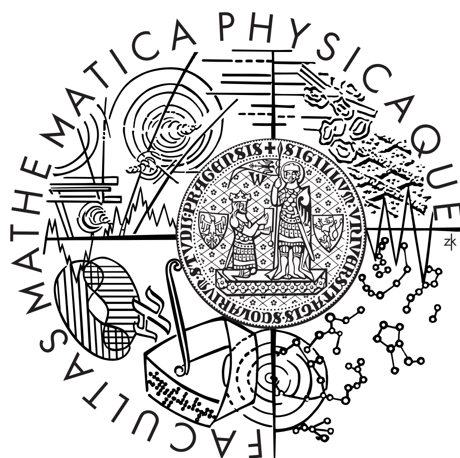
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Marcel Kikta

Vizualizace geometrických algoritmů

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.

Studijní program: Informatika

Studijní obor: Programovaní

Praha 2012

Chcel by som sa poďakovať svojmu vedúcemu RNDr. Martinovi Pergelovi, Ph.D. za rady a čas, ktoré mi venoval počas písania bakalárskej práce. Ďalej by som sa chcel poďakovať mojím rodičom, ktorí ma podporovali počas štúdia.

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne a výhradne s použitím citovaných prameňov, literatúry a ďalších odborných zdrojov.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona v platnom znení, hlavne skutočnosť, že Univerzita Karlova v Prahe má právo na uzavretie licenčnej zmluvy o používaní tejto práce ako školského diela podľa §60 odst. 1 autorského zákona.

V .... dňa .... Podpis autora

Názov práce: Vizualizace geometrických algoritmů

Autor: Marcel Kikta

Katedra: Kabinet software a výuky informatiky

Vedúci bakalárskej práce: RNDr. Martin Pergel, Ph.D.

Abstrakt: Práca sa zaoberá návrhom programovacieho jazyka, prekladača a interpretera určeného na vizualizovanie geometrických algoritmov. Dôraz je kladený na návrh čo najjednoduchšieho jazyka a na kvalitu vizualizačného prostredia. Počas implementácie prekladača vznikol aj generátor syntaktického LL(1) automatu. Práca obsahuje ako ukážku funkčnosti implementáciu niekoľkých geometrických algoritmov naprogramovaných v navrhnutom jazyku.

Kľúčové slová: vizualizácia, geometria, algoritmus

Title: Geometric algorithm visualization

Author: Marcel Kikta

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Martin Pergel, Ph.D.

Abstract: This thesis deals with designing programming language, compiler and intepreter assigned for visualization of geometric algorithms. Emphasis is put on designing the simplest language and on quality of visualization environment. During compiler implementation a syntactic LL(1) parser generator was created. As a demonstration of functionality thesis contains implementation of some geometric algorithms written in designed language.

Keywords: visualization, geometry, algorithm

Obsah

Úvod **3**

1 Štruktúra prekladača **4**

1.1 Lexikálna analýza 4

1.2 Syntaktická analýza 6

1.2.1 Množiny FIRST a FOLLOW 6

1.2.2 LL(1) gramatiky 7

1.2.3 Konštrukcia LL(1) automatu 9

1.3 Sémantická analýza 10

1.4 Medzikódy 10

2 Návrh riešení **12**

2.1 Návrh jazyka 12

2.2 Syntaktická analýza 13

2.3 Tabuľky symbolov 14

2.4 Medzikód a sémantická analýza 15

2.5 Reprezentácia medzikódu 16

2.6 Dynamicky alokované štruktúry 16

2.7 Predávanie parametrov do funkcie referenciou 17

2.8 Návrh grafického rozhrania 17

2.9 Krokovanie algoritmov 18

3 Popis vizualizovaných algoritmov **19**

3.1 3D konvexný obal algoritmom Gift wrapping 19

3.1.1 2D verzia 19

3.1.2 3D verzia 20

3.2 Algoritmus na hľadanie najbližšej dvojice bodov v rovine 21

3.3 Hľadanie najmenšej gule ohraničujúcej dané body v trojrozmernom priestore 21

4 Užívateľská príručka **24**

4.1 Popis jazyka 24

4.1.1 Typy 24

4.1.2 Premenné 25

4.1.3 Komentáre 25

4.1.4 Riadiace štruktúry 25

4.1.5 Funkcie 26

4.1.6 Operátory 26

4.2 Spustenie programu 28

4.3 Interpreter 28

5 Implementácia **31**

5.1 Generátor zásobníkového automatu 31

5.2 Prekladač 34

5.3 Interpreter 35

Záver **37**

Zoznam použitej literatúry **38**

Popis prílohy na CD **39**

Úvod

Výuka algoritmov je neodmysliteľnou súčasťou štúdia informatiky. Algoritmy môžu byť reprezentované buď ako pseudokód alebo slovným popisom. Medzi najrýchlejšie metódy pochopenia fungovania algoritmov patrí metóda vizualizácie.

Hlavným cieľom tejto práce je návrh programovacieho jazyka, prekladača a interpretera určených na vizualizovanie geometrických algoritmov v trojrozmernom priestore. Práca sa skladá z dvoch častí, a to prekladača a interpretera. Prekladač prekladá jednoduchý procedurálny jazyk do zásobníkového medzikódu. Vygenerovaný medzikód následne vykonáva interpreter jazyka, ktorý umožňuje vizualizovať algoritmy rôznou rýchlosťou a zároveň ich posúvať smerom vzad. Pri implementácii prekladača vznikol aj nástroj na generovanie inštrukcií syntaktického analyzátora, ktorý dokáže spracovať akúkoľvek LL(1) gramatiku.

Existuje niekoľko projektov, ktoré sa zaoberajú vizualizáciou algoritmov. Práve geometrickými algoritmami sa zaoberá Ondřej Skutka vo svojej diplomovej práci [9]. Práca Ondřeja Skutky vizualizuje rovnakú triedu algoritmov, ale neobsahuje programovací jazyk navrhnutý pre tento účel, namiesto toho umožňuje pridávať algoritmy pomocou zásuvných modulov. Ďalší projekt Luďka Kučeru Algovize [8] sa zaoberá vizualizovaním rôznych skupín algoritmov, ale neumožňuje editovať kód algoritmov. Projekt, ktorý obsahuje jazyk špecializovaný práve na geometrické algoritmy, by sme asi hľadali ťažko. Preto môžeme porovnať vizualizátor geometrických algoritmov s klasickým objektovým jazykom, kde by užívateľ na vizualizovanie použil nejakú 3D grafickú knižnicu. Táto práca má výhodu v jednoduchosti navrhnutého jazyka bez potreby študovania zložitej knižnice. Ďalšou výhodou je možnosť posúvania algoritmu po krokoch dozadu, ktorá v klasickom jazyku chýba, a teda odráža unikátnosť tohto projektu.

Prvá kapitola obsahuje teoretický úvod do prekladačov, a to hlavne teóriu, ktorá bola použitá pri implementácii prekladača. V druhej kapitole popisujeme návrh jazyka a analyzujeme možné riešenia implementačných problémov. V tretej kapitole sa nachádza popis vizualizovaných algoritmov, ktoré sú ukážkou funkčnosti projektu. Podrobný popis jazyka a užívateľská príručka je vo štvrtej kapitole. Piata kapitola opisuje vytvorený program z pohľadu programátora.

1. Štruktúra prekladača

Hlavnou časťou tejto práce je návrh jazyka a implementácia prekladača vhodného na vizualizovanie geometrických algoritmov. V tejto kapitole sa budeme zaoberať jednotlivými časťami prekladača. Väčšina informácií v spísaných v nasledujúcich podkapitolách sú prevzaté z knihy *Compilers Principles, Techniques and Tools*[1]. Prekladačom rozumieme program, ktorý prevádza zdrojový kód na cieľový kód. Vedľajším výstupom prekladača sú rôzne chybové hlásenia.

Prekladač môžeme rozdeliť na prednú (*front end*) a zadnú časť (*back end*). Front end prekladá zdrojový kód do medzikódu, ktorý by mal byť nezávislý na platforme. Back end prevádza tento kód do cieľových strojovo závislých inštrukcií a pritom vykonáva rôzne optimalizácie.

Front end sa ďaľej delí na lexikálnu, syntaktickú, sémantickú časť, generátor kódu a optimalizátor kódu. Do lexikálnej časti vstupuje program ako postupnosť znakov a je prevedený na postupnosť tokenov. Tieto tokeny vstupujú do syntaktickej časti. Syntaktický parser ich podľa pravidiel gramatiky poskladá do syntaktického (derivačného) stromu. Sémantická analýza kontroluje derivačný strom, z ktorého generátor kódu generuje medzikód. Medzikód je vstupom do poslednej časti front endu (optimalizátoru), ktorý vykonáva rôzne vysokoúroňové optimalizácie.

1.1 Lexikálna analýza

Prvá časť prekladača má ako hlavnú úlohu načítať znaky zo vstupného programu, spájať ich do lexémov a na výstup poslať postupnosť tokenov. Lexikálny analyzátor funguje ako podprogram syntaktického analyzátora. Lexikálny analyzátor sa stará aj o odstránenie komentárov.

Existuje niekoľko dôvodov pre oddelenie syntaktickej a lexikálnej časti prekladača:

1. Zjednodušenie návrhu - oddelenie lexikálnej časti od syntaktickej podstatne zjednodušuje návrh syntaktického analyzátora.

2. Zlepšenie efektivity - oddelený lexikálny analýzator nám dovoľuje použiť špecializované techniky, ktoré slúžia len na lexikálnu časť a nezaoberajú sa parsovaním. Techniky bufferovania vstupu dokážu značne zrýchliť celý prekladač.

