VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

JAZYK PRO PROCEDURÁLNÍ GENEROVÁNÍ

LANGUAGE FOR PROCEDURAL GENERATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ROMAN DOBIÁŠ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20847/2017/xdobia11

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Dobiáš Roman**
Obor: Informační technologie
Téma: **Jazyk pro procedurální generování**
Language for Procedural Generation
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte L-systémy a šumy pro procedurální generování a knihovnu OpenGL.
2. Navrhněte vlastní jazyk umožňující procedurální generování.
3. Implementujte multiplatformní knihovnu pro procedurální generování. Knihovna bude pracovat s textovým konfiguračním souborem.
4. Vytvořte jednoduchou vizualizační aplikaci, která využije OpenGL a implementovanou knihovnu. Vytvořte sadu příkladů, na kterých budete demonstrovat schopnosti navržené knihovny.
5. Vytvořte video s demonstrací implementované knihovny.

Literatura:

- dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a kostra aplikace.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Milet Tomáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Práca sa zaoberá návrhom a implementáciou knižnice s jazykom pre procedurálne generovanie, vychádzajúcim z L-systémov. Zmyslom práce je vytvoriť prakticky použiteľnú a jednoducho integrovateľú knižnicu, ktorá sa bude dať využiť v celej rade aplikácií, obzvlášť v 3D vykresľovacích enginoch alebo editoroch. Práca sa zaoberá nutnou teóriou procedurálneho generovania a L-systémov, téoriou formálnych jazykov, a návrhom a implementáciou daného systému. Výsledkom práce sú ukážkové projekty využívajúce knižnicu a početné príklady dosiahnuté pomocou generovania.

Abstract

This thesis deals with designing and implementing a library with language devoted to procedural generation extending L-systems. Emphasis is put on practical usage of the library which is aimed to be used by a wide spectrum of real-world applications, especially by 3D rendering engines and editors. The thesis covers theory of procedural generation, L-systems, theory of compilers, and design and implementation of the library. In conclusion, case study projects are introduced which embed the library and numerous examples are given.

Kľúčové slová

L systémy, Procedurálne generovanie, Korytnačia grafika, Formálne jazyky, Prekladače, Bison, Flex, C++, OpenGL, SVG

Keywords

L-systems, Procedural generation, Turtle graphics, Formal languages, Compilers, Bison, Flex, C++, OpenGL, SVG

Citácia

DOBIÁŠ, Roman. *Jazyk pro procedurální generování.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet

Jazyk pro procedurální generování

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Tomáša Mileta. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Roman Dobiáš
14. mája 2018

Poďakovanie

Táto práca by nevznikla bez bezhraničného nadšenia vedúceho práce a jeho časovo bohatým konzultáciam. Ďalej taktiež ďakujem tvorcom všetkého slobodného softwaru, ktorý technicky umožnil vznik práce, hlavne LaTeX, InkScape, GIMP, Vim, Fedora a mnoho ďalších.
Počas písania tejto práce neboli poškodené žiadne shadery ani grafické karty. Práca nebola napísaná na Slovensku. Autor v osudné dni septembra 2017 navštívil osobne *5th Avenue*, nestretol sa ale s človekom, ktorého krsné meno je George.

> The darkest places in hell are reserved for those who maintain their neutrality in times of moral crisis.
>
> Dante Alighieri

Obsah

Kapitola 1

Úvod

V súčasnej dobe sa v ľudskej spoločnosti objavuje idea obecnej automatizácie, ktorá by nahradila monotónnu ľudskú prácu a umožnila ľuďom koncetrovať sa výhradne na kreatívne, vysoko abstraktné zmýšlanie. V oblasti počítačovej grafiky ale podobné techniky už dávno existujú pod názvom *procedurálne generovanie* a predstavujú proces syntézy dát, ktorý je popísaný algoritmicky (procedúrami). V praxi je možné tento prístup využiť napríklad k tvorbe komplexných a detailných dát len na základe malého objemu vstupných dát. V počiatkoch prvých video hier našli napríklad tieto techniky svoje uplatnenie v generovaní obsahu, textúr a iný herných prvkov z dôvodu pamäťových obmedzení. Namiesto uloženia predvytvorených herných levelov bol herný svet vygenerovaný vždy pred spustením hry, priamo v pamäti. Tento postup zároveň umožnil predĺženie hernej doby z dôvodu straty monotónnosti obsahu hry. V súčasnej dobe sa ich využitie spája s *generovaním rozsiahlych trojrozmerných herných svetov*, či scén vo filmoch z dôvodu zníženia ceny, ako aj náročnosti tvorby vizuálne bohatých scén (napr. prostredie sveta vo filme Avatar).

Pre využitie procedurálneho generovania je nutné zostrojiť systém, ktorý by umožnil implementovať tieto techniky. K tomuto účelu je možné napríklad použiť obecné programovacie jazyky. Programovanie v nich avšak vyžaduje riešenie správy pamäte, práce s ukazateľmi, inštanciaciu štruktúr či tried a podobné technické problémy, čo predstavuje zvýšenú záťaž pre programátora. Preto má zmysel uvažovať o vytvorení nového programovacieho jazyka, ktorý umožní tvoriť programy bližšie abstrakcii procedurálneho generovania a tým zlepší komfort pri programovaní.

V dnešnej dobe sú dostupné komerčné produkty, ktoré umožňujú generovať objekty pomocou preddefinovaných postupov na základe vytvorených gramatík. Obecne nie sú ale dostupné systémy, ktoré by bolo možné upraviť a použiť pre generovanie ľubovoľných entít, napr. zvukových stôp alebo jednoducho integrovať do vlastných aplikácií.

Táto práca sa teda zaoberá teóriou procedurálného generovania (kapitola 2) a vytvorením nového programovacieho jazyka v prenositeľnej knižnici. Cieľom je rovnako zabezpečiť dostatočnú expresívnosť jazyka pre jeho využitie k obecnému generovaniu. V kapitole 3 sú preto predstavené zaužívané formálne štruktúry a prístupy, ktoré sa obecne používajú k tvorbe programovacích jazykov. V kapitole 4 je popísaný návrh jazyka, teda obecné konštrukcie, ktoré zlepšujú expresívnosť jazyka ako aj komponenty, ktoré zabezpečujú jeho sémantiku v skutočnom počítačovom systéme. Ako presne vyzerá konkrétny systém z pohľadu kódu v jazyku `C++` a jeho nasadenia, je potom vysvetlené v kapitole 5. V záverečnej kapitole 6 sú popísané prípadové štúdie, využívajúce novozniknutú knižnicu.

Kapitola 2

Procedurálne generovanie

V tejto kapitole je popísané procedurálne generovanie, jeho zmysel a využitie, a niektoré obvyklé metódy, pomocou ktorých sa v praxi realizuje. Tieto techniky nám poslúžia ako východiskový bod pre neskorší návrh jazyka.

2.1 Definícia a využitie

Podľa knihy [18, str. 1] je možno formálne definovať procedurálne generovanie ako spôsob *vytvárania obsahu na základe algoritmov bez, alebo s obmedzeným ľudským zásahom.*

V praxi sa jedná o automatizované vytvorenie obsahu, ktorý vzniká z predom definovaných dát, transformovaných pomocou postupov (algoritmov), ktoré sú riadené parametrami. Parametre generovania sú obvykle závislé od entít, ktoré generujeme a od úrovne abstrakcie v generovaní. V prípade, že naším cieľom je napríklad generovať mestá, potom parametrami takéhoto generovania môžu byť hustota zástavby, výškový limit mesta, množstvo zelene a podobne.

Pod slovom obsah je obecne možné dosadiť akýkoľvek prejav ľudského intelektu, od textu, obrazu až po hudbu, ktorý by pri klasickej tvorbe bol vytvorený manuálne človekom.

Výhody [8, str. 268-269] tohoto prístupu teda spočívajú v:

- možnosti získania rozsiahlych dát na základe malej vstupnej množiny (angl. *data-base amplification*)
- získaní výsledkov, ktoré sú si podobné, zároveň vyzerajú jedinečne
- v jednoduchej tvorbe výstupov, ktoré požadujú pravideľnú alebo opakujúcu sa štruktúru
- v ušetrení nákladov na tvorbu obsahu

Svoje uplatnenie nachádza napríklad vo filmoch. Moderné filmy sú často zasadené v fiktívnom svete, ktorý je dielom grafických umelcov. Generovanie uľahčuje prácu umelcov pri vytváraní modelov, ktoré sú si podobné, ako sú napríklad modely vegetácie. K tomu účelu je využívaný napríklad komerčný systém *SpeedTree* [1] , ktorého výstup je znázornený na obrázku 2.1. Podobne je generovanie využitie pri rôznych filmárskych efektoch ako sú explózie, časticové efekty v podobe ohňa[5] alebo hmly, ktoré by bolo zložité animovať manuálne.

[1] http://speedtree.com - SpeedTree Vegetation Modeling

Obr. 2.1: Komerčný systém SpeedTree. Na obrázkoch sú zobrazené stromy, ktorých geometria bola vygenerovaná týmto systémom. Prevzaté z webu [1].

Táto technika sa taktiež uplatňuje vo video hrách. V počiatkoch vzniku video hier boli počítačové systémy limitované veľkosťou trvalej pamäte, čo motivovalo tvorcov hier, akou je napríklad hra Elite, ku vytvoreniu herného levelu v čase spustenia [18, str. 4]. V súčasnosti je pomocou procedurálneho generovania možné generovať celé mestá [15].

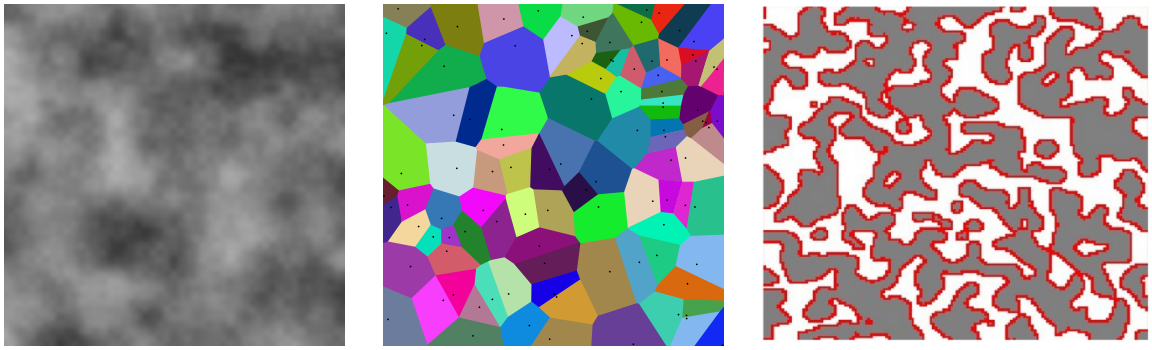
Samotné generovanie môže prebiehať úplne nezávisle, alebo môže byť ovplyvnené vstupom uživateľa, prípadne obmedzeniami prostredia. Napríklad metóda [16] popisuje generovanie stromov, ktorých rast je adaptívny a je ovplyvnený *tropizmami* – entitami, ktoré spôsobujú priťahnutie alebo odtiahnutie konárov. Takými entitami môžu byť prekážky v priestore, prípadne svetlo alebo gravitácia.

Obr. 2.2: Príklad generovaných stromov, ktorých rast je adaptovaný na prekážky v okolí a množstvo osvetlenia v priestore. Obrázok prezvatý z práce [16].

