobrazovanie matematických výrazov

Doteraz sme pracovali v našom vizuálnom prostredí iba s oknom XML editora a oknom grafu. Existujú ale algoritmy, pri ktorých nevyužívame okno grafu ale potrebujeme **algoritmus vizualizovať iným spôsobom**. Príkladom takéhoto algoritmu je násobný algoritmus, ktorý slúži na výpočet dominantného vlastného vektoru a k nemu príslušného vlastného čísla. V ňom prakticky okrem vizualizácie aproximácie samotných číselných hodnôt, nie je nič čo by sme prakticky, vo všeobecnom príklade algoritmu, mohli vizualizovať. Už pri tvorbe aplikácie a XML jazyka bola v našich myšlienkach zahrnutá predstava animovania matematických výrazov, ktoré popisujú stav animovaného algoritmu. Z našich skúseností pri používaní knižnice **MathJax** vieme v prostredí HTML stránok vykreslovať matematické výrazy a z našich skúseností ich dokážeme vo webovom prostredí dynamicky aj meniť. Našim cieľom je teda vytvorenie okna matematického výrazu v rámci vizualizačného prostredia, ktorý vykreslí daný reťazec jazyka TeX v novom okne nášho vizualizačnom prostredí.

3.6.1 Knižnica MathJax

K vykresľovaniu matematických výrazov používame na tento účel určenú a v Javascripte písanú knižnicu MathJax. K jej použitiu ju stačí importovať do súbora `*.html` webovej stránky a to v jej hlavičkou použitím nasledovnej značky načítania

```
<script src='https://cdnjs.cloudflare.com/ajax/libs/mathjax
/2.7.4/MathJax.js?config=TeX-MML-AM_CHTML' async></script>
```

Na celej vykresľovanej webovej stránke môžme po načítaní našej knižnice používať oddeľovače $$ pre zobrazovanie matematických výrazov v móde *display* a po nastavení parametra `inlineMath`, objektu `tex2jax` v konfigurácii knižnice aj oddeľovače $ pre zobrazovanie matematických výrazov v móde *inline*. Jednotlivé módy zobrazovania sú ilustrované na nasledovnom obrázku.

Gamma funkcia splňujúca predpoklad $\Gamma(n) = (n-1)! \quad \forall n \in \mathbb{N}$ (mód "inline") je definovaná Eulerovým integrálom ako

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\,dt\,.$$

(mód "display").

Obrázek 3.2: Módy zobrazovania matematických výrazov knižnice MathJax

Celkové znenie konfigurácie knižnice v našej aplikácii vyzerá nasledovne:

```
MathJax.Hub.Config({
  showProcessingMessages: false,
  messageStyle: "none",
```

```
  showMathMenu: false,
  tex2jax: { inlineMath: [['$','$'],['\\(','\\)']],
  processClass: "text",
  ignoreClass: "mathjax-ignore" },
});
```

Nastavením parametrov `showProcessingMessages` a `messageStyle` rušíme zobrazovanie procesných správ pri vykresľovaní matematických výrazov. Hodnotou `false` parametru `showMathMenu` deaktivujeme zobrazovanie kontextového menu po kliknutí pravým tlačidlom myši na objekt vykresleného matematického výrazu. V parametri `tex2jax` nastavujeme **oddeľovače** pre vykreslovanie matematických výrazov v riadkovom móde (mód *inline*). Určením hodnoy kľúča `processClass` nastavujeme elementy tých tried, ktoré sú prioritne určené k vykresľovaniu a naopak vyplnením kľúča `ignoreClass` zakazujeme knižnici MathJax vykresľovať matematické výrazy v elementoch danej triedy.

Pri vývoji aplikácie používame hlavne dva konštrukty na riadené vykresľovanie matematických výrazov. Prvým z nich je konštrukt `Typeset`:

```
MathJax.Hub.Queue(["Typeset", MathJax.Hub, element]);
```

Čo sa v tomto záhadnom volaní deje? Metóda `Queue` objektu `MathJax.Hub` pridáva do fronty volania funkcií knižnicou MathJax, volanie funkcie `Typeset`. `MathJax.Hub` je objekt základnej kontrolnej štruktúry knižnice MathJax. Knižnica MathJax spracováva práve každý textový reťazec jazyka TeX, oddelený oddeľovačmi $$ a $, a vykresľuje ho postupným, sekvenčným vykonávaním funkcií a metód, ktorých volania si skladuje vo vlastnej fronte. Volaním metódy `Queue` pridávame do tejto fronty ďalšie volanie funkcie. Metóda `Queue` akceptuje jediný argument a to argument tzv. **špecifikácie volania** spätnej funkcie. Ide o argument poľa, ktorého prvá položka môže byť volaním konkrétnej funkcie napr. `[function() ]`, alebo názvom funkcie určenej k volaniu. Druhou položkou poľa je objekt, z ktorého je funkcia a teda metóda volaná a každá ďalšia položka je argumentom danej funkcie. Volanie `Typeset` vykresľuje matematické výrazy v danom HTML elemente alebo zozname elementov predaných v argumente funkcie. Element môže byť v zozname identifikovaný pomocou CSS selektoru `id` alebo priamym odkazom na objekt štruktúry DOM. Ak funkcii `Typeset` nie je predaný žiadny element, sú **preprocesory** a následne procesory **vykresľovania** matematických výrazov aplikované na celý HTML dokument - čo znamená, že dochádza k opätovnému vykresľovaniu všetkých matematických výrazov dokumentu.

Ďalším konštruktom, alebo teraz už vieme, že metódou, je metóda `Text`. V nasledujúcom kóde získavame z elementu tzv. *Jax* objekt.

```
var math = MathJax.Hub.getAllJax(element)[0];
MathJax.Hub.Queue(["Text", math, "x+1"]);
```

Ide o knižnicou MathJax vytvorenú štruktúru resp. objekt popisujúci jeden matematický výraz daného elementu. Volaním metódy `Text`, ktorá sa aplikuje na tomto objekte, nastavujeme novú hodnotu matematického výrazu. Ten sa následne vykresľuje.

Táto metóda `Text` je určená k dynamickému obnoveniu vykresleného matematického výrazu a v starších verziách knižnice, na základe našich skúseností, vieme potvrdiť, že bola efektívnejšia ako vykresľovanie matematického výrazu nanovo. V novej verzii knižnice sa jej **efektivita zdanlivo neprejavuje** a ako je písané aj v dokumentácii knižnice MathJax, metóda `Text` urýchľuje len nájdenie matematických výrazov na stránke alebo vo vybranom elemente. K úplnému prekresleniu matematického výrazu dochádza tak či tak. To ovplyvnilo aj náš postup, pri navrhovaní okenného komponentu matematického výrazu.

3.6.2 Okno matematického výrazu

Podobne ako sme vytvorili okno `Graph`, pomocou ktorého zobrazujeme okná grafov na plátne našej aplikácie, vytvárame komponent `Formula`, ktorý bude slúžiť k **zobrazovaniu matematických výrazov**. Vytvárame ho v zložke `app/javascripts/components`. Základná verzia jeho návratovej HTML štruktúry v metóde `render` bude vyzerať nasledovne:

```
<Node>
  <div>{"\$\$" +this.props.content+ "\$\$"}</div>
</Node>
```

Podobne ako komponent `Graph` aj komponent `Formula` používa okenné rozhranie definované komponentom `Node`. Komponent `Formula` získava znenie matematického výrazu jazyku TeX v atribúte `content` a to následne vykresľuje oddeľovačmi $$, ktoré určujú vykreslenie výrazu v móde *display*.

Už pri popise knižnice MathJax sme spomínali konštrukty na vykresľovanie a obnovovanie matematických výrazov. Nebolo to náhodou. Tieto konštrukty plánujeme použiť v **systémových volaniach komponentov frameworku React** a to volaniach `componentDidMount` a `componentDidUpdate`. Elementu matematického výrazu dopĺňame atribút `ref - ref=(el) => this.el = el`. Pomocou neho si odkaz na daný element ukladáme do inštančnej premennej komponentu - `el`. K danému odkazu budeme pristupovať v systémových volaniach za účelom vykreslenia matematického výrazu. A to nasledovným spôsobom:

```
componentDidMount() {
  MathJax.Hub.Queue(["Typeset", MathJax.Hub, this.el]);
}
```

```
componentDidUpdate() {
  var math = MathJax.Hub.getAllJax(this.el)[0];
  MathJax.Hub.Queue(["Text", math, this.props.content]);
}
```

Komponent máme prakticky hotový a na základe našich predošlých skúseností do štýlopisu kaskádových štýlov CSS v `app/assets/stylesheets/application.css` pridávame štýly k vykresľovaným elementom knižnicou MathJax k odstráneniu modrého podsvietenia matematických výrazov po kliknutí a tiež k odstráneniu bieleho odsadenia matematických výrazov.

Pri **zmene veľkosti okna** matematického výrazu potrebujeme zmeniť veľkosť písma samotného výrazu, pretože ako sme si to už stihli vyskúšať, pri takomto stave aplikácie sa pri zmene rozmerov okna matematických výrazov nedeje so samotným výrazom absolútne nič. Veľkosť písma musí byť ale menená v pomere zmeny veľkosti rozmerov okna. Z našich predošlých skúseností vieme, že toto nefunguje úplne dobre a to hlavne preto, lebo pre každý výraz, môže byť ten pomer iný. To je dané tým, že pri **zmene veľkosti písma** niektorých výrazov, môže vykreslený výraz rásť rapídne do svojej šírky, pri zmene veľkosti písma iných výrazov, môže omnoho rýchlejšie rásť do svojej výšky. Pre každý výraz a pre každú zmenu veľkosti jeho okna by sme si potrebovali tento pomer počítať a zo skúsenosti vieme, že ani to k bezchybnej zmene veľkosti výrazov nestačilo. Nakoľko to nie je ani predmetom našej práce, zmenu veľkosti okien matematických výrazov nebudeme v našej aplikácii podporovať a pre komponent okenného rozhrania `Node`, vytvoríme booleovský atribút `resizeDisabled`, ktorým znemožňujeme zmenu veľkosti matematických výrazov.

3.6.3 Element `<Formula />`

Ku komponentu `Formula` budeme navrhovať k nemu príslušnú XML značku. Tá bude mať štandardne meno v atribúte `name`, aby sme k nej mohli pristupovať v rámci javascriptového objektového zápisu snímkov. Podobne bude mať atribúty `content` a `ref`, ktoré budú predávať komponente vykresľovaný obsah.

V príslušnej parsovacej triede elementu `Formula` vytvárame funkciu `hash`, ktorá generuje textový reťazec javascriptového objektového zápisu atribútov komponentu `Formula`.

```
def hash
  h = {}
  h[:content] = ref if ref.present?
  h[:content] = "'#{content}'" if content.present?
  h.to_json.gsub('"', "")
end
```

Všimnime si, že atribút `content` má v značke väčšiu prioritu ako atribút `ref`. Zatiaľ čo hodnota `ref` je priradená do obsahu ako možný identifikátor už v programe použitej premennej, hodnota `content` je umiestnená do úvodzoviek, čo charakterizuje jej interpretáciu ako textového reťazca. Našim cieľom je aj to, aby v rámci matematického výrazu sme mohli vkladať hodnoty existujúcich premenných. To sa dá dosiahnuť ak predávaným textovým reťazcom atribútu `content` komponentu `Formula` bude textový reťazec zlúčený s odkazom na javascriptovú premennú. Takýto zlúčený reťazec by mohol vyzerať takto: `"x = "+x`. Potrebujeme teda nejakým spôsobom povedať nášmu parseru, že na tomto a tomto mieste reťazca TeX matematického výrazcu bude vložený odkaz na túto a túto premennú. Inšpirovaný jazykom Ruby, vytvárame **operátor substitúcie** `@{}`. Do neho budeme umiestňovať názov premennej, na ktorú sa odkazujeme nasledovným spôsobom:

```
<Var name="x" value="10" />
<Formula content="x = @{x}" name="1" />
```

Zároveň upravujeme podobu výsledného reťazca na `"x = "+x+""` nakoľko nechceme programovo zisťovať, či operátor substitúcie sa nachádza na konci alebo na začiatku reťazca a na základe toho neumiestňovať znak `+` po jednej z jeho strán. Takýmto spôsobom jednoducho nahradíme všetky výskyty operátorov substitúcie `@{myvar}` v reťazci matematického výrazu za `'+myvar+'`. To uskutočňujeme v parsovacej triede `FormulaElement` regulárnym výrazom a metódou `gsub` takto:

```
content.gsub /@{([^}]*)}/, "'+(\\1)+'"
```

Náš nový vytvorený komponent `Formula` a k nemu príslušnú XML značku `<Formula />` ihneď testujeme na algoritme simulovaného žíhania doplnením ku značke grafu:

```
<Formula content="x = @{x}" name="x" />
<Formula content="y = @{y}" name="y" />
<Graph name="1" ref="data">
  <Marker x="x" />
</Graph>
```

Pri testovaní sme si všimli hneď niekoľko nedostatkov a nimi sa zaoberáme v nasledujúcich sekciách.

3.7 Stavy okien

Jedným s problémov, s ktorým sme sa stretli po vytvorení a testovaní komponentu `Formula` bolo zobrazenie všetkých okien na tom istom mieste. Bolo by dobré, keby sme rozmiestnenie okien mohli určiť programovo v rámci zápisu algoritmu v našom XML jazyku. Ďalším problémom bola **strata informácie o polohe okna**. Pri zmene snímkov animácie sa predošlé posunutie okna anuluje - okno sa vráti do pôvodnej pozície, a ak užívateľ na obrazovke mal okná usporiadané istým spôsobom, toto usporiadanie sa stráca pri akejkoľvek zmene momentálneho snímku. K tomu dochádza na základe prekreslovania celej HTML štruktúry, vykresľovanej komponentom `App` pri zmene jeho stavu, čo je súčasťou procesov knižnice React a jej paradigmy reaktívneho programovania. Podobne, ak zmeníme rozmery XML editora alebo okna grafu a následne prejdeme na iný snímok animácie, rozmery okien sa vrátia do ich pôvodných hodnôt. Musíme si teda hodnoty rozmerov a polôh okien pamätať.

Za schopnosť posúvania okien a za možnosť zmeny ich veľkosť sme vďační komponentom `Draggable` a `Resizable` použitým v našom komponente `Node`. Tieto komponenty manažujú svoj vlastný stav, v ktorom si ukladajú svoju momentálnu pozíciu resp. veľkosť. Ako sme už ale spomínali, sú dcérskymi komponentami komponentu `Node` a ten je dcérskym komponentom komponentov `Graph` a `XML` a tie sú dcérskymi komponentami komponentu `App`. Komponent `App` definuje svoj vlastný stav a pri jeho zmene znova prekresľuje svoj obsah čo znamená opätovnú inicializáciu komponentov `Graph` a `XML` a to znamená **opätovnú inicializáciu** komponenov `Node` a to zas spôsobuje novú inicializáciu komponentov `Draggable` a `Resizable`. To má prirodzene za následok stratu predošlého stavu polohy a veľkosti okna.

Dobrou správou je, že tak ako komponent `Draggable` tak aj komponent `Resizable` prijímajú v atribútoch hodnoty aktuálnej pozície resp. veľkosti. Tie sú uprednostnené pred interným stavom pozície a rozmerov komponentov. To čo chceme docieliť je pamätanie si pozícií a rozmerov okien a ich simultánna zmena pri akcii poťahovania okna alebo pri udalosti zmeny jej rozmerov. **Pozície a rozmery** jednotlivých okien si budeme **pamätať** v kolekcii uloženej v stave komponentu `App` pod kľúčom `nodes`. Aplikácia pracuje len s jedným oknom XML editora a preto si jeho stav bude pamätať pod kľúčom `xml`. Stav ostatných okien si bude pamätať pod kľúčmi `canvas-#{name}` a `formula-#{name}`, kde na mieste premennej `name` je uložený okna resp. XML značky. Jednotlivé hodnoty pozícií a rozmerov sú uložené pod kľúčmi `x`, `y`, `h`, `w`.

```
nodes: { xml: { x: 100, y: 100, h: 400, w: 480 } }
```

Tie sú predávané ako atribúty jednotlivým komponentom až kým sú predané komponentom `Draggable` a `Resizable`. Podobným spôsobom sú udalosti posúvania komponentu `onDrag` a zmeny jeho veľkosti `onResize` z komponentov `Draggable` a `Resizable` predávané „smerom nahor“ rodičovským komponentom až do komponentu `App`, ktorý metódami `rememberState` a `onGraphResize` nastavuje momentálny stav okna. Pri metóde `onGraphResize`, ktorá je volaná v prípade zmeny veľkosti grafu, získavame skutočnú veľkosť grafu vlastnosťami `offsetHeight` a `offsetWidth`, od ktorých musíme ale odčítať odsadenie plátna grafu.

Okrem zachovania zmeny pozície a rozmerov okien naprieč snímkami animácie, sme pre XML značky `Graph` a `Formula` pridali atribúty `x` a `y`, v prípade značky `Graph` aj `height` a `width`. Tie určujú **počiatočné hodnoty pozície** okna resp. jeho rozmerov a môže byť využité pri písaní programu, v ktorom si chceme vopred nakonfigurovať usporiadanie okien na plátne obrazovky.

Atribúty počiatočnej pozície a rozmerov sú pri predávaní komponentom prepisované hodnotami klientskych atribútov aktuálnej pozície resp. rozmerov. Toto prepisovanie je určené poradím rozvinutia jednotlivých objektov vlastností ilustrované napr. pri atribútoch v komponente `App` zobrazovaného komponentu `Graph`. To zobrazujeme v nasledovnej ukážke, ktorá tiež ilustruje čo sa deje pri rozbaľovaní javascriptovových objektov do atribútov komponentov.

```
<Graph {...this.state.animation.canvases[key]}
       {...this.state.nodes["canvas-"+key]} />
<Graph { x: 100, y: 100 } { x: 110, y: 120, w: 300, h: 150 } />
<Graph { x: 110, y: 120, w: 300, h: 150 } />
<Graph x="110" y="120" w="300" h="150" } />
```

3.7.1 Posúvanie okien matematických výrazov

Krátko potom ako sme sa tešili z toho, že aplikácia si teraz pamätá stav pozície okna a zachováva si ho aj naprieč rôznymi snímkami, sme objavili zásadný problém. Pri akejkoľvek snahe potiahnuť okno matematického výrazu, je samotný výraz vykresľovaný nanovo. Prečo sa toto deje? Po krátkom zamyslení usudzujeme na základe našich poznatkov a skúsenosti: Pri posúvaní okna sa mení stav

a zmena stavu má za následok prekreslenie HTML štruktúry, ktorá od stavu závisí. Nakoľko komponent `Formula` má teraz predávané stavové atribúty `x` aj `y` určujúce jeho polohu, jeho HTML obsah sa pri zmene stavu prekresľuje. Pri prekreslení HTML štruktúry sa prekreslené elementy vykresľujú a osadzujú nanovo a toho dôsledkom je okrem zmazania starého elementu aj nové volanie metódy `componentDidMount` čo v prípade komponentu `Formula` má za následok **neželané nové vykreslenie** matematického výrazu.

Tento problém riešime vytvorením ďalšieho komponentu s názvom `Mathjax`. Tento komponent nemá za úlohu nič iné, len vykresliť matematický výraz jemu zadaný. Nakoľko prijíma iba jediný atribút `content`, jeho obsah pri zmene jeho rodičovského komponentu `Formula` nebude menený a komponent nebude pri posúvaní okna `Formula` vykresľovaný a osadzovaný do HTML štruktúry nanovo.