3. Prenositeľnosť - pri zmene kódovania vstupného textu postačia zmeny v lexikálnej časti.

Definícia 1. V nasledujúcom budeme využívať tieto pojmy:

- Lexém *je postupnosť znakov, slovo daného jazyka.*

- Vzorom *nazveme pravidlá, ktoré popisujú množinu vstupných lexémov pre token.*

- Token *je výstupný symbol lexikálnej analýzy. Ak existuje viac lexémov pre jeden token, lexikány analyzátor uloží do tokenu dodatočné informácie o lexéme.*

Jednotlivé vzory sa dajú jednoducho zapísať pomocou regulárnych výrazov. Regulárne výrazy sú jazyky práve rozpoznávané konečnými automatmi [2]. Lexikálny analyzátor postaví pre všetky vzory konečný automat a v každom výstupnom stave vráti token. Po každom rozpoznanom tokene sa automat reštartuje. Problém môže nastať, ak je vzor tokenu prefixom pre iný token (napr.: token $>$ a $>=$). V tomto prípade automat pokračuje vo výpočte a snaží sa spracovať čo najdlhší lexém.

V prípade že automat narazí na neznámy znak, ktorý nie je vo vstupnej abecede alebo je na vstupe slovo, ktoré nezodpovedá žiadnemu vzoru, tak nastáva chyba. Automat zahlási chybu a pokúsi sa zotaviť z chyby. Existuje niekoľko metód zotavovania z chýb:

- ignorovanie znakov až pokým automat nenájde správny znak na vytvorenie tokenu

- nenahlásenie chyby a vrátenie špeciálneho tokenu, ktorý nie je vo vstupnej abecede; túto chybu potom spracuje syntaktický analyzátor

- zmazanie prebytočného znaku

- vloženie chýbajúceho znaku
- náhrada nesprávneho znaku správnym
- prehodenie dvoch susedných znakov

Medzi lexikálne chyby nepatrí preklep v kľúčovom slove. Napríklad ak programátor zadá *fi* namiesto *if*, analyzátor to nespracuje ako if token ale ako identifikátor. Tieto chyby potom rieši až syntaktický parser.

1.2 Syntaktická analýza

Syntaktický analyzátor zisťuje, či daná postupnosť tokenov z lexikálnej analýzy tvorí vetu v gramatike daného jazyka. Ďalšou úlohou je výstavba derivačného stromu, ktorý sa v sémantickej časti prekladača ďalej spracuváva.

V praxi sa používajú 2 typy parserov. Derivačný strom môže byť stavaný zhora nadol alebo zdola na hor. Uvedeným dvom postupom zodpovedajú 2 typy gramatík LL(k) a LR(k). V skratkách LL(k) a LR(k) prvé písmeno označuje, že zdrojový kód sa číta zľava (L), druhé písmeno označuje, či sa vytvára ľavá (L) alebo pravá derivácia (R). Posledný znak k označuje počet tokenov, ktoré syntaktický analyzátor potrebuje aby sa rozhodol, ktoré prepisovacie pravidlo gramatiky použije.

Ďalej sa budeme zaoberať iba LL(1) gramatikami a parsermi, pretože tieto nástroje boli použité aj pri implemetácii Vizualizátora geometrických algoritmov. Trieda gramatík LL(1) je dostatočne veľká na to, aby pokryla potreby väčšiny programovacích jazykov. Aby sme si mohli uviesť definíciu LL(1) gramatík, potrebujeme si zaviesť nasledujúce pojmy.

1.2.1 Množiny FIRST a FOLLOW

Definícia 2. *FIRST(α), kde α je reťazec symbolov gramatiky, je množina terminálov, ktorými začínajú terminálne slova, ktoré môžeme prepísať z α. Ak α generuje prázdne slovo (λ) tak $\lambda \in$ FIRST(α).*

Konštrukcia množiny $FIRST(A)$:

1. Ak je A terminál, tak $FIRST(A) = A$

2. Ak je A neterminál a $A \Rightarrow B_1, B_2...B_k$ je pravidlo gramatiky,pre nejaké $k > 0$, tak vložíme terminál a do $FIRST(A)$, ak pre nejaké i $a \in FIRST(B_i)$ a $\lambda \in FIRST(B_1)...FIRST(B_{i-1})$; ak $\lambda \in FIRST(B_j)$, pre všetky $j = 1..k$, tak vložíme λ do $FIRST(A)$;

3. Ak existuje pravidlo $A \Rightarrow \lambda$, pridáme λ do $FIRST(A)$

Definícia 3. *Pre neterminál A definujeme FOLLOW(A) ako množinu terminálov a, ktorá sa môže objaviť hneď napravo od A, to znamená, že existuje derivácia* $S \overset{*}{\Rightarrow} \alpha A a \beta$

Koštrukcia množiny $FOLLOW(A)$:

1. vložíme znak konca súboru($) do $FOLLOW(A)$, kde A je štartovací neterminál gramatiky

2. Ak existuje pravidlo $A \Rightarrow \alpha B \beta$, tak všetko z $FIRST(\beta)$ okrem λ vložíme do $FOLLOW(B)$

3. Ak existuje pravidlo $A \Rightarrow \alpha B$ alebo pravidlo $A \Rightarrow \alpha B \beta$, kde $FIRST(\beta)$ obsahuje λ, tak vložíme všetko z $FOLLOW(A)$ do $FOLLOW(B)$

1.2.2 LL(1) gramatiky

Definícia 4. *Gramatika G je v LL(1) práve vtedy a len vtedy ak* $A \rightarrow \alpha \mid \beta$ *sú dva jednoznačné pravidlá a platia nasledujúce podmienky:*

1. *Neexistuje terminál a, pre ktorý pravidlá α a β generujú terminálne slovo začínajúce na a.*

2. *Najviac jedno z pravidiel α a β generuje prázdne slovo.*

3. *Ak β vygeneruje prázdne slovo, potom α negeneruje žiadne terminálne slovo začínajúce terminálom z $FOLLOW(\alpha)$. Podobne ak α vygeneruje prázdne slovo, potom β negeneruje žiadne terminálne slovo začínajúce terminálom z $FOLLOW(\beta)$.*

Do triedy LL(1) gramatík nepatria *nejednoznačné gramatiky* a *ľavo rekurzívne gramatiky.*

Definícia 5. *Gramatika je nejednoznačná, ak existujú 2 rôzne ľavé derivácie, ktoré generujú to isté terminálne slovo. Príklad nejednoznačnej gramatiky, vyskytujúcej sa v programovacích jazykoch:*

$$S \Rightarrow if\ then\ S\ else\ S \mid if\ then\ S \mid s$$

Derivované slovo *if then if then s else s* má dva významy. Buď to môžeme interpretovať ako *if then (if then s else s)* alebo *if then (if then s) else s*. Existuje niekoľko riešení:

1. syntaktická chyba (Algol 60)

2. **else** patrí k najbližšiemu **if**; to sa docieli prepísaním gramatiky na:

$$S \Rightarrow M \mid O$$

$$M \Rightarrow if\ then\ M\ else\ M \mid s$$

$$O \Rightarrow if\ then\ S \mid if\ then\ M\ else\ O$$

   Toto riešenie funguje však iba u LR parserov.

3. pridanie zátvoriek **begin** a **end**

Definícia 6. *Gramatika je ľavo rekurzívna, ak pre neterminál A a reťazec symbolov gramatiky α, existuje derivácia $A \overset{*}{\Rightarrow} A\alpha$.*

Gramatiky s ľavorekurzívnymi pravidlami sa dajú jednoducho prepísať bez ľavej rekurzie. Napríklad gramatiku

$$A \Rightarrow A\alpha \mid \beta$$

prepíšeme na

$$A \Rightarrow \beta X$$

$$X \Rightarrow \alpha X \mid \lambda$$

LL(1) automat si tiež nevie poradit spravidlami typu:

$$A \Rightarrow \alpha\beta$$

$$A \Rightarrow \alpha\gamma$$

Pri vyberaní pravidiel nie je jasné ktoré sa má použiť. Riešenie je odložiť rozhodovanie a pravilá prepísať na:

$$A \Rightarrow \alpha X$$
$$X \Rightarrow \beta$$
$$X \Rightarrow \gamma$$

1.2.3 Konštrukcia LL(1) automatu

LL(1) je klasický deterministický zásobníkový automat, ktorý prijíma prázdnym zásobníkom. Na vstupnej páske má tokeny (terminály) z lexikálnej analýzy, na zásobníku má len počiatočný neterminál. K každom kroku odoberie vrchol zásobniku a urobí jednu z nasledujúcich možností:

- ak je na vrchole terminál, tak prečíta terminál zo vstupu a zo zásobníka odstráni vrchol
- ak je na vrchole neterminál, tak sa podľa rozhodovacej tabuľky a terminálu na vstupe rozhodne, ktoré prepisovacie pravidlo použije a ním nahradí vrchol zasobníka; vstup v tomto kroku načíta.

Ak máme korektnú LL(1) gramatiku môžeme si skonštruovať tabuľku (M). M má riadky indexované neterminálmi a stĺpce terminálmi. Konštrukcia automatu pre pravidlo gramatiky $A \Rightarrow \alpha$:

1. Pre každý neterminál a z $FIRST(\alpha)$ pridáme $A \Rightarrow \alpha$ do $M(A, a)$
2. Ak $\lambda \in FIRST(\alpha)$, tak pre každý neterminál b z $FOLLOW(A)$ pridáme $A \Rightarrow \alpha$ do $M(A, b)$
3. Ak $\lambda \in FIRST(\alpha)$ a zároveň $\$ \in FOLLOW(A)$ pridáme $A \Rightarrow \alpha$ do $M(A, \$)$

$FOLLOW(A)$ stačí konštruovať len pre neterminály, ktoré generujú λ, pre ostatné by to bolo zbytočné. Na prázdne miesta tabuľky pridáme hlásenie chyby. Ak v nejakej bunke tabuľky sa nachádza 2 a viac pravidiel, tak spracovaná gramatika nepatrí do LL(1) a je potrebné ju upraviť.

1.3 Sémantická analýza

Úlohou sémnatickej analýzy je napĺňanie tabuliek symbolov a kontrola kontextových závislostí. V tabuľkách symbolov si prekladač udržuje informácie o definovaných typoch a funkciách, deklarovaných premenných a konštantách. Informácie v tabuľke symbolov sú používané na riešenie kontextových väzieb, typovú kontrolu a generovanie medzikódu [3].