2.2 Metódy procedurálneho generovania

V dnešnej dobe sa v oblasti generovania ustálili techniky, ktoré si následne rozoberieme. Obecne ich môžeme rozdeliť do dvoch kategórií:

- **bodovo-orientované techniky**
  Tieto techniky pracujú nad jednotlivými bodmi v N-rozmernej matici (napr. 2D obraze). Patria sem napríklad šumy.
- **štruktúrne orientované techniky**
  Metódy, ktoré pracujú s abstraktnými štruktúrami (napr. so symbolmi v gramatikach).



Obr. 2.3: Zľava Pelinov šum, pripomínajúci obklady alebo dym. Obrázok prebratý z webu [2]. V strede Voronoi diagram, oddelujúci priestor do uzavretých buniek. Obrázok prevziaty z webu [3]. Sprava celuárny automat, predstavujúci pôdorys 2D herného levelu. Obrázok prevzatý z článku [12].

2.2.1 Šumy

Obecne je šum N-rozmerný signál, ktorého hodnoty podliehajú istej náhodnosti. V počítačových systémoch nie je možné získať skutočné stochastické javy z dôvodu deterministického výpočtu. Typickým príkladom šumu sú preto pseudonáhodné generátory čísiel. Obecné náhodné veličiny sa ale v počítačovej grafike nepoužívajú z dôvodu nespojitosti ich hodnôt. Sú ale potrebné pre syntézu *spojitých šumov*, kde slúžia k získaniu náhodnych počiatočných dát, napríklad v metóde *Perlinov šum* [9].

Spojité šumy majú v počítačovej grafike uplatnenie hlavne v oblasti vytvárania textúr. Pre každý bod takejto textúry je vyhodnotená hodnota spojitého šumu pre dané súradnice. Bežne sa používa napríklad *Perlinov šum*, ktorého dvojrozmerná vizualizácia, zobrazená na obrázku 2.3, pripomína oblaky či dym. V praxi sa často šumy kombinujú pomocou jednoduchých operácií.

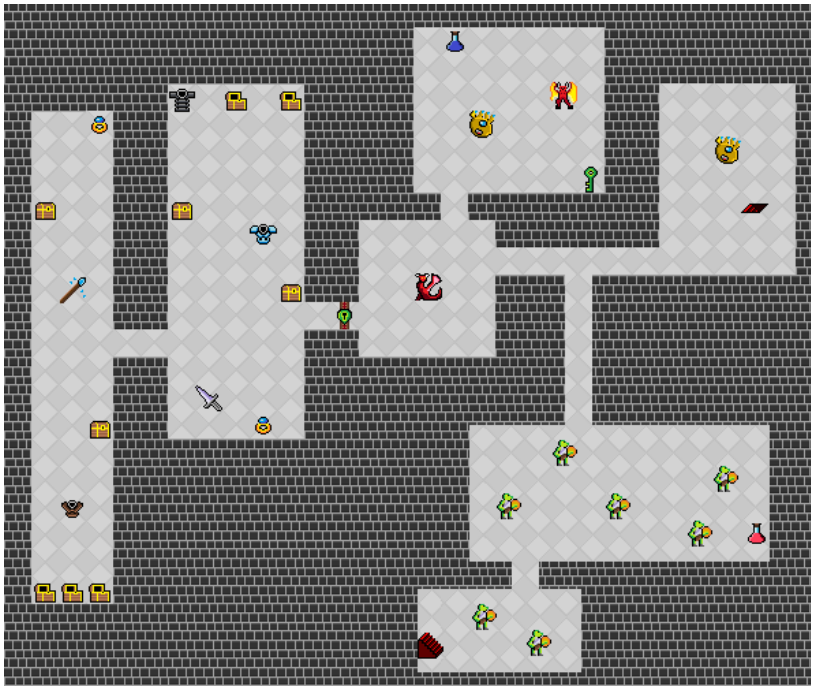
Okrem textúr je možné využiť šumy k tvorbe 3D terénu. V takom prípade sa hodnoty šumu interpretujú ako výšková mapa. Výška jednotlivých bodov terénu následne zatrieďuje danú plochu do rôznych kategórií. Napríklad pri generovaní prírodného terénu podobného našej zemi je možné interpretovať nízke pozície ako vodné plochy, teda moria, prípadne jazerá, kdežto vyššie pozície môžu mať textúru trávy, prípadne kameňov. Najvyššie pozície je možno interpretovať ako vrcholky hôr a nastaviť im textúru snehu.

2.2.2 Voronoi diagram

Voronoi diagram je obraz, ktorý je rozdelený do N ohraničených podoblastí, definovaných podľa rozdelujúcich bodov. Oblasti sa delia podľa príslušnosti pixelov ku jednotlivým rozdelujúcim bodom [2]. Bod oblasti patrí vždy najbližšiemu bodu.

Praktickým príkladom využitia Voronoiových diagramov je napríklad generovanie pôdorysov mestských oblastí [7]. Samotný diagram vizuálne pripomína plast v úle. Jeho kolorizovaná podoba je zobrazená na obrázku 2.3.

[3] https://en.wikipedia.org/wiki/Perlin_noise
[3] https://en.wikipedia.org/wiki/Voronoi_diagram



Obr. 2.4: Príklad vygenerovaného herného levelu pomocou metódy BSP. Algoritmus BSP vytvorí množinu podpriestorov, ktoré sú následne prepojené. Vďaka algoritmu BSP sa jednotlivé podpriestory neprekrývajú. Obrázok prevzatý z práce [18, str. 38].

2.2.3 Celuárne automaty

Celuárne automaty [19] predstavujú matematický model diskrétnej simulácie, kde je systém rozdelený na diskrétne podsystémy – *bunky*, ktoré majú navzájom (v rámci okolia) vzťahy. Hodnoty jednotlivých buniek sa vyvíjajú pararelne v diskrétnych krokoch pomocou identických pravidiel. Týmto spôsobom je možné simulovať priestoré problémy, napríklad presun áut v rámci križovatky, či šírenie ohňa v ohraničenom priestore s prekážkami.

Obecne sa celuárne automaty delia do štyroch kategórií, z ktorých vizuálne najzaujímavejšia je kategória 3 – *chaotické celuárne automaty*.

Celuárne automaty je možné využiť aj pri tvorbe hier. Na obrázku 2.3 môžeme vidieť pôdorys vygenerovaného herného levelu pomocou metódy [12].

2.2.4 Metóda BSP

V procedurálnom generovaní sa vyvinuli techniky umožňujúce *delenie priestoru* (angl. *space partitioning* [18, str. 33]), ktoré rozdelia priestor do diskrétnych podoblastí. Tie je možné následne použiť ku generovaniu herného levelu.

Jednou z týchto techník je metóda *binary space partition* (BSP), ktorá iteratívne delí priestor vždy na dve nové části (buď horizontálne alebo vertikálne), čím vzniká binárny strom. Následne zpätným spájaním jednotlivých uzlov stromu a prepojením podoblastí, ktoré reprezentujú, je možné dosiahnuť prepojený graf. Príklad vygenerovaného levelu je zobrazený na obrázku 2.4.

2.3 L-systémy

Lindenmayer systémy (skrátene L-systémy) boli pôvodne predstavené v roku 1968 matematikom Lindenmayerom ako matematický formalizmus pre popis topológie rastlín. Formálne definície ako aj popis systémov vychádzajú z knihy [17], v ktorej sú tieto systémy podrobne popísané a preto predstavuje vhodný študijný zdroj pre záujemcov.

Definícia L-systémov vychádza z *prepisovacích systémov*, ktoré ale upravujú o pararelné prepisovanie symbolov, čo umožňuje popísať komplexné štruktúry pomocou jednoduchých gramatík. V počítačovej grafike sa obvykle využívajú spolu s *korytnačiou grafikou* (angl. *turtle graphics*). Tá predstavuje jednoduchý, ale mnohostranný spôsob ako vizualizovať reťazce týchto systémov. Bližšie je táto metóda popísana v podkapitole 2.3.2.

2.3.1 Definícia D0L systému

Najjednoduchší L-systém je D0L systém. D predstavuje deterministickosť a 0L bezkontextový Lindenmayerov systém.

Definícia 2.1. *Nech V je abeceda, V^* množina všetkých slov nad abecedou V, V^+ množina všetkých neprázdnych slov nad abecedou V. Potom L-systém je usporiadaná trojica $G = (V, \omega, P)$, kde V abeceda systému, $\omega \in V^+$ je počiatočný reťazec (axióm) systému a $P \subseteq V \times V^*$ je množina produkčných pravidiel. Produkcia je dvojica $(a, \chi) \in P$ sa značí ako $a \Rightarrow \chi$. Písmeno a predstavuje predchodcu, χ následníka odvodenia. Pre každé písmeno a z abecedy V existuje aspoň jedno slovo χ také, že $(a, \chi) \in P$. V opačnom prípade sa predpokladá existencia implicitného pravidla $(a, a) \in P$. Systém je deterministický práve vtedy, ak platí, že pre každé $a \in V$ existuje práve jedno $\chi \in V^*$ také, že $a \Rightarrow \chi$.*

Definícia 2.2. *Nech $\mu = a_i \dots a_m$ je slovo nad V. Potom slovo $v = \chi_1 \dots \chi_m \in V^*$ je priamo odvodené zo slova μ, zapisujeme $\mu \Rightarrow v$, práve vtedy, ak $a_i \to \chi_i$ pre všetky $i = 1, \dots, m$. Slovo v je reťazcom N-teho kroku systému G vtedy, ak existuje postupnosť slov $\mu_0, \mu_1, \dots, \mu_n$ taká, že platí $\mu_0 \Rightarrow \mu_1 \Rightarrow \dots \Rightarrow \mu_n$.*

Príklad 2.1. *Uvažujme abecedu $V = \{A, B\}$, počiatočný reťazec $\mu = A$, množinu pravidiel $P = \{A \to AB, B \to A\}$ a nech $G = (V, \mu, P)$ je D0L systém. Kroky tohoto systému sú znázornené v tabuľke 2.1.*

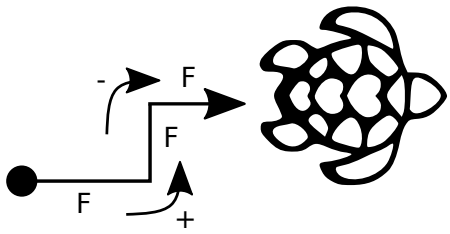
Iterácia	Reťazec
n = 0	A
n = 1	AB
n = 2	ABA
n = 3	ABAAB
n = 4	ABAABABA

Tabuľka 2.1: Prebieh derivácie v D0L systéme, definovanom v príklade 2.1. Každý riadok znázorňuje výsledok po N-tej derivácii.

2.3.2 Korytnačia grafika

Korytnačia grafika (angl. *turtle graphics*) predstavuje jednoduchý spôsob ako interpretovať reťazce L-systémov. Jej princip fungovania vychádza z pohybu pomyslenej korytnačky v piesku – pri pohybe sa korytnačka pohybuje vpred, pričom za sebou zanecháva stopu, ktorej tvar záleží na dĺžke kroku, prípadne smerovej orietácii korytnačky.