Vrámci komponentu `Mathjax` je potrebné spomenúť našu snahu vylepšiť obnovovanie matematických výrazov pri prepínaní jednotlivých snímkov animácie. To sme robili napríklad spôsobom, že sme po zmene atribútu `content`, komponent zneviditeľnili zmenou jeho CSS vlastností a zobrazili volaním funkcie ako argumentu funkcie `MathJax.Hub.Queue`, až po vykreslení matematického výrazu. To sa ukázalo byť pri prehrávaní animácií algoritmov veľmi nepraktické, nakoľko samotná vykreslená hodnota matematického výrazu bola ihneď ukrytá pretože dochádzalo vzápätí k jej ďalšej zmene. Nakoľko volanie metódy `Text` sa v rámci efektivity ukázalo byť takmer zbytočné, nakoľko matematický vzorec je zhodný z celým textovým reťazcom, ktorý vstavaný parser knižnice MathJax prehľadáva, voláme tak ako pri osadení nového komponentu `Mathjax`, tak aj pri jeho obnovení jeho metódu `renderMathjax`, ktorá nanovo vykresľuje matematický vzorec volaním metódy `Typeset`.

3.8 Animácia viacerých okien

Jedným z nedostatkov, ktoré sme si všimli po implementácii komponentu `Formula` na príklade algoritmu simulovaného žíhania bolo, že pre komponent `Graph`, na ktorom sa nám zobrazoval vodiaci bod, a aj pre komponent `Formula` sa nám individuálne vytvorili nové snímky. Pritom, každá dvojica takýchto snímkov popisovala jediný stav. To čo chceme dosiahnuť je **meniť stav oboch okien** v rámci **jedinej snímky**. Prvý nápad, ktorý nám prišiel na rozum je vytvorenie XML značky `Frame`, ktorá bude obaľovať všetky vizuálne komponenty, ktorých zmeny sa majú uskutočniť v rámci jedinej snímky animácie. Po úprave už raz demonštrovanej časti XML kódu algoritmu simulovaného žíhaní, by tak simultánna zmena stavu všetkých troch komponentov, mohla byť zapísaná takto:

```
<Frame>
  <Formula content="x = @{x}" name="x" />
  <Formula content="y = @{y}" name="y" />
  <Graph name="1" ref="data">
    <Marker x="x" />
  </Graph>
</Frame>
```

Vo výslednom kompilovanom Javascripte daného XML kódu nechceme robiť

nič iné len zlučovať kľúče pre jednotlivé grafy a matematické výrazy v rámci jediného snímku nasledovným spôsobom:

```
frames.push(m(
  frames[frames.length-1],
  { canvases: { 1: { data: data } } },
  { formulas: {
      x: { content: 'x = '+x+'' },
      y: { content: 'y = '+y+'' }
  } }
));
```

Parsovacia trieda `FrameElement` by v takomto prípade vyzerala nejako takto:

```
class FrameElement < Element
  def start_print
    "frames.push(m(frames[frames.length-1], {"
  end

  def end_print
    "}));"
  end
end
```

Do nej ale potrebujeme vkladať dva kľúče `canvases` a `formulas`, ktoré musia byť uzatvorené a my nemáme garanciu, že užívateľ nebude na miesto dcérskych elementov elementu `Frame` **vpisovať neusporiadane** raz element `Graph`, raz element `Formula` a pod. Takisto nedokážeme v parseri naraz zvažovať, či daný element napr. grafu je elementom posledným a či kľúč `canvases` sa už nemá uzatvoriť. Parser by potreboval poznať nasledujúce znaky XML programu na určenie nasledujúcich elementov a to **nie je úlohou parseru**. Takýmto spôsobom by sme veci začali komplikovať. Miesto toho sa ale ponúka jednoduchá a šikovná myšlienka zavedenia XML značiek `Formulas` a `Graphs` určených pre obalenie jednotlivých okenných elementov `Formula` a `Graph` daného snímku. XML značky `Formulas` a `Graphs` môžu pri parsovaní blokovo otvárať a zatvárať svoje príslušné kľúče výsledného javascriptového objektu:

```
class FormulasElement < Element
  def start_print
    "formulas: {"
  end

  def end_print
    "},"
  end
end
```

Element `Formula` bude umiestňovať do otvoreného kľúča `formulas` reťazec typu `"x:  content: 'x = '+x+" "`. V prípade viacero formúl, musí za týmto

textovým reťazcom nasledovať čiarka. Tu je vhodné pozastaviť sa nad jedným dôležitým problémom, ktorý sa častokrát v programátorskej praxi opakuje: **Čo s poslednou čiarkou?** Tým, že sme oddelili parsované XML značky do jednotlivých tried, skrz tieto triedy nemáme prístup k nejakému globálnemu stavu parsera, v ktorom by sme si napríklad mohli pamätať, či danému matematickému výrazu predchádzal iný matematický výraz a či teda je písanie čiarky pred matematickým výrazom nevyhnutné. Najjednoduchšie by bolo, ak by existoval **znak**, ktorý by mazal svoj predošlý znak. Ten by sme mohli umiestniť na začiatku vráteného textového reťazca metódy `end_print` a tak by **mazal čiarku** po poslednej formuli. Ale on taký znak existuje! Je to znak s poradovým číslom 9 (resp. 8. od nultého indexu) ASCII tabuľky znakov a ide o znak `backspace` alebo `\b`. Testujeme teda jeho funkčnosť v jazyku Ruby a zisťujeme, že tento znak sa chová štandardne ako každý iný znak a svoj predošlý znak v textovom reťazci nemaže. Čo ale nie je, môže byť a tomuto procesu sami pomôžeme nahradením každej dvojice znakov, z ktorej druhým znakom je znak `\b`, prázdnym znakom. To uskutočnujeme v kontroléri `AlgorithmsController` regulárnym výrazom a metódou `gsub` aplikovanou na výslednom textovom reťazci javascriptového kódu nami kompilovaného XML programu nasledovne:

```
gsub(/^[\b]+|([^\b](\g<1>)*[\b])/, '')
```

Lexikálna jednotka `\g<1>` v regulárnom výraze je vyjadrenie volania tzv. **subrutiny**. V tomto prípade subrutina volá regulárny výraz prvej zachytenej skupiny (`([^\b](\g<1>)*[\b])`). Metódy parsovacích tried `FormulasElement` alebo obmenene `GraphsElement` vyzerajú po doplnení našej myšlienky so znakom `\b` nasledovne:

```
def start_print
  "formulas: { "
end

def end_print
  "\b},"
end
```

Všimnime si **taktické umiestnenie medzery** na konci návratového reťazca v metóde `start_print` pre prípad, že značka `<Formulas>` nemá žiaden dcérsky element.

3.8.1 Efektivita parseru

Naša predošlá snaha o lepšiu kontrolu tých okien, ktorých stav sa má simultánne meniť pri zmene snímkov animácie sa nám podarilo vytvoriť niekoľko nových značiek, ktoré začínajú mierne komplikovať zápis našich XML programov. Už predošlý XML kód potrebujeme zapisovať takto:

```
<Frame>
  <Formulas>
    <Formula content="x = @{x}" name="x" />
```

```
    <Formula content="y = @{y}" name="y" />
  </Formulas>
  <Graphs>
    <Graph name="1" ref="data">
      <Marker x="x" />
    </Graph>
  </Graphs>
</Frame>
```

a v prípade ak pracujeme iba s individuálnym oknom grafu alebo matematického výrazu, musíme ho obaľovať do XML značiek `Frame` aj `Formulas` resp. `Graphs`. Chceme **zjednodušiť zápis XML programov** pre užívateľa a zvýšiť tak efektivitu a konsekventne aj obľúbenosť nášho jazyka. To napríklad spôsobom, že užívateľ nemusí používať značku `Frame` v prípade, že chce meniť stav iba jediného okna. Zároveň ale nechceme komplikovať náš parser nastavovaním zložitých podmienok, ktoré si pamätajú, či sme práve vstúpili do značky `Frame` a ak nie tak doplňujeme návratový reťazec triedy `FormulaElement` a ak áno, tak ho nedoplňujeme... Ideálne potrebujeme aby sme podporovali aj jednoduchší zápis XML programov, aby sa ten jednoduchší zápis transformoval do zložitejšie, ktorý je následne parsovaný. K tomu potrebujeme nástroj na zmenu XML programu alebo zápisu na iný XML zápis a máme šťastie, pretože takýto **nástroj** existuje - volá sa **XSLT**.

3.8.2 XSLT Transformácia

XSLT je skratka pre eXtensible Stylesheet Language Transformation. Ako uvádza D. Tidwell [12], motiváciou k vytvoreniu XSLT bolo vytvorenie nástroja, ktorý dokáže **transformovať dáta** zapísané vo formáte XML do formátov, ktoré môžu definovať spôsob zobrazenia daných dát a ktorého formát je zároveň dostupný pre niekoho, od koho nie je očakávaná znalosť programovať. Zatiaľ čo primárnou doménou XML je dátový obsah a jeho sémantický alebo významový popis, XSLT umožňuje transformáciu XML dokumentov do ich prezentačných variantov ako PDF, SVG, VRML, Java, JPEG atď. ale aj do iného dokumentu XML. Štýlopis XSLT je sám o sebe XML dokumentom čo je často využívané pre transformáciu XSLT štýlopisov do iných štýlopisov.

Predtým než sa pustíme do návrhu štýlopisu našej XSLT transformácie potrebujeme definovať jej požiadavky. Našim prvotným cieľom je zabezpečiť aby bol každý element `Graph` a `Formula`, ktorý sa už raz nenachádza v elemente `Frame` **individuálne obalený** elementom `Formulas` resp. `Graphs`, teda elementom svojho plurálu, a elementom `Frame`. Takisto chceme docieliť to, že užívateľ vôbec nebude musieť používať elementy `Formulas` a `Graphs` a teda, že jednotlivé dcérske elementy elementu `Frame` budú **vytriedené do elementov svojích skupín**. Pred integráciou do našej aplikácie testujeme XSLT transformáciu nástrojom `xsltproc`. Vytvárame štýlopis `frames.xslt` a ten aplikujeme na testovací súbor `test.xml` príkazom `xsltproc frames.xslt test.xml`. Prvým cieľom transformácie je vôbec vypísať pôvodný dokument, nakoľko transformácia nemá pôvodný dokument nahradiť úplne novým iným dokumentom, ale skôr len mierne upraviť. To dosahujeme nasledovnou šablónou identity vnútri štýlopisu:

```
<xsl:template match="node() | @*">
  <xsl:copy>
    <xsl:apply-templates select="node() | @*" />
  </xsl:copy>
</xsl:template>
```

Šablóna prechádza všetkými elementami a ich atribútmi, kopíruje ich a následne aplikuje šablóny na všetky dcérske elementy a ich atribúty. Šablóna takto prechádza celým XML dokumentom. To ďalšie čo chceme transformovať sú všetky tie elementy `Graph` a `Formula`, ktoré nepatria rodičovskému elementu `Frame`. Nasledujúca šablóna nájde s využitým technológie **XPath**, využitej v atribúte `match`, všetky takéto elementy, získa textový reťazec ich množného čísla a následne ich obalí elementom `Frame` a v závislosti od nájdeného elementu elementom `Graphs` alebo `Formulas`.

```
<xsl:template match="*[not(self::Frame)]/Graph|Formula">
  <xsl:variable name="el" select="concat(name(), 's')" />

  <xsl:element name="Frame">
    <xsl:element name="{$el}">
      <xsl:apply-templates select="node() | @*" />
    </xsl:element>
  </xsl:element>
</xsl:template>
```

To čo nám ostáva je už len vytriediť elementy `Graph` a `Formula` v prípade ak sú vopred umiestnené v elemente `Frame` do ich skupinových elementov. Hľadáme teda element `Frame` a v ňom vytvárame elementy `Formulas` a `Graphs` pričom do nich kopírujeme tie elementy `Formula` a `Graph`, ktoré sú priamo dcérskymi elementami nájdeného elementu `Frame` (`<xsl:copy-of select="Formula"/>`) ale zároveň aj tie elementy, ktoré už boli obdržané v elemente `Formulas` (`<xsl:copy-of select="Formulas/Formula"/>`). Analogicky vykonávame to isté pri elementoch grafov.

```
<xsl:template match="Frame">
  <xsl:copy>
    <xsl:if test="Formula|Formulas">
      <Formulas>
        <xsl:copy-of select="Formula"/>
        <xsl:copy-of select="Formulas/Formula"/>
      </Formulas>
    </xsl:if>
    <xsl:if test="Graph|Graphs">
      <Graphs>
        <xsl:apply-templates select="Graph" />
        <xsl:apply-templates select="Graphs/Graph"/>
      </Graphs>
    </xsl:if>
```

```
  </xsl:copy>
</xsl:template>
```

So znalosťou XSLT transformácie sa cítime posilnení a hneď nám napadajú **dve ďalšie vylepšenia**. Doteraz sme vedeli definovať pre objekt grafu jediný vodiaci bod. Podobne ako sme v javascriptovom objekte snímkov definovali kľúče pre matematické výrazy a grafy, definujeme kľúč `markers` pre objekt grafu (`canvases: { 1: { markers: [] }}`). Narozdiel od okien formúl a grafov pre nás nie je potrebné aby element vodiaceho bodu mal meno a preto pod kľúčom `markers` nebude kolekcia ďalších kľúčov ale **pole objektov vodiacich bodov**. Po pridaní kľúča `markers`, pridávame parsovaciu triedu `MarkersElement`, ktorá otvára a zatvára pole vodiacich bodov grafu, analogickú k `FormulasElement` a `GraphsElement`. Nakoľko nechceme aby užívateľ musel zbytočne zabaľovať elementy vodiacich bodov do elementu `<Markers></Markers>` aj pre tento prípad vytvoríme šablónu v našej XSLT transformácii:

```
<xsl:template match="Graph" name="graph">
  <xsl:copy>
    <xsl:apply-templates select="*[not(self::Marker)] | @*"/>
    <xsl:if test="name(.) = 'Graph'">
      <xsl:element name="Markers">
        <xsl:copy-of select="Marker"/>
      </xsl:element>
    </xsl:if>
  </xsl:copy>
</xsl:template>
```

Tá kopíruje všetky dcérske elementy elementu `Graph` okrem elementov `Marker`. Nakoľko pri volaní šablóny po mene sa neoveruje správnosť jej cesty v atribúte `match`, potrebujeme overovať meno samotné elementu, na ktorý bola šablóna volaná. Následne vytvárame element `Markers`, do ktorého umiestňujeme všetky elementy `Marker`.

Jedným z problémov bolo, že pre grafy, ktoré sa nenachádzajú v elemente `Frame`, šablóna pre nastavenie vodiacich bodov takýmto grafom nepridáva element `Markers`. Po dlhšom skúmaní sme zistili, že je to dané tým, že pre daný, už raz nájdený element grafu, sa iné šablóny netestujú. Nahradzujeme teda volanie `<xsl:apply-templates select="node() | @*"/>` vnútri **šablóny pre nájdenie grafov** a formúl bez rodičovského elementu `Frame` na `<xsl:call-template name="graph"/>`. A pridávame šablóne pre vytvorenie elementu `Markers` meno `graph`.

Vodiacim bodov tiež pridávame booleovský atribút `vertical`, ktorý určuje, či sa k bodu má zobrazovať **vertikálna vodiaca čiara** smerujúca až k spodnej hrane plátna grafu. Jej zobrazovanie implementujeme v metóde `renderMarkers` komponentu `Graph` jednoducho:

```
if (vertical) {
  ctx.beginPath();
  ctx.moveTo(cX, h - bH);
  ctx.lineTo(cX, cY);
```

```
  ctx.lineWidth = 1;

  // set line color
  ctx.strokeStyle = '#ff0000';
  ctx.stroke();
  ctx.strokeStyle = '#000000';
}
```

Kde premenné `cX` a `cY` sú absolútne súradnice vodiaceho bodu a rozdiel `h - bH` je celková výška plátna bez spodného odsadenia.

Posledným zásadným vylepšením našich transformácii je nastavenie **zobrazovaného rozsahu grafu**. Je zložité pri nastavovaní rozsahu rozmýšľať o veľkosti intervalu a potom o posune. Väčšine ľudí si predstaví pri rozsahu minimálne a maximálne hodnoty jednotlivých súradníc, ktoré určujú ich zobrazovaný interval. Našim cieľom je teda dosiahnuť to, aby užívateľ nemusel používať element `<Shift />` a mohol rozsah aj s posunom definovať v elemente `<Range />` nasledujúcim spôsobom:

```
<Range x="-1..2" y="0..3" />
```

Všetko čo nám stačí urobiť je vypočítať rozdiel dvoch čísel v atribúte danej súradnice čím získame veľkosť intervalu. Tú zapíšeme do atribútu toho istého názvu ale atribútu nového elementu `Range`. Posunom je prvé číslo rozsahu. K rozdeleniu textového reťazca používame XSLT funkcie `substring-after` a `substring-before` v prípade počítania veľkosti intervalu nasledovne:

```
<xsl:value-of
     select="substring-after(., '..') - substring-before(., '..')" />
```

To ale vykonávame iba v prípade, že hodnota atribútu obsahuje **dve bodky**, inak vraciame tú istú hodnotu atribútu. Pre lepšiu orientáciu uvádzame celú finálnu verziu šablóny pre transformáciu elementu `<Range />`.

```
<xsl:template match="Range">
  <xsl:element name="Range">
    <xsl:for-each select="@*">
      <xsl:attribute name="{name()}">
        <xsl:choose>
          <xsl:when test="contains(., '..')">
            <xsl:value-of select="
              substring-after(., '..') - substring-before(., '..')" />
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="." />
          </xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
    </xsl:for-each>
  </xsl:element>
```

```
  <xsl:element name="Shift">
    <xsl:for-each select="@*">
      <xsl:attribute name="{name()}">
        <xsl:choose>
          <xsl:when test="contains(., '..')">
            <xsl:value-of select="substring-before(., '..')" />
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="0" />
          </xsl:otherwise>
        </xsl:choose>
      </xsl:attribute>
    </xsl:for-each>
  </xsl:element>
</xsl:template>
```

3.9 Zjednodušenie značiek `Var` a `set`

Pri všetkých doterajších zápisoch algoritmov a programov v nami tvorenom XML jazyku, sme pre každú operáciu deklarácie premennej alebo priradenia hodnoty k premennej museli zakaždým použiť značku `Var` alebo `set`. To občas vytváralo celkom **zdĺhavý zápis** sekvencie značiek **deklarácii a priradení** ako napr.

```
<Var name="alpha" value="0.9" />
<Var name="new_x" />
<Var name="new_y" />
<Var name="p" />
<Var name="t" value="1.0" />
<Var name="t_min" value="0.00001" />
<Var name="x" />
...
```

alebo

```
<set name="gradientx" value="dfx(x)" />
<set name="gradienty" value="dfy(y)" />
<set name="x" value="x - lrate * gradientx" />
<set name="y" value="y - lrate * gradienty" />
```

Na základe odporúčania konzultanta našej práce a pre praktické dôvody efektívnosti zápisu algoritmov sme sa **rozhodli skrátiť** niekoľkonásobný zápis deklarácií premenných a zápis priradenia hodnoty premenným tak aby v ideálnom prípade vyžadoval iba **jediné použitie značky** `Var` resp. `set`. Nami pôvodne zamýšľaná myšlienka mala najskôr takúto podobu:

```
<Var names="a, b, c" values="1, 'string', a + 1" />
```

Pri takejto podobe deklarácii ale **strácame prehľad** o tom, akej premennej je priradená aká hodnota, čo znižuje efektivitu práce v danom jazyku. Nakoľko vieme, že v SAX parseri máme okamžitý prístup k atribútom elementov a k ich menám, núka sa nám zrejmé návrhové riešenie:

```
<Var a="1" b="'string'" c="a+1" />
```

V takomto riešení pri deklarované premenné bez priradenej hodnoty by mohli byť prázdnymi atribútmi (napr. `<Var a />`). Podľa štandardu XML [13] ale takáto **definícia prázdneho atribútu nie je odporúčaná** a aj po našom odskúšaní, môžeme potvrdiť, že pri parsovaní je **ignorovaná**. Deklarované atribúty so žiadnou vopred definovanou hodnotou sme nútení značiť spôsobom `<Var a=""/>`. Ako teda bude vyzerať metóda parsovacej triedy nášho elementu `<Var />`? Vieme, že v triede `AlgorithmDocument` priradíme atribúty elementov do inštančných premenných inštancií ich jednotlivých parsovacích tried. Stačí nám teda prechádzať všetkými inštančnými premennými, získať ich meno, hodnotu a pre každú vrátiť reťazec deklarajúci premennú v Javascripte. Tieto reťazce potrebujeme následne zlúčiť. Takto teda vyzerá naša metóda `start_print`:

```
def start_print
  self.instance_variables.map do |variable|
    value = self.instance_variable_get(variable)
    "let #{variable.to_s.sub(/^@/, '')}
     #{(value.present? ? " = #{value}" : "")};"
  end.join
end
```

Názov inštančnej premennej v jazyku Ruby v sebe zahrňuje aj znak `@` určujúci inštančnú premennú. Ten odstránime jeho nahradením za žiaden znak. Overujeme, či hodnota atribútu je prítomná. Metóda `present?` vracia hodnotu `false` aj v prípade, že testovaná premenná je prázdnym reťazcom. V závislosti od hodnoty atribútu premennej priraďujeme hodnotu aj v javascriptovom kompilovanom kóde. Deklarácie individuálnych premenných zlučujeme metodou `join` ponad metódu `map`, ktorá prechádza poľom inštančných premenných a vracia nové, upravené pole.