Na sémantickú analýzu sa používajú atribútové gramatiky. Ku každému symbolu gramatiky je pridaná množina atribútov. Tieto atribúty môžu reprezentovať čokoľvek. Hodnoty atribútov sú definované sémantickými pravidlami, ktoré sú pridané k syntaktickým pravidlám gramatiky. Existujú dva typy atribútov:

1. *Syntetizované atribúty:* hodnota atribútu je vypočítaná z atribútov seba a synov v derivačnom strome.
2. *Dedičné atribúty:* hodnota atribútu je vypočítaná z atribútov otca, seba a súrodencov v derivačnom strome.

Aby sme mohli určiť poradie vykonávania sémantických pravidiel, je potrebné zostrojiť graf závislosti. Graf závislosti je orientovaný graf, kde vrcholom je každý atribút symbolu derivačného stromu. Ak hodnota atribútu X vznikla z atribútov $Y_1 \dots Y_n$, tak budú viesť orientované hrany z $Y_1 \dots Y_n$ do X. Topologickým zotriedením môžeme určiť poradie výpočtu sémantických pravidiel.

1.4 Medzikódy

Informácie o medzikódoch sú čerpané z učebného textu *Překladače* [3]. Medzikód je výstup z front endu prekladača. Samozrejme prekladače ho nemusia generovať a môžu priamo prekladať do cieľového kódu. Iné prekladače generujú ako výsledný produkt medzikód, ktorý je potom interpretovaný. Použitie medzikódu má nasledujúce výhody:

1. Zjednodušuje návrh prekladača pre iný cieľový jazyk.
2. Medzikód môžeme optimalizovať využitím rôznych stojovo nezávislých optimalizácii.

Druhy medzikódu:

- syntaktický strom alebo DAG (acyklický orientovaný graf)
- zásobníkový medzikód
- trojadresový medzikód

Za stromový medzikód môžeme považovať upravený syntaktický strom. Táto forma medzikódu je výhodnejšia na intereptovanie a menej vhodná na ďalší preklad.

Zásobníkový medzikód je linearizovaná reprezentácia syntaktického stromu. Je vhodný ako na interpretovanie, tak na další preklad do cieľového kódu. Je tvorený postupnosťou operácii nad zásobníkom. Každá z týchto inštrukcií predstavuje vloženie hodnoty alebo konštanty na zásobník, vykonanie aritmetickej alebo logickej operácie na zásobníku alebo uloženie vrcholu zásobníku do premmenej.

Trojadresový kód má inštrukcie typu $a = b\ op\ c$,kde a, b, c sú premenné, konštanty alebo dočasné premmené vytvorené prekladačom. *Op* predstavuje v podstate ľubovolnú operáciu. Hlavný rozdiel medzi trojadresovým a zásobníkovým kódom je, že trojadresový pomenúva operandy a značne zjednodušuje vysokoúrovňové optimalizácie.

2. Návrh riešení

V tejto kapitole je zachytený postupný vývoj jazyka a prekladača. Ďalej analyzujeme jednotlivé možnosti riešenia implementačných problémov.

2.1 Návrh jazyka

Prvou úlohou bolo navrhnúť jazyk vhodný na vizualizáciu geometrických algoritmov. Vzhľadom k časovej zložitosti sme vylúčili objektovo orientovaný jazyk a zamerali sme sa na procedurálny jazyk podobný Pascalu. Boli stanovené požiadavky, ktoré by mal spĺňať navrhovaný programovací jazyk:

1. syntax podobná Jave alebo C++
2. silne typovaný
3. riadiace štruktúry ako cykly (*for*, *while*) a podmienky (*if*)
4. možnosť dynamicky alokovať pamäť (možnosť vytvoriť spojový zoznam alebo binárny strom)
5. možnosť vytvárať funkcie a rekurzívne ich volať
6. interpretovaný jazyk

Syntax podobnú známemu programovaciemu jazyku sme zvolili, aby bol jazyk jednoduchý na naučenie. Podmienka silnej typovej kontroly bola pridaná do návrhu z dôvodu lepšej detekcie chýb programátora. Interpretovaný jazyk bol zvolený z časových dôvodov a jednoduchosti implementácie. Ostatné podmienky zabezpečovali, aby jazyk umožňoval implementovať akýkoľvek algoritmus.

Vytvorili sme jednoduchú gramatiku, ktorá dovoľovala deklarovať funkciu alebo štruktúru kdekoľvek v kóde okrem tela cyklu alebo funkcie. Aby bola splnená podmienka dynamickej alokácie, pridali sme operátor *new* a rozdelili sme typy na hodnotové a referenčné. Medzi hodnotové typy sme dali vstavané typy ako **int** (32 bitové celé číslo so znamienkom), **long** (64 bitové celé číslo so znamienkom), **real** (číslo s plavúcou desatinnou čiarkou), **bool** (typ s dvoma hodnotami true

alebo false) a **string** (ľubovoľne dlhý reťazec). Všetky štruktúry, ktoré si vytvorí programátor sú referenčné.

Aby bol tento jazyk špecializovaný na vizualizáciu geometrických algoritmov, rozhodli sme sa pridať vstavané štruktúry **point**, **line**, **polygon**, **color**. Tieto štruktúry už patria medzi referenčné. Všetky geometrické algoritmy okrem algoritmov pracujúcich s kružnicam si vystačia s navrhnutými tvarmi. Oblé tvary ako guľa je možné vygenerovať pomocou trojuholníkov.

Na zobrazovanie boli navrhnuté funkcie na vykreslenie alebo mazanie geometrických tvarov. Ďalej sme vytvorili funkce, ktoré môžu prefarbiť daný objekt alebo zmeniť prednastavenú farbu. Na vykresľovanie mnohostenov v 3D bolo potrebné dodať možnosť upravovať priehľadnosť objektov, preto sme navrhli funkcie, ktoré umožňovali nastaviť tento atribút. Takmer v každom geometrickom algoritme sú potrebné matematické funkcie. Z tohto dôvodu sa do návrhu dostali goniometrické (*sin*, *cos* ...), zaokrúhlovacie (*round*, *ceil*, *floor*) a matematické (*log*, *pow*, *sqrt*, *abs*, *rand*) funkcie. Aby tento programovací jazyk spĺňal všeobecné štandardy kladené na jazyk, boli do neho pridané vstavané funkcie umožňujúce prácu so vstupom a výstupom na konzolu a do súboru.

Keďže sa budú vizualizovať algoritmy, bolo potrebné navrhnúť spôsob krokovania, aby si užívateľ mohol krok po kroku v ľubovoľnej rýchlosti pozrieť algoritmus. Pre tento účel bola navrhnutá funkcia *step*.

2.2 Syntaktická analýza

Hlavným dôvodom pre výber LL(1) (viď podkapitola 1.2.2) gramatiky bolo, že tieto gramatiky sa dajú spracúvať aj bez automatizovaných prostriedkov. S hotovou gramatikou sme sa snažili vytvoriť inštrukcie zásobníkového automatu. Ručná tvorba množiny $FIRST$ (viď def. 2) bola jednoduchá. Pri použití gramatiky s približne 100 pravidlami však tvorba $FOLLOW$ (viď def. 3) predstavovala netriviálny problém. Z tohto dôvodu sme pristúpili ku kroku napísať si vlastný generátor inštrukcií zásobníkového stroja. Toto riešenie bolo veľmi praktické, lebo aj pri najmenších zmenách v gramatike nebolo nutné nanovo ručne vytvárať inštrukcie.

Pri generovaní inštrukcií vznikol problém nejednoznačnosti gramatiky (viď def. 5). Týkalo sa to dobre známeho problému nazvaného *dangling else*. Nejednoznačná gramatika vyzerala takto:

$$S \Rightarrow if\ (\ podmienka\ )\ S\ E$$

$$S \Rightarrow statement$$

$$E \Rightarrow else\ S \tag{2.1}$$

$$E \Rightarrow \lambda \tag{2.2}$$

Generátor zásobníkového automatu vygeneroval do rozhodovacej tabuľky na pozíciu $(E, else)$ dve prepisovacie pravidlá (2.1 a 2.2). Odstránením pravidla 2.2 z tabuľky bol vyriešený problém nejednoznačnosti a zároveň syntaktický analyzátor pripája *else* vždy k najbližšiemu *if*.

2.3 Tabuľky symbolov

Tabuľka symbolov bola navrhutá ako jedna trieda, ktorá poskytovala funkcie na vkladanie a vyhľadávanie premenných, funkcií a deklarovaných typov. Za účelom reprezentácie elementov tabuľky bola napísaná abstraktná trieda *Symbol*. Súčasťou jazyka by mali byť vstavané typy a zároveň by mal byť schopný premenovať typy, deklarovať si vlastné štruktúry a vytvárať polia všetkých typov. Z tohto dôvodu boli napísané triedy *Type*, *Struct* a *Field*, ktoré reprezentovali tieto štruktúry. Na reprezetovanie funkcií, ich parametrov a premenných boli navrhnuté triedy *Function*, *Variable* a *Parameter*. Tieto triedy sú zdedené z triedy *Symbol*.

Pôvodne sme chceli aby všetky deklarované objekty boli uložené v jednej hašovacej tabuľke. Hašovaciu tabuľku sme zvolili z dôvodu rýchleho vkladania a vyhľadávania. Pri neskoršom testovaní bolo zistené, že ak identifikujeme všetky objekty len podľa mena, tak nemôžeme pomenovať typ aj premennú rovnakým názvom. Nasledujúci kód nebol možný:

```
point point=new point;
```

Z tohto dôvodu sme sa rozhodli ukladať funkcie do jednej tabuľky, premenné do druhej a typy do tretej tabuľky.