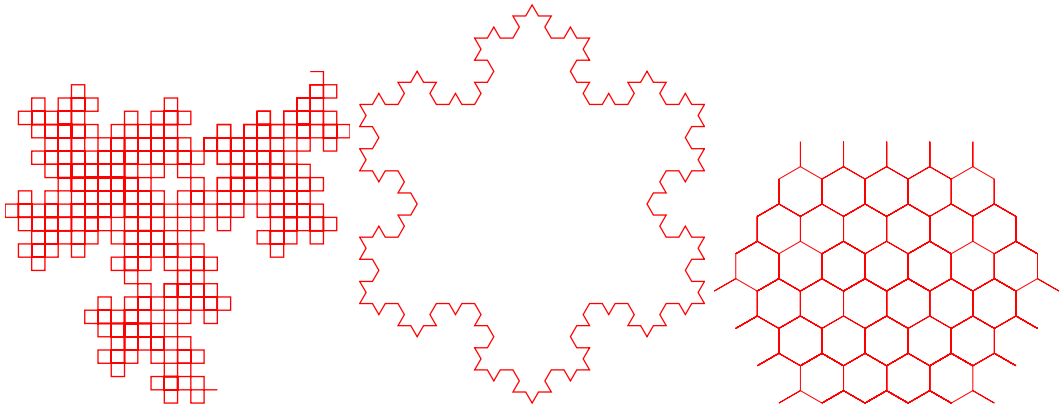
Podobnú analógiu možno zaviesť pri vizualizácii reťazca symbolov. Virtuálnu korytnačku, ktorá kreslí čiaru, je možné ovládať pomocou definovaných termínalnych symbolov z abecedy L-systému, ktoré zodpovedajú pohybu a rotácii. Vizualizácia teda pozostáva zo sekvenčného prechodu reťazca a interpretácie symbolu reťazca v konkrétnom kroku. Príklad vizualizácie je znázornený na obrázku 2.5.



Obr. 2.5: Vizualizácia reťazca F + F - F korytnačkou. Symbol F predstavuje pohyb vpred, + rotáciu o 90 stupňov. Obrázok z webu [4].

Definícia 2.3. *Nech je stavom korytnačky trojica* (x, y, α)*, kde x,y sú súradnice v rovine, a uhol* α *predstavuje smer. Nech je definovaný uhol* ω*, predstavujúci zmenu uhlu, a* δ *predstavujúca dĺžku kroku. Potom nech existujú nasledujúce pravidlá, definujúce zmenu stavu korytnačky:*
F: pohyb korytnačky z bodu (x,y) o smerový vektor $(cos(\alpha), sin(\alpha))$*, o dĺžke kroku* δ*.*
+: zmena uhlu α *o uhol* ω*.*
-: zmena uhlu α *o uhol* $-\omega$*.*

Napriek jednoduchosti je možné pomocou D0L systémov a takto definovanej korytnačej grafike vytvárať vizuálne pôsobivé obrazce. Vybrané príklady sú zobrazené na obrázku 2.6.



(a) F: F+FF+F+F+FF
$\alpha = 90°$ ω = F

(b) F: F-F++F-F
$\alpha = 60°$ ω = F++F++F

(c) F: +F+F+
$\alpha = 60°$ ω = F

Obr. 2.6: Vizuálne komplexné vzory, vytvorené pomocou D0L systému a jednoduchej korytnačej grafiky podľa definície 2.3.

[4] http://www.clipartpanda.com/clipart_images/turtlehawaiianhearts-41340657

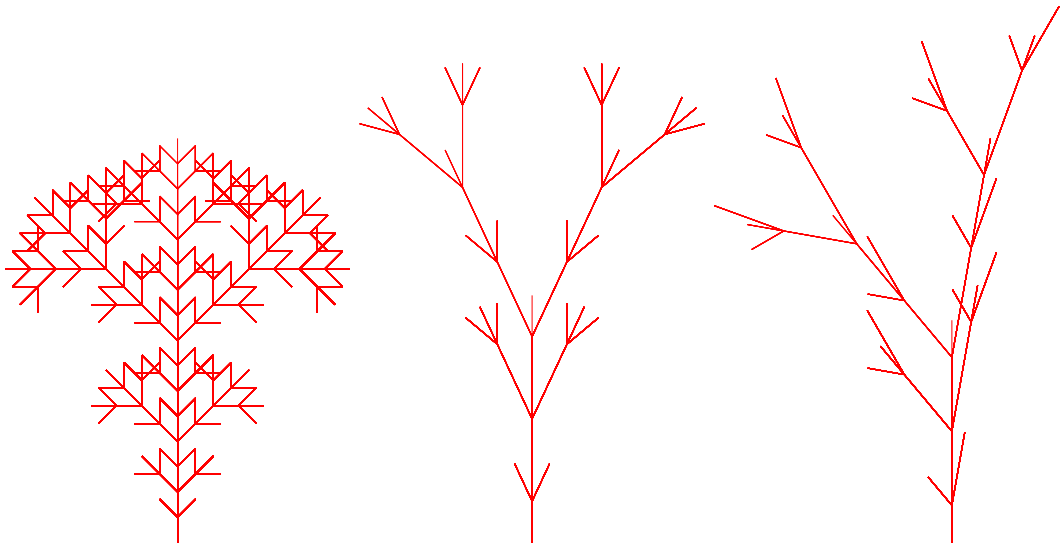
2.3.3 Zátvorkové L-systémy

Z definície 2.3 vyplýva zásadné obmedzenie vizualizácie – je možné vizualizovať len spojitú krivku. Z tohoto dôvodu existuje rozšírená aplikácia D0L systémov s názvom *zátvorkové L-systémy*, ktorá rozširuje metódu korytnačej grafiky z definície 2.3 o *zásobník*. Pomocou zásobníku je možné uložiť stav korytnačky v istom momente vizualizácie a neskôr tento stav obnoviť a napríklad pokračovať vo vizualizácii iným smerom. Pre prácu so zásobníkom je potrebné rozšíriť množinu terminálov o uloženie stavu [a načítanie naposledny uloženého stavu z vrcholu zásobníka pomocou].

Definícia 2.4. *Nech je trojica $G = (x, y, \alpha)$, kde x je súradnicou X, y súradnicou Y a uhol α predstavuje smer, stavom korytnačky, a nech je definovaný uhol ω, predstavujúci zmenu uhlu, a δ predstavujúca dĺžku kroku. Zároveň nech existuje zásobník ς, uchovávajúci stavy G. Potom nech existujú nasledujúce pravidlá, definujúce zmenu stavu korytnačky:*
F: pohyb korytnačky z bodu (x,y) o smerový vektor $(cos(\alpha), sin(\alpha))$, o dĺžke kroku δ.
+: zmena uhlu α o uhol ω.
-: zmena uhlu α o uhol $-\omega$.
[: uloženie aktuálneho stavu korytnačky G na vrchol zásobníku ς
]: vyňatie stavu G z vrcholu zásobníka σ a nastavenie aktuálneho stavu týmto stavom

Vďaka zásobníku je možné pomocou D0L systémov a metódy z definície 2.4 vytvárať členité útvary s vetvami. Vetvenie zároveň predstavuje vhodné využitie pararelného prepisovania L-systémov a umožňuje vytvárať jednoduché D0L gramatiky, ktorých vizualizácia pripomína tvar rastlín.

Prakticky je možné využiť takto definované D0L systémy ku generovaniu vektorových textov a obrázkov, ktoré je bližšie popísané v podkapitole 6.1.

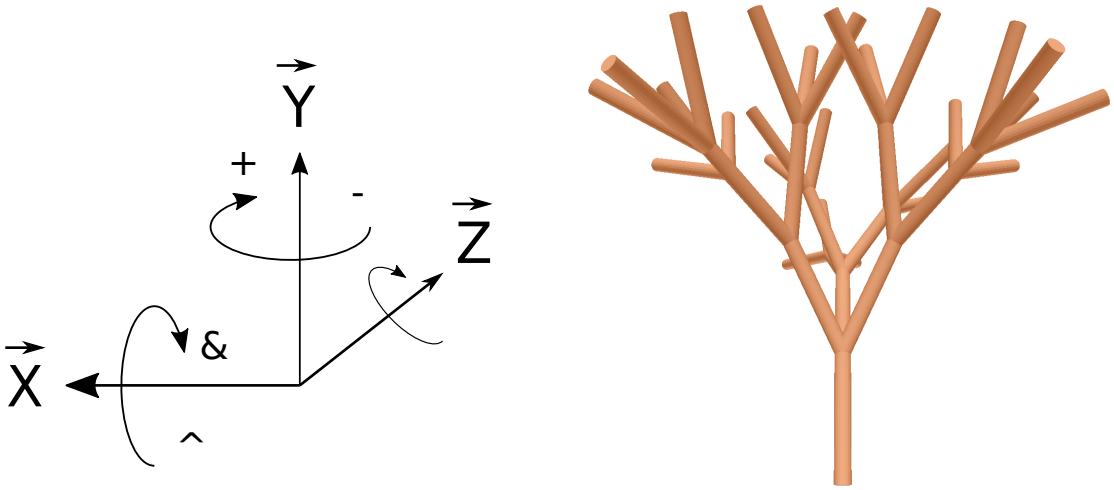


(a) F: F+FF+F+F+FF
$\alpha = 90°$ ω = F

(b) F: F-F++F-F
$\alpha = 60°$ ω = F++F++F

(c) F: +F+F+
$\alpha = 60°$ ω = F

Obr. 2.7: Príklady zátvokových D0L gramatík. Vďaka pridanému zásobníku je možné dosiahnuť vetvenie, podobné rastlinným štruktúram.



Obr. 2.8: Znázornenie rozšírenia zátvorkových systémov do 3D priestoru. Príklad takéhoto systému je na obrázku vpravo.

2.3.4 Zátvorkové L-systémy v 3D

V doposial popísaných L-systémoch nás zaujímali prioritne dvojrozmerné obrazce. Pomocou jednoduchého rozšírenia je ale možné využiť zátvorkové systémy aj pre generovanie 3D modelov. Pôvodné symboly pre rotáciu v 2D ploche je možné nahradiť rozšírenými symbolmi, umožňujúcimi rotáciu okolo obecných osí X, Y, Z o pevne definovaný uhol. Podobne je nutné rozšíriť definíciu stavu korytnačky o aktuálnu rotáciu okolo osí X a Z.

Príklad generovania 3D modelu pomocou zátvorkového L-systému je na obrázku 2.8.

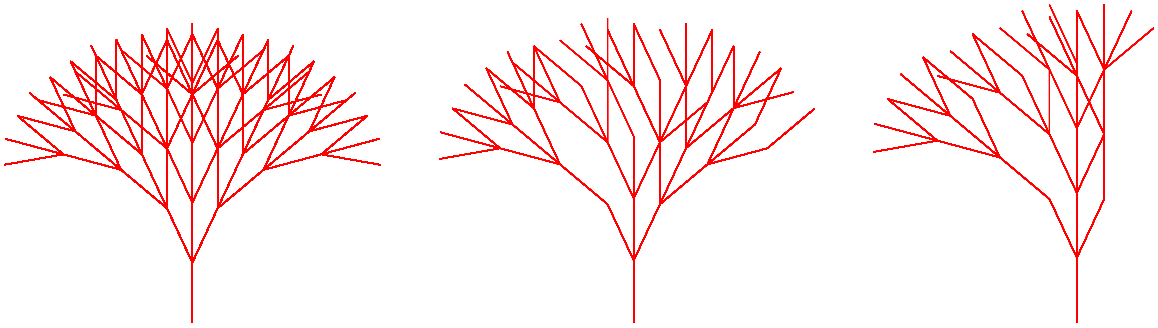
2.3.5 Stochastické L-systémy

Deterministickosť D0L systémov umožňuje získať pre rovnaký počiatočný reťazec rovnaký výsledok generovania, čo je vhodné v istých aplikáciach, kde je požadovaný predikovaný výsledok, napríklad v generovaní herného príbehu alebo scény v grafickom deme.

Aby bolo možné dosiahnuť modely podobné reálnemu svetu, je nutné zakomponovať do L-systémov náhodnosť. *Stochastické L-systémy* [10, str. 20] preto umožňujú pre každý symbol jazyka systému definovať viacero pravidiel a navyše určiť pravdepodobnosť výberu konkrétno pravidla v prepisovacom cykle.