Pri definícii značky `set`, parsovacia trieda a jej metóda `start_print` funguje analogicky. Značka pozná iba jeden špeciálny prípad na rozdiel od značky `Var`. Pri implementácii násobného algoritmu sme sa stretli s priradením hodnoty do položky poľa `<set name="v[1][0]"value="v[1][0] / v[minI][0]"/>`. Takáto položka poľa ale nemôže byť názvom atribútu XML značky, nakoľko atribút vo svojom názve nemôže obsahovať podľa XML špecifikácie [13] hranaté zátvorky. Potrebujeme preto nejakých spôsobom zachovať tzv. *fallback* pre takéto prípady a tým najjednoduchším spôsobom, ktorý sa ponúka, je zachovanie priradenia pomocou atribútov `name` a `value`. V parsovacej triede značky `<set />` teda nastavujeme, že v prípade ak značka má oba atribúty `name` aj `value`, tak všetky ostatné atribúty ignoruje a priraďuje do identifikátora v atribúte `name` hodnotu atribútu `value`.

3.10 Implementácia násobného algoritmu

S využitím novej značky `Formula` a nového zápisu značiek `Var` a `set` implementujeme v našom XML jazyku na záver našej práce násobný algoritmus alebo metódu násobnej iterácie. Ten ako sme už popisovali, nachádza **postupným násobením matice dominantný vlastný vektor** a k nemu príslušné vlastné číslo. V algoritme si môžeme všimnúť sémantický posun značky `Function`, ktorá už neslúži len pre vykresľovanie funkcií ale je používaná aj v rámci programu resp. v rámci nášho algoritmu ako funkcia. Ďalšie nami implementované algoritmy je možné nájsť v prílohe práce.

```
<!-- Deklarácia potrebných premenných -->
<Var i="0" lambda="" matrix="[[2, -12], [1, -5]]" v="[[1], [1]]" />

<!-- Deklarácia potrebných funkcií -->
<Function name="product" formula="
  [ [m[0][0]*v[0][0] + m[0][1]*v[1][0]],
    [m[1][0]*v[0][0] + m[1][1]*v[1][0]] ]" />
<Function name="eigenvalue" formula="a[0][0] / v[0][0]" />

<!-- Hlavný cyklus násobného algoritmu -->
<while cond="i < 70">

  <!-- Počítame nový vektor v -->
  <set v="product(matrix, v)" />

  <!-- Vo vektore 'v' nájdeme hodnotu
       s minimálnou absolútnou vzdialenosťou od nuly -->
  <Var min="Number.MAX_SAFE_INTEGER" minI="0" j="0" />
  <while cond="j < v.length">
      <Var abs="Math.abs(v[j])" />
      <if cond="abs < min">
        <set min="abs" minI="j" />
      </if>
      <set j="j + 1" />
  </while>

  <!-- Delíme hodnoty vektoru tak
       aby na indexe tejto hodnoty bola 1 -->
  <Var l="0" />
  <while cond="l < v.length">
    <set name="v[l][0]" value="v[l][0] / v[minI][0]" />
    <set l="l + 1" />
  </while>

  <!-- Inkrementujeme iterátor a vypočítame
       novú hodnotu vlastného čísla -->
  <set i="i + 1" lambda="eigenvalue(product(matrix, v), v)" />

  <!-- Vykreslíme matematický zápis definície vlastného čísla a
       vlastného vektoru dosadený o naše hodnoty -->
```

```
  <Formula name="f" content=" \left( \begin{array}{cc}
      @{matrix[0][0]} \& @{matrix[0][1]} \\
      @{matrix[1][0]} \& @{matrix[1][1]}
    \end{array} \right) \left( \begin{array}{c}
      @{v[0][0]} \\ @{v[1][0]}
    \end{array} \right) = @{lambda} \cdot \left( \begin{array}{c}
      @{v[0][0]} \\ @{v[1][0]}
    \end{array} \right)" x="100" y="100" />
</while>
```

4. Užívateľská dokumentácia

V nasledujúcej kapitole sa pokúsime popísať praktické aspekty inštalácie, spustenia a používania nástroja, ktorý sa nám podarilo v predošlej kapitole navrhnúť a implementovať.

4.1 Popis inštalácie

K lokálnemu spusteniu aplikácie potrebujeme mať nainštalovaný programovací jazyk Ruby, framework Rails, v prostredí ktorého je aplikácia vytvorená a javascriptové behové prostredie. K inštalácii jazyka ruby použijeme jeho interpreter, manažér verzií a behové prostredie rbenv. K aplikácii je pripojený inštalačný súbor `install-debian.sh` určený najmä pre unixové systémy stavané na distribúcii debian a vytvorený na základe inštalačného postupu L. Tagliaferri [14]. Inštalačný skript obnoví odkazy na inštalačné balíčky operačného systému, inštaluje behové prostredie jazyka ruby rbenv, inštaluje samotný jazyk ruby a to jednu z posledných stabilných verzií 2.4.1, inštaluje ruby balíček alebo tiež tzv. gem s názvom `bundler` čo je manažér inštalovaných ruby balíčkov a ich závislostí a balíček Rails čo je webový framework, v ktorom je naša aplikácia naprogramovaná. Vyhneme sa sťahovaniu dokumentácii k jednotlivým balíčkom, čo nám ušetrí mnoho času. Ako doplnok inštalačný skript inštaluje javascriptové behové prostredie pre potreby niektorých ruby balíčkov, určených hlavne na interpretáciu a kompiláciu javascriptu a jeho syntaktických foriem na strane klienta v rámci frameworku Rails.

Skript spúšťame, po priradení práv na exekúciu, z príkazového riadku unixového systému distribúcie debian nasledovne:

```
chmod +x install-debian.sh
sh install-debian.sh
```

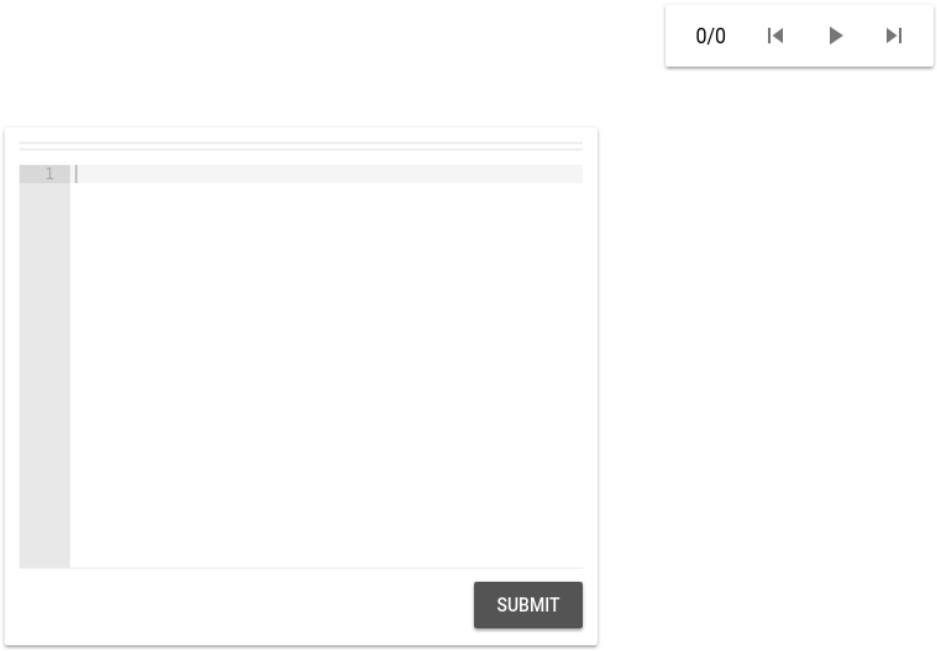
Aplikácia je určená predovšetkým k behu na vyhradenom serveri a dobre funguje v kombinácii so softvérovým webovým servrom Nginx.

4.1.1 Spustenie aplikácie

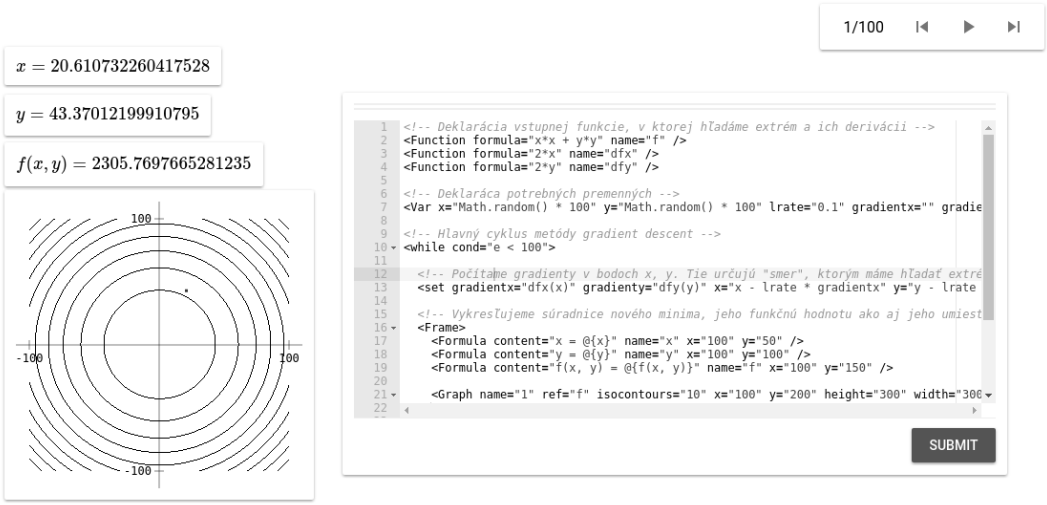
Aplikáciu v adresári spúšťame príkazom `rails s`, ktorý spúšťa beh lokálneho servera našej aplikácie. K tej pristupujeme prostredníctvom webového prehliadača na lokálnej adrese http://localhost:3000.

4.2 Ovládanie prostredia

Po spustení aplikácie vidíme okno editora ACE, do ktorého môžeme písať zadanie algoritmu v našom XML jazyku a kontrolný panel animácie, ktorý môžeme označiť tiež aj „panel prehrávania“, s intuitívne známymi tlačidlami dozadu, dopredu a prehrávať. Tie kontrolujú prehrávanie animácie algoritmu. Vľavo od tlačidiel si môžeme všimnúť popisok „0/0“, ktorý určuje číslovanie momentálneho snímku animácie z celkového počtu snímkov.



Obrázek 4.1: Prostredie aplikácie po spustení programu



Obrázek 4.2: Prostredie aplikácie po načítaní algoritmu gradient descent ilustrovaný na funkcii dvoch premenných

Okná je možné po uchopení myšou presúvať ľubovoľne po obrazovke. V prípade okna XML editora môžeme okno za účelom presúvania uchopiť iba za jeho hlavičku resp. uchopovaciu lištu označenú dvoma vodorovnými čiarami podobným spôsobom ako v okenných prostrediach väčšiny operačných systémov. Po uchopení plochy je možné hýbať celou pracovnou plochou prostredia.

Uchopením jedného z pásiem pravej alebo spodnej hranice okna môžeme okno podľa potreby zväčšovať alebo zmenšovať v jeho šírke alebo výške. Princíp je opäť analogický ako v okenných prostrediach väčšiny operačných systémov. Veľkosť okien matematických výrazov z dôvodu potreby rozsiahlejšej implenentácie dynamického osadzovania textu matematického výrazu reagujúceho na zmenu veľkosti okna je nemožné meniť.

Tlačidlom „Submit“ kompilujeme algoritmus zapísaný v našom XML jazyku. Nezmenený popisok „0/0“ v paneli prehrávania i po odoslaní XML zápisu nášho algoritmu indikuje chybovosť v kompilovanom kóde algoritmu alebo v jeho samotnom XML zápise. V prípade úspešnej kompilácie nášho XML kódu sa nám v prostredí objavia nám požadované okná a pripraví sa animácia algoritmu, ktorú následne môžemme prehrávať prostredníctvom tlačidiel v paneli prehrávania.

4.3 Tvorba prvých programov

V nasledujúcej sekcii si demonštrujeme tvorbu XML programov v našom vizualizačnom prostredí. Po spustení aplikácie vidíme na plátne umiestnené okno XML editora. V tomto editore budeme zapisovať náš program. Vyskúšajme si vytvoriť prvý jednoduchý program *Hello world* vypísaním nasledujúcej značky do editora.

```
<Formula content="Hello\ world!" name="hello" />
```

Po stlačení tlačidla „Submit“ sa značka resp. program kompiluje a na obrazovke sa nám objaví nami definované nové okno s obsahom „Hello world!“. Takýmto spôsobom môžeme v našej aplikácii tvoriť programy. Nami vytvorený program „Hello world“ pozostával iba z jediného snímku a to pre nás nie je zaujímavé. Skúsme si vytvoriť program pozostávajúci z viacerých snímkov a to nasledovným spôsobom:

```
<Var i="0" str="’’" />
<while cond="i < 5">
  <set str="str + ’hello\\ ’" />
  <Formula content="@{str}" name="hello" />
  <set i="i + 1" />
</while>
```

V programe deklarujeme premenné iterátora a prázdneho textového reťazca a uskutočňujeme chod cyklu, ktorého vnorené príkazy bežia presne 5-krát. V ňom pripájame k hodnote existujúceho reťazca `str` reťazec `"hello "`. Nakoľko reťazec je vykresľovaný ako reťazec jazyka TeX, k napísaniu medzery potrebujeme medzeru uviesť lomítkom. Nakoľko sa textový reťazec kompiluje do jazyka Javascript, potrebujeme samotný znak lomítka odsadiť ďalším lomítkom. Na premennú sa

vrámci obsahu okna matematického reťazca v atribúte `content` odkazujeme konštruktom `@{mojapremenna}`.

Po stlačení tlačidla „Submit“ si všimneme zmenu v paneli prehrávania v pravom hornom rohu obrazovky. Celkový počet snímkov je teraz päť a tlačidlami dopredu a dozadu môžeme jednotlivými snímkami prechádzať a tak meniť hodnotu textového reťazca, ktorý sa vykresľuje v pohybovateľnom okne na našej obrazovke. Počet snímkov je určený počtom výskytov vizualizačných elementov `Formula` a `Graph` v chode programu.

Skúsme si vyskúšať zobraziť práve graf funkcie a to hneď graf funkcie dvoch premenných:

```
<Function formula="Math.pow(x, 3) + Math.sin(y)" name="f" />
<Graph name="1" ref="f" />
```

Môžeme si všimnúť, že značka `Graph` odkazuje na vykreslenie premennej `f`. V nej je uložená funkcia matematického výrazu uvedeného v značke `Function`. Po odoslaní programu, sa nám zobrazí okno grafu, s ktorým môžme pracovať.

Môžeme taktiež nastaviť rozsah zobrazovanej funkcie v grafe. Doteraz náš graf zobrazuje nami určenú funkciu na intervaloch $[0, 1]$. Tieto intervaly, ale môžeme meniť v atribútoch vnorenej značky `Range`, tak ako popisujeme v nasledujúcom príklade. Všimnime si v ňom tiež nastavenie hodnoty atribútu `isocontours`, ktorá nám definuje počet izokontúr, resp. čiar grafu, ktoré sa budú vykresľovať. Po stlačení tlačidla „Submit“ vidíme okno iného grafu, narozdiel od predchádzajúceho príklad, ale ide o graf tej istej matematickej funkcie.

```
<Function formula="Math.pow(x, 3) + Math.sin(y)" name="f" />
<Graph isocontours="20" name="1" ref="f">
  <Range x="-1..1" y="-1..1"></Range>
</Graph>
```

Kombináciou predošlých príkladov, môžeme vykresliť naraz okno matematického výrazu a okno grafu. Ich nové načítanie vrámci programu sa štandardne premietne do vytvorenia novej snímky. Môžeme ale vrámci jedinej snímky meniť obe okná a to umiestnením ich značiek do spoločnej značky `Frame`. Značka `Graph` tiež pozná niečo ako dcérsku značku `Marker`. Tá určuje vodiaci bod alebo čiaru na grafe a v nasledujúcom príklade sa tento bod bude pohybovať po grafe v závislosti od iterátora cyklu. Skúsme si nechať zkompilovať nasledujúci príklad a následne prechádzať jednotlivými snímkami animácie. Pre lepšie porozumenie fungovania značky `Frame`, môžeme skúsiť kompilovať ten istý program, v ktorom sa značky `Formula` a `Graph` nebudú nachádzať v značke `Frame`.

```
<Function formula="Math.pow(x, 3) + Math.sin(y)" name="f" />
<Var i="0" str="''" />
<while cond="i < 6">
  <set str="'['+i/5+','+i/5+']'" />
  <Frame>
    <Formula content="@{str}" name="hello" x="100" y="100" />
    <Graph isocontours="20" name="1" ref="f" x="100" y="150">
      <Range x="-1..1" y="-1..1"></Range>
```

```
      <Marker x="i/5" y="i/5" />
    </Graph>
  </Frame>
  <set i="i + 1" />
</while>
```

Na jednoduchých príkladoch sme sa oboznámili so základmi fungovania aplikácie a nášho programovacieho jazyka založeného na syntaxi XML. Demonštrovaná funkcionalita ale nie je všetkým čo sa môže vrámci aplikácie dosiahnuť a preto prikladáme detailnú dokumentáciu všetkých XML značiek určených k použitiu. Rovnako, v prílohe na konci práce môžeme nájsť zložitejšie XML programy nami v práci použitých algoritmov, na ktorých demonštruujeme funkčnosť nášho XML jazyka a jeho vizualizačného prostredia. Ich funkčnosť si môžeme takisto vyskúšať a to vkladaním ich kódu do XML editora aplikácie a jeho následnou kompiláciou, spôsobom, ktorý sme si už stihli v tejto sekcii popísať na jednoduchších príkladoch programov.

4.4 Značky

K zápisu algoritmu v XML jazyku využívame vlastné definované značky, ktoré môžeme rozdeliť na vizualizačné a programové, pričom programové môžeme ešte deliť na deklaračné a riadiace resp. kontrolné. Vizualizačné značky sú `Formula`, `Graph` a `Frame` a podznačky grafu `Marker`, `Range` a `Shift`. Programové deklaračné sú `Array`, `Function` a `Var` a kontrolné, pre lepšie odlíšenie začínajúce malým písmenom, `if`, `set` a `while`.