2.4 Medzikód a sémantická analýza

Spomedzi možných foriem medzikódu (viď kap.1.4) sme sa rozhodoli pre zásobníkový kvôli predchádzajúcej znalosti tohto kódu a relatívnej zložitosti ostatných dvoch typov medzikódu v porovnaní s ním. Hlavnými výhodami zásobnikového medzikódu sú:

- jednoduchšie generovanie inštrukcií a interpretovanie oproti trojadresovému kódu
- jednoduchšia reprezentácia oproti stromovému medzikódu

Použitie atribútových gramatík sme vylúčili, lebo ich pridanie do generátora zásobníkových inštrukcií by značne skomplikovalo program. V syntaktickej analýze bol vygenerovaný syntaktický strom, ktorý sme prechádzali do hĺbky. Vďaka prechodu do hĺbky bolo možné zároveň generovať kód bez potreby vytvárania grafu závislosti. V takmer každom uzle bolo potrebné vykonať nejakú sémantickú kontrolu, a tak sme do generátora zásobníkového automatu ku každému syntaktickému pravidlu pridali možnosť vložiť názov funkcie, ktorá sa pri prechode stromom zavolala. Použili sme na to *Java RMI* (technológia, ktorá dokáže volať funkciu pomocou názvu). V priebehu programovania sémantickej analýzy sa však ukázalo, že volať len jednu funkciu nestačí. Preto sme sa rozhodli pridať možnosť volať viac funkcií. Pri prechode do hĺbky je možné presne určiť, kedy sa funkcia zavolá. Ak sa program nachádza v uzle, ktorý ma n synov, môže zavolať $n + 1$ rôznych funkcií, a to pri vstupe do uzlu, pri návrate z prvého syna, pri návrate z druhého syna atď. Táto funkčnosť dovoľovala bez komplikácií vykonať sémantickú analýzu a generovanie medzikódu.

Atribútové gramatiky si ukladajú potrebné informácie o uzloch derivačného stromu. V každom uzle samozrejme treba uchovávať iné informácie. Do každého uzlu by sme tak mohli vložiť všetky dátové štruktúry alebo navrhnúť spôsob ako tam uložiť len tie potrebné. Nakoniec sme sa rozhodli pre vytvorenie statickej triedy *Atributes*, do ktorej sme vložili všetky potrebné dátové štruktúry. Toto riešenie bolo najhodnejšie z dôvodu jednoduchého prístupu ku všetkým atribútom bez potreby hľadania atribútov v iných uzloch stromu.

Predchádzajúcimi úpravami generátora zásobníkového stroja vznikol nástroj, ktorý sa dá použiť na parsovanie akejkoľvek LL(1) gramatiky. Zároveň značne uľahčuje aj sémantickú analýzu.

2.5 Reprezentácia medzikódu

V prekladači je výsledný kód reprezentovaný ako pole objektov. Každý z týchto objektov je zdedený z abstraktnej triedy *Instruction*. Pri návrhu formátu súboru uloženia výsledného kódu boli prehodnocované tri možnosti:

1. uložiť kód do *XML*
2. uložiť kód do vlastného formátu
3. uložiť kód pomocou *Java serialization api*

Vytváranie vlastného formátu by bola asi najpracnejšia možnosť a bolo by potrebné vytvoriť vlastný parser cieľového kódu. Uloženie do *XML* bola o niečo lepšia alternatíva z dôvodu jednoduchého parsovania. Pri implementovaní tejto metódy sme zistili, že vytváranie inštrukcií z reťazca v *XML* je dosť nešikovné. Najjednoduchšie uloženie a parsovanie poskytlo nakoniec *Java serialization api*. Čo sa týka rýchlosti parsovania *serialization api* bolo približne rovnako rýchle ako *XML*. Výstupný súbor prekladača, ktorý obsahuje kód, má príponu *.ser*.

2.6 Dynamicky alokované štruktúry

V prípade, že by jazyk nepodporoval dynamicky alokované štruktúry, postačovalo by všetky premenné uložiť na spodok zásobníka. Pri dynamickej alokácii bolo navrhnuté, aby sa na zásobník uložil iba ukazateľ na pole alebo štruktúru a dáta by boli uložené inde. Prvý nápad bol vytvoriť si nejaké pole v pamäti a tam zapisovať dynamicky alokované štruktúry. Toto riešenie by však prinieslo priveľa komplikácii. *Garbage Colector Javy* (GC) by tieto štruktúry nevymazal a preto by musel byť napísaný vlastný GC. Ďalším problémom by bolo prideľovanie takto vytvorenej pamäte s ohľadom na efektivitu. Lepším nápadom bolo vytvárať dy-

namicky alokované objekty ako štruktúry v Jave a nechať starosti s prideľovaním pamäti na Javu.

Navrhovaný jazyk podporoval zložité štruktúry ako napríklad trojrozmerné pole štruktúry, v ktorej je uložené ďalšie dvojrozmerné pole. Linearizovať túto štruktúru by predstavovalo netriviálny problém. Preto sme sa rozhodli uložiť viacrozmerné polia do hierarchie jednorozmerných polí. V poslednom poli budú buď referencie na referenčné typy alebo hodnoty hodnotových typov. Pri tomto návrhu je zjednodušené pristupovanie k jednotlivým prvkom viacrozmerných polí.

2.7 Predávanie parametrov do funkcie referenciou

Možnosť predávania parametru do funkcie referenciou je užitočná najmä pri písaní rekurzívnych algoritmov, a preto bola pridaná do návrhu. Pri technickej realizácii nastal problém vytvorenia ukazateľa na premennú, lebo *Java* takúto konštrukciu nepodporuje. Rozmýšľali sme nad riešením získania adresy, kde je uložený ukazateľ na objekt, ktorý chceme predávať referenciou, ale to sa tiež nepodarilo uskutočniť z dôvodu obmedzení *Javy*. V navrhnutej hierarchii ukladania objektov popísanej v predchádzajúcej kapitole 2.6 si môžeme všimnúť, že všetky globálne a lokálne premenné sú uložené na zásobniku a ostatné hodnoty sú buď v štruktúre alebo v poli. Riešením tohto problému bolo vytvoriť ukazateľ na premennú pomocou dvoch parametrov:

1. pole rep. zásobník, v ktorom je premenná uložená
2. pozícia v poli

2.8 Návrh grafického rozhrania

Keďže hlavným grafickým prvkom je vizualizačná plocha, bola umiestnená do stredu grafického okna. Na pravú stanu bol pridaný panel umožňujúci ovládanie vizualizovania algoritmu. Pre potreby konzolového vstupu a výstupu boli do spodnej časti pridané 2 konzoly, jedna na vstup a druhá na výstup. Na vizualizovanie 2D a 3D algoritmov sme sa rozhodli použiť grafickú knižnicu *Java 3D* [10]. Táto

knižnica poskytovala všetky potrebné funkcie a bola vyvýjaná výrobcom *Javy* a preto sme ju zvolili pre tento projekt.

2.9 Krokovanie algoritmov

Posúvanie algoritmu smerom vpred po krokoch bolo implementačne jednoduché. Vďaka navrhnutej funkcii *step* mohol užívateľ algoritmus krokovať. Do grafického okna sme pridali možnosť nastavovať rýchlosť algoritmu. Užívateľ si môže určiť, či sa má algoritmus v každom kroku zastaviť a čakať na spustenie užívateľa alebo len zastaviť na niekoľko milisekúnd.

Aby sa užívateľ mohol v algoritme vracať naspäť, bolo žiadúce uložiť stav interpretera v danom momente do pamäte. To znamená, že bolo potrebné uložiť premenné v programe a stav zavolaných funkcií. Ďalej sme museli zabezpečiť, aby sa vstupná a výstupná konzola uložila. Mohli sme všetky tieto objekty skopírovať do pamäte, ale jednoduchšie riešenie bolo uložiť ich opäť pomocou *Java serialization api*.

Ukladanie vizualizovaných objektov sa nedalo spraviť takto jednoducho, lebo tieto objekty neimplementovali rozhranie *serializable*. Pri krokovaní algoritmu sa vykresľuje alebo maže v každom kroku len malé množstvo objektov, a teda výhodnejšie bolo uložiť len zmeny.

Po pridaní vstupu a výstupu do súboru sa ukladanie interpretera skomplikovalo. Krokovanie čítania a zápisov do súboru nebolo možné z nasledujúcich dôvodov:

- za predpokladu krokovania zápisu do súboru by sa pri kroku späť museli zmazať dáta zapísané v danom kroku
- pri krokovaní čítania zo súboru by sme sa museli vrátiť späť v súbore, čo by síce nebolo nemožné, avšak obtiažne

Riešením tohto problému bolo nevykonávať uloženie stavu interpretera, ak je otvorený nejaký súbor na čítanie alebo zápis. Toto obmedzenie nie je príliš zásadné. Klasický algoritmus funguje tak, že načíta dáta, vypočíta a zapíše výsledok. Z hľadiska vizualizácie je zaujímavá len fáza výpočtu, pri ktorej sa väčšinou zo súboru nič nečíta ani nezapisuje.

3. Popis vizualizovaných algoritmov

V tejto kapitole si popíšeme algoritmy, ktoré sú vizualizovné pomocou Vizualizátora geometrických algoritmov. V kapitole 3.1 je rozobratý algoritmus na hľadanie 3D konvexného obalu nazvaný Gift wrapping. V kapitole 3.2 si priblížime algoritmus na hľadanie najbližšej dvojice bodov v rovine. Tento algoritmus beží v čase $O(n \log(n))$. Posledný vizualizovaný algoritmus (3.3) je určený na nájdenie najmenšej gule obsahujúcej zadané body v trojrozmernom priestore.