Definícia 2.5. *Nech stochastický OL-systém je usporiadaná štvorica $G = (V, \omega, P, \Pi)$, kde V abeceda systému, počiatočný reťazec ω a množina produkčných pravidiel P, sú identické podľa definicície 2.1. Funkcia $\pi : P \to \Re$ zobrazuje množinu produkčných pravidiel na množinu nezáporných čísiel s označením pravdepodobnostné súčinitele.*

Definícia 2.6. *Nech $\hat{P}(\mu, s) \subset P$ označuje podmnožinu produkčný pravidiel množiny P, ktoré označujú slovo μ na pozícii s. Ak neexistuje žiadne také pravidlo, ktoré by označilo slovo μ na pozícii s, potom množina $\hat{P}(\mu, s)$ obsahuje pravidlo identity, ktoré skopíruje symbol na pozícii s s pravdepodobnostným faktorom 1. Derivácia $\mu \to v$ je stochastická derivácia v systéme G práve vtedy, ak pre každú pozíciu s slova μ je pravdepodobnosť aplikovaného pravidla $p_i \in \hat{P}(\mu, s)$ rovná výpočtu pomocou rovnice 2.1.*



Obr. 2.9: Príklad stochastického generovania stromu. Prvý strom je generovaný deterministickou gramatikou $A : F[+A][-A][A]$. Zvyšné dva stromy sú generované stochastickou gramatikou, obsahujúcou pravidlo navyše $A : [A]$. Pomer pravdepodobnosti oboch pravidiel je $55.5 : 44.4\%$. Stromy sa líšia odlišným počiatočným nastavením generátora (angl. *seed*).

$$prob(p_i) = \frac{\pi(p_i)}{\sum_{p_k \in \hat{P}(\mu,s)} \pi(p_k)} \tag{2.1}$$

Príklad náhodného generovania tvaru stromu je znázornený na obrázku 2.9.

2.3.6 Parametrické L-systémy

Doposial prezentované L-systémy umožňovali vyjadriť vizuálne pôsobivé 2D či 3D obrazce, ktorých štruktúra bola uniformná – jednotlivé grafické primitíva mali rovnakú dĺžku, šírku, či uhol rotácie. Napríklad v prípade generovaniu rastu stromu nie je možné postupne zkracovať dĺžku, či znižovať šírku konárov.

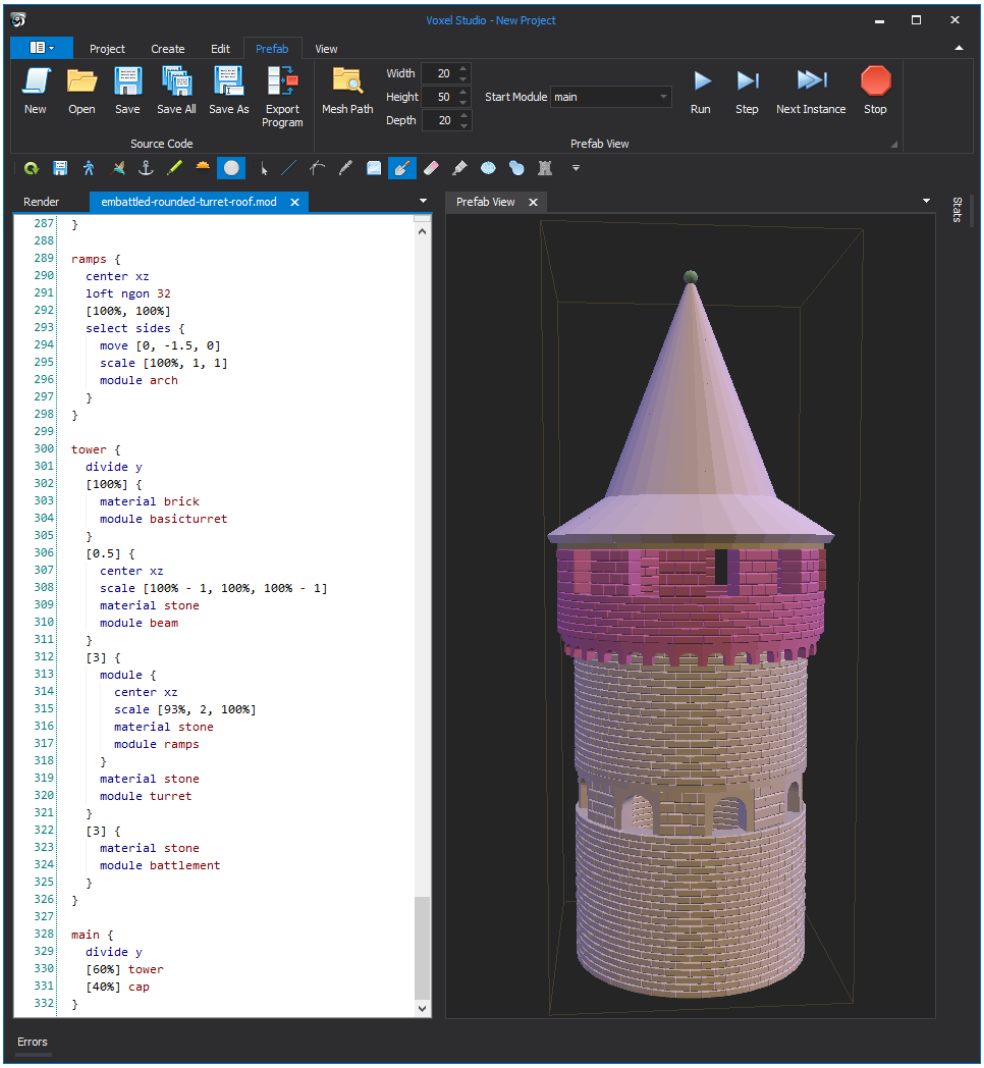
Tento problém umožňujú riešiť *parametrické L-systémy*. V ich definícii sa zo symbolu stáva obecná konečná štruktúra bez zanorenia, umožňujúca reprezentovať konečné vektory hodnôt. Zároveň sú prepisovacie pravidlá rozšírené o jednoduchú manipuláciu s hodnotami symbolov a jednoduché logické predikáty.

2.4 Existujúce riešenia

Nakoľko techniky, ktoré boli predstavené v predchádzajúcej sekcii správy, sa objavili na scéne počítačovej grafiky už osemdesiatych rokoch, vznikli medzi časom implementácie podobných jazykov, ktoré je vhodné zbežne predstaviť, prípadne definovať ich obmedzenia.

- **L-systems online** [4]
  Nástroj implementuje vlastný jazyk, ktorú umožňuje definovať bezkontextové parametrické stochastické L-systémy. Implementácia nástroja je v jazyku JavaScripte a projekt má webové rozhranie, umožňujúce zobrazovať SVG a 3D grafiku pomocou WebGL. Zdrojové kódy nástroja sú voľne dostupné. Nástroj je prezentovaný ako vizualizácia L-systémov, bez dokumentácie potrebnej ku integrácii do iných aplikácií. Nevýhodou nástroja môže byť implementácia v jazyku JavaScript, ktorá vyžaduje interpret tohoto jazyka.

- **VoxelFarm** [5]
  VoxelFarm je komerčný nástroj, integrovaný do grafického enginu Unreal Engine [6] a Unity [7]. Nástroj umožňuje definovať procedurálnu geometriu pomocou vlastného jazyka vychádzajúceho z L-systémov. Vygenerovaná geometria je voxelizovaná. Cieľom nástroja je umožňiť hráčom manipuláciu s herným svetom priamo v hre pomocou editácie voxelov. Príklad vygenerovaného modelu pomocou nástroja `VoxelFarm` je znázornený na obrázku 2.10.



Obr. 2.10: Príklad použitia nástroja VoxelFarm pre generovanie architektúry. Obrázok prevzatý z blogu [8].

[5] www.voxelfarm.com - Voxel Farm
[6] https://www.unrealengine.com Herný engine od spoločnosti Epic Games
[7] https://unity3d.com Herný engine od spoločnosti Unity Technologies
[7] www.procworld.blogspot.cz/2014/11/life-without-debugger.html - Procedural World

Kapitola 3

Formálne jazyky a prekladače

Táto kapitola veľmi stručne popisuje základné znalosti z teórie formálnych jazykov a prekladačov, potrebné k návrhu a implementácii programovacieho jazyka. Formálne definicície vychádzajú z knihy [14], ktorá predstavuje úplny zdroj informácii ku formálnym jazykom a automatom.

3.1 Obecná definícia gramatiky

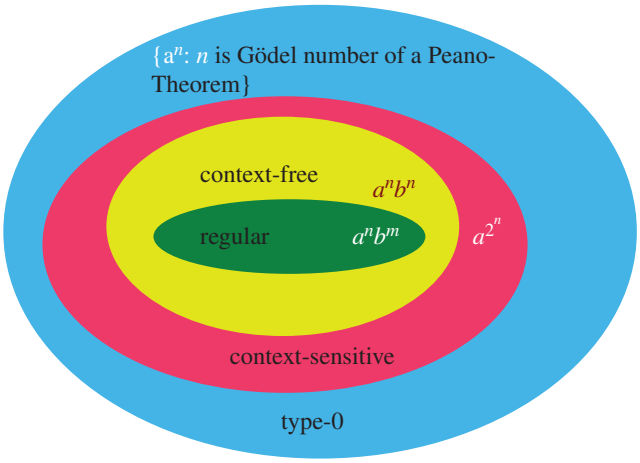
Jazyk akým je slovenčina, čestina, či C++, je obecne *množina reťazcov (slov), patriacich do jazyka.* Aby bolo možné vyjadrovať sa formálne o jazykoch a algoritmoch nad nimi, sú jazyky exaktne definované pomocou *gramatík* – systémov, ktoré obsahujú symboly a pravidlá pre ich popis.

Definícia 3.1. *Gramatika je usporiadaná štvorica* $G = (\Sigma, T, S, P)$*, kde* Σ *je konečná neprázdna množina nonterminálnych symbolov,* $T, T \cap \Sigma = \emptyset$ *je konečná neprázdna množina terminálnych znakov,* $S \in \Sigma$ *je počiatočny symbol gramatiky a* $P = (\Sigma \cup T)^+ \times (\Sigma \cup T)^*$ *je množina produkčných pravidiel. Nech reťazce* $s_1 = xyz$ *a* $s_2 = xwz$ *sú platnými reťazcami gramatiky* G *a zároveň nech* $p = y \times w$*, zapisujeme* $y \to w$ *je pravidlom z množiny* P*. Potom produkčný krok* $xyz \Rightarrow xwz$ *značíme ako* $s_1 \Rightarrow s_2$*.*

Definícia 3.1 predstavuje obecnú definíciu gramatík, ktoré definujú jazyky. V praxi sa jazyky ďalej klasifikujú do *Chomskeho hierarchie*[11] podľa vyjadovacej sily jazyka. Z pohľadu konštrukcie prekladača programovacieho jazyka sú podstatné len *regulárne* a *bezkontextové jazyky.*

3.2 Štruktúra prekladačov

Z pohľadu teórie je teda ľubovoľný programovací jazyk definovateľný pomocou gramatiky. Aby bolo možné programovať v programovaciom jazyku, je nutné zostrojiť program, ktorý prevedie vstupný text (program v programovaciom jazyku) na iný jazyk, spracovateľný pomocou počítaču. Takýto systém sa obecne nazýva *prekladač* a obecne prekladá *vstupný jazyk* na *cieľový jazyk.* Cieľovým jazykom môžu byť napríklad inštrukcie procesora, ktoré sú priamo vykonávateľné procesorom, alebo iný programovací jazyk (napríklad Haskell-to-C). V prípade, že cieľovým jazykom je jazyk slúžiaci k výpočtu, ktorý nie je priamo spracovateľný procesorom, je nutné zostrojiť program, ktorého vstupom bude *medzijazyk* a jeho činnosťou bude výpočet riadený týmto jazykom. Takýto program sa obecne nazýva *interpret.*



Obr. 3.1: Chomskeho hierarchia jazykov. Pre potreby konštrukcie prekladačov sú podstatné len *regulárne jazyky* a *bezkontextové jazyky*. Pri každom typy je znázornený tvar najkomplexnejšieho reťazca, ktorý dokáže daný jazyk vyjadriť. Prebraté z článku [11].