Niektoré značky obsahujú atribút `name`. Ten je pri programových značkách ako `Array`, `Function` a `set` použitý ako názov identifikátoru priradenej štruktúry, na ktorú sa môžeme jeho prostredníctvom odkazovať. Pri vizualizačných značkách je úlohou mena odlíšiť jednotlivé inštancie vizualizačných značiek. Napríklad, ak chceme zmeniť hodnotu obsahu už raz definovaného matematického výrazu `<Formula name="f1"content="x = 1"/>` tak naň musíme odkazovať znova prostredníctvom mena `f1`.

4.4.1 Značka `<Formula />`

Formula je značka slúžiaca k vykresleniu matematického výrazu, ktorý je zapísaný v jazyku TeX. Matematický výraz je predaný značke prostredníctvom atribútu `content` a jeho súčasťou môže byť aj odkaz na premennú použitú v algoritme. Ten je zapísaný skrz zástupný syntaktický konštrukt `@{meno}`, v ktorom „meno“ je identifikátor premennej.

```
<Formula content="x = @{x}" name="x" />
```

Atribút `content`

Atribút určený k predaniu reťazca matematického výrazu zapísaného v jazyku TeX

Atribút `name`

Povinný a jedinečný názov výrazu.

Atribút `ref`

Voliteľný odkaz na názov premennej, v ktorej môže byť uložená hodnota matematického výrazu. V prípade použitia nahradzuje obsah atribútu `content` hodnotou premennej, na ktorú sa odkazuje.

Atribúty `x, y`

Atribúty `x` a `y` určujú pozíciu x-ovej resp. y-ovej súradnice vykresleného okna matematického výrazu.

4.4.2 Značka `<Graph></Graph>`

Slúži k vykresleniu grafu. Môže vykresľovať pole bodov ako aj odkazovanú funkciu. V oboch prípadoch je pole alebo funkcia predaná značke `Graph` ako názov jej identifikátora prostredníctvom atribútu `ref`. Značka `Graph` má dve vnorené značky `Marker` a `Range`, pričom prvá definuje vizuálny znak, terč v grafe, ktorým chceme zvýrazniť vybraný bod alebo funkčnú hodnotu a značka `Range` definuje rozsah funkcie, resp. tú jej časť ktorú chceme zobrazovať.

Atribút `isocontours`

Atribút definujúci počet kontúr funkcie, ktoré chceme zobraziť. Je využitý iba v prípade zobrazovania funkcie dvoch premenných.

Atribút `name`

Povinný a jedinečný názov grafu.

Atribút `ref`

Povinný odkaz na názov premennej, ktorá definuje buď pole alebo funkciu, ktorú chceme vykresliť.

Atribúty `x, y`

Atribúty `x` a `y` určujú pozíciu x-ovej resp. y-ovej súradnice vykresleného okna grafu.

Atribúty `height, width`

Atribúty `height` a `width` určujú výšku resp. šírku vykresleného okna grafu.

4.4.3 Značka `<Marker />`

Definuje vizuálne označenie bodu v grafe. Je to voliteľná vnorená značka značky `Graph`.

Atribút `x`

Povinný atribút. Určuje pozíciu x-ovej súradnice vykreslovaného bodu vrámci grafu. Pokiaľ graf vykresľuje pole hodnôt, hodnota je interpretovaná ako index poľa. Ak graf vykresľuje funkciu, hodnota je interpretovaná ako skutočná vstupná hodnota funkcie.

Atribút `y`

Voliteľný atribút. Určuje pozíciu y-ovej súradnice vykreslovaného bodu vrámci grafu. Ak nie je určený tak jeho hodnotou sa automaticky stáva funkčná hodnota vykresľovaného poľa alebo funkcie grafu na základe atribútu `x`.

Atribút `vertical`

Voliteľný boolovský atribút. Ak je prítomný tak dokresľuje vertikálnu čiaru vedúcu od vykresľovaného bodu k spodnej hrane plátna, na ktorom je vykresľovaný graf.

4.4.4 Značka `<Range />`

Určuje intervalový rozsah zobrazovanej časti grafu. Je to voliteľná vnorená značka značky `Graph`. Jej predvolené hodnoty sú 0 až 1 v oboch súradniciach.

Atribúty `x, y`

Oba atribúty sú voliteľné. Určujú intervalový rozsah zobrazovanej časti grafu v orientácii x-ovej, resp. y-ovej súradnice. Prijímajú hodnotu v dvoch formátoch a to ako celé alebo desatinné číslo napr.

```
<Range x="2" y="1.5" />
```

čo nastavuje intervalový rozsah zobrazovanej časti grafu na hodnoty 0 až 2 v x-ovom intervale a 0 až 1.5 v y-ovom intervale alebo vo formáte dvojice čísel oddelených dvoma bodkami napr.

```
<Range x="-2..2" y="1..1.5" />
```

čo určuje intervalový rozsah zobrazovanej časti grafu na hodnoty -2 až 2 v na x-ovej súradnici a 1 až 1.5 na y-ovej súradnici.

4.4.5 Značka `<Frame></Frame>`

Môže obaliť viacero značiek `Graph` a `Formula`, ktorých zmena sa má udiať simultánne vrámci jedinej snímky animácie. U tých značkách `Graph` a `Formula`, ktoré nie sú obalené spoločnou značkou `Frame` sa zmena premietne do individuálnej snímky pre každú značku.

4.4.6 Značka `<Array />`

Deklaruje pole určenej dĺžky. Využíva sa hlavne preto, že ho môže vyplniť náhodnými hodnotami. Samotné pole môžeme deklarovať aj ako premennú skrz značku `Var`.

Atribút `length`

Povinný atribút. Určuje dĺžku poľa.

Atribút `name`

Povinný názov poľa.

Atribút `random`

Boolovský atribút. Určuje či sa pole má vopred vyplniť náhodnými hodnotami rozsahu 0 až 1. Ak je jeho hodnota negatívna, tak sa pole vyplní hodnotami 0.

4.4.7 Značka `<Function />`

Slúži na deklaráciu funkcie. Značka bola pôvodne určená pre zápis funkcie v zmysle matematického výrazu vyjadreného v jazyku javascript, ale je ju možné použiť aj na deklaráciu funkcií ako takých. Funkcia je deklarovaná iba menom bez zápisu jej argumentov. Tie sú odčítané z obsahu funkcie a každá nová premenná je interpretovaná ako ďalší argument. Takto deklarovaná funkcia teda nepredpokladá deklaráciu nových premenných v jej obsahu. Poradie použitia premenných vo funkcii určuje poradie argumentov funkcie. Napríklad,

```
<Function name="eigenvalue" formula="a[0][0] / v[0][0]" />
```

značí funkciu `eigenvalue(a, v)`.

Atribút `formula`

Javascriptový zápis obsahu funkcie, teda jej výslednej návratovej hodnoty alebo matematického výrazu.

Atribút `name`

Povinný atribút určujúci identifikátor funkcie.

4.4.8 Značka `<Var />`

Deklaruje premenné s možnosťou priradenia hodnoty. Nemá žiadne vopred definované atribúty a ku každému atribútu sa správa ako k novej deklarovanej premennej. Ak premennej nechceme priradiť hodnotu, definujeme atribút ako prázdny.

```
<Var str="retazec" x="1" y="" z />
```

Premenná s názvom `z` v tomto prípade nie je deklarovaná ale premenná `y` je.

4.4.9 Značka `<if></if>`

Podmieňuje exekúciu vnoreného obsahu značky na základe podmienky vpísanej v atribúte `cond`.

Atribút `cond`

Atribút prijímajúci podmienku v podobe výrazu napísaného v jazyku javascript, ktorý vracia boolovskú hodnotu.

4.4.10 Značka `<set />`

Operátor priradenia hodnoty do už deklarovanej premennej alebo premenných. Podobne ako značka `Var` sa ku každému atribútu správa ako k premennej, ktorej je priradzovaná jej hodnota. Narozdiel od značky `Var` má ale špeciálny mód priradenia. Ak použijeme naraz atribúty `name` a `value`, značka priradí do identifikátora vloženého do atribútu `name` hodnotu atribútu `value` a všetky ostatné atribúty ignoruje. Tento mód je využívaný najmä vtedy ak priraďujeme hodnotu na konkrétne miesto v poli premenných, napr.

```
<set name="array[3]" value="3" />
```

4.4.11 Značka `<while />`

Podmieňuje opakovanú exekúciu vnoreného obsahu značky na základe podmienky vpísanej v atribúte `cond`.

Atribút `cond`

Atribút prijímajúci podmienku v podobe výrazu napísaného v jazyku javascript, ktorý vracia boolovskú hodnotu.

5. Programátorská dokumentácia

V nasledujúcej kapitole budeme technicky a podrobne popisovať nami vytvorenú aplikáciu vizualizačného prostredia a interpreta nášho XML jazyka. Budeme sa venovať prostrediu, v ktorom naša aplikácia beží, nástrojom, ktoré využíva a budeme charakterizovať jednotlivé súčasti aplikácie a ich funkcionalitu z pohľadu vývojára.

5.1 Nástroje a prostredie

Pri tvorbe našej aplikácie používame framework - aplikačný rámec s názvom Rails alebo Ruby on Rails. Pod pojmom „framework“ môžeme rozumieť rámec sofvérových riešení problémov, ktorých výskyt sa pri tvorbe istých vybraných druhov aplikácii opakuje. Framework nám poskytuje nástroje a konvencie v pohobe podporných programov, knižníc, rozhraní a rôznych návrhových vzorov.

Ako doplnok k frameworku Ruby on Rails používame framework React. Zatiaľ čo Rails nám poskytuje celkový rámec pre tvorbu aplikácii ako takých a je písaný v jazyku Ruby, React je klientsky framework poskytujúci rámec pre tvorbu aplikácii generovaných na strane klienta. Je potrebné odlišovať tzv. „backendovú“ a „frontendovú“ vrstvu aplikácie. Zatiaľ čo tzv. „backend“ pristupuje k dátam aplikácie a zabezpečuje ich spracovanie a zmeny, „frontendová“ vrstva aplikácie má zodpovednosť za spôsob zobrazovania pripravených dát. Súčasťou takéhoto zobrazenie je aj možnosť vizuálnej interakcie s dátami a práve to nám framework React uľahčuje. Framework React integrujeme do frameworku Ruby on Rails skrz adaptér alebo tzv. gem - balíček jazyka Ruby s názvom `react-rails` a s podporným balíčkom `webpacker`, ktorý slúži k manažmentu javascriptových modulov vo frameworku Rails.

V aplikácii tiež využívame Railsom importovanú knižnicu Nokogiri, ktorá nám sprístupňuje nástroje ako SAX parser alebo XSLT transformáciu na prácu s dátami vo formáte XML. Vďaka tomu sme schopný kompilovať zápisy algoritmov v našom XML jazyku do Javascriptu, cez ktorých ich následne vizualizujeme v našom prostredí.

5.1.1 Ruby on Rails

Ruby on Rails alebo skrátene Rails je, už ako sme spomenuli, framework slúžiaci na tvorbu webových aplikácii. Vybrali sme si ho na základe našej dobrej skúsenosti a efektívnosti jeho používania. Dôvody našej inklinácii k Railsu krátko popisujeme aj v nasledujúcej sekcii.

Prvá verzia frameworku bola publikovaná v roku 2005. Inováciou frameworku bolo zahrnutie nástrojov uľahčujúcich samotný proces vývoja webových aplikácii. Pre porovnanie, jeho konkurencie frameworkov písaných v jazykoch Java, PHP alebo .NET bola založená na poskytnutí tried a objektov k tvorbe webovej aplikácie a k ním určeného obrovského manuálu, ktorý popisoval ich použitie. Pri vývoji aplikácie písanej vo frameworku Rails, začína programátor s hotovou funkčnou aplikáciou, ktorá je pripravená k upravovaniu. Framework Rails je zameraný na

vývojára a riešenia vývojár vo frameworku Rails dnes nehľadá v rozsiahlych dokumentáciach ale na rôznych webových portáloch, ktoré zahŕňajú komunitu ľudí, ktorí sa zapájajú do tvorby riešenia daného problému. Tie sú niekedy následné implementované aj do samotného frameworku alebo do jeho podporných knižníc.

Ruby

Ďalším aspektom obľúbenosti frameworku Rails je programovací jazyk Ruby, v ktorom je písaný. Ide o objekto-orientovaný skriptovací jazyk. Ide o jazyk, ktorý slovami Rubyho je nepochopiteľné strohý [6] a jeho vlastnosťou je jednoduchá čitateľnosť, ktorá plynie z toho, že jazyk sa snaží nepoužívať mnoho špeciálnych znakov, bloky oddeľuje odsadzovaním a pri pomenovávaní tried a metód sa snaží používať deskriptívne, ničmenej krátke a jednoducho zapamätateľné názvy.

Princípy DRY a CoC

Základnými filozofickými princípmi frameworku sú princípy DRY - „Don't repeat yourself“ a CoC - „Convention over configuration“. Princíp DRY reprezentuje myšlienku, že každý kus kódu môže byť v aplikácii vyjadrený na jedinom mieste. Inými slovami princíp DRY popisuje, že tá istá logika je popísaná v aplikácii iba jediný krát v snahe zamedziť akejkoľvek duplikácii kódu. Zavedenie princípu DRY malo za cieľ predísť situácii, v ktorej v dôsledku zmeny napr. databázovej schémy bolo potrebné meniť kód aplikácie na viacerých miestach.

Princíp „Convention over configuration“ nadradzuje konvenciu - predvolené hodnoty aplikácie frameworku Rails nad potrebou ich nanovo konfigurovať pri každej novej aplikácii. Tento princíp okrem toho povzbudzuje vývojárov k nasledovaniu odporúčaných riešení. To podporuje čítateľnosť kódu aj lepšie porozumenie kódu vývojármi v tímoch. Samozrejme, Rails sprístupňuje aj možnosť vytvárania vlastných, nekonvenčných riešení. Aj vďaka tomuto princípu je Rails nástrojom, ktorý sa nesnaží kopírovať webové štandardy, ale sám ich aktívne vytvára.

Nokogiri

Nokogiri je názov knižnice importovanej vo frameworku Rails a písanej v jazyku Ruby, ktorej súčasťou sú nastroje pre spracovanie dát vo formáte XML. Ide hlavne o rôzne druhy parserov ako napr. parser štruktúry DOM, SAX parser, Push parser, nástroj XPath na vyhľadávanie obsahu v XML dokumentoch, nástroj pre XSLT transformácie a iné. Z knižnice využívame najmä SAX parser a nástroj pre XSLT transformáciu.

5.1.2 React

React je knižnica slúžiaca k vytváraniu komponentov. Komponentom chápeme nami definovaný HTML element doplnený o programovateľnú funkcionalitu. V našej aplikácii ho využívame pri realizácii interaktívneho prostredia slúžiaceho na vizualizáciu algoritmov. Rôzne druhy okien, ktoré používame v našom grafickom rozhraní môžeme priradiť práve rôznym komponentom vytvoreným v knižnici React. React využíva myšlienku tzv. „reaktívneho programovania“. Reaktívne

programovanie je programovacia paradigma, ktorej cieľom je zabezpečiť rýchlu reaktívnosť aplikácie pre užívateľa.

Knižica React využíva aj myšlienku tzv. funkcionálneho šablónovania („functional templating“), ktorá značí priraďovanie specifických funkcií jednotlivým komponentom. Písanie oddelenej funkcionality do nezávislých komponentov má za úlohu aj oddelenie zodpovedností programovej logiky, v anglickom originále „separation of concerns“.

Knižnica React využíva ECMAScript 6, čo je špecifikácia jazyka Javascript, a syntaktické doplnenie JSX. JSX nám umožňuje vkladať HTML kód priamo, bez použitia úvodzoviek označujúc začiatok textového reťazca, do javascriptového kódu. Špecifikácia ECMAScript 6 nie je adaptovaná všetkými webovými prehliadačmi a tak súčasťou balíčka `react-rails` je aj nástroj Babel, ktorý kompiluje nepodporované funkcie špecifikácie ECMAScript 6 do jazyka Javascript.

5.2 Kompilácia

Jednou zo zásadných súčastí našej aplikácie je kompilovanie XML kódu zápisu nášho algoritmu do Javascriptu. Tá sa uskutočňuje odoslaním HTTP požiadavky metódy `POST` s obsahom daného XML kódu predaného ku kompilácii na adresu webového servra našej aplikácie. Tá následne XML kód predáva nástrojom knižnice Nokogiri k spracovaniu. Kód najprv prechádza XSLT transformáciou, ktorá značky doplní alebo pozmení pre jednoduchší proces parsovania. Transformované XML je následne parsované SAX parserom, ktorý vracia javascriptový kód odpovedajúci nášmu zápisu algoritmu v jazyku XML. Ten kontrolér vracia ako súčasť odpovede na HTTP požiadavok vo formáte JSON čo je konvencia pre tzv. javascriptovú objektový zápis.

Obdržaný Javascriptový kód na strane klienta je následne exekuovaný komponentom `App` a algoritmom vygenerované snímky sú priradené do stavu klientskej aplikácie čo zabezpečí načítanie animácie. Proces obdržania príslušného javascriptového kódu k algoritmu je bližšie popísaný pri komponente `App` a jej metódy `submit`.

5.2.1 Cesty (Routes)

V predošlej sekcii sme popisovali odoslanie HTTP požiadavky na adresu našej aplikácie. Je potrebné povedať, že naša aplikácia pri jej spustení je spúšťaná cez webový server, ktorý takéto požiadavky môže prijímať na rôznych adresách. Podobu adries definujeme v súbore `config/routes.rb`.

V našom definičnom súbore týchto ciest `routes.rb` definujeme ako prvú koreňovú cestu a tá definuje mapovanie adresy `/` a všetkých požiadavkov metódy `GET` na akciu `index` kontroléra `WelcomeController`, ktorý tieto požiadavky následne spracováva a štandardne zobrazuje úvodnú stránku aplikácie, ktorú nájdeme na príslušnej ceste `views/welcome/index.html.haml`. V súbore `routes.rb` používáme tiež konštrukt `resources` na vytvorenie tzv. „RESTful“ ciest. Pod skratkou REST alebo v plnom znení „Representational State Transfer“ chápeme architektúru rozhrania, v ktorej časti programu bežia na rôznych strojoch. Pre ich komunikáciu je využívaná sada konvencií a súčasťou tejto sady konvencií je aj

odporúčanie ako by dostupné URL adresy daných častí programu mali vyzerať. Konštrukt `resources` vytvára 7 ciest:

URL	method	action
/algorithms/	GET	index
/algorithms/new	GET	new
/algorithms/	POST	create
/algorithms/:id	GET	show
/algorithms/:id/edit	GET	edit
/algorithms/:id	PATCH/PUT	update
/algorithms/:id	DELETE	destroy

kde „method“ je metóda prijímanej požiadavky a „action“ je názov príslušnej metódy alebo akcie kontroléra, ktorá spracuvúva HTTP požiadavok. Z konštruktu `resources`, v našom konkrétnom prípade nechávame generovať iba jedinú cestu `create` tá je mapovaná na akciu `create` kontroléra `AlgorithmsController`.

5.2.2 Kontrolér `AlgorithmsController`

`AlgorithmsController` je tzv. kontrolér alebo radič. Účelom kontrolérov je spracovávať užívateľské požiadavky posielané na dostupné adresy aplikácie, na ich základe pristupovať k dátam a tie v spracovanej podobe posielať šablónam k zobrazeniu. Metóda kontroléru je tiež nazývaná akciou. Vrámci frameworku Rails je každá akcia volá predvolene metódu `render` určenú na vykreslenie šablóny.

`create`

Metóda `create` je akciou kontroléru `AlgorithmsController` a jej cieľom je obdržať XML zápis algoritmu, transformovať ho, parsovať jeho transformovanú podobu a odpovedať HTTP požiadavku objektom vo formáte JSON zahrňujúcim kompilovaný algoritmus v jazyku Javascript.