3.1 3D konvexný obal algoritmom Gift wrapping

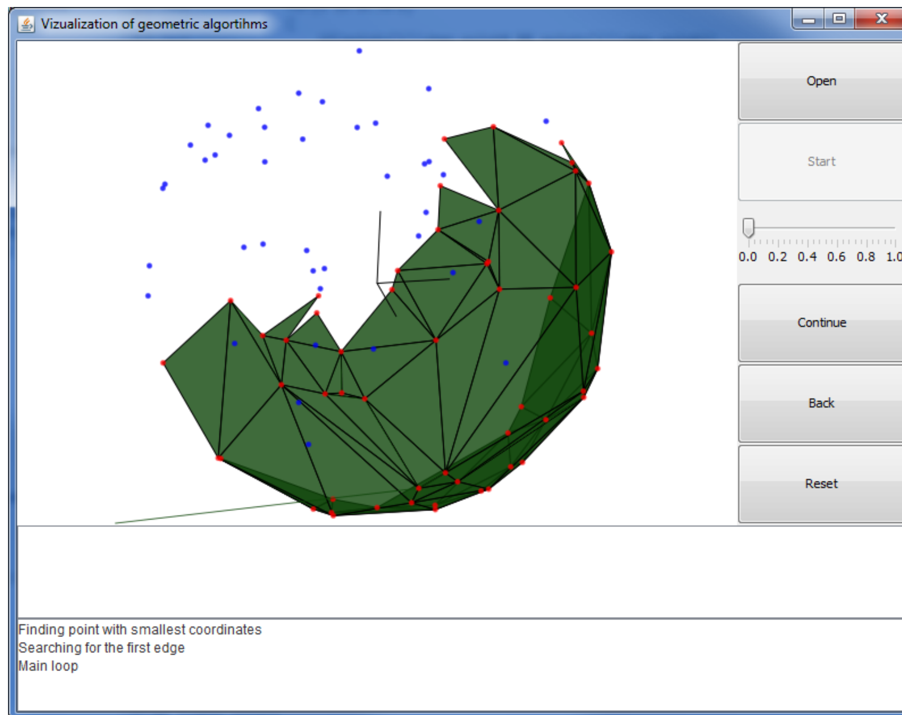
3.1.1 2D verzia

2D verzia algoritmu Gift wrapping je prevzatá z knihy Discrete and Computational Geometry [4]. Donald Chand a Sham Kapur v roku 1970 prvýkrát navrhli tento algoritmus a primárne je určený pre konvexné obaly vo vyšších dimenziách.

Majme množinu bodov v rovine, ktoré sa nachádzajú v všeobecnej polohe. Vyberieme bod s najmenšou y-ovou súradnicou. Ak existuje viac takýchto bodov, zvolíme ten, ktorý je najviac napravo, t.j. má najvyššiu x-ovú súradnicu. Vybraný bod si označme P. Bod P spojíme s ostatnými bodmi a vyberieme úsečku, ktorá zviera s x-ovou osou najväčší uhol. Zvolená úsečka je prvá hrana konvexného obalu.

Každú ďalšiu hranu konvexného obalu nájdeme pomocou predchádzajúcej hrany, ktorú si označíme H. Koncový (otvorený) bod úsečky H spojíme so všetkými ostatnými bodmi. Za ďalšiu hranu konvexného obalu vyberieme z vytvorených úsečiek tú, ktorá zviera s H najväčší uhol. Takto pokračujeme ďalej, kým nenarazíme na bod P.

Časová zložitosť algoritmu je $O(hn)$, kde h je počet bodov konvexného obalu a n je celkový počet bodov. Keďže konvexný obal može obsahovať všetky body, časová zložitosť je v najhoršom prípade $O(n^2)$.



Obr. 3.1: Postupná výstavba 3D konvexného obalu.

3.1.2 3D verzia

3D verzia algoritmu je prezvatá z učebného textu Geometrické algoritmy I [5]. Algoritmus sa dá priamo zovšeobecniť do trojrozmerného priestoru (viď obr. 3.1). Prvú hranu konvexného obalu môžeme nájsť pomocou 2D verzie algoritmu tak, že premietneme body do roviny xy. Nájdenú úsečku zvolíme za aktuálnu a označíme si ju AB. Pomocou úsečky AB a každého bodu vytvoríme rovinu. Z vytvorených rovín vyberieme tú, ktorá rozdeľuje priestor na 2 podpriestory tak, že jeden obsahuje všetky body a druhý žiadne. Bod, ktorý vytvára s úsečkou AB túto rovinu, označme P. Vytvoríme trouholník ABP, ktorý je zaroveň stenou konvexného obalu. Tento postup opakujeme, pokiaľ nie je konvexný obal uzavretý. Ak nájdeme neuzavretú hranu, označíme ju ako aktuálnu a cyklus opakujeme.

Časová zložitosť algoritmu je $O(fn)$ [4], kde f je počet stien a n je počet bodov. V najhoršom prípade môže byť $f = O(n)$, z čoho vyplýva, že celková zložitosť algoritmu v najhoršom prípade je $O(n^2)$. Hľadanie steny je možné implementovať v čase $O(n)$ tak, že porovnávame uhol už nájdenej steny, ktorá obsahuje aktuálnu hranu, a n možných stien. Rovinu zvierajúcu najväčší uhol použijeme na vytvorenie ďalšej steny.

3.2 Algoritmus na hľadanie najbližšej dvojice bodov v rovine

Jednoduchý algoritmus na hľadanie najbližších bodov v rovine porovnáva vzdialenosť každej dvojice bodov a vyberie tú najmenšiu. Vykoná dokopy $\binom{n}{2}$ krokov, to znamená že beží v čase $\Theta(n^2)$.

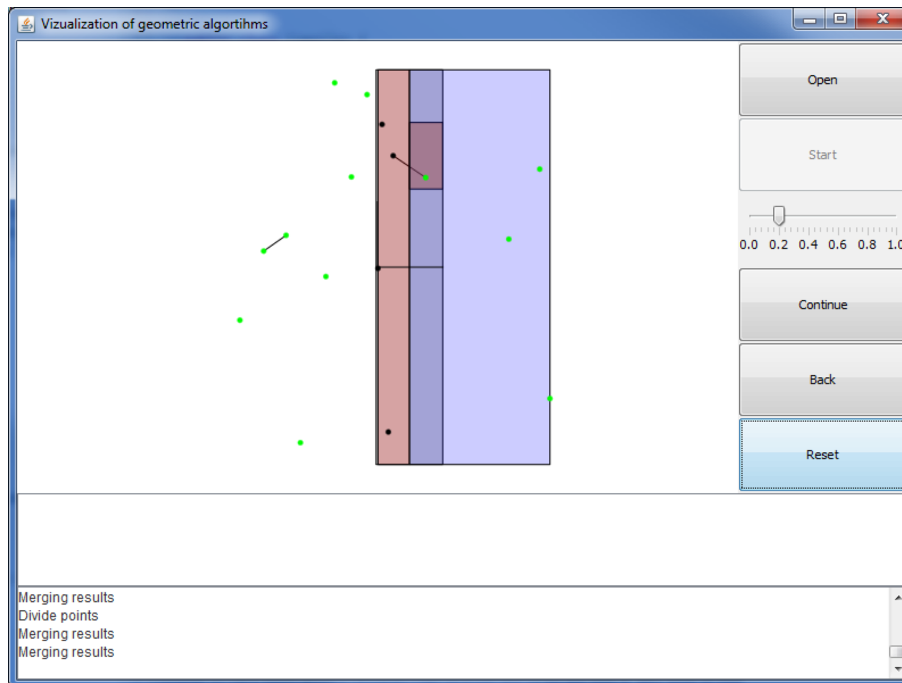
Rýchlejší algoritmus (viď obr. 3.2), prevzatý z knihy Introduction to Algorithms [6], dokáže nájsť dvojicu najbližších bodov v čase $O(n \log(n))$. Využíva metódu Rozdeľuj a panuj. Ak je počet bodov väčší ako tri, rozdelí body podľa x-ovej súradnice na rovnako veľké množiny (ľavú a pravú) a zavolá sa rekurzívne na obe množiny. V opačom prípade nájde najmenšiu dvojicu hrubou silou. Po vynorení z rekurzie máme zistené najmenšie vzdialenosti z ľavej (δ_l) a pravej (δ_p) časti. Označme zatiaľ výpočítanú minimálnu vzdialenosť $\delta = \min(\delta_l, \delta_p)$. Vieme, že najbližšia dvojica bodov bude ležať v ľavej alebo v pravej množine alebo jeden bod bude vľavo a jeden vpravo. Zostáva nám teda porovnať body, ktoré sa nachádzajú v rôznych množinách a zároveň majú maximálnu vzdialenosť δ od deliacej čiary. Označme si ich M_l a M_p. Pre každý bod z M_l existuje obdĺžnik (R) veľkosti $\delta \times 2\delta$, v ktorom sa môžu potenciálne najbližšie body z M_r nachádzať. Keďže body v M_r sú od seba vzdialené minimálne δ, tak v obdĺžniku R sa môže nachádzať maximálne šesť bodov. Tým pádom nám stačí pre každý bod v M_l urobiť maximálne $O(1)$ porovnaní, teda dokopy vykonáme $O(n)$ porovnaní.

Aby sme pri implementácii porovnávali len body z obdĺžnika R, je potrebné spoločne prechádzať zoznamy bodov M_l a M_p zotriedené podľa y-ovej súradnice. Tieto zoznamy môžeme získať predtriedením celej množiny bodov.

Časová zložitosť algoritmu je $T(n) = T(n/2) + O(n)$, čo je podľa Master theorem $O(n \log(n))$.

3.3 Hľadanie najmenšej gule ohraničujúcej dané body v trojrozmernom priestore

Pravdepodobnostný algoritmus (viď obr. 3.3) na nájdenie najmenšej gule obsahujúcej množinu zadaných bodov sme prevzali z publikácie Smallest enclosing

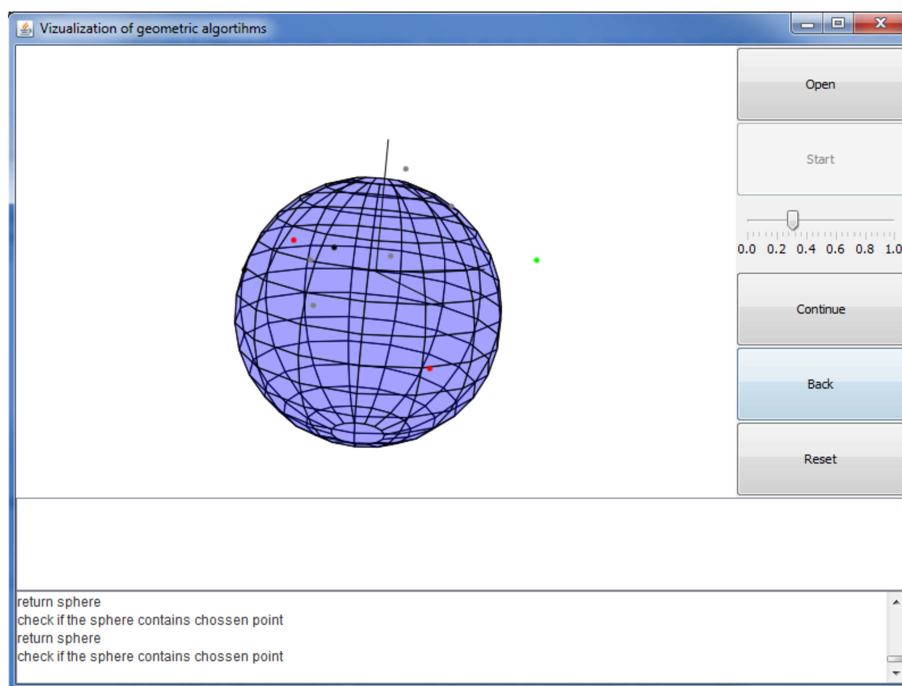


Obr. 3.2: Algoritmus na hľadanie najbližšej dvojice bodov v rovine vo fázi porovnávania bodov z rôznych množín.

disks (balls and ellipsoids) [7].