Obecne sa prekladač a naň pripojený interpret skladajú z nasledujúcich častí:

- **lexikálna analýza**
- **syntaktická analýza**
- **sémantická analýza**
- **generátor kódu**
- **výpočet**

Popis obecného prekladača vychádza z knihy [1, kap. 1].

3.2.1 Lexikálna analýza

Aby bolo možné spracovať vety vstupného jazyka, je nutné naprv klasifikovať jednotlivé slová viet do vetných členov – *lexikálnych kategórii*. Každé slovo (angl. *lexém*) je klasifikované do tej kategórie, ktorej jazyk produkuje identický reťazec ako skúmané slovo. Jazyk jednotlivých vetných členov je definovaný pomocou *regulárnych jazykov* a v prípade automatizovaného spracovania pomocou *regulárnych výrazov*, ktoré sú ekvivalentné týmto jazykom. Činnosť lexikálnej analýzy je ilustrovaná príkladom 3.1.

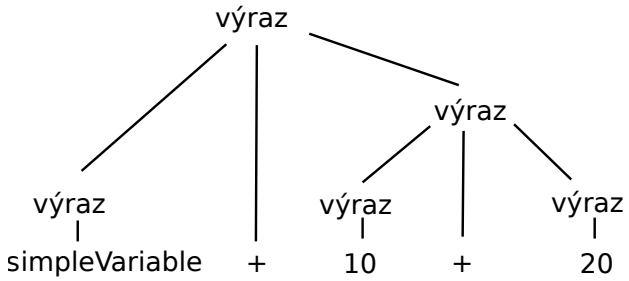
Príklad 3.1. *Uvažujme jednoduchý jazyk G, ktorého vetné členy nech sú čísla a identifikátory, a nech čísla sú definované regulárnym výrazom* `[0-9]+` *a identifikátory výrazom* `[a-zA-Z]+`*. Potom slovo* `00133132` *je platným číslom jazyka G. Podobne slovo roman je identifikátorom jazyka G. Naopak, slovo #peace nie je slovom jazyka G, pretože nepatrí do jazyka žiadneho z definovaných vetných členov.*

Výsledkom lexikálnej analýzy je teda štruktúra (angl. *token*) , obsahujúca typ vetného člena a hodnotu, reprezentujúcu lexém (napríklad literál v prípade identifikátora).

3.2.2 Syntaktická analýza

Vstupom syntaktickej analýzy je sekvencia tokenov, reprezentujúcich vetné členy. Každý jazyk je definovaný ako množina pravidieľ definujúcich konštrukcie, teda povolené postupností vetných členov, ktoré definujú množinu reťazcov, ktorá spĺňa tieto pravidlá.

Príklad 3.2. *Uvažujme o jazyku G, definujúcom číselné výrazy, schopné výpočtu. Nech vetné členy tohoto jazyka sú čísla (zložené z číslic), premenné (zložené z alfa znakov) a symboly predstavujúce numerické operátory (napr. násobenie). Uvažujme nasledujúce pravidlá: vyraz → premenna, vyraz → cislo, vyraz → vyraz+vyraz. Pre reťazec* `mojaPremenná+10+20` *je možné zostrojiť derivačný strom, ktorý je zobrazený na obrázku 3.2.*



Obr. 3.2: Derivačný strom pre reťazec `mojaPremenná + 10 + 20`. Nakoľko gramatika jazyka G je vágna, je možné zostrojiť aj druhý syntaktický strom pre daný reťazec.

Syntaktická analýza je v praxi realizovaná konštrukciou *syntaktického analyzátora* (angl. *parser*). Existujú rôzne typy analyzátorov, ktoré sa líšia komplexnosťou a množinou gramatík, ktoré dokážu spracovať.

Analyzátory môžeme rozdeliť na dva obecné typy:

- **zhora-dole**
  Analyzátor vytvára syntaktický strom zhora (od neterminálnych uzlov) smerom dole (k terminálnym). Patrí sem metóda napríklad `LL(n)`.
- **zdola-hore**
  Analyzátor vytvára syntaktický strom zdola hore, napríklad `LR`.

Výstupom syntaktickej analýzy je *syntaktický strom* (angl. *parsing tree*), ktorý predstavuje rozbor vety podľa pravidiel jazyka.

3.2.3 Sémantická analýza

Obecne sémantická analýza kontroluje platnosť zadaných konštrukcií jazyka v zmysle významu konštrukcií – dochádza ku overovaniu existencie referencovaných funkcií, premenných, ku kontrole typovej kompability a prípadnému pretypovaniu.

3.2.4 Generovanie kódu

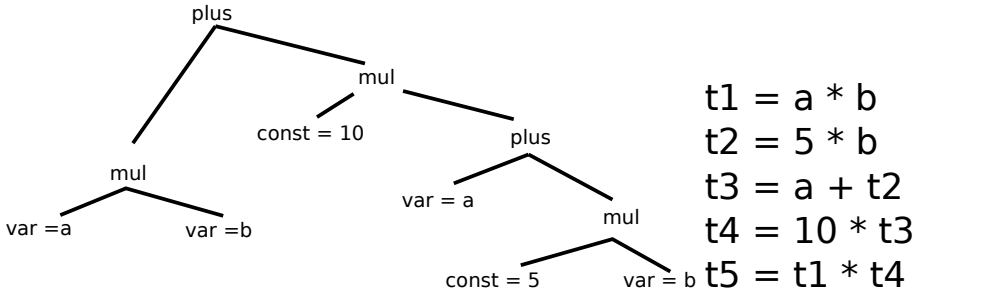
Táto časť prekladu je závislá na konečnom cieli prekladu. V prípade interpretácie tu vzniká naviazanie interpretovateľných štruktúr na jednotlivé uzly *derivačného stromu*.

Výstupom sémantickej analýzy môže byť rozšírený syntaktický strom (angl. *abstract syntax tree* alebo postupnosť inštrukcií, ktorá sa nazýva trojadresný kód (angl. *three-adress code*).

Abstraktný syntaktický strom [1, kap. 6] reprezentuje program vo forme stromu, ktorého uzly vyjadrujú konkrétnu operáciu, napríklad súčet dvoch podstromov.

Trojadresný kód [1, kap. 6] reprezentuje program vo forme sekvencie operácii, ktorých vykonanie je lineárne. Štrukturovanosť a podmienenosť kódu je vyjadrená špeciálnymi inštrukciami, reprezentujúcimi skoky vo vykonávaní programu.

Príklad 3.3. *Uvažujme jednoduchý číselný výraz* `a*b + 10*(a+(5*b))`. *Reprezentácia tohoto výrazu obomi metódami je ilustrovaná na obrázku 3.3.*



(a) Abstraktný syntaktický strom (b) Trojadresný kód

Obr. 3.3: Realizácia výrazu z príkladu 3.3 pomocou oboch spomenutých metód.

3.2.5 Výpočet

Výpočet realizuje vykonávanie kódu (príkazov) programu. V prípade *trojadresného kódu* sa jedná o jednoduché lineárne spracovanie sekvencie inštrukcií a občasné preskočenie inštrukcií alebo skok naspäť, ktoré realizuje príkazy typu `return`, `break` alebo cykly.

V prípade reprezentácie programu stromom je interpretácia programu zabezpečená vyhodnotením jednotlivých uzlov a ich poduzlov. Napríklad, hodnotu výrazu na obrázku 3.3a je možné získať prehľadaním stromu do hĺbky a aplikáciou operácie, reprezentovanej uzlom na hodnoty podstromov.

3.3 Konštrukcia prekladača

Prekladač jazyka je možné implementovať manuálne na základe spomenutých princípov a metód. Výsledný program ale nebude dostatočne flexibilný voči zmenám jazyka. Korektná ručná implementácia niektorých metód (napr. LR parsing) môže byť taktiež náročná a náchylná ku vzniku chýb.

Z týchto dôvodov sa vyvinuli nástroje, ktoré umožňujú generovať zdrojové kódy, realizujúce jednotlivé časti prekladu. Táto bakalárska práca využíva nástroje `Flex` a `Bison`.

3.3.1 Flex

Pre spracovanie vstupného textu je využitý voľne dostupný, prenositeľný generátor Flex, šírený pod licenciou BSD. Flex umožňuje definovať tokeny v komfortnej podobe a následne vygenerovať deterministický konečný automat [13, str. 2], zapísaný v jazyku C alebo C++.

Vo Flexe sa jednotlivé tokeny definujú pomocou regulárneho výrazu. Prípadne kolízie regulárnych výrazov sú vrámci tohoto generátora riešené dĺžkou označeného reťazca, prípadne chronológiou zoznamu pravidiel, teda skôr zadané pravidlo má prioritu voči neskôr zadanému [13, str. 22].

Pravidlá sa zadávajú v následujúcej forme:

```
"<REGULARNY-VYRAZ>" {<KOD>}
```

`<REGULARNY-VYRAZ>` predstavuje reťazec, ktorého prečítaním zo vstupu sa vykoná kód `<KOD>`. Výsledkom kódu je obvykle návratová hodnota, ktorá je predaná lexéru pomocou konštrukcie `return`.

Typ návratovej hodnoty je explicitne špecifikovať priamo v zdrojovom súbore s pravidlami, alebo importovať z výstupu Bisona.

3.3.2 Bison

Táto časť projektu je založená na generátore syntaktického analyzátora GNU/Bison, ktorý pre zadanú gramatiku v BNF [1] forme vytvorí preložiteľný C/C++ kód, predstavujúci LALR[3, str. 1] sytaktický analyzátor.

Bison umožňuje generovať parser vo forme triedy C++, ktorú je možné viacnásobne inštanciovať počas behu programu a spustiť viacero nezávislých prekladov. Samotný objekt parseru je reentrantný [13, str. 132], teda umožňuje opätovný preklad bez nutnosti vytvoriť novu inštanciu.

Bison rozližuje medzi terminálnym a non-terminálnym symbolom. Terminálne symboly sú definované pomocou špecialneho príkazu a predstavujú jednotlivé tokeny, získané lexérom.

Non-terminálne symboly sú definované pravidlami gramatiky. Pravidlá sú definované v následujúcom formáte:

```
<nazov-symbolu>:
        <symbol1> <symbol2> ... { }
        | <symbol3> <symbol4> ... { }
```

Pravidlá môžu mať viacero alternatív, ktoré sú v definícii oddelené pomocou znaku `|`.

[1] Backus-Nauerová forma

Kapitola 4

Návrh jazyka pre procedurálne generovanie

Táto kapitola popisuje konkrétne aspekty navrhovaného jazyka, princípy jeho fungovania ako aj abstraktný návrh realizácie.

4.1 Požiadavky na jazyk

V kapitole 2 boli predstavené jednotlivé techniky generovania. Cieľom tejto práce je vytvoriť taký programovací jazyk, ktorý by umožnil implementovať tieto techniky, prípadne ich rozšírenejšie varianty.