Metóda `create` najprv inicializuje objekt parseru definovaného triedou `AlgorithmDocument` popísanou neskôr. Následne obaľujeme textový reťazec XML zápisu algoritmu značkou `<algorithm></algorithm>`, pretože parser aj XSLT štýlopis predpokladajú existenciu jediného koreňového algoritmu v XML dokumente, ktorý prijímajú na vstupe. Metódami `gsub` nahradzujeme ľavé a pravé špicaté zátvorky, ktoré sa môžu vyskytovať v atribúte podmienky značiek `while` a `if`. Nahradzujeme ich tzv. HTML entitami. Ide o reprezentácie špeciálnych znakov V HTML dokumente. Zabezpečujeme tak, že znaky špicatých zátvoriek nebudú misinterpretované sa znaky určujúce začiatok alebo koniec XML značky. V ďalšom riadku načítame súbor, štýlopis XSLT transformácie. S ním vzápätí transformujeme prijatý XML reťazec a výsledok transformácie ihneď parsujeme SAX parserom. Výsledok parsovania je ukrytý v inštančnej premennej `result` objektu `parser`. Pri parsovaní vkladáme pri vybraných značkách do výsledného kompilovaného javascriptu znaky
`b`, ktoré označujú „backspace“. Ide o trik, ktorým sa vyhneme zložitejšej programovaj logike v kóde parsovania individuálnych elementov. Znaky backspace majú za účel určiť ich predošlý znak k vymazaniu. Znaky backspace a znaky, ktoré ich predchádzajú mažeme metódou regulárnym výrazom, metódou `gsub`. V závere akcie vraciame objekt JSON obsahujúci pôvodne znenie algoritmu v jazyku XML a znenie algoritmu v kompilovanom kóde Javascript.

`algorithm_params`

Súkromná metóda `algorithm_params` je určená na filtráciu kontrolérom prijímaných parametrov a je súčasťou konvencie frameworku Rails s názvom „silné parametry“ (strong parameters). V našom prípade povoľujeme kontroléru prijať v tele HTTP požiadavky iba parameter `xml`.

5.2.3 XSLT Transformácia

XSLT alebo tiež eXtensible Stylesheet Language Transformation je nástroj pre transformáciu XML dokumentov. Na svojom vstupe prijíma tzv. rozšírený štýlopis alebo tiež súbor, ktorý popisuje transformáciu a dokument určený k transformácii. Následne upravuje podľa daného štýlopisu štruktúru a obsah XML dokumentu.

XSLT transformáciu vykonávame za účelom zjednodušiť proces parsovania XML dokumentu. Dovoľte mi uviesť príklad. Pri použití individuálnej značky `Formula` alebo `Graph` nemusíme značky obalovať značkou `Frame` pre určenie snímku. Ak chceme ale meniť viacero komponentov, okien typu `Formula` alebo `Graph` po prechode jedného snímku, potrebujeme značky obaliť značkou `Frame`. V kompilovanom kóde Javascript je ale výsledok rovnaký a v oboch prípadoch musíme predpokladať vytvorenie novej snímky. Aby sme v parseri nemuseli riešiť logiku existencie značky `Frame`, XSLT transformáciou všetky individuálne značky `Formula` a `Graph` značkou `Frame` obalíme a tak zabezpečíme jednoduchšiu podobu nášho parsera.

`lib/xslt/frames.xslt`

Pri transformácii v kontroléri `AlgorithmsController` načítavame ku XSLT transformácii súbor `lib/xslt/frames.xslt`. Súbor obsahuje XSLT štýlopis, podľa ktorého transformujeme XML zápis algoritmu. V jeho úvode špecifikujeme verziu XML a použitej XSLT transformácie. Nastavujeme tiež odsadzovanú formu výstupu. Štýlopis sa skladá zo šablón, ktoré po nájdení k nim zodpovedajúcich elementov v XML dokumente, tieto elementy transformujú a ich transformovanú štruktúru vracajú na výstupe. Prvá šablóna definovaná s menom `copy` kopíruje nájdené elementy a atribúty a overuje na ne všetky ostatné možné šablóny danej transformácie. Šablóna je hneď použitá na koreňový element XML dokumentu a vytvára tak tzv. transformáciu identity, pomocou ktorej kopírujeme celý dokument.

Tretia šablóna nachádza cez tzv. vyhľadávací formát značiek v dokumente - XPath, použitý v atribúte `match`, všetky tie značky `Graph` a `Formula`, ktoré nemajú rodičovský element `Frame`. Do premennej s názvom `el` si šablóna ukladá množné číslo názvu nájdeného elementu `Graph` alebo `Formula`. Šablóna vracia element `Frame` obsahujúci dcérsky element `Graphs` alebo `Formulas` v závislosti od pôvodnej nájdenej značky, v ňom nachádzajúcu sa pôvodne nájdenú XML značku, na ktorú je v prípade grafu ešte aplikovaná šablóna s názvom `graph`.

Štvrtá šablóna rieši situáciu lenivého užívateľa, ktorý do značky `Frame` ľubovoľne nahádzal značky `Graph` a `Formula`, popr. niektoré z nich umiestnil do ich skupinových elementov `Formulas` alebo `Graphs`. Táto šablóna pre uľahčenie práce parsera umiestňuje a vytrieďuje všetky značky `Formula` do značky `Formulas` a všetky značky `Graph` do značky `Graphs`. Značkou `xsl:copy` určujeme kopírovanie rodičovského elementu `Frame`, ktorý tieto značky ako najvrchnejší element obaľuje.

Piata šablóna rieši špeciálnu značku elementu `Graph` a to značku `Range`. Tá určuje zobrazovaný rozsah grafu funkcie. V kompilovanom kóde, ale rozlišujeme rozsah a posun grafu, pričom rozsah v tomto prípade určuje iba veľkosť intervalu, nie jeho posun. Transformáciu popísanou v piatej šablóne dosahujeme zmenu elementu `Range`

z formy `<Range x="-2..2" y="-3..1.5" />` na `<Range x="4" y="4.5" /><Shift x="-2" y="-3" />`. A to tak, že pre každý nájdený atribút elementu `Range` zisťujeme, či obsahuje reťazec „..“ funkciou `contains(., '..')`. Tá sa odkazuje na hodnotu momentálne prechádzaného atribútu. Ak nie, vraciame hodnotu atribútu bez zmeny. Ak atribút obsahuje „..“, vraciame rozdiel dvoch hodnôt oddelených bodkami ako veľkosť intervalu. Pri prechádzaní elementu `Range` tiež vytvárame element `Shift` pričom elementu `Shift` vytvárame tie isté atribúty ako má element `Range`. Hodnota atribútov elementu `Shift` je nulová, ak ten istý atribút v prípade elementu `Range` neobsahuje „..“. Ak ten istý atribút v elemente `Range` obsahuje reťazec „..“, tak do hodnoty atribútu elementu `Shift` vkladáme prvú hodnotu z dvojice hodnôt oddelených dvoma bodkami určujúcu posun po danej súradnici zobrazovaného grafu.

Posledná šablóna s názvom `graph` najprv aplikuje všetky ostatné šablóny na elementy grafu s výnimkou elementov `Marker`. Následne, v prípade, že prechádzaný element je elementu grafu `Graph` (čo nemusí byť nutne pravda, ak je šablóna volaná menom), obaľuje všetky elementy `Marker` značkou alebo elementom `Markers`, čo nám uľahčuje parsovanie výsledného transformovaného dokumentu.

5.3 SAX Parser

SAX Parser je rozhranie na prechádzanie textového reťazca XML dokumentu, ktoré v ňom detekuje jednotlivé elementy. V našej aplikácii ho využívame ku kompilovaniu XML zápisu algoritmov na javascriptový kód. K jeho vytvoreniu vrámci prostredia frameworku Rails používame knižnicu Nokogiri. V akcii `create` ho definujeme skrz triedu `AlgorithmDocument`.

5.3.1 Parsovací dokument `AlgorithmDocument`

Trieda `AlgorithmDocument` je trieda SAX parsera v knižnici Nokogiri a nájdeme ju v súbore `lib/parser/algorithm_document.rb`. Obsahuje štandardné metódy `start_element` a `end_element`, ktoré sú volané pri identifikácii začiatku a konca nájdeného elementu. Trieda má jedinú verejne dostupnú inštančnú premennú `result`, do ktorej sa ukladá výsledný javascriptový kód kompilovaného algoritmu zapísaného v jazyku XML.

initialize

Metóda `initialize` sa v jazyku Ruby volá vždy po inicializácii inštancie triedy. V našej metóde definujeme hodnoty dvoch inštančných premenných `@result` a `@stack`. Ako sme už spomínal, premenná `result` ukladá výsledný javascriptový kód kompilovaného algoritmu vo forme textového reťazca a premenná `stack` je premenná zásobníku prechádzaných elementov. Elementy, do ktorých práve parser vchádza sa do zásobníku ukladajú, zatiaľ čo elementy, z ktorých parser práve vychádza sa zo zásobníka vyberajú. Zásobník slúži na to, aby pri vychádzaní z elementov vrámci XML dokumentu, sme mohli volať metódu `end_print` objektu práve uzatvárajúceho sa elementu. Tento objekt je inicializovaný v metóde `start_element` a zásobník je jediný spôsob ako sa k nemu metóda `end_element`, môže dostať, bez toho aby musela inicializovať nový objekt prechádzaného elementu.

start_element(tagname, attrs = [])

Metóda `start_element` indikuje začiatok elementu v prechádzanom textovom reťazci XML zápisu kompilovaného algoritmu alebo programu. Zvyčajne je súčasťou tejto

metódy konštrukt `switch` (v prípade jazyka Ruby `case`), ktorý na základe názvu elementu, vráti textový reťazec resp. bude ďalej vetviť program. My, nasledujúc už raz popísaný princíp frameworku Rails - DRY, pre každý element vytvárame inštanciu k nemu pridelenej triedy. Tá obsahuje metódy `start_print` a `end_print`, ktoré voláme v metódach `start_element` a `end_element`. Takto separujeme programovú logiku jednotlivých elementov do zvláštnych tried, ktoré môžme rozširovať o ich špecifické metódy, ale ktoré majú verejnú štruktúru (metódy `start_print` a `end_print`) rovnakú. Na základe názvu elementu teda inicializujeme inštanciu k nemu priradenej triedy a následne k nemu priraďujeme inštančné premenné podľa jeho atribútov. Voláme jeho metódu `start_print`, ktorá na základe týchto obdržaných údajov generuje textový reťazec časti kompilovaného Javascriptu, zodpovedajúcej sa práve danému elementu. Element v závere umiestňujeme do zásobníka aby sme na jeho konci mohli volať metódu `end_print`.

end_element(tagname)

Metóda `end_element` indikuje koniec elementu v prechádzanom textovom reťazci XML zápisu kompilovaného algoritmu alebo programu. Funkciou `pop` z poľa zásobníku vyberá posledný vložený element a volá jeho metódu `end_print`, ktorej návratovú hodnotu ihneď ukladá do inštančnej premennej parsovacieho dokumentu - `result`, zhromažďujúcej textový reťazec výsledného javascriptového kódu.

5.3.2 Parsovacie elementy `Parser::Elements`

Súbor `lib/parser/elements.rb` obsahuje modul nášho parsera `Parser::Elements` obsahujúci triedy jednotlivých elementov. Tie dedia všetky od triedy `Element`. Tá obsahuje dve predvolené metódy `start_element` a `end_element` vracajúce prázdne reťazce.

Trieda `ArrayElement`

Obdržiava 3 atribúty - dĺžku `length` určujúcu veľkosť poľa, názov `name` určujúci identifikátor poľa a booleovský atribút `random` určujúci predvolené vyplnenie poľa náhodnými číslami rozsahu od 0 do 1. Vracia javascriptový kód, ktorý deklaruje pole a cyklom `for` ho vypĺňa, na základe atribútu `random`, náhodnými hodnotami alebo hodnotami 0, ak je hodnota atribútu `random false`.

Trieda `FunctionElement`

Obdržiava atribúty `formula` a `name`. Pôvodne značka slúžiaca pre zápis matematických výrazov, môže slúžiť aj na definíciu jednoduchých funkcií. Atribút `name` určuje názov identifikátora funkcie a atribút `formula` prijíma samotné znenie funkcie. Trieda `FunctionElement` vracia v metóde `start_print` javascriptový kód definície funkcie, ktorej návratová hodnota je hodnota atribútu `formula`. Trieda `FunctionElement` sama zisťuje názvy a počet argumentov pre definovanú funkciu. To činí skenovaním znenia funkcie pre identifikátory premenných. Identifikátor premennej nájde regulárnym výrazom ako textový reťazec, pred alebo za ktorým sa nenachádza znak bodky a za ktorým sa nenachádza ľavá oblá zátvorka určujúca volanie funkcie. Vrámci jednej definície funkcie, môže byť ten istý identifikátor premennej použitý viackrát, preto zabezpečíme, že pole identifikátorov obsahuje unikátne hodnoty volaním metódy `uniq`.

Trieda `FormulaElement`

Trieda `FormulaElement` a zároveň značka `Formula` obdržiavajú atribúty `content`, `name`, `ref`, `x` a `y`. Posledné dva určujú polohu okna `Formula` - okna matematického výrazu na obrazovke. Atribút `name` určuje meno daného výrazu. Atribút `content` určuje textový obsah výrazu, pričom ide o textový reťazec písaný v jazyku TeX. Atribút `ref` určuje odkaz na premennú, ktorá obsah daného výrazu môže určovať. Metóda `hash` triedy `FormulaElement` ukladá jednotlivé atribúty inštancie triedy do kolekcie `h`. Atribút `content` má pri tomto ukladaní väčšiu prioritu ako atribút `ref`. V textovom reťazci obsahu matematického výrazu sa môže nachádzať aj reťazec odkazu na premennú používanú v algoritme a to ako zástupná konštrukcia znakov `@{myvar}`, kde `myvar` je názov premennej, na ktorú sa chceme odkazovať. Takýmto spôsobom je hodnota atribútu `content` v tvare `x = @{x}` kompilovaná do `'x = '+x+"`, kde za `x` obklopenú znakmi plus bude v exekúcii javascriptu dosadená hodnota skutočnej premennej `x`. Kolekcia `h` je následne transformovaná do formátu JSON, z ktorého sú vymazané úvodzovky. Tie sa v javascriptovom kóde nepoužívajú a boli dôvodom jeho chybného spracovania.

Výsledným reťazcom, ktorý trieda `FormulaElement` vracia, je kľúč s názvom danej značky Formula definovaným v atribúte `name` a jeho hodnota, ktorá je zlúčením hodnoty prázdneho matematického výrazu a výrazu, ktorého objekt vlastností sme vytvorili v metóde `hash`. Tento celý kľúč je súčasťou objektu snímku.

Trieda `GraphElement`

Trieda `GraphElement` a zároveň značka `Graph` obdržiavajú atribúty `height`, `isocontours`, `ref`, `x`, `y` a `width`. Atribúty `x` a `y` určujú polohu okna grafu na obrazovke v aplikácii. Atribút `name`, podobne ako pri iných elementoch, určuje identifikačný názov grafu. Atribúty `height` a `width` určujú výšku a šírku grafu. Atribút `isocontours` počet izokontúr grafu, čo sú vrstevnicové čiary použité v prípade, že graf vykresľuje funkciu dvoch premenných. Atribútom `ref` určujeme, z ktorej premennej má graf získať dáta k vykresleniu. Premennou môže byť funkcia 1 alebo 2 premenných, alebo pole hodnôt. Metóda `start_print` vracia v Javascripte zapísaný kľúč objektu s názvom grafu získaným z atribútu `name` a jeho hodnotou, ktorou je zlúčenie predvoleného objektu prázdneho grafu a objektu vlastností grafu, ktorého značku práve parsujeme. Nakoľko XML značka `Graph` môže mať dcérske elementy, je jej objekt vlastností pripájaný k celkovému textovému reťazcu, reprezentujúcemu zodpovedajúci javascriptový kód algoritmu, až po parsovaní dcérskych elementov, na konci elementu `Graph`. Dcérske elementy elementu `Graph`, takto generujú kľúče kolekcie (alebo objektu) vlastností grafu. Pri vlastnostiach `isocontours`, `h`, `x`, `y` a `w` určujeme aj ich predvolenú hodnotu v prípade, že atribút v značke, a tak následne ani v inštancii triedy, nebol určený.

Trieda `MarkerElement`

Trieda `MarkerElement` zodpovedá kompilácii značke `Marker`, ktorá značí vodiacu čiaru alebo bod vykresľovaného grafu. Jej súčasťou sú tri atribúty `x`, `y` a `vertical`. Hodnota atribútu `x` značí index alebo vstupnú hodnotu funkcie, v závislosti od toho, či graf vykresľuje pole generovaných bodov alebo funkciu. V prípade, že značka `Marker` neobsahuje atribút `y`, vykresľuje sa vodiaci bod na súradniciach funkčnej hodnoty funkcie alebo poľa v hodnote `x`. V opačnom prípade sa vodiaci bod vykresľuje na presnom mieste danom súradnicami `x`, `y`. Atribút `vertical` určuje, či k vodiacemu bodu má byť dokreslená aj vodiaca čiara spájajúca bod kolmo so spodnou horizontálnou hranicou plátna kde je predpokladané miesto vykreslenia x-ovej súradnice. Metóda `start_print`

vracia textový reťazec javascriptového objektu obsahujúceho tri atribúty triedy, v prípade, že atribúty `y` a `vertical` nie sú nulové.

Trieda `MarkersElement`

Trieda `MarkersElement` zodpovedá kompilácii značky `Markers` a tá je doplňovaná do XML dokumentu XSLT transformáciou. Javascriptový objekt grafu má tzv. „markre“ alebo vodiace vizuálne prvky uložené v poli `markers`. Dáva nám zmysel preto vypisovať kľúč tohoto poľa osobitným elementom a triedou. Trieda rovnako ako značka nemajú žiadne atribúty a v metódach `start_print` a `end_print` otvárajú a zatvárajú kľúč poľa vodiacich prvkov, ktoré sú v poli uložené ako objekty generované triedou `MarkerElement`. Je potrebné si uvedomiť, že každá značka `<Marker />` generuje textový reťazec javascriptového objektu ukončený čiarkou, v prípade, že po danej značke nasleduje ďalšia značka `Marker`. Miesto toho aby sme si nejakým zbytočne zložitým spôsobom počítali jednotlivé značky a tak registrovali poslednú značku `Marker`, po ktorej už čiarku písať nebudeme, pri koncovom výpise kľúča `markers` triedy `MarkersElement` na začiatku reťazca vkladáme špeciálny znak
`b`, ktorý značí znak backspace, v tabuľke ASCII znakov na deviatom mieste. Tento znak nám indikuje, že jeho predošlý znak a teda čiarku po poslednom objekte `marker` poľa `markers`, je potrebné vymazať. Toto vymazávanie znakov backspace a im znakov predošlých sa uskutočňuje regulárnym výrazom v kontroléri `AlgorithmsController`.

Trieda `RangeElement`

Trieda `RangeElement` generuje kľúč definujúci intervalový rozsah zobrazovanej časti grafu. Tak ako je element `Range` dcérskym elementom elementu `Graph`, je kľúč generovaný danou triedou súčasťou objektu grafu generovaného triedou `GraphElement`. Trieda a teda aj značka prijímajú dva atribúty `x`, `y` označujúce rozsahy intervalov vrámci príslušných súradníc. Ich predvolené atribúty sú `1` a `1`.

Trieda `ShiftElement`

Trieda `ShiftElement` generuje podobným spôsobom ako trieda `RangeElement` kľúč, tentokrát definujúci posun zobrazovaného rozsahu grafu. Ten je súčasťou generovaného javascriptového objektu grafu triedou `GraphElement`. XSLT transformáciou generovaná značka `Shift` ako aj prislúchajúca trieda majú dva atribúty `x` a `y` s predvolenými hodnotami 0. To znamená graf zobrazuje predvolene prvý kvadrant funkcie od počiatku súradnicovej sústavy.