Celú funkčnosť zabezpečuje jedna rekurzívna funkcia $b_minidisk$ s parametrami P,R, kde P a R sú množiny bodov. Ak je P prázdna, vytvoríme kružnicu z bodov v R. V opačnom prípade vyberieme náhodný bod $p \in P$ a zavoláme rekurzívne funkciu $b_minidisk$ s parametrami $P - \{p\}$ a R, ktorá nám vráti guľu G. Ak $p \in G$, tak vrátime G. Pri zistení, že $p \notin G$, vieme, že p je jedným z bodov, ktoré sa nachádzajú na povrchu gule. Preto zavoláme znova funkciu $b_minidisk$ s parametrami $P - \{p\}$ a $R \cup \{p\}$ a tá nám vráti najmenšiu guľu obsahujúcu všetky body z $P \cup R$.

V najhoršom prípade môže algoritmus bežať v exponenciálnom čase, práve vtedy, ak vždy náhodne zvolíme bod z povrchu gule. Priemerne ale vykoná iba lineárny počet operácií [7].



Obr. 3.3: Algoritmus na hľadanie namenšej gule práve porvnáva, či zelený bod patrí do gule.

4. Užívateľská príručka

V tejto kapitole podrobne popíšeme súčasti navrhnutého jazyka (typy, premenné, komentáre, riadiace štruktúry, funkcie, operátory). V kapitole 4.3 si interpreter bližšie rozoberieme z pohľadu užívateľa.

4.1 Popis jazyka

Navrhnutý jazyk je procedurálny a silne typovaný. Gramatika jazyka je priložená na CD.

4.1.1 Typy

Jazyk obsahuje vnorené hodnotové typy:

1. **int** – 32-bitové celé číslo reprezentované v dvojkovom doplnku
2. **long** – 64-bitové celé číslo reprezentované v dvojkovom doplnku
3. **real** – 64-bitové číslo s plávajúcou desatinnou čiarkou podľa štandardu IEEE754
4. **bool** – typ s hodnotami **true** a **false**
5. **string** – ľubovoľne dlhý reťazec

Medzi vnorené referenčné typy patria:

1. **point** (bod) – štruktúra obsahujúca tri položky typu **real**: **x**, **y**, **z**
2. **line** (úsečka) – štruktúra obsahujúca dve položky typu **point**: **point0**, **point1**
3. **polygon** (mnohouholník) – štruktúra obsahujúca pole typu **point** (**array**) a položku typu **int** (**size**)
4. **color** (farba) – štruktúra obsahujúca tri položky typu real: **red**, **green**, **blue**

4.1.2 Premenné

Existujú 2 typy premenných: globálne a lokálne. Globálne premenné sú viditeľné v celom programe. Lokálne sú viditeľné len vo vnútri funkcií.

Pokiaľ premenná nebola inicializovaná, tak obsahuje prednastavenú hodnotu, ktorá sa rovná **null** pre všetky referenčné typy. Číselné hodnotové typy (**int**, **real**, **long**) majú prednastavenú hodnotu **0**, hodnota typu **bool** je **false** a typu **string** prázdny reťazec.

Pri inizializovaní referenčného typu pomocou operátoru **new** sa všetky položky vytváranej štruktúry nastavia na prednastavenú hodnotu popísanú v predchadzajúcom odstavci.

4.1.3 Komentáre

Programátor môže používať 2 typy komentárov:

1. jednoriadkové:

```
//comment
```

2. viacriadkové:

```
/* comment
*/
```

Viacriadkové komentáre nemôžu byt vnorené, to znamená, že komentár končí pri prvom výskyte symbolov `*/`.

4.1.4 Riadiace štruktúry

Mezi základné riadiace štruktúry patria podmienky (*if-else*) a cykly (*for*, *while*). Podmienka *if* funguje rovnako ako v iných programovacích jazykoch. Vyhodnocovaný výraz musí byť typu **bool**. Ak má hodnotu **true** vykoná sa kód vo vetve za *if*, ak nie, vykoná sa kód v *else* (v prípade že, štruktúra obsahuje vetvu *else*).

Cyklus *while* funguje podobne. Pokým má výraz hodnotu **true** tak sa telo cyklu vykonáva. Cyklus *for* sa však od bežných jazykov mierne odlišuje. Pracuje

len s premennými typu **real**, **long** a **int**. Na začiatku cyklu sa riadiaca premenná inicializuje na hodnotu prvého výrazu. Po každom vykonaní tela cyklu sa hodnota tejto premennej zvýši o jedna. Cyklus pokračuje kým je táto hodnota menšia alebo rovná ako druhý výraz.

4.1.5 Funkcie

Jazyk obsahuje ako vstavané funkcie (dispozícii na CD), tak i možnosť deklarácie vlastných funkcií programátora. Za kľúčovým slovom *of* sa určuje návratový typ funkcie. Funkcii, ktorá nevracia hodnotu, sa zadefinuje návratový typ na *void*.

Parametre môžu byť predávané do funkcie buď hodnotou alebo referenciou. V prípade predávania parametrov hodnotou sa pre hodnotový typ jednoducho skopíruje hodnota, pre referenčný typ sa analogicky kopíruje referencia. Parameter predávaný do funkcie referenciou sa v syntaxi jazyka označuje znakom &. V prípade predávania premennej referenciou sa vytvorí ukazateľ na danú premennú a pri priradení do nej sa zmení obsah aj mimo funkcie.

4.1.6 Operátory

Operátory $==$ a $!=$ sú definované na všetkých typoch. Pri hodnotových porovnávajú hodnotu, pri referenčných porovnávajú referenciu a výsledok je typu **bool**.

Binárny operátor $=$ je priraďovací operátor definovaný na všetkých typoch. Pri hodnotových kopíruje hodnotu, pri referenčných kopíruje referenciu. Za predpokladu, že operátor $=$ je použitý na číselné typy, ktoré sa zároveň nezhodujú, tak sa typ výrazu z pravej strany skonvertuje na typ, ktorý ma premenná na ľavej strane. Takto môže dôjsť ku strate informácií, napríklad pri priradení čísla 1.2585 do premennej typu long sa hodnota zmení na číslo 1. Tieto automatické pretypovania fungujú aj pri vracaní hodnoty z funkcie a pri predávaní parametrov hodnotou do funkcie. Stratové pretypovanie funguje nasledovne:

- pri pretypovaní **long** na **int** sa použije na int spodných 32 bitov
- pri pretypovaní **real** na **long** resp. **int** sa z reálneho čísla odrežú čísla za desatinnou čiarkou a v prípade, že by hodnota bola väčšia ako maximálna

hodnota **long** resp. **int**, dosadí sa maximálna hodnota **long** resp. **int**

Aritmetické binárne operátory $+, -, *, /, \%, >, <, >=, <=$ sú definované na číselných typoch **int**, **real**, **long**. Výnimku tvorí operátor modulo (%), ktorý nie je zadefinovaný pre typ **real**. Výsledok porovnávacích operátorov $(>, <, >=, <=)$ je typu **bool**. Ak sa niektorý z týchto operátorov použije na 2 rôzne typy, tak sa jeden z nic skonvertuje na ten všeobecnejší:

- pri použití premenných typov **int** a **long** sa **int** pretypuje na **long**
- pri použití premenných typov **int** a **real** sa **int** pretypuje na **real**
- pri použití premenných typov **long** a **real** sa **long** pretypuje na **real**

Výsledkom aritmetických operátorov $+, -, *, /, \%$ je:

- **int** pri použití premenných typov **int** a **int**
- **long** pri použití premenných typov **long** a **int** alebo **long** a **long**
- **real** pri použití premenných typov **int** a **real** alebo **long** a **real** alebo **real** a **real**

Pokiaľ aspoň jeden výraz operátora / je typu real, vykonáva sa reálne delenie. V opačnom prípade sa vykonáva celočíselné delenie. V jazyku sú definované aj unárne operátory $+, -$ na číselnych typoch. Návratová hodnota má rovnaký typ ako vstupný. Pri aritmetických operáciach môže dôjsť k pretečeniu alebo podtečeniu čísla. Takéto chyby nie sú hlásené.

Logické operátory &&, || a ! sú definované na výrazoch typu **bool** a vracajú opäť typ **bool**. Vyhodnocovanie binárnych operátorov && a || je vždy úplné, t.j. aj ked je výsledok istý vyhodnotí sa všetko. Binárny operátor && vykonáva logickú operáciu *and*, binárny operátor || vykonáva logickú operáciu *or* a unárny operátor ! vykonáva operáciu *not*.

Operátor [] je binárny a slúži k prístupu k prvkom poľa. Môže byť zavolaný na poli a argument musí byť **long** alebo **int**. V prípade zavolania s argumentom typu long sa automaticky pretypuje na **int**. Pri tejto zmene môže dôjsť k strate dát. Binárny operátor . je možné zavolať na štruktúre a pristupovať pomocou

neho k jednotlivým položkám štruktúry.