Preto jazyk, ktorý bol navrhnutý v tejto práci vychádza vo svojom základe z *L-systémov* z podkapitoly 2.3. Dôvodom je vysoká expresívnosť týchto systémov ako aj priame zakomponovanie rekurzie v podobne prepisovania symbolov. Rozšírením jazyka o numerické výpočty a obecný imperatívny programovací model bude možné implementovať ľubovoľný L-systém a zároveň aj obecnejšie algoritmy, akými sú napr. BSP (zo sekcie 2.2.4), či *bodovo-orientované metódy* z podkapitoly 2.2.

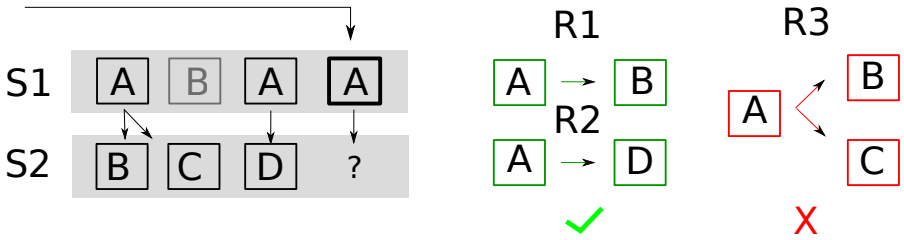
Nakoľko cieľom je vytvoriť obecný jazyk pre procedurálne generovanie, bude potrebné zabezpečiť komfortnú, ale zároveň efektívnu *obojstrannú komunikáciu so systémom*, ktorý bude integrovať jazyk a umožní tak pracovať s výsledkami generovania.

4.2 Definícia jazyka

Fungovanie programu popísaného navrhovaným jazykom je založené na prepisovaní typovaných symbolov, ktoré sú spoločne s pravidlami definované uživateľom a v dobe behu programu sa ich štruktúra nemení.

Po spustení programu sa vytvorí uživateľom definovaný počiatočný reťazec symbolov s počiatočnými hodnotami. Následne prebiehajú takzvané *prepisovacie cykly*. V každej iterácii sa sekvenčne pre každý symbol aktuálneho reťazca vyhodnotia všetky pravidlá, definované pre jeho typ a z množiny prípustných pravidieľ sa náhodne vyberie jedno pravidlo. Prípustnosť pravidla je založená na vyhodnotení logiky pravidla.

Vybrané pravidlo je aplikované a jeho aplikáciou môžu vniknúť nové symboly, ktoré sa pridajú na koniec reťazca pre ďalšiu iteráciu. V prípade, že žiadne z pravidieľ pre daný symbol nie je aplikovateľné, je aplikované *implicitné pravidlo*, ktoré tento symbol skopíruje k ďalšej iterácii.



Obr. 4.1: Písmená A, B, C, D predstavujú typy symbolov. Obrázok znázorňuje stav systému pri prepisovaní posledného symbolu aktuálneho reťazca $S1$. Pravidlo $R3$ nemá splniteľný predikát, pretože symbol B , ktoré pravidlo preskakuje (agreguje), nenasleduje za aktuálnym symbolom A. Pre aktuálnym symbol sú splnené pravidlá $R1$ a $R2$ a jedno z nich bude náhodne vybrané a aplikované.

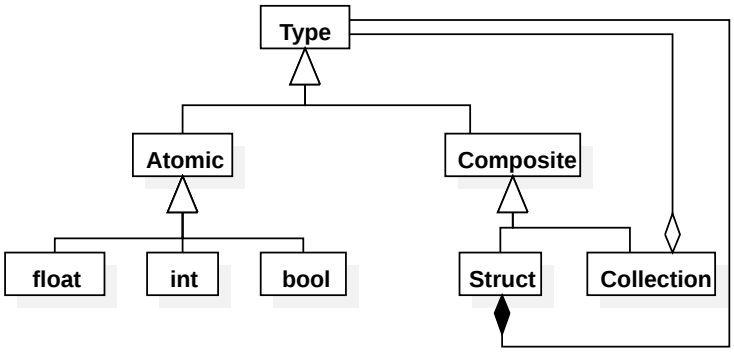
Výpočet je ukončený po skončení prepisovacej iterácie ak počas iterácie nebolo použité ani jedno z uživateľských pravidiel. V opačnom prípade výpočet pokračuje až do splnenia tejto podmienky, v prípade zadania maximálneho počtu iterácií len po daný počet iterácií.

4.2.1 Symboly

Symboly sú typované. Každý uživateľom definovaný symbol predstavuje unikátny typ, ktorý predstavuje štruktúru, zloženú z *atomických* (preddefinovaných typov) alebo z iných *kompozitných typov*.

Atomické typy pozostávajú z jednoduchého konečného celočíselného typu alebo jednoduchého typu s pohyblivou desatinnou čiarkou.

Nad všetkými typmi je možné definovať *zložené typy* v podobe štruktúry pevných členov alebo kolekciu ľubovoľných prkov. Tieto štruktúry je možné zanorovať definovaním nových nadtypov. Jednotlivé zložky majú vrámci štruktúry unikátny názov, ktorý predstavuje ich identifikáciu. Štruktúry sú vzájomne rôzne aj v prípade ekvivalentných podtypov a ich usporiadaní.



Obr. 4.2: Diagram, znázorňujúci typovú hierarchiu. Typy je možno rozdeliť na atomické (celočíselný, logický, desatinný typ) alebo zložené (kompozitné), ktoré sú zložené z iných typov.

Špeciálnym typom je typ *kolekcia*. Kolekcia umožňuje vytvoriť nehomogénne dynamické pole prvkov. Aby bolo možné pracovať s kolekciou v typovanom jazyku, je nutné zaviesť univerzálny typ *any*, ktorému je možné priradiť ľubovoľnú hodnotu iného typu. Zároveň je ale nutné zaviesť špeciálne operácie pre získanie hodnoty tohoto typu. V praxi sa jedná o *dynamické pretypovanie*, ktoré v dobe behu overí, či hodnota premennej je naozaj hodnotou želaného typu. Komplementárne k tomuto mechanizmu je nutné definovať *typové predikáty*, ktoré umožnia logické rozhodovanie a následne pretypovanie *any* premennej.

4.2.2 Pravidlá a funkcie

Uživateľ taktiež špecifikuje pravidlá. Každé pravidlo je zložené z dvoch časti: *predikátovej funkcie* a *procedúry*.

Predikátová funkcia zobrazuje parametre symbolu na boolovskú hodnotu, ktorá určuje, či je pravidlo aplikovateľné na aktuálny symbol.

Procedúra pravidla predstavuje akciu, resp. súbor akcií, ktoré sa majú vykonať pri aplikácii pravidla na symbol.

Pri špecifikovaní pravidla uživateľ definuje obe zložky pravidla. Telo každej zo zložiek je tvorené sekvenciou *príkazov*. Príkaz je buď jednoduchý (napr. priradenie) alebo zložený (cyklus, podmienka). Príkazy rovnako pracujú s *výrazmi*, ktoré predstavujú vyčísliteľnú hodnotu s typom.

Pre zvýšenie komfortu je rovnako možné definovať vlastné pomocné funkcie, ktoré je následne možné použiť v iných funkciach a výrazoch. Jazyk ale striktne nepovoľuje rekurziu vo funkciách.

4.2.3 Derivačný strom

Pre zabezpečenie kontextovosti jazyka je zavedený takzvaný prístup k derivačnému stromu. Pomocou tohoto mechanizmu je možné pristupovať k symbolom, ktoré boli derivované v tej istej iterácii systému (pravý a ľavý kontext) ako aj ku symbolu, ktorého deriváciou vznikol.

4.2.4 Vlastnosti jazyka

Z predchádzajúcej definície jazyka môžeme konštantovať niekoľko vlastností jazyka. Jazyk má nasledujúce vlastnosti:

- **parametrickosť** - zabezpečená definovaním symbolu ako štruktúry, zavedení predikátovej funkcie
- **stochastickosť** - umožnená definíciou viacerých pravidiel pre identický symbol, kde práve jedno z pravidiel je finálne vybrané k prepisu symbolu
- **kontextovosť** - vďaka dostupnosti mechanizmu pre prístup ku derivačnému stromu a tým ovplyvniť výpočet na základe okolia (kontextu)

Je preto možné vytvoriť algoritmus prevodu ľubovoľného parametrického, stochastického L-systému na program navrhovaného jazyka. Týmto spôsobom je možné dokázať vyjadrovaciu silu navrhovaného jazyka, ktorý má potom minimálne takú silu ako spomínaný L-systém.

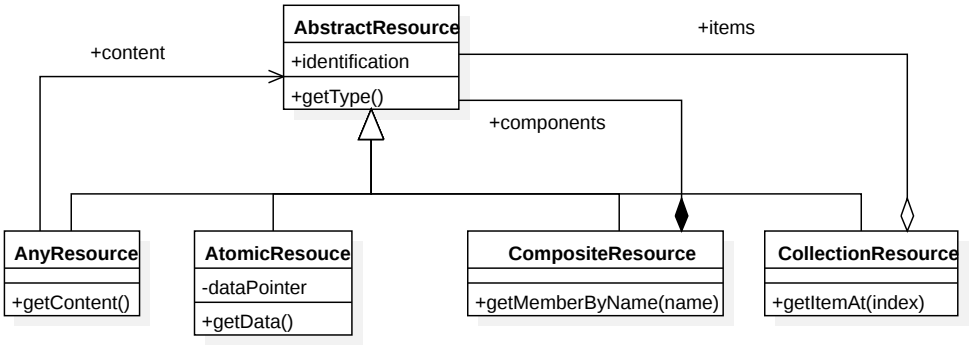
4.3 Objektový návrh implementácie jazyka

Pred samotnou implementáciou jazyka je nutné premyslieť ako sa budú jednotlivé aspekty jazyka mapovať na štruktúry v kóde, ako budú jednotlivé štruktúry uložené v pamäti a ako budú vyzerať ich vzájomné väzby. Vzhľadom na rozhodnutie implementovať jazyk v jazyku C++ je vhodné zvoliť objektovo orientovanú analýzu a návrh.

4.3.1 Symboly

Ako už bolo zmienené, symboly sú typované štruktúry. V praxi predstavujú typy hierarchiu podtypov, ktorej listovej uzly sú atomické typy. Tento vzťah je možné modelovať štruktúrnym návrhovým vzorom *composite*[6], ktorý definuje ako vytvárať rozšíriteľné triedne hierarchie.

Obrázok 4.3 znázorňuje vizualizáciu hiararchie pomocou jazyka `UML` [1], reprezentujúcej štrukturované hodnoty typov.



Obr. 4.3: Návrh hierarchie hodnôt typov, zapísaný v jazyku UML. `AnyResource` predstavuje univerzálny typ s hodnotou iného typu, `AtomicResource` predstavuje neštruktúrovaný typ (float, int). `CompositeResource` a `CollectionResource` predstavujú zložené typy z inýćh typov – štruktúru a nehomogénne dynamické pole

.

Každý typ sa obecne skladá zo svojej definície, teda metainformácií a z množiny svojich hodnôt, teda v prípade OOP z kolekcie inštancií tried.

Hodnoty typov sú reprezentované abstraktnou triedou `AbstractResource`, ktorá uchováva identifikáciu typu. Táto trieda zároveň poskytuje metódy pre prístup k hodnotám, názvu typu ako aj spôsobu, ako kopírovať hodnotu inštancie do inej inštancie totožného typu.

V prípade atomických typov obsahuje inštancia triedy `AbstractResource` priamo uloženú hodnotu typu. V opačnom prípade obsahuje inštancia odkazy na jednotlivé podtypy v hierarchii.