Trieda `FrameElement`

Trieda `FrameElement` definuje triedu pre kompiláciu elementu `Frame`. Neprijíma žiadne atribúty. Trieda v metódach `start_print` a `end_print` vracia textový reťazec javascriptového kódu, v ktorom pridáva ďalšiu snímku do poľa snímok. Snímka je výsledkom zlúčenia predošlej snímky a novej snímky pozostávajúcej z objektov generovaných dcérskymi elementami elementu `Frame`. Každý dcérsky element elementu `Frame` generuje textový reťazec javascriptového kódu ukončený čiarkou pre predpoklad existencie nasledujúceho elementu. Takáto čiarka vytvorená posledným elementom je určená k vymazaniu znakom
`b`. Pri zlučovaní predošlého snímku a zmien nasledujúceho snímku, upravujeme možnosti zlučovania funkcie `m` odkazujúcej na funkciu `merge` knižnice `deepmerge`. Novou definíciou zlučovacej funkcie `arrayMerge` zabezpečujeme prepísanie poľa objektov na pole novej snímky. Argumenty `d` a `s` odkazujú na tzv. `destination` - originálne pole

a `source` - pole určené k novému snímku. Bez definície tohoto zlučovania, pri zmene vodiaceho bodu bol vždy nechcene generovaný nový bod.

Trieda `FormulasElement`

Trieda `FormulasElement` definuje triedu pre kompiláciu elementu `Formulas`, ktorý zlučuje všetky elementy `Formula` patriace do jedinej snímky. V metódach `start_print` a `end_print` generuje kľúč `formulas` pre ukladanie jednotlivých matematických výrazov a teda javascriptových objektov zodpovedajúcich elementom `Formula`. Trieda ani značka neprijímajú žiadne atribúty. Textový reťazec metódy `end_print` začíname znakom
`b`, ktorý určuje jeho predošlý znak k vymazaniu. Tento predošlý znak je čiarka na konci generovaného textového reťazca, ktorý zodpovedá individuálnemu elementu `Formula`. Ide o princíp vymazávania čiarky posledného objektu poľa alebo kolekcie, ktorý sme popísali už napríklad pri triede `FrameElement` alebo `MarkersElement`.

Trieda `GraphsElement`

Trieda `GraphsElement` funguje presne tým istým spôsobom ako `FormulasElement` a zlučuje objekty grafov do spoločného kľúča `canvases`, ktorý je kľúčom objektu snímky. Trieda ani značka nemá žiadne atribúty a využíva už zaužívaný koncept mazania čiarky posledného dcérskeho objektu použitím znaku backspace
`b`.

Trieda `SetElement`

Trieda `SetElement` spracuváva všetky atribúty značky `<set />` ako skutočné priradenia hodnôt do premenných. Pozná iba jedinú výnimku a to v prípade, že značka `<set />` má oba atribúty `name` a `value`. V takom prípade je do identifikátora `name` priradzovaná, vo výslednom javascriptovom kóde, hodnota `value`. V akomkoľvek inom prípade, pri kompilácii XMLL značky, trieda vezme všetky inštančné premenné inštancie triedy, čo sú všetky atribúty XML značky prijímanej na vstupe, získa ich hodnoty a vráti textový reťazec priradenia hodnoty do premennej. Vo vrátenom textovom reťazci individuálneho atribútu, premennej, je vymazaný znak `@` zo začiatku mena inštančnej premennej, ktorý pridáva do jej názvu jazyk Ruby. Priradenia všetkých atribútov, premenných sú spojené z mapovaného poľa do jedného textového reťazca metódou `join`.

Trieda `VarElement`

Trieda `VarElement` funguje na presne takom istom princípe priradenia hodnôt do premenných ako trieda `SetElement`. Trieda `VarElement` nepozná výnimku simultánneho použitia atribútov `name` a `value` ako v prípade elementu `<set />`. Pre každú inštančnú premennú triedy reprezentujúcu atribút XML značky je vytvorený textový reťazec deklarácie premennej a voliteľného priradenia hodnoty do premennej. Pre každú deklaráciu premennej, musí atribútu značky byť priradená nejaká hodnota. Pre deklaráciu premennej bez hodnoty, priraďujeme atribútu v XML značke prázdny reťazec ako bolo už ukázané v užívateľskej dokumentácii. Podobne ako v prípade triedy `SetElement` je z názvu inštančnej premennej odstránený znak `@`. Premennej sa priradzuje hodnota len vtedy ak hodnota jej príslušného atribútu v XML značke nie je prázdna.

Trieda `AlgorithmElement`

Trieda `AlgorithmElement` resp. značka `Algorithm` neprijíma žiadne atribúty a definuje javascriptový kód na začiatku a konci algoritmu. V ňom v metóda `start_print` vytvára funkciu s názvom `algorithm`, v ktorej bude kompilovaný javascriptový kód algoritmu a na začiatku nej iniciuje pole snímkov `frames`, ktorého inicializačná hodnota je pole jedinej položky - snímky prázdnej animácie. V koncovej metóde `end_print`, trieda generuje javascriptový kód, ktorý maže prvú položku poľa frames v prípade, že do neho bol pridaná aspoň jedna snímka a vracia samotné pole snímok ako návratovú hodnotu funkcie `algorithm`.

Trieda `IfElement`

Trieda `IfElement` obdržiava jediný atribút `cond` definujúci podmienku podnecujúcu vykonanie vnoreného kódu. Metóda `start_print` otvára vo výslednom javascriptovom kóde blok podmienky `if` a metóda `end_print` ho zatvára.

Trieda `WhileElement`

Podobne ako v prípade triedy `IfElement` elementu `<if></if>`, trieda `WhileElement` obdržiava jediný atribút `cond`, ktoré definuje podmienku podnecujúci ďalšiu iteráciu cyklu. Metóda `start_print` otvára vo výslednom javascriptovom kóde iteračný blok cyklu `while` a metóda `end_print` ho zatvára.

5.4 CSS Štýlopis

Celý štýlopis CSS je v aplikácii popísaný v jedinom súbore `app/assets/stylesheets/application.css` a definuje vizuálny štýl aplikácie. Definujeme v ňom všeobecne štýly elementu `body`, štýl voliteľnej rukoväte komponentov slúžiacej na ich posun po obrazovke, štýly kontrolného panelu prehrávania animácie, štýl poťahovateľného plátna, štýly XML editora a nastavenia niektorých štýlov zobrazovania matematických výrazov.

Štýl `overflow-x` v elemente `body` zabraňuje zobrazovaniu horizontálneho posuvníka v okne aplikácie. Pri štýlovaní úchopnej rukoväte máme na mysli, že rukoväť je vykreslená iba v tých obsahových elementoch triedy `.card-content`, ktoré sú označené súčasne triedou `.with-handle`. So zobrazením rukoväte sa mení výška rodičovského elementu, ten ale výplňa svojou výškou stopercentne svoj rodičovský element a preto je potrebné od neho odčítať výšku lišty, ktorá automaticky od 100% neodčítava. Nastavením stopercentnej výšky a šírky karty `.card` vyplňujeme jeho obsahom rodičovský uzol okna komponentu. Triedou `.player-menu` nastavujeme fixované zobrazenie panela prehrávania v pravom hornom rohu a triedu `.player-state` zarovnávame číslovanie momentálne zobrazovanej snímky. Trieda `.react-draggable` opravuje zobrazovanie poťahovateľných komponentov balíčka `react-draggable`, ktoré majú predvolene nastavené relatívne umiestnenie čo spôsobuje chybu v umiestňovaní komponentov pri ich poťahovaní. Použitie tried `.plane` a `.plane-inner` nám pomáha k vytvoreniu poťahovateľného prostredia alebo plátna. Zatiaľ čo element triedy `.plane` staticky ostáva na tom istom mieste, vyplňuje priestor celej obrazovky a reaguje na udalosti poťahovania myšou, element triedy `.plane-inner` je ten, ktorý sa následne pri poťahovaní posúva. Nadmerná šírka elementu `.plane-inner` zabezpečuje nezalamovanie textu vnútri komponentov, ktorý sa v dôsledku vlastnosti `display: inline-block` prispôsobuje šírke najbližšieho rodičovského okna typu `display: block`. Trieda `.xml-submit` nastavuje štýlovanie tlačidla na odoslanie XML zápisu algoritmu a trieda `.ace_editor` dopĺňa

XML editor o spodný rámik. Triedy súvisiace s knižnicou MathJax odstraňujú modré podsvietenie po kliknutí na matematický výraz a biele horné a spodné odsadenie matematických výrazov.

5.5 Komponenty

Ako sme už popisovali pri knižnici React, komponentami sa dajú chápať nami definované HTML elementy rozšírené o nami definovaní a programovanú funkcionalitu. Každý komponent je určený k plneniu konkrétnej funkcie. Jednotlivé komponenty nájdeme v priečinku aplikácie `app/javascript/components`. Každý komponent definovaný v knižnici React dedí od rodičovského komponentu `React.Component` a predpokladá existenciu najdôležitejšej metódy `render`, ktorá vracia HTML štruktúru komponentu. Bez správne definovanej metódy `render` komponent nie je funkčný a nemôže byť použitý.

V nasledujúcej kapitole budeme používať slovo komponent na označenie programových tried jednotlivých komponent a slovo uzol na označenie okna v rámci aplikačného prostredia.

`constructor()`

Konštruktor volaný vždy po inicializácii objektu triedy musí vo frameworku React obsahovať volanie konštruktura dedenej triedy a to hneď v prvom riadku. Konštruktoru sú predané ako argument HTML atribúty daného komponentu. Opakovaným vzorom vrámci nami definovaných niekoľkých komponentov je tzv. „binding“ metód komponentov - `this.next = this.next.bind(this)`. Ide o „zviazanie“ metódy s kontextom `this`. Normálne ak je metóda volaná vrámci podnietenej udalosti jej kontext a teda premenná `this` sa odkazuje na okno, alebo na element, z ktorého bola udalosť podnietená. My chceme dosiahnuť aby volaním odkaz `this` sme mohli aj napriek kontextu udalosti pristupovať k inštancii triedy komponentu. To dokážeme práve metódou `bind`, ktorou viažeme tento kontext `this` na inštanciu nášho komponentu, ku ktorej môžme v „naviazanej“ metóde už pristupovať.

`componentDidMount()`

Systémová metóda knižnice React `componentDidMount` je volaná vždy po nasadení komponentu do štruktúry DOM webovej aplikácie zobrazovanej webovým prehliadačom.

`componentDidUpdate()`

Systémová metóda knižnice React `componentDidMount` je volaná vždy po aktualizácii komponenty k čomu môže dôjsť napríklad pri dynamickej zmene atribútu komponenty.

`render()`

Systémová metóda knižnice React `render` má za úlohu vykreslenie HTML štruktúry, ktorú vracia ako funkcia v návratovej hodnote. Vďaka syntaxu `*.jsx` nemusíme HTML štruktúru obalovať do úvodzoviek ako textový reťazec, ničmenej HTML štruktúra, ktorú vraciame musí mať jediný koreňový element.

5.5.1 Komponent `App`

Komponenta `App` obaľuje všetky ostatné komponenty a je de facto klientskym prostredím aplikácie. Komponenta `App` zachycuje udalosť odoslania algoritmu zapísaného v XML v komponente XML a reaguje naň odoslaním HTTP požiadavky na koncový bod aplikačného rozhrania lokálneho Rails serveru na adresu `/algorithms` prostredníctvom knižnice na vysielanie HTTP požiadaviek `axios`. Po obdržaní kompilovaného algoritmu v jazyku javascript, tento algoritmus je cez príkaz `eval` uložený do premennej `algorithm`. Vygenerovaný javascriptový kód algoritmu je následne spustený a vracia jednotlivé snímky algoritmu, ktoré je možné prehrávať.

Pre generované komponenty (alebo tiež uzly - "nodes"), ktoré algoritmus vyžaduje sú uložené lokálne stavy pozícii a rozmerov uzlov a to jednotlivo grafov a matematických výrazov (alebo formúl). Ukladanie týchto stavov mimo stavovú premennú `animation` nám zabezpečí zachovanie stavov naprieč snímkami animácie.

`constructor()`

Na začiatku konštruktora definujeme stav našej aplikácie. Na základe tohoto stavu a jeho zmien bude aplikácia vykresľovaná. V stave definujeme niekoľko kľúčov. Kľúč `animation` zodpovedá práve zobrazovanému snímku animácie. Kľúč `frame` určuje index zobrazovaného snímka vrámci poľa `frames`, ktoré ukladá všetky snímky animácie. Kolekcia `nodes` ukladá lokálne stavy okien aplikácie ako napr. ich pozíciu a rozmery. Dôvodom oddelenia týchto atribútov od atribútov definovaných v jednotlivých snímkoch bol fakt, že chceme zachovávať pozíciu a rozmery okien naprieč snímkami animácie. Kľúč `plane` ukladá momentálnu pozíciu posunu plátna aplikácie, na ktorom sú zobrazované jednotlivé okná. Booleovský znak `playing` definuje, či animácia je práve prehrávaná a kľúč `xml` udržuje obsah XML editora. Štandardne pri využití balíčka komponentu ACE Editor, nie je potrebné tento obsah udržiavať. V dôsledku zmeny snímok animácie sa obsah komponentu `App` prekresľuje a s ním, vrámci konvencie reaktívneho programovania a knižnice React, aj element XML editora, ktorý po prekreslení zostáva prázdny ako v jeho počiatočnom stave. Z prirodzeného dôvodu si musíme teda obsah XML editora pamätať v stave vlastného komponentu.

V konštruktore tiež definujeme nastavenie balíčka `axios`, ktorý slúži na odosielanie HTTP požiadavkov nášmu servru. Zaradením hlavičiek `X-Requested-With` a `X-CSRF-TOKEN` do každého odoslaného požiadavku balička `axios` zabezpečujeme požiadavku prejdenie bezpečnostnou verifikáciou frameworku Rails. Tá ma za účel ubrániť aplikáciu pred tzv. CSRF útorok, alebo tiež „Cross-site Request Forgery“ útokom. Ide o útok, v ktorom navštívením inej stránky, môžeme nechať túto stránku poslať nechcený HTTP požiadavok na adresu našej aplikácie, ktorá môže meniť naše údaje vrámci aplikácie. CSRF ochrana frameworku Rails vyžadením autentikačného kódu zabraňuje takýmto požiadavkom meniť dáta vrámci aplikácie. Autentikačný znak alebo token, obdržíme z meta hlavičky práve zobrazovanej stránky našej webovej aplikácie. Pre zachovanie tejto konfigurácie vrámci globálne prístupnej premennej, uchovávame odkaz na balíček `axios` s danou konfiguráciou v globálnom objekte okna webového prehliadača.

Na konci metódy konštruktor nastavujeme viazania, ktorých princíp a účel sme už vo všeobecnosti popísali vyššie. Ide o viazania jednotlivých metód komponentu z kontextom jeho inštancie.

prev()

Metóda `prev()` je metóda posúvania animácie algoritmu a posúva animáciu o 1 snímok dozadu. Po dekrementácii indexu aktuálneho snímku nastavuje nový index aj v premennej stave komponentu `App` a nastavuje aj nový snímok do kľúča `animation`. Metóda posúva animáciu iba vtedy ak je možné ešte animáciu spätne posúvať.

play()

Metóda `play()` je metóda prehrávania animácie algoritmu. Najprv obnovuje animáciu do stavu prvej snímky a zároveň nastavuje stav aktuálneho prehrávania na pravdivý - `true`. Následne animáciu začne po snímkoch posúvať dopredu v pol sekundových intervaloch.

stop()

Metóda `stop()` zastavuje chod animácie algoritmu. Obnovuje animáciu do stavu prvej snímky a nastavuje stav aktuálneho prehrávania na nepravdivý - `false`. Tiež ukončuje pravidelné volanie metódy `next()` vymazaním intervalu jej volania a to funkciou `clearInterval`.

next()

Metóda `next()` je metóda posúvania animácie algoritmu a posúva animáciu o 1 snímok dopredu. Po inkrementácii indexu aktuálneho snímku nastavuje nový index aj v premennej stave komponentu `App` a nastavuje aj nový snímok do kľúča `animation`. Metóda nastavuje stav prehrávania na `false` v prípade, že práve posúva animáciu do posledného snímku. Takto zabraňujeme zachovaniu stavu prehrávania, aj keď je animácia už dávno na konci. Metóda posúva animáciu iba vtedy ak sa animácia nenachádza v stave posledného snímku. V prípade, že sa v tomto stave nachádza, v prípade prehrávania dovršujeme proces dokončenia prehrávania vymazaním intervalu, ktorý nastavuje pravidelné posúvanie animácie o snímok dopredu. Takto zabránime vytvorením nových a nových intervalov, v prípade opätovného spustenia nahrávania.

getPlayerState()

Metóda `getPlayerState()` vracia číslovanie aktuálneho snímku z celkového počtu snímkov k zobrazeniu na kontrolnom paneli prehrávania. Je potrebné si uvedomiť, že hodnota aktuálneho snímku je indexová hodnota a začína svoje číslovanie od nuly. Preto k indexu pripočítavame 1. V prípade nulového počtu celkových snímkov číslovanie špeciálne upravujeme aby zobrazovalo za aktuálny nultý snímok.

planeDrag(e)

`planeDrag(e)` je metóda volaná komponentom `DraggableCore` pri poťahovaní daného komponentu. V metóde overujeme, či poťahujeme plátno a nie len jeho komponent. V prípade, že poťahujeme plátno, nastavujeme nový stav súradníc plátna pripočítaním posunu, ktorý je hodnotou atribútov `movementX` a `movementY` objektu udalosti komponentu `DraggableCore`.

rememberState(key, hash)

`rememberState(key, hash)` je metóda využívaná k zmene momentálnych stavov okien, a to ich pozícii a rozmerov. Na základe prvého argumentu kľúča určuje okno

ktorého vlastnosti sa majú zmeniť, zatiaľ čo kopíruje stav všetky ostatných okien pomocou syntaxu trojbodky. Trojbodka ako prefix kolekcie v špecifikácii ECMAScript 6 rozširuje objekt do ktorého je kolekcia vnorovaná o svoje vlastné kľúče. Vybraný uzol je obnovaný ako zjednotenie jeho doterajšieho stavu a kolekcie nových hodnôt vybraných vlastností `hash`.

`onGraphResize(id, el)`

`onGraphResize(id, el)` je metóda, ktorá je volaná pri udalosti zmeny veľkosti grafu. Vzápätí vola metódu `rememberState`, skrz ktorú nastavuje výšku a šírku daného grafu, na základe jeho identifikátora. Od hodnôt nových rozmerov odčíta hodnotu 24 čo je súčet dolného a horného, resp. ľavého a pravého odsadenia grafu. Momentálne rozmery získava cez metódy `offsetWidth` a `offsetHeight` objektového modelu DOM.

`xmlChange()`

`xmlChange()` je metóda volaná pri zmene obsahu XML editora a ukladá si jeho nový obsah do stavu komponenty `App`. Takto prechádzame strate obsahu XML editora, ku ktorej dochádza vrámci prekreslovania HTML šablóny komponenty `App` pri zmene stavu súvisiaceho s komponentom `XML` ako napr. jeho pozície.

`submit()`

Metóda `submit()` je volaná po stlačení tlačidla „Submit“ XML editora cez volanie skrz atribút `onSubmit` komponenty XML. Jej hlavnou úlohou je odoslať algoritmus v zápise XML na adresu aplikácie a obdržať k nemu prisluchajúci generovaný javascriptový kód. Metóda odosiela algoritmus zapísaný v XML jazyku ako textový reťazec, ktorý je súčasťou tela HTTP požiadavky metódy `POST` vyvolanej knižnicou `axios`, určenou na odosielanie HTTP požiadaviek. HTTP požiadavku posiela na lokálnu adresu aplikácie `/algorithms`.