Poradie operátorov podľa priority:

1. operátor ! a unárne $+, -$
2. operátory $*, /, \%$
3. binárne operátory $+, -$
4. operátory $>, <, >=, <=, ==, !=$
5. operátor &&
6. operátor $||$

4.2 Spustenie programu

Vizualizátor geometrických algoritmov je napísaný v *Jave*. To znamená že je prenositeľny a funguje na operačných systémoch *Windows* a *Linux*. Pre správnu funkčnosť programu je potrebné mať nainštalovanú Java verziu 6 alebo novšiu. Prekladač spustíme pomocou príkazu:

```
java -jar compiler.jar file
```

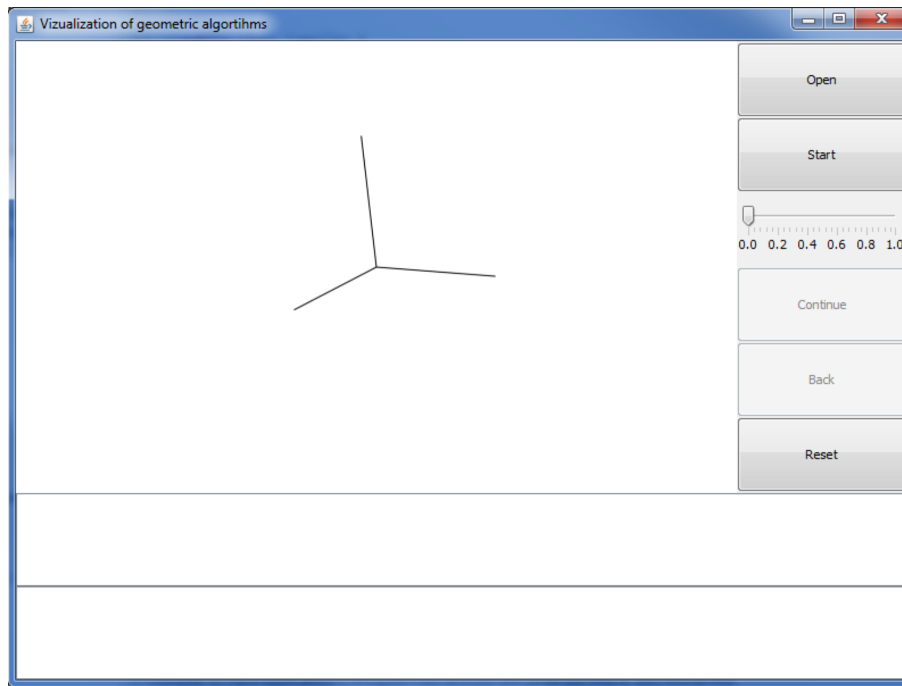
Ak bol program preložený bez chýb, tak výstupom je súbor s rovnakým názvom a príponou *.ser* (*Java serialization api*). Interpreter môžeme spustiť buď pomocou príkazu

```
java -jar interpreter.jar file.ser
```

alebo dvojklikom na ikonu súboru *interpreter.jar*. Pri spúšťaní interpretera z príkazového riadku nie je nutné zadať súbor so zdrojovým kódom.

4.3 Interpreter

V prípade spustenia interpretera z príkazového riadku bez názvu súboru alebo pri spustení dvojklikom je nutné načítať zdrojový kód pomocou tlačidla *Open*.



Obr. 4.1: Interpreter pred spustením algoritmu

Interpreter je zobrazený na obrázku 4.1. Program sa spúšťa tlačidlom *Start*. Rýchlosť vykonávania algoritmu je možné meniť pomocou posuvnej lišty. Hodnoty na tejto lište označujú dobu v sekundách, na ktorú interpreter pri zavolaní funkcie *step* zaspí. Ak je ukazateľ posunutý úplne vpravo, program sa zastaví a znova môže byť spustený pomocou tlačidla *Continue*. Zastaviť bežiaci program je možné pomocou tlačidla *Pause*, ktoré sa objaví na mieste tlačidla *Continue* pri behu programu. Posúvať algoritmus späť je možné tlačidlom *Back*, avšak program musí byť zastavený. Interpreter ukladá len posledných 10 krokov, z dôvodu šetrenia pamäťou. Funkcia kroku späť sa dá vypnúť zavolaním funkcie *disable_step_back*. Stlačením tlačidla *Reset* sa zastaví program, reštartuje sa interpreter, vymaže sa obsah konzol a vymažú sa zobrazené objekty.

Priebeh algoritmu sa zobrazuje na vizualizačnej ploche. Užívateľ môže pomocou kolieska na myši približovať a oddiaľovať, pomocou ľavého tlačidla rotovať a pomocou pravého tlačidla posúvať zobrazené objekty.

Výstupná konzola sa nachádza v spodnej časti okna. Text v konzole sa nedá editovať a slúži len na výstup programu. Zápis na výstupnú konzolu je možný pomocou funkcií *write∗*.

Vstupná konzola sa nachádza nad výstupnou konzolou. Editovať sa dá vždy

len posledný napísaný riadok. Program môže čítať tento vstup pomocou funkcií $read*$. Všetky tieto funkcie sú popísané v prílohe na CD.

5. Implementácia

5.1 Generátor zásobníkového automatu

V balíku *stack_automat_generator* sa nachádza generátor zásobníkového automatu. V súbore obsahujúcom gramatiku sa môžu nachádzať buď prázdne riadky, riadky obsahujúce definície terminálov alebo riadky obsahujúce pravidlá. Príklad pravidla:

```
<endif> ::= else <block> @else_block_begin#1 @process_if_else#2
```

Neterminály sú uzavreté v špicatých zátvorkách a funkcie, ktoré sa volajú počas sémantickej analýzy, majú formát `@názov#číslo`. Číslo za mriežkou určuje, v ktorom momente sa má daná funkcia zavolať pri prechode derivačným stromom do hĺbky. Nula znamená pri vstupe do uzlu a číslo n znamená po návrate z n-tého syna. Vo všeobecnosti pravidlo obsahuje neterminál, znak `::=`, terminály a neterminály a funkcie. Štartovací neternimál je `<program>` a prázdne slovo je označné `<lambda>`.

Príklad definície tokenu:

```
new 13
```

Príklad definície združeného tokenu:

```
REL_OPERATOR 14 < > <= >= != ==
```

Tokeny sú zadefinované identifikátorom, ktorý sa použije pri definícii pravidla a kladný číslom, ktoré identifikuje token z lexikálnej analýzy. Združený token obsahuje naviac zoznam tokenov, ktoré môže obsahovať. Tento zoznam sa použije pri syntaktickej chybe prekladača. Číslo tokenu sa neodporúča používať veľmi vysoké lebo výsledkom generátora je tabuľka indexovaná číslom terminálu a vygenerovaným číslom neterminálu.

Generátor prechádza zdrojový kód gramatiky dvakrát. Pri prvom prechode počíta počet pravidiel gramatiky a kontroluje správnosť formátu súboru. Pri druhom čítaní nahráva pravidlá do pamäte pomocou triedy *Rule*. Následne vykonáva

kontrolu správnosti gramatiky. Kontroluje či každý použitý token bol zadefinovaný a či je pre každý použitý neterminál zadefinované aspon jedno pravidlo. Aby mohol ďalej pokračovať vo výpočte, skontroluje, či každé pravidlo gramatiky generuje terminálne slovo. Na túto kontrolu je použitý nasledujúci algoritmus:

1. Do pomocného poľa vložíme neterminály, ktoré obsahujú pravidlo $A \Rightarrow \beta$, kde β je terminálne slovo.

2. Postupne prechádzame pravidlá a kontrolujeme, či sú zložené len z terminálov a neterminálov v pomocnom poli. Ak áno, vložíme daný neterminál do poľa.

3. Iterujeme dovtedy, pokým pomocné pole narastá. Ak na konci algoritmu obsahuje pole všetky neterminály, tak gramatika prešla kontrolou.

Ďalšou kontrolou je zisťovanie, či gramatika neobsahuje ľavú rekurziu. Pre tento účel si vytvoríme pomocný graf. Vrcholy sú neterminály a orientovaná hrana vedie z neterminálu A do neterminálu B, pokiaľ existuje pravidlo $A \Rightarrow \alpha B \beta$, kde α je buď λ alebo postupnosť neterminálov, ktoré generujú prázdne slovo. Ak tento graf obsahuje kružnicu tak je gramatika ľavo rekurzívna. Algoritmus na zisťovanie kružnice v grafe funguje následovne:

1. V každom kroku odstránime vrcholy, z ktorých nevedie žiadna hrana.

2. Ak na konci ostane prázdny graf, tak gramatika neobsahuje ľavú rekurziu.

Generátor ďalej vypočítava množinu $FIRST$ (viď def. 2) simulovaním ľavej derivácie. Pre každé pravidlo $A \Rightarrow B\alpha$ sa nahradzuje prvý neterminál dovtedy, kým na začiatku derivácie nie je terminál a. V tomto momente sa vloží pravidlo $A \Rightarrow B\alpha$ do tabuľky na pozícu $M(A, a)$ do zásobníkového stroja. Výpočet prebieha pomocou štruktúry $FIFO$ a aplikuje všetky ľavé derivácie okrem skracujúcich pravidiel $A \Rightarrow \lambda$.

Množinou $FOLLOW$ (viď def. 3) stačí vytvárať len pre skracujúce pravidlá. Výpočet prebieha v rekurzívnej funkcii *computeFollow* nasledovným spôsobom:

1. Pre každé pravidlo $A \Rightarrow \lambda$ nájdeme všetky pravidlá splňajúce $B \Rightarrow \alpha A \beta$.

2. Ak $\beta = \lambda$, tak vložíme inštrukciu $A \Rightarrow \lambda$ do tabuľky $M(A, x)$ pre $x \in FOLLOW(B)$. $FOLLOW(B)$ vypočítame rekurzívne.

3. V opačnom prípade vložíme inštrukciu $A \Rightarrow \lambda$ do tabuľky $M(A, x)$ pre $x \in FIRST(\beta)$. Slovo β si rozpíšeme ako postupnosť $C_1, C_2 \dots C_k$, kde C_i je terminál alebo neterminál. Pokial C_i generuje λ vložíme inštrukciu $A \Rightarrow \lambda$ do tabuľky $M(A, x)$ pre $x \in FIRST(C_i)$.