Pomocou mechanizmu triednej dedičnosti sú potom definované jednotlivé typy typov:

- `AtomicResource` definuje typy, ktorých hodnotu možno serializovať do binárneho reťazca.
- `CompositeResource` definuje štruktúrované typy a pomocné metódy, ktoré umožňujú prístup k podtypom

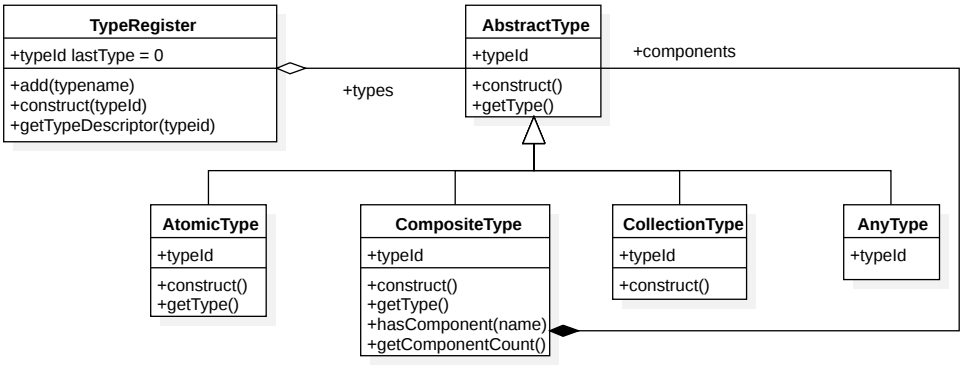
[1] Unified Modeling Language

- `CollectionResource` definuje obecný typ kolekcie ľubovoľných štruktúr a pomocné metódy pre pridanie, odstránenie a manipuláciu s členmi kolekcie

Z hľadiska uloženia metadát sa javí neefektívne ukladať úplný popis typu v každej inštancii typu. Preto je nutné zaviesť akýsi register, ktorý by uchoval popis typu na jednom mieste. Z toho dôvodu je pridaná trieda `TypeRegister`, ktorej účelom je spravovať definované typy ako aj ponúkať mechanizmus k tvorbe inštancii takýchto typov. Zároveň predstavuje register typov jednoduchý mechanizmus overenia existencie typu nutný pre sémantickú analýzu.

Pre každý typ uchováva tento register popis metadát typu vo forme inštancie triedy typu `AbstractType`, asociovanú s menom typu ako aj konštruktor hodnoty tohoto typu. Registrácia typov prebieha počas inicializácie systému.

Na obrázku 4.4 je znázornená vizualizácia vzťahov spomenutých tried.



Obr. 4.4: Triedny diagram hierarchie reprezentujúcej typy. Jednotlivé triedy uchovávajú metainformácie o konkrétnom type. Napríklad trieda `CompositeType` uchováva odkazy na názvy členov štruktúry a ich `AbstractType`, ktoré ich reprezentujú.

Vzhľadom na fakt, že typ môžeme byť buď atomický alebo kompozitný, je nutné aj popis typu členiť do hierarchie tried. Trieda `TypeDecriptor` predstavuje potom abstraktné rozhranie k triedam `AtomicType` a `CompositeType`. Trieda `AtomicType` uchováva popis atomického typu. Trieda `CompositeType` obsahuje odkazy na iné `AbstractType` a k nim asociované názvy zložiek. Poskytuje preto predikát pre overenie prítomnosti zložky typu.

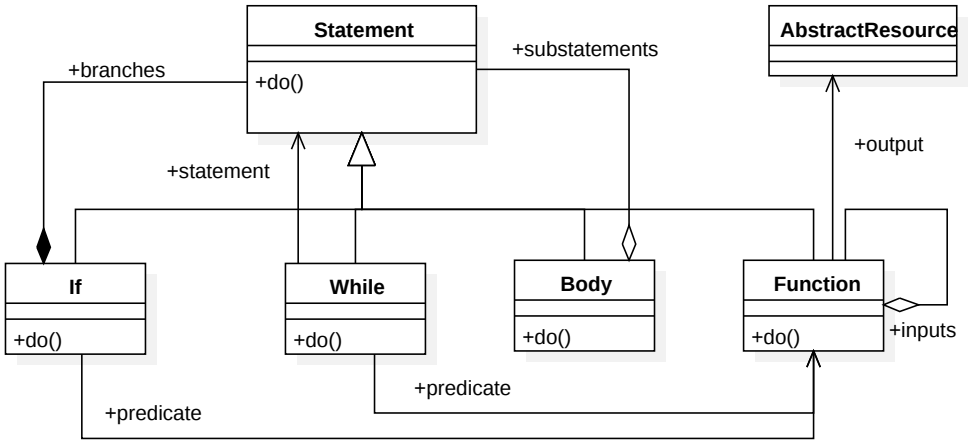
4.3.2 Príkazy a výrazy

Pre realizáciu pravidiel je nutné navrhnúť spôsob, akým bude implementovaná logika pravidla, teda mechanizmus prepisu vstupnej štruktúry na nové symboly v ďalšej iterácii. Pre tento jazyk bola zvolená logika založená na príkazoch. Obecne pre ľubovoľný príkaz platí, že je vykonateľný a výsledkom jeho vykonanie je zmena stavu.

Keďže existuje niekoľko typov príkazov, môžeme modelovať príkaz ako abstraktnú triedu a konkrétne typy príkazov ako triedy, ktoré rozširujú tento príkaz.

Objektový návrh reprezentácie príkazov a funkcií je vizualizovaný na obrázku 4.5.

Jednoduché výrazy sú modelované triedou `Function`. Táto trieda prestavuje modul, ktorý má vstupy a výstup. Vstup je realizovaný pomocou odkazov na iné inštancie triedy `Function`, kdežto výstup sa viaže na inštanciu triedy `AbstractResource`. Tento model umožňuje vytvárať komplexné výrazy, ktoré sú potom realizované vyhodnotením podvý-



Obr. 4.5: UML diagram pre výpočetnú hierarchiu. `Statement` predstavuje obecný výpočetný blok. Pomocou odvodených blokov `If`, `While`, `Body` je možné vyjadriť ľubovoľný algoritmus pomocou kompozície blokov. Blok `Function` umožňuje získať výpočtom obecnú hodnotu.

razov a operáciou nad nimi. Taktiež musí existovať špeciálna tzv. *nulárna funkcia*, ktorá umožňuje sprístupniť typovú hodnotu (inštanciu `AbstractResource`) ako vstup do modulu.

Výrazy sú teda realizovateľné ako stromy, ktorých listami sú nulárne funkcie, uzlami operácie a výsledkom je hodnota.

Príkaz môže byť taktiež zložený a vtedy sa jeho vykonaním postupne vykonajú príkazy, z ktorých sa skladá. Takýto príkaz sa nazýva sekvenciou príkazov a je modelovaný triedou `Body`.

Aby bolo možné implementovať príkazy, ktorých vykonanie je podmienené stavom premenných, je nutno pridať podmienený príkaz. Ten reprezentuje trieda `If` a umožňuje vykonať jeden z dvoch príkazov v závislosti na platnosti podmienky, teda logického výrazu.

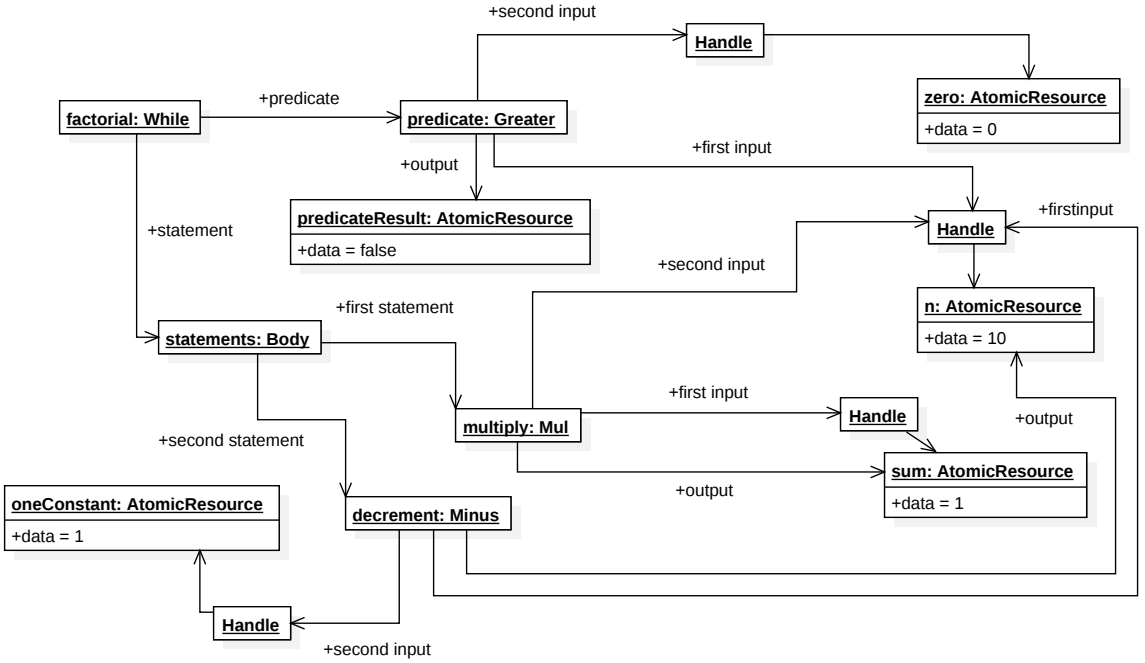
Nakoniec, pre úplný výpočetný model je nutné pridať konštrukciu, ktorá umožní opakovaný výpočet na základe splnenia výrazy. To je zabezpečené triedou `While`, ktorá zastupuje rovnomenný cyklus.

Konštrukcie, ktoré sme si práve uviedli nám umožňujú realizovať ľubovoľnú sekvenciu príkazov. Pre uživateľa by bolo vhodné keby si mohol vytvoriť vlastnú funkciu, ktorá by mala vstupy a výstupy a vnútri by bola realizovaná pomocou obyčajných príkazov. Z tohoto dôvodu je nutné zaviesť triedu `FunctionRegister`, ktorá umožní definovať uživateľské funkcie a následne vytvoriť ich inštancie, ktoré môžu byť ďalej použité vo hierarchii objektov.

Príklad realizácie výpočtu faktoriálu čísla, ktorého pseudokód je na výpisku 4.1, je znázornený na obrázku 4.6. Reprezentácia algoritmu objektami vytvára acyklický graf.

```
1 int n = 10; int sum = 1;
2 while(n > 0) {
3     sum = sum * i;
4     n = n - 1;
5 }
```

Výpis 4.1: Pseudokód, ktorý realizuje algoritmus pre výpočet faktoriálu čísla N. Reprezentácia tohoto algoritmu objektami je znázornená na obrázku 4.6.



Obr. 4.6: Objektový diagram, znázorňujúci objektovú hierarchiu, ktorá implementuje výpočet faktoriálu (výpis 4.1). Objekty triedy `Handle` predstavujú pomocné *nulárne funkcie*, ktoré sprístupňujú konečnú hodnotu. Koreňom grafu je objekt `faktorial`. Pri spustení jeho výpočtu je periodicky overená platnosť predikátu a následne vyčíslený blok `statements`. Triedy `Mul`, `Greater` a `Minus` reprezentujú príslušné binárne operácie $*$, $>$ a $-$.

4.4 Výpočtový model jazyka

V tejto časti návrhu je bližšie definovaný výpočtový model jazyka, možnosti riešenia niektorých jeho problémov ako aj popísanie komunikácie jazyka s okolím.