V prípade úspešného spracovania požiadavku aplikácia obdrží kompilované znenie algoritmu v programovacom jazyku Javascript ako dátový parameter `algorithm` obdržanej odpovede. Prijatý javascriptový kód je exekuovaný funkciou `eval`. V samotnom kóde sa algoritmus ukladá do deklarovanej premennej `algorithm` a používa sa v ňom premenná `m` ako odkaz na funkciu `merge` určenú na zlučovanie javascriptových kolekcií alebo objektov. Je potrebné si uvedomiť, že k deklarovaným premenným v kontexte exekuovaného kódu ako argumentu funkcie `eval` nemáme mimo kontext funkcie `eval` - mimo kontext samotného exekuovaného kódu prístup. Podobne, funkcia `eval` nemá prístup k nám používanými globálnym premenným ako napr. k funkcii `merge` importovanej z knižnice `deepmerge`. Preto na prístup ku kódu, ktorý je exekuovaný vrámci funkcie `eval`, potrebujeme deklarovať premennú `algorithm` v lokálnom rozsahu („local scope“), t.j. vrámci najbližšej rodičovskej funkcie, v ktorej kód prebieha. Tak k nemu budeme mať prístup ako k premennej aj vrámci exekuovaného kódu funkcie `eval`. Podobne vytvárame premennú `m`, cez ktorú sa odkazujeme na globálne definovanú funkciu `merge`. Premennú `m` podobne používame v exekuovanom kóde funkcie `eval`.

Exekúciou samotného obdržaného javascriptového kódu sme iba uložili funkciu nášho algoritmu do premennej `algorithm`, preto túto premennú exekuujeme a ukladáme si jej návratovú hodnotu, ktorá je poľom snímkov, ktoré tvoria animáciu daného algoritmu.

Predtým než dosadíme pole snímkov animácie do stavu aplikácie a nastavíme animáciu na jej počiatočný snímok, nastavujeme tiež počiatočné hodnoty stavov všetkých

vykreslovaných okien. To robíme jednotlivo pre okná grafov a okná matematických výrazov. Tie môžu vo svojich príslušných značkách XML zápisu prijať atribúty definujúce pozíciu okna a u grafov tiež aj jeho rozmery. Prirodzene ak u objektov grafov a matematických výrazov (formúl) tieto atribúty nenachádzame, nastavujeme predvolené hodnoty pozícii a rozmerov ich okien. V kolekci uzlov identifikujeme jednotlivé okna matematických výrazov a grafov na základe ich kľúča, ktorý sa skladá z identifikácie ich typu (`"canvas-"`, `"formula-"`) a ich názvu (resp. kľúča, pod ktorým sú uložené v kolekciách `canvases` a `formulas`).

V závere metódy načítavame do stavu snímky animácie, lokálne stavy pozícií a rozmerov okien, XML zápis algoritmu a nastavujeme stav našej vizualizácie na jej prvý snímok.

`render()`

V metóde `render()` si do konštanty `planeInnerStyle` ukladáme štýl, ktorý určuje polohu vnútornej, pohyblivej časti plátna. Na základe momentálne stavu prehrávania animácie vytvárame tlačidlo k zastaveniu alebo prehrávaniu animácie, ktoré umiestňujeme do kontrolného panela animácie. V karte kontrolného panelu k prehrávaniu zobrazuje zaradom číslovanie momentálneho snímku z celkového počtu snímkov, tlačidlo na posun animácie do predošlého snímku, tlačidlo na prehrávanie resp. zastavenie animácie a tlačidlo na posun animácie do nasledujúceho snímku. V HTML štruktúre, ktorú vraciame v metóde `render()` používame komponenty knižnice `material-ui`. Material UI je knižnica vytvorená spoločnosťou Google za účelom vytvorenia predvoleného vizuálneho štýlu aplikácii určeného k ich rýchlemu a efektívnemu navrhovaniu.

V druhej časti vykresľovanej HTML štruktúry zobrazujeme posúvateľné plátno, zložené s elementov tried `.plane` a `.plane-inner`. Element triedy `.plane` pokrýva celú obrazovku a nemení svoju polohu. Je určený ako rukoväť (tzv. „handle“) komponentu `DraggableCore`, za ktorý sa plátno poťahuje. Element triedy `.plane-inner` svoju polohu v dôsledku podnietenia udalosti poťahovania mení. Zároveň všetky udalosti myši, ktoré by mohli podnietiť akciu poťahovania sú z elementu triedy `.plane-inner` ignorované atribútom `cancel` komponentu `DraggableCore`. Komponenta `DraggableCore` je komponenta programovateľnej funkcionality poťahovania elementov po obrazovke. Načúvame jej udalosť `onDrag`, skrz ktorú meníme poloho vnútorného elementu triedy `.plane-inner`.

Na plátne zobrazujeme grafy pre každý prvok grafu momentálneho snímku uloženého v kolekcii `canvases`. Štandardne ako pri každom zobrazovaní zoznamu komponentov v knižnici React, potrebujeme nastaviť ich kľúč `key`. Graf prijíma atribúty `range` a `shift`, ktoré určujú rozsah a posun zobrazovanej časti funkcie v grafe. V prípade, že tieto atribúty sú súčasťou objektu vrámci snímku animácie v `canvases[key]`, sú tieto predvolené hodnoty prepísané trojbodkovou notáciou rozvinutia kľúčov objektu. Následne graf doplňujeme a atribúty určujúce jeho pozíciu a rozmery z kolekcie `nodes`. Pri zobrazovaní grafu načúvame jeho dvom udalostiam a to `onDrag` a `onResize`, ktoré nechávajú bežať metódy `rememberState` resp. `onGraphResize` ukladajúce zmenu pozície grafu na plátne a jeho rozmerov. Zobrazujeme komponent XML editora. Podobne ako v prípade komponentov grafov, reagujeme na zmenu jeho pozície a rozmerov cez atribúty `onDrag` a `onResize`. Definujeme jeho obsah atribútom `xml` a reagujeme na zmeny jeho obsahu cez volanie funkcie v atribúte `onChange` a na stlačenie tlačidla k odoslaniu jeho obsahu volanou funkciou v hodnote atribútu `onSubmit`.

Podobným spôsobom ako v prípade grafov, z momentálneho snímku animácie načítavame všetky komponenty matematických výrazom, štandardne pri zobrazovanie zoznamu komponentov v knižnici React im priradzujeme kľúč a rozbaľujeme všetky ich atribúty z objektu snímky animácie a atribútu určujúce ich pozíciu z kolekcie `nodes`, z

premennej stavu aplikácie. Nakoľko matematické výrazy nemenia svoje rozmery, predovšetkým z dôvodu zložitosti implementácie neustáleho prekreslovania matematického výrazu pri zmene jeho veľkosti, reagujeme iba na zmenu jeho pozíciu funkciou v atribúte `onDrag`.

5.5.2 Komponent `XML`

Komponent `XML` definuje okenné rozhranie na zápis algoritmov v jazyku XML.

`blur(e, editor)`

Metóda blur(e, editor) je volaná po tom čo kurzor opustí textové pole XML editora. Za takéhoto stavu predpokladáme, že užívateľ práve dopísal a je pravdepodobné, že bude interagovať ďalej s aplikáciou. Nakoľko akákoľvek interakcia s komponentom XML editora mimo písanie má za následok prekreslenie celého XML editora a stratu jeho obsahu, preventívne získavame jeho obsah a posúvame ho atribútu `onChange`, cez ktorý sa v komponente `App` následne volá metóda `xmlChange`. Tá ukladá obsah XML editora do stavu našej aplikácie.

`submit()`

Metóda submit je volaná po stlačení tlačidla „Submit“ XML editora. Metóda získava obsah XML editora a posúva ho atribútu `onSubmit`, ktorého predpokladaná hodnota je funkcia. Tou funkciou je metóda `submit` komponentu `App` slúžiaca k odoslaniu XML kódu na jeho kompiláciu.

`render()`

Metóda `render` zobrazuje samotný komponent `XML`. Komponent obsahuje komponent `AceEditor` importovaný knižnicou `react-ace`. Komponent `AceEditor` má mód písania nastavený na XML a používa farebnú schému s názvom `github`. Odkaz na komponent ACE editora je definovaný v inštančnej premennej `aceEditor`. Komponent vypĺňuje maximálnu šírku a výšku svojho uzlového obalu, s výnimkou odrátania výšky tlačidla „Submit“. Hodnota obsahu XML editora je načítavaná z atribútu `xml` komponenty `XML`. Atribút editora `onBlur` volá odkazovanú metódu `blur` po deaktivovaní textového poľa ACE editora.

Komponent `Node` spoločne obaľuje tlačidlo na odoslanie algoritmu zapísaného v jazyku XML a komponent ACE editora. Definuje použitie rukoväte (`handle`) na poťahovanie okna po plátne obrazovky, a prijíma atribúty rozmerov (`height`, `width`) a pozície (`x`, `y`) ako hodnoty tých istých atribútov komponenty `XML`. Podobným spôsobom prenáša komponent `XML` na komponent `Node` aj odkazy na metódy, ktoré sa vykonávajú pri poťahovaní uzle resp. pri zmene jeho veľkosti.

5.5.3 Komponent `Graph`

Komponent `Graph` má za úlohu vykresliť referované dáta, tie môžu byť predané komponentu ako funkcia ale aj ako pole. V prípade funkcie, komponent na základe svojich rozmerov generuje pole bodov dostatočnej veľkosti, ktoré následne vykresľuje.

Atribúty: `h`, `w`, `x`, `y`, `range`, `shift`, `markers`, `onDrag`, `onResize`

`constructor()`

Okrem už pre nás známych častí konštruktora, v konštrukture komponentu `Graph` definujeme šírku vnútorného rámika vrámci elementu `canvas`. Táto šírka slúži ako biele miesto, na ktoré sa už nevykresľujú hodnoty funkcie resp. predaných dát, ale môže slúžiť ako priestor na vykreslenie súradníc a orientačných čísel.

`componentDidMount()`

Po nasadení komponentu do štruktúry DOM webovej aplikácie žiadame prehliadač o exekúciu metódy `draw` pri ďalšom prekreslení obsahu okna webového prehliadača.

`setCanvas(canvas)`

Metóde `setCanvas` je predaný odkaz na element `canvas` a tá tento odkaz ako aj odkaz na dvojrozmerný kontext elementu `canvas`, ktorý je použitý na vykresľovanie dvojrozmerných objektov ukladá do inštančných premenných `this.canvas` a `this.ctx`. K ním máme následne prístup vrámci celej inštancie komponentu.

`draw()`

Generuje a vykresľuje súčasti grafu. Pred každým vykreslením najprv vyčistí plátno, následne vykreslí súradnice a k ním orientačné hodnoty, ktoré určujú rozsah vykreslovaných dát resp. funkcie. V prípade funkcie musí dáta, ako pole vykreslovaných bodov, generovať na základe rozmerov grafu. Po vykreslení samotných dát vykresluje všetky predané značky resp. vodiace body. Na konci volá globálnu metódu rozhrania okna webového prehliadača `window.requestAnimationFrame`. Metóda `window.requestAnimationFrame()` žiada prehliadač o exekúciu predanej funkcie pri ďalšom prekreslení obsahu okna webového prehliadača. Tá sa uskutoční pred samotným prekreslením.

`projectX(x)`

Metóda `projectX(x)` prijíma hodnotu indexu bodu vrámci poľa vygenerovaných resp. obdržaných bodov a vracia jej absolútnu polohu na plátne.

`projectY(y)`

Metóda `projectY(y)` prijíma hodnotu y-ovej súradnice relatívnej polohy vykreslovaného bodu vrámci y-ového rozsahu vykreslovaných bodov a vracia jej absolútnu polohu na plátne. Absolútnu hodnotu y-ovej súradnice vykreslovaného bodu ešte otáča o celkovú výšku plátna, nakoľko číselné označenie bodov na plátne je zoradené smerom zhora nadol a my pri vykreslovaní hodnôt poľa alebo funkcie pracujeme v prvom kvadrante, teda po súradnici `y` smerom zdola nahor.

`renderAxes()`

Vykresľuje súradnice grafu a ich orientačné hodnoty. Definuje premenné rozmerov bieleho rámika, ktorý ohraničuje samotnú vykresľovanú funkciu grafu `bH` a `bW`, premenné rozsahu zobrazovaných dát `rX` a `rY` a premenné posunu grafu `sX` a `sY`. V prípade, že graf zobrazuje funkciu, hodnoty `rX` a `sX`metóda berie z atribútov komponentu. Ak graf má za cieľ zobraziť pole bodov, je rozsah určený dĺžkou takéhoto poľa a posun po súradnici x je nulový. Nastavujeme predvolené štýly vykresľovania obsahu elementu `<canvas></canvas>` a definujeme funkciu `setStyle` určenú k prepnutiu štýlu na štýl

vykresľovania pomyselných čiar súradníc. Tie sa vykresľujú v prípade, že súradnica vo svojej originálnej polohe je mimo zobrazovanej časti grafu, čo overujeme podmienkou porovnania y-ovej absolútnej polohy x-ovej súradnice uloženej v premennej `xPos`. Ak je súradnica na svojej originálnej polohe súčasťou zobrazovanej časti grafu vykresľuje sa štandardným štýlom ako celá čiara čiernej farby. K danej x-ovej súradnici vykresľujeme orientačné čísla po stranách súradnice. Získavame a ukladáme si najbližšie celé čísla x-ových hodnôt po stranách súradnice do premenných `xLeftNum` a `xRightNum`. Ak má byť najbližšie celé popisné číslo 0, teda hodnota počiatku súradnicovej sústavy, tak zobrazujeme minimum resp. maximum danej súradnice ako popisné číslo bez ohľadu na to, že je číslom desatinným. K nájdeným popisným číslam následne projekciou `projectXNum` získavame ich absolútnu polohu. S argumentami hodnoty daného čísla, jej x-ovej a y-ovej súradnice a znaku `right`, ktorý označuje, na ktorú stranu popisné číslo patrí, ho následne funkciou `drawNumberWLine` vykresľujeme. Strana ako argument v prípade funkcie `drawNumberWLine` nám umožňuje popisné čísla posunúť mimo vnútorneho plátna grafu, tak aby neprekrývali samotnú líniu alebo kontúry grafu. To oceníme hlavne vtedy, ak má byť popisným číslom číslo s niekoľkými desatinnými miestami.

Po obnovení niektorých štýlov vykresľovania, získavame podobným spôsobom absolútnu polohu y-ovej súradnice na plátne. V prípade, že súradnica je mimo zobrazovanej časti grafu, zobrazujeme súradnicu v strede grafu a nastavujeme jej vizuálny štýl na sivú farbu a na vykresľovanie prerušovanej čiary. K vykresľovaniu orientačných čísel získavame najprv najbližšie celé čísla po dĺžke súradnice do ukladáme ich do premenných `yBottomNum` a `yTopNum`. Následne funkciou `projectYNum` získavame ich absolútnu polohu na plátne. Funkciou `drawNumberHLine`, ktorá akceptuje hodnotu vykresľovaného čísla a súradnice jej absolútnej polohy vykresľujeme dané orientačné číslo s miernym posunom tak aby neprekrývalo vykreslenú súradnicu a bolo podľa možností čitateľné.

Konštrukcia `parseFloat(...).toString()` použitá vo funkciách `drawNumberWLine` a `drawNumberHLine` zabezpečuje odstránenie zbytočných núl na konci desatinného čísla, po zaokrúhlení metódou `toFixed(4)` na 4 desatinné miesta.

`generateData()`

`generateData()` je metóda na generovania poľa dát, ktoré sa následne vykresľujú v grafe. Používa sa v prípade, v ktorom komponent `Graph` vykresľuje funkciu a teda, v prípade kedy jeho atribút `data` je funkciou. Body vykresľujeme vo vnútornom rámiku plátna. V grafe vždy vykresľujeme množinu bodov a vykresľovanie funkcie nie je pre nás takisto nič iné ako iba vykreslenie množiny bodov. Túto množinu bodov v tejto metóde generujeme. V cykle overujeme či v premennej `data` je funkcia jednej alebo dvoch premenných. Na základe toho definujeme nové pole `arr`, ktoré môže byť pole funkčných hodnôt na zobrazovanom rozsahu funkcie alebo pole polí, ktorých položky ukladajú funkčné hodnoty pozdĺž celého rozsahu pozdĺž y-ovej osi. Pri generovaní bodov funkcií dvoch argumentov si pamätáme tiež najmenšiu a najväčšiu funkčnú hodnotu danej funkcie. To vzápätí použijeme pri vykresľovaní izokontúr danej funkcie. Funkcia `wrap` prijíma ako argument pole polí, ktoré sa dá predstaviť ako mriežka hodnôt a túto mriežku obaľuje zo všetkých strán jedným stĺpcom resp. riadkom. To nám umožňuje predpokladať, že každé signifikantné políčko mriežky má 4 priamych susedov (v1avo, vpravo, nahor a nadol), čo nám uľahčuje prácu pri prechádzaní políčok. Definujeme tiež funkciu `getIsocontours`, ktorá na základe vstupného argumentu počtu vykresľovaných izokontúr a predtým určenej minimálnej a maximálnej funkčnej hodnoty vykresľovanej funkcie dvoch argumentov v premenných `min` a `max`, vracia pole hodnôt, ktoré reprezentuje funkčné hodnoty pozdĺž, ktorých sa izokontúry budú vykresľovať. Premenná `segment`, určuje vzdialenosť funkčných hodnôt medzi jednotlivými izokontúrami. Vo finálnej časti generovania dát pre funkciu dvoch premenných, prechádzame rozšíreným

poľom funkčných hodnôt a hľadáme také políčka funkčných hodnôt, ktoré sú väčšie alebo rovné funkčnej hodnote vybranej izokontúry a zároveň majú priameho susedo, ktorého funkčná hodnota je menšia ako funkčná hodnota vybranej izokontúry. To nám definuje potrebný kontrast na vykreslenie bodu na danom políčku. Generujeme takto mriežku bodov, kde existencia vykresľovaného bodu je určená hodnotou 1. Mriežku bodov ukladáme do inštančnej premennej komponentu `data`.

`renderData()`

Úlohou metódy `renderData` je vykresliť jednotlivé body grafom zobrazovanej funkcie alebo poľa. Vo funkcií prechádzame inštančnou premennou `data`, do ktorej sme metódou `generateData` priradili body k vykresleniu. Metóda overuje, či pole `data` je pole polí, čo značí vykresľovanie funkcie dvoch premenných. V takom prípade prechádzame každým riadkom takejto mriežky a v prípade výskytu hodnoty 1, vykresľujeme bod. Je potrebné povedať, že riadok mriežky, je v skutočnosti vykresľovaným stĺpcom a tento stĺpec je obrátený naopak, nakoľko pri vykresľovaní objektov v HTML elemente `<canvas>` hodnoty y-ovej súradnice stúpajú zhora nadol. Pri vykresľovaní bodu, jeho y-ovú súradnicu teda otáčame. V prípade, že poľe `data` je poľom hodnôt, získavame pre tieto hodnoty ich absolútne súradnice vrámci elementu `<canvas>`, na ktorých vykresľujeme príslušné body.