Zo súboru *StackAutomatSource*, ktorý obsahuje kód zásobníkového automatu, generujeme kód spolu s vytvorenými štruktúrami do súboru *StackAutomat.java* medzi riadky označené komentármi `/*init*/` a `/*/*init*/`.

Príklad gramatiky:

```
number 42
+ 23
* 20
( 0
) 1
end_of_file 45
<program> ::= <e> end_of_file
<e> ::= <term> <e2>
<e2> ::= + <term> <e2>
<e2> ::= <lambda>
<term> ::= <factor> <term2>
<term2> ::= * <factor> <term2>
<term2> ::= <lambda>
<factor> ::= number
<factor> ::= ( <e> )
```

Keďže je gramatika v správnom tvare, kontroly nenájdu chyby a začnú sa generovať inštukcie. Ako príklad si môžeme popísať vytváranie inštrukcií z množiny *FIRST* pre neterminál `<e>`. Pri simulovaní ľavej derivácie sa do rady najskôr vloží pravidlo `<e>::=<term><e2>`. Pretože `<term>` je neterminál, použijú sa všetky prepisovaci pravidlá, ktoré majú na ľavej strane `<term>`. V tejto gramatike je

len jedno a do radu sa vloží derivácia <e>::=<factor><term2><e2>. V ďalšom kroku sa prepíše <factor> na number a (<e>). V rade budú 2 derivácie:

```
<e> ::= number <term2> <e2>
<e> ::= ( <e> ) <term2> <e2>
```

Obe derivácie obsahujú na prvom mieste terminál, a tak do zásobníkového stroja vložíme na pozíciu $M[e, number]$ a $M[e, (]$ pravidlo <e>::=<term><e2>.

Ako ukážku vypočtu množiny $FOLLOW$ si uvedieme výpočet inštrukcií pre pravidlo <e2>::=<lambda>. Najskôr nájdeme pravidlá, ktoré na pravej strane obsahujú <e2>. Pretože existujú taktéto pravidlá dve a obe majú <e2> na konci, rekurzíve vypočítame $FOLLOW$ pre neterminál <e>. Ak by sme chceli rekurzíve vypočítať aj pre <e2> znova, zacyklili by sme sa a preto túto vetvu výpočtu odrežeme. Pri výpočte $FOLLOW$ pre $<e>$ opäť nájdeme pravidlá, kde sa <e> nachádza na pravej strane a pozrieme sa na symbol za ním. Keďže v oboch prípadoch sú to terminály, tak vlozíme pravidlo <e2>::=<lambda> do tabuľky na miesta $M[e2, (]$ a $M[e2, end_of_file]$.

5.2 Prekladač

Hlavnou časťou prekladača je trieda *Compiler*, ktorá číta zdojový súbor, prekladá program a generuje kód. Prekladač pozostáva z:

1. lexikálneho analyzátora, ktorý je reprezentovaný triedou *Ylex.*

2. zásobníkového automatu, ktorý je reprezentovaný triedou *StackAutomat.*

3. sémantického analyzátora, ktorý je reprezentovaný triedou *SemanticAnalyser.*

Lexikálny analyzátor je vygenerovaný pomocou nástroja **Jflex** a na reprezentovanie tokenu používa triedu *Ytoken.*

Zásobníkový automat, vygenerovaný pomocou generátora zásobníkového automatu, funguje ako klasický $LL(1)$ automat a jeho funkčnosť je popísaná v podkapitole 1.2.3. Výstupom je derivačný strom, ktorý pozostáva z uzlov triedy *Vertex.* Tento strom má v každom liste ukazateľ na token. V nelistových uzloch sa

nachádza pole ukazateľov na synov. Pre účely ladenia je možné vypísať derivačný strom do *XML*. Slúži na to metóda *printDerivationTree*.

Derivačný strom je ďalej spracúvaný semantickým analýzatorom. Do hĺbky prechádza strom a volá funckie definované pri každom pravidle. Každá funkcia berie ako argument uzol stromu na ktorom bola zavolaná a vracia **int**. Zároveň sa generuje zásobníkový kód, ktorý je reprezentovaný triedou *InstructionBlock*. V balíku *Instruction* sa nachádzajú všetky inštrukcie. Inštrukcie, ktoré nemajú argument, sú zdedené z *Instruction* a inštrukcie s argumentom *InstructionWithArgument*.

Trieda *Atributes* slúži na ukladanie pomocných dát a trieda *Tables* reprezentuje tabuľky prekladača. *Tables* nám poskytuje metódy na ukladanie deklarovných premenných, funkcií a typov. Funkcie, premenné, parametre funkcií, typy, polia a štruktúry sú reprezentovné triedami *Function*, *Variable*, *Parameter*, *Type*, *Field* a *Struct*. Všetky tieto triedy dedia od abstraktnej triedy *Symbol*.

Výstupom z prekladača je pole inštrukcií, ktoré sa uloží do súboru pomocou *Java serialization api*. Pre účel ladenia programu sa generuje aj *XML* výstup obsahujúci inštrukcie.

5.3 Interpreter

Hlavnou triedou interpretera je *InterperterMain*. Táto trieda obaľuje intepreter a zabezpečuje načítanie knižnice *Java3D*, v prípade, že nie je nainštalovaná. Triedy *InterperterMain* a *ExtensionsClassLoader* sú prevzaté od Emmanuela Puybareta z projektu *Sweet Home 3D* [11].

Trieda *Interpreter* je vlákno, ktoré vykonáva inštrukcie preloženého programu a obsahuje implementáciu vnorených funkcií jazyka. Na zastavenie programu slúži premenná *started*. Interpreter pred každou inštrukciou skontroluje, či má zastaviť. Ak áno vlákno sa uspí na zámku premennej *lock*.

Krokovanie algoritmu je implementované pomocou *Java serialization api*. Celý stav zásoniku sa uloží do poľa bytov, ktoré sa zapíše do fronty. Pri zobrazení alebo mazaní geometrického tvaru sa ukladajú tieto operácie do fronty pomocou interface *GuiOperation*. Pri zavolaní kroku späť sa nahrá naposledy uložený stav

zásobníku a vykonajú sa reverzné operácie s geometrickými útvarmi. Oddelovač jednotlivých krokov v rade operácí s útvarmi je **null**.

Grafické rozhranie je implementované triedou *Gui*, ktorá zabezpečuje pridávanie a odoberanie geometrických útvarov do 3D vizualizovanej plochy.

Záver

Cieľom práce bol návrh jazyka, implementácia prekladača a interpretera určených na vizualizáciu geometrických algoritmov. Tento cieľ sa podarilo splniť, keďže dôraz bol kladený na jednoduchosť navrhovaného jazyka a kvalitu vizualizácie geometrických algoritmov.

Z časových dôvodov neboli implementované niektoré riadiace štruktúry, ktoré je možné nájsť v iných jazykoch ako napríklad: skoky *goto*, *break*, *continue* alebo operátory ++, += atď. Skoky ako *goto* by ani do štruktúrovaného programovania nemali patriť. Ďalej by bolo možné navrhovaný jazyk rozšíriť o preťažovanie funkcií. Všetky tieto štuktúry sa samozrejme dajú nahradiť pomocou implemetovaných konštrukcií jazyka.

Tento projekt by bolo možné rozšíriť o vyššie zmienené neimplementované štruktúry. Ďalej je možné rozšíriť navrhovaný jazyk o štruktúry určené na vizualizovanie iných tried algoritmov. Keďže projekt obsahuje aj generátor syntaktického LL(1) automatu, je možné jednoducho upravovať gramatiku a pridať sémantické akcie.

Táto práca má reálne využitie ako pomôcka pri výuke geometrických algoritmov, resp. po rozšírení projektu aj iných skupín algoritmov.

Zoznam použitej literatúry

[1] Aho A., Lam M., Sethi R., Ullman J.: *Compilers Principles, Techniques and Tools*, Addison Wesley, 2007, ISBN 0-321-49169-6

[2] Chytil M.: *Teorie automatů a formálních jazyků*, Státní pedagogické nakladatelství Praha 1978

[3] Češka M., Hruška T., Beneš M.: *Překladače* http://www.fit.vutbr.cz/~meduna/fjp/skripta.pdf

[4] Devadoss S., O'Rourke J.: *Discrete and Computational geometry*, Princeton University Press, 2011, ISBN 978-0-691-14553-2

[5] Slovák J: *Geometrické algoritmy I* http://www.math.muni.cz/~slovak/Vyuka/slidy.pdf

[6] Cormen T., Leiserson C., Rivest R., Stein C.: *Introduction to Algorithms*, MIT Press, 2001, ISBN 0-262-03293-7

[7] Welzl E.: *Smallest enclosing disks (balls and ellipsoids)*, Lecture Notes in Computer Science, 1991, Volume 555/1991, 359-370

[8] Kučera L.: *Algovize* http://www.algovision.org/

[9] Skutka O.: *Vizualizace geometrických algoritmů pro potřeby výuky*, Diplomová práca, Masarikova univerzita v Brně, 2006, http://vigeal.webzdarma.cz/

[10] Grafická knižnica Java 3D http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html

[11] Puybaret E.: *Sweet Home 3D* http://www.sweethome3d.com/

[12] Java http://www.java.com/

[13] Ant http://ant.apache.org/

Popis prílohy na CD

V adresári *zdrojove kody* sa nachádzajú zdrojové kódy. Preložiť je ich možné pomocou nástroja *Ant* [13] verzie aspoň 1.8.2 a je zároveň je potrebné mať nainštalovanú knižnicu *Java3D* [10] verziu aspoň 1.5.2. Adresár *spustitelne subory* obsahuje spustiteľné súbory prekladača a interpretera. Na ich spustenie potrebujeme nainštalovať *Javu* [12] verziu 6 alebo vyššiu. Spustiteľné súbory môžeme spustiť aj bez nainštalovanej knižnice *Java 3D*. *Javadoc* celého projektu sa nachádza v adresári *dokumentacia* a tento adresár obsahuje aj popis vstavaných funkcií jazyka. Elektronickú verziu textu tejto práce môžeme nájsť v priečinku *praca*. Vizualizované algoritmy sú umiestnené v adresári *algoritmy*.