`renderMarkers()`

Má za úlohu vykresliť značky ktoré označujú vodiace body na plátne. Vodiace body sú komponentu `Graph` predané v atribúte `markers`. Metóda obsahuje internú pomocnú funkciu `renderMarker`, ktorá obdrží vodiaci bod, alebo značku vo forme objektu `{ x: int, y: int, vertical: boolean }`, kde hodnoty `y` a `vertical` sú voliteľné. Na základe toho, či komponent `Graph` vykresľuje hodnoty funkcie alebo poľa, získavame hodnoty premenných `x` a `y`, ktoré udávajú relatívnu polohu vodiaceho bodu vrámci grafu. Tú je potrebné premietnuť do absolútnych rozmerov nášho plátna. Pri vykresľovaní vodiaceho bodu vrámci funkcie je potrebné premietnuť hodnotu `x` do hodnoty indexu bodu vrámci už vytvoreného pola vykresľovaných hodnôt `data`. To sa uskutočňuje na riadku

```
Math.floor( ( marker.x - sX) * data.length / rX )
```

Nakoľko ide o index poľa, potrebujeme zaokrúhlovať pomerovú hodnotu horizontálnej súradnice umiestnenia bodu na celé číslo. V prípade, že hodnota parametra `vertical` je `true`, dokresľujeme k bodu vertikálnu vodiacu čiaru, ktorá končí na spodnej hrane vykreslovaného grafu. Toto uskutočňujeme pre všetky vodiace objekty kolekcie predanej v atribúte grafu `markers`.

`render()`

Vraciame element `canvas`, na ktorom vykreslujeme graf obalený uzlovým komponentom, ktorý definuje univerzálne okenné rozhranie. K určenej šírke komponentu je pridané ešte vnútorné odsadenie v hodnote po stranách 12 pixelov. Jednotlivým elementom sú predané atribúty komponentu definujúce rozmery elementu `<canvas>`, pozíciu a rozmery okna, metódy reagujúce na zmenu pozície a rozmerov okna `onDrag` a `onResize` a metóda `setCanvas`, ktorá v atribúte `ref`, ktorý je interpretovaný knižnicou React, nastavuje odkaz na daný element `canvas`.

5.5.4 Komponent `Node`

Komponent uzla definuje okno nášho aplikačného rozhrania a s ním univerzálnu funkcionalitu posúvania okien alebo zmeny ich veľkosti. K posunu uzla a k zmene jeho veľkosti využívame balíčky `react-draggable` a `re-resizable`.

Atribúty: `handle`, `height`, `width`, `x`, `y`, `resizableDisabled`

`handleStart()`, `resizeStart()`

Udalosti v Javascripte sa propagujú od najvrchnejšieho elementu (napr. `<button />`) až po ten najspodnejší `<body></body>`. Zastavením propagácie udalosti kliknutia myši pri začiatku zmeny veľkosti uzla alebo posunu uzla, zabezpečíme to, že udalosť kliknutia nebude propagovaná k pracovnej ploche našej aplikácie, kde by mala za následok simultánny posun pracovnej plochy, čo nie je želané správanie aplikácie.

`render()`

Jedným z atribútov komponenty `Node` je atribút `handle`, ktorý určuje, či vrámci daného uzla chceme zobrazovať lištu, ktorá funguje ako rukoväť k posunu daného uzla. To špecifikujeme v atribúte `handle` elementu `Draggable`. Tie uzly, ktoré rukoväť nemajú je možné posúvať po uchopení na ktoromkoľvek bode v rámci daného uzla.

Predvolene, elementy `Draggable` aj `Resizable` interne manažujú svoju pozíciu resp. veľkosť avšak v našom prípade dochádza k opätovnému vykresleniu okien pri prechode jednotlivými snímkami animácie, preto si musíme udržiavať stav veľkosti a pozícii jednotlivých okien sami. To zabezpečíme explicitným načítaním pozície a veľkosti v atribútoch `position` a `size`. Pre zachovanie konzistencie správania aplikácie povoľujeme zmenu veľkosti uzla iba na pravej a spodnej hrane elementu v atribúte `enable`.

K zobrazeniu okna využívame dizajnový jazyk Material Design, vyvinutý v roku 2014 korporáciou Google a to konkrétne elementy karty, `Card` a `CardContent`, ktorých umiestnenie jemne upravujeme v závislosti od použitia rukoväte vrámci uzla.

`Node.defaultProps`

Cez rozhranie `defaultProps` nastavujeme predvolenú hodnoty metódy `onResize`, ktorá je volaná pri zmene velkosti uzla v prípade ak inštancia komponenty `Node` ju neobdrží ako atribút. Takýmto spôsobom sa vyhneme možnému zlyhaniu a nezvratnému zastaveniu chodu aplikácie v dôsledku neexistujúcej metódy.

5.5.5 Komponent `Formula`

Komponent `Formula` rámcuje uzol, okno aplikácie, ktoré zobrazuje matematický výraz.

Atribúty: `content`, `x`, `y`, `onDrag`

`render()`

V metóde `render` vraciame komponent `Mathjax`, ktorý má za úlohu vykresliť matematický výraz zapísaný v jazyku TeX obalený uzlovým komponentom. U toho je deaktivovaná funkcionalita zmeny veľkosti, nakoľko k nej by bolo potrebné prepočítavať veľkosť zobrazovaného matematického výrazu vzhľadom na jeho štruktúru a použitú veľkosť jeho písma. To by bolo zbytočne výpočetne aj vývojovo náročné a nie je to obsahom tejto práce. Jednotlivým komponentom sú predané atribúty komponentu `Formula`.

5.5.6 Komponent `Mathjax`

Mathjax je komponent, ktorej jediná úloha je rámcovať funkcionalitu zobrazovania matematických výrazov.

Atribúty: `content`

Dôvodom k oddeleniu komponenty `Mathjax` od komponenty `Formula` je fakt, že komponenta `Formula` je zodpovedná aj za umiestnenie samotného uzla. Umiestnenie uzla je súčasťou atribútov komponenty `Formula` a pri zmenách atribútov dochádza k obnove celej komponenty, čo by v prípade zjednotenie komponent `Mathjax` a `Formula`znamenalo, že pri každej zmene pozície komponenty by dochádzalo k opätovnému vykreslovaniu vzorca i v prípade, že by samotný vzorec ostával bez zmeny.

`componentDidMount()`

Po nasadení komponentu do DOM štruktúry webovej aplikácie voláme metódu, ktorá je zodpovedná za vykreslenie matematického výrazu v rámci danej komponenty.

`componentDidUpdate()`

Po aktualizácii komponenty voláme metódu, ktorá je zodpovedná za vykreslenie matematického výrazu v rámci danej komponenty.

`renderMathjax()`

Metóda `renderMathjax()` má za účel opätovné vykreslenie matematického vzorca vrámci individuálnej komponenty. V metóde posúvame volanie metódy `Typeset` objektu `MathJax.Hub` do fronty knižnice typografického procesora matematických výrazov MathJax. Opätovné vykreslovanie matematického výrazu uskutočňujeme iba vrámci elementu `div` súčasnej komponenty `Mathjax`, na ktorý sa odkazujeme v premennej triedy `formula`.

`render()`

Vraciame výslednú HTML štruktúru, pričom ukladáme referenciu na obaľovací `div` element do premennej inštancie komponentu `Mathjax` s názvom `formula`. Matematický výraz zapísaný v jazyku TeX obalujeme oddeľovačom `$$`, ktorý knižnici MathJax určuje vykreslenie matematického vzorca.

6. Záver

V našej práci sme sa pokúsili vytvoriť programovací jazyk využívajúci syntaxe XML za účelom vizualizácie vybraných numerických algoritmov a k nemu určené vizualizačné prostredie. V teoretickej časti sme sa oboznámili s numerickými algoritmami určenými k vizualizácii, so znakmi, princípmi a kritériami programovacích jazykov a tieto poznatky sme postupne aplikovali pri vývoji nášho XML jazyka a jeho vizualizačného prostredia. Pri tvorbe tohoto XML jazyka sme si zvolili postup, v ktorom sme najprv definovali potreby numerických algoritmov, neskôr potreby vizualizácie a až na základe nich sme navrhovali samotný programovací jazyk. Samotný proces tvorby programovacieho jazyka a jeho vizualizačného prostredia bol ovplyvnený náročnosťou jednotlivých požiadaviek algoritmov na programovací jazyk a na jeho prostredie. Okrem toho sme chceli tiež overiť vhodnosť XML syntaxe k vytvoreniu programovacieho jazyka. Na základe našej praktickej skúsenosti usudzujeme, že programovací jazyk využívajúci XML syntaxe môže nájsť svoje okrajové uplatnenie vo vybraných oblastiach. Takisto usudzujeme, že samotná XML syntax má svojou štruktúrou blízko k funkcionálnej paradigme programovania. Preto pri tvorbe akéhokoľvek programovacieho jazyka založeného na syntaxi XML odporúčame nasledovanie a aplikovanie funkcionálneho prístupu pri výbere a návrhu štruktúry programovacieho jazyka.

Nami vytvorený XML jazyk a k nemu vytvorená aplikácia vizualizačného prostredia ponúka mnoho príležitostí na vylepšenie. Jedným z nevyhnutných možných vylepšení XML jazyka je poskytnutie spätnej väzby vypisovania chýb pri jeho kompilácii za účelom lepšej informovanosti programátora, ktorý v ňom zapisuje algoritmus alebo program. Takisto sa dá uvažovať o vylepšení štruktúry XML jazyka v záujme zníženia závislosti od programovacieho jazyka Javascript, do ktorého je náš XML jazyk kompilovaný. Inými slovami, o nahradení niektorých dlhých hodnôt atribútov XML značiek, ktoré tvoria časti javascriptového kódu, novými XML značkami. Ďalším funkčným vylepšením aplikácie, pri rozširovaní oboru aplikáciou vizualizovateľných algoritmov, je rozšírenie komponentu grafu o zobrazovanie vektorov. Z tých menších vizuálnych vylepšení je možné uvažovať o vodiacich horizontálnych čiarach v okne grafu alebo o schopnosti meniť veľkosť matematických výrazov. Dá sa takisto uvažovať o vytvorení ďalšieho druhu komponentu pre zapisovanie obyčajného textu. Výkonnosť aplikácie môže byť vylepšená menej častým volaním funkcie prekreslenia grafu, napr. iba v prípadoch zmeny snímky animácie alebo načítania novej animácie. Zobrazovanie a prekreslovanie matematických výrazov môže byť efektívnejšie ak budeme matematické výrazy pri každej zmene snímku a načítaní novej animácie vykreslovať naraz. Z tých ambicióznejších nápadov sa ponúkajú myšlienky o možnosti deklarovania vlastných XML značiek vrámci jazyka alebo existencie nových typov okien vo vizualizačnom prostredí ako napríklad okna určeného na vykreslovanie matematickej štruktúry grafu ako štruktúry uzlov a hrán, čo by otvorilo možnosť vizualizácie celej ďalšej kategórie algoritmov.

Okrem pedagogického nástroja na popis obmedzenej množiny numerických algoritmov aplikácia môže poskytnúť užitočné riešenia problémov, s ktorými sa vývojár stretáva pri vytváraní okenných rozhraní, vykreslovaní matematických funkcií ako aj pri kompilovaní XML dokumentov.

Zoznam použitej literatúry

[1] A. Toffler. *Future Shock*. Random House, 1745 Broadway, New York, Spojené štáty americké, 1970.

[2] J. Solomon. *Numerical Algorithms: Methods for Computer Vision, Machine Learning, and Graphics*. CRC Press, Boca Raton, Florida, Spojené štáty americké, 2015.

[3] D. Flanagan. *Canvas Pocket Reference*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, Spojené štáty americké, 2011.

[4] M. Might. A very short introduction to html5's canvas tag element: Graphing mathematical functions in javascript. `http://matt.might.net/articles/rendering-mathematical-functions-in-javascript-with-canvas-html/`, Október 2012. (Posledný krát navštívené 18.05.2018).

[5] D. Geary. *Core HTML5 Canvas: Graphics, Animation, and Game Development*. Prentice Hall, Upper Saddle River, New Jersey, Spojené štáty americké, 2012.

[6] S. Ruby, D. Thomas, and D. H. Hansson. *Agile Web Development with Rails 5*. The Pragmatic Bookshelf, 9650 Strickland Rd, Raleigh, Spojené štáty americké, 2016.

[7] W3C XML Working Group (WG). Extensible markup language (xml) 1.0: W3c recommendation 10-february-1998. `https://www.w3.org/TR/1998/REC-xml-19980210`, Február 1998. (Posledný krát navštívené 18.05.2018).

[8] E. Castro. *XML: For the world wide web*. Peachpit Press, 1249 Eighth Street, Berkeley, Spojené štáty americké, 2001.

[9] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 340 Pine Street, San Francisco, Spojené štáty americké, 2000.

[10] C. K. Louden and A. K. Lambert. *Programming Languages: Principles and Practice*. Cengage Learning, 20 Channel Center Street, Boston, Spojené štáty americké, 2011.

[11] R. Wenger. *Isosurfaces: Geometry, Topology, and Algorithms*. AK Peters/CRC Press, Boca Raton, Florida, Spojené štáty americké, 2013.

[12] D. Tidwell. *XSLT: Mastering XML Transformations*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, Spojené štáty americké, 2008.

[13] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0 (fifth edition). `https://www.w3.org/TR/2008/REC-xml-20081126/#NT-Name`, November 2008. (Posledný krát navštívené 18.05.2018).

[14] L. Tagliaferri. How to install ruby on rails with rbenv on ubuntu 16.04. `https://www.digitalocean.com/community/tutorials/how-to-install-ruby-on-rails-with-rbenv-on-ubuntu-16-04`, August 2016. (Posledný krát navštívené 18.05.2018).

Seznam obrázků

Prílohy

6.1 Algoritmus simulovaného žíhania

```
<!-- Generovanie poľa s obsahom 200 náhodných hodnôt -->
<Array length="200" name="data" random="true" />

<!-- Deklarácia potrebných premenných -->
<Var alpha="0.9" new_x="" new_y="" p="" t="1.0" t_min="0.00001" x=""
     y="data[Math.floor(Math.random() * (data.length - 1))]" />

<!-- Hlavný cyklus algoritmu simulovaného žíhania -->
<!-- Overujeme či nová teplota ešte nedosiahla minimum -->
<while cond="t > t_min">

  <!-- Vnútorný cyklus algoritmu simulovaného žíhania -->
  <Var i="0" />
  <while cond="i < 100">

    <!-- Náhodne vyberáme index nášho poľa náhodných hodnôt a podľa
         teploty určíme pravdepodobnosť s akou zmeníme naše
         odhadované maximum -->
    <set new_x="Math.floor(Math.random() * (data.length - 1))"
         new_y="data[new_x]" p="Math.E * (new_y - y) / t" />

    <if cond="p > Math.random()">
      <!-- Nastavíme hodnoty nového maxima -->
      <set x="new_x" y="new_y" />

      <!-- Vykreslíme snímok našej animácie, vrátane indexovej aj
           funkčnej hodnoty nášho maxima a jeho polohy na grafe -->
      <Frame>
        <Formula content="x = @{x}" name="x" x="100" y="100" />
        <Formula content="y = @{y}" name="y" x="100" y="150" />
        <Graph name="1" ref="data" x="100" y="200">
          <Marker x="x" vertical="true" />
        </Graph>
      </Frame>
    </if>

    <set i="i + 1" />
  </while>

  <!-- Znižujeme teplotu o koeficient alfa -->
  <set t="t * alpha" />
</while>
```

6.2 Metóda gradient descent

```
<!-- Deklarácia vstupnej funkcie a jej derivácie -->
<Function formula="x*x + 5" name="f" />
<Function formula="2*x" name="df" />

<!-- Deklarácia potrebných premenných -->
<Var x="Math.random() * 100" lrate="0.1" gradient="" e="0" />

<!-- Hlavný cyklus metódy gradient descent -->
<while cond="e < 100">

  <!-- Počítame gradient v bode x. Gradient určuje
       "smer", ktorým algoritmus hľadá extrém -->
  <set gradient="df(x)" x="x - lrate * gradient" />

  <!-- Vykresľujeme hodnotu nového minima
       a jeho umiestnenie na grafe funkcie -->
  <Frame>
    <Formula content="x = @{x}" name="x" x="100" y="100" />
    <Formula content="f(x) = @{f(x)}" name="y" x="100" y="150" />
    <Graph name="1" ref="f" x="100" y="210">
      <Marker x="x" />
      <Range x="-10..10" y="-100..100" />
    </Graph>
  </Frame>

  <!-- Inkrementujeme iterátor -->
  <set e="e + 1" />
</while>
```

6.3 Metóda gradient descent (2 premenné)

```
<!-- Deklarácia vstupnej funkcie a jej derivácii -->
<Function formula="x*x + y*y" name="f" />
<Function formula="2*x" name="dfx" />
<Function formula="2*y" name="dfy" />

<!-- Deklarácia potrebných premenných -->
<Var x="Math.random() * 100" y="Math.random() * 100" lrate="0.1"
     gradientx="" gradienty="" e="0" />

<!-- Hlavný cyklus metódy gradient descent -->
<while cond="e < 100">

  <!-- Počítame gradienty v bodoch x, y. Tie určujú "smer",
       ktorým náš algoritmus hľadá extrém -->
  <set gradientx="dfx(x)" gradienty="dfy(y)"
       x="x - lrate * gradientx" y="y - lrate * gradienty" />

  <!-- Vykresľujeme súradnice nového minima, jeho funkčnú hodnotu
       a jeho umiestnenie na grafe funkcie -->
  <Frame>
    <Formula content="x = @{x}" name="x" x="100" y="50" />
    <Formula content="y = @{y}" name="y" x="100" y="100" />
    <Formula content="f(x, y) = @{f(x, y)}" name="f" x="100" y="150" />

    <Graph name="1" ref="f" x="100" y="200"
           isocontours="10" height="300" width="300">
      <Range x="-100..100" y="-100..100" />
      <Marker x="x" y="y" />
    </Graph>
  </Frame>

  <!-- Inkrementujeme iterátor -->
  <set e="e + 1" />
</while>
```

6.4 Metóda násobnej iterácie

```
<!-- Deklarácia potrebných premenných -->
<Var i="0" lambda="" matrix="[[2, -12], [1, -5]]" v="[[1], [1]]" />

<!-- Deklarácia potrebných funkcií -->
<Function name="product" formula="
  [ [m[0][0]*v[0][0] + m[0][1]*v[1][0]],
    [m[1][0]*v[0][0] + m[1][1]*v[1][0]] ]" />
<Function name="eigenvalue" formula="a[0][0] / v[0][0]" />

<!-- Hlavný cyklus násobného algoritmu -->
<while cond="i < 70">

  <!-- Počítame nový vektor v -->
  <set v="product(matrix, v)" />

  <!-- Vo vektore 'v' nájdeme hodnotu
       s minimálnou absolútnou vzdialenosťou od nuly -->
  <Var min="Number.MAX_SAFE_INTEGER" minI="0" j="0" />
  <while cond="j < v.length">
      <Var abs="Math.abs(v[j])" />
      <if cond="abs < min">
        <set min="abs" minI="j" />
      </if>
      <set j="j + 1" />
  </while>

  <!-- Delíme hodnoty vektoru tak
       aby na indexe tejto hodnoty bola 1 -->
  <Var l="0" />
  <while cond="l < v.length">
    <set name="v[l][0]" value="v[l][0] / v[minI][0]" />
    <set l="l + 1" />
  </while>

  <!-- Inkrementujeme iterátor a vypočítame
       novú hodnotu vlastného čísla -->
  <set i="i + 1" lambda="eigenvalue(product(matrix, v), v)" />

  <!-- Vykreslíme matematický zápis definície vlastného čísla a
       vlastného vektoru dosadený o naše hodnoty -->
  <Formula name="f" content=" \left( \begin{array}{cc}
      @{matrix[0][0]} \& @{matrix[0][1]} \\
      @{matrix[1][0]} \& @{matrix[1][1]}
    \end{array} \right) \left( \begin{array}{c}
      @{v[0][0]} \\ @{v[1][0]}
    \end{array} \right) = @{lambda} \cdot \left( \begin{array}{c}
      @{v[0][0]} \\ @{v[1][0]}
    \end{array} \right)" x="100" y="100" />
</while>
```