4.4.1 Pravidlá a mechanizmus derivácie

Aby bolo možné korektne implementovať jazyk a jeho výpočtový model, je nutné formálne definovať algoritmus, podľa ktorého bude výpočet prebiehať.

Algoritmus 1 formálne definuje proces prepisovania štruktúr v procedurálnom jazyku. Funkcia `GetSymbol` vracia i-ty symbol z reťazca s, ďalšia funkcia `GetSetOfApplicableRules` získa množinu pravidiel pre daný symbol, ktorých predikát bol splnený, následne funkcia `ApplyRuleOnSymbol` spustí procedúru pravidla nad prepisovaných symbolom a novovytvorené symboly do množiny.

4.4.2 Problém zastavenia

V predchádzajúcej časti boli definované elementárne časti jazyka ako aj mechanizmus jeho sémantiky. V tomto momente je zrejmé, ako definovať symboly, teda štruktúry a ako zaviesť pravidlá, ktoré umožnia transformáciu na iné symboly. Sme teda schopní definovať základné teoretické L-systémy. V praxi je ale nutné zaistiť, aby bol výpočet v konečnom čase ukončený.

Problémom je ako zaistiť, aby výpočet skončil. Z pohľadu programátora preto existuje niekoľko spôsobov, ako zastaviť výpočet:

Algoritmus 1: Výpočtový model derivácie, realizujúci prepisovací proces.

```
Data: vstupný reťazec, definované typy symbolov a ich pravidlá
Result: Reťazec štruktúrovaných symbolov AbstractResource
s ← inicializacia vstupneho reťazca;
isOver ← false;
while not isOver do
    s_n ←'';
    isOver ← true;
    for i ← 1 to s.length do
        sym ← GetSymbol(i);
        rules ← GetSetOfApplicableRules(i);
        if not rules.empty then
            isOver ← false;
            symbolsToSkip ← 0;
            rule ← RandomlyChooseOneFromSet(rules);
            outputSymbols ← ApplyRuleOnSymbol(rule, sym,symbolsToSkip);
            StringAppend(s_n, outputSymbols);
            i ← i + symbolsToSkip;
        end
        else
            StringAppend(s_n, ApplyImplicitRule(sym));
        end
        i ← i + 1;
    end
end
SavePreviousString(s);
s = s_n;
```

- **Obmedzenie počtu krokov**
  Programátor explicitne stanoví *maximálny počet derivačných krokov* a výpočet tým pádom skončí najneskôr po stanovem počte iterácií.

- **Obmedzenie pravidla podľa parametru**
  Vzhľadom na algoritmus je výpočet ukončený v momente, keď pre všetky symboly reťazca nie sú prípustné iné ako implicitné pravidlá. Preto programátor môže po istom počte krokov ukončiť výpočet negatívnym vyhodnotením predikátu pravidla a to buď:

  - na základe globálnej premennej
  - na základe vstavanej premennej systému, definujúcej číslo iterácie
  - na základe hodnôt z prepisovaného symbolu

4.4.3 Problém vstupu a výstupu jazyka

Pri vytváraní dynamického systému, ktorého činnosť je ovplyvnená okolím, je nutné uvažovať na spôsobe akým system bude komunikovať so svojím okolím.

Komunikácia programovacieho jazyka s okolím je v bežných viacúčelových jazykoch realizovaná procedúrami pre prácu so súborovým systémom, prípadne zápisom do štandardných súborov. V prípade jazyka pre procedurálne generovanie toto nie je možné, nakoľko sa očakáva vstavanie systému do iného, komplexného systému.

Riešenie spočíva v uvedomení si štruktúry systému. Nakoľko jazyk realizuje generovanie symbolov z počiatočného reťazca, je nutné zabezpečiť pred začiatkom výpočtu naplnenie tohoto reťazca symbolov počiatočnými symbolmi. Podobne, výsledkom generovania je len ďalší reťazec symbolov.

Počiatočné nastavenie generovania jazyka je možné ovplyvniť nasledujúcimi spôsobmi:

- **Inicializačná funkcia**
  Volaním špeciálnej procedúry si skript sám vytvorí štartovný reťazec. Tento spôsob sám o sebe neumožňuje otvorenosť systému, ale predstavuje praktický spôsob definície generovania.

- **Globálne parametre**
  Na ovplyvnenie generovania je možno použiť globálne premenné, ktorých obsah bude možno zmeniť pomocou rozhrania systému. Tento model je inšpirovaný globálnymi premennými v jazyku `GLSL` [2], ktorý umožňuje parametrizovanie programov, bežiacich na grafických procesoroch pomocou programu na klasickom CPU. Nakoľko zápis spočíva v zapísaní jednoduchých hodnôt, je tento mechanizus limitovaný na atomické typy akými sú `float` či `int`.

- **Vstupný reťazec**
  Poslednou možnosťou je umožniť prístup nielen ku serializovaným výstupom generovania, ale v podobnom formáte takto dáta vkladať na počiatku generovania. Tento spôsob ale značne predpokladá znalosť vnútornej štruktúry generovacieho skriptu, nakoľko je jazyk typovaný a preto by serializovaná štruktúra musela obsahovať metainformácie o jednotlivých typoch konštrukcie. Výhodou je ale jednoduché zreťazenie generovania, ktoré je pojednávané v podkapitole 4.4.4.

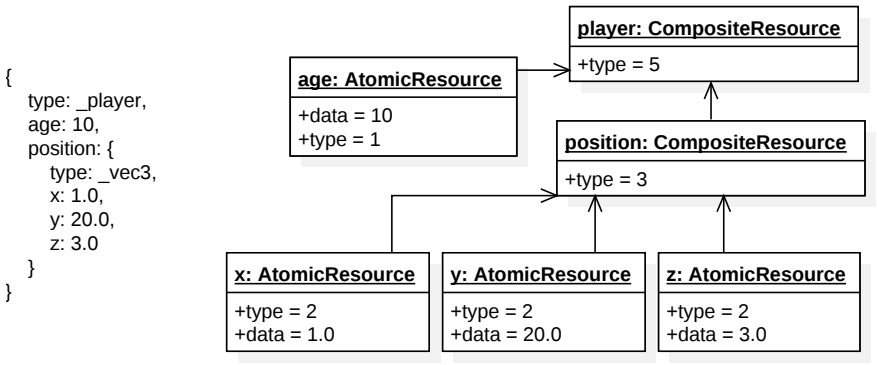
Týmito spôsobmi je teda zabezpečený vstup jazyka a ovplyvnenie jeho činnosti. Ďalšou kľúčovou otázkou je navrhnúť, ako sprístupniť výsledky generovania.

Z podstaty jazyka a generovania vieme, že výsledkom je vždy reťazec symbolov, ktorými sú obecné zanorené štruktúry alebo kolekcie. Vzhľadom na zanorenosť budeme musieť zvoliť minimálne bezkontextový serializačný formát, ktorý nám umožní zakódovať hodnoty štruktúr do textovej reprezentácie.

Serializáciu bude možno prevádzať buď na celom reťazci, alebo postupne na jednotlivých symboloch. Dôvodom sú pamäťové obmedzenia. Napríklad pri generovaní fraktálnych útvarov je možné vytvoriť už pri malom počte iterácií niekoľko tisíc objektov. V prípade, že by mal byť každý objekt reprezentovaný ako text, zaberala by v pamäti jeho textová reprezentácia minimálne toľko miesta koľko jeho binárna reprezentácia. Dôsledkom toho boli dvojnásobné pamäťové nároky pri generovaní. Naopak, postupným serializovaním jednotlivých symbolov sa nezmení časová zložitosť, ktorá bude stále lineárna, ale *zlepší sa pamäťová zložitosť* na veľkosť textovej reprezentácie najväčšieho zo symbolov.

Príklad serializácie do formátu `JSON` [3] je znázornený na obrázku 4.7.

[2] OpenGL Shading Language
[3] JavaScript Object Notation



Obr. 4.7: Príklad konverzie medzi textovým serializačným formátom `JSON` a objektovou hierarchiou v procedurálnom jazyku.

4.4.4 Generovacia hierarchia

Vďaka textovej modulárnosti navrhnutého jazyka je možné jednoducho zabezpečiť dekompozíciu skriptov na jednotlivé časti, prípadne jednotlivé úrovne generovania.

Pomocou možnosti serializácie do formatú `JSON` a spätnej deserializácie na inštancie objektov v návrhu jazyka je možné obojstranne komunikovať s jazykom. Táto vlastnosť zároveň otvára priestor k vytvoreniu *hierarchie generovania*.

Hierarchia generovania predstavuje koncept, ktorý umožňuje vytvoriť systém generovacích modulov, ktorých vstupy a výstupy by boli navzájom prepojené. Príkladom nech je generovanie mestskej zástavby. Na vrchole hierarchie by existoval modul, ktorý by generoval pôdorys mesta, pozemky pre domy, cesty a podobne. Konečným výstupom tohoto modulu by boli metaobjekty, predstavujúce pozíciu a ohraničenie domu, pozíciu pre strom a podobne, na ktoré by bolo možné naviazať moduly pre generovanie konečných modelov z konkrétnych metaobjektov.

Príkladom môže byť generovanie generovanie scény hradu, ktorá sa skladá z terénu, stromov a samotného hradu. Tento príklad je znázornený a popísaný na obrázku 4.8.

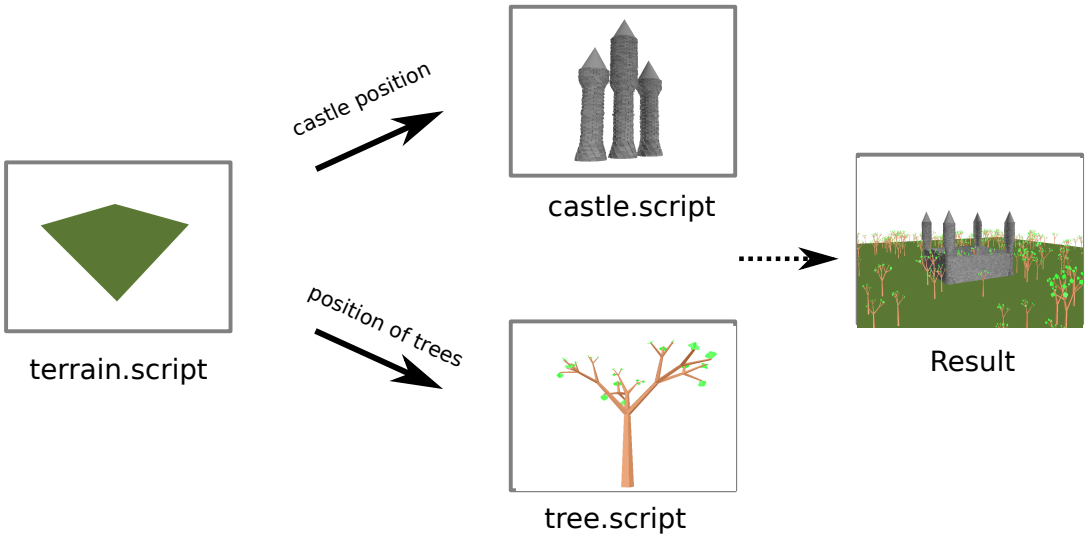
Výhodou tohoto systemu by bola silnejšia dekompozícia na samostatné moduly, ktoré by sa dali jednoducho znovu využiť. Zvýšila by sa flexibilita generovania, nakoľko modul pre generovanie domov by bolo možné nahradiť iným, ktorý by napr. generoval panelové domy, či mrakodrapy.

Podmienkou pre zavedenie takéhoto systému je *jasne definované rozhranie modulov* a možnosť prepojenia vstupov a výstupov. Vďaka serializačnej vlastnosti navrhovaného jazyka je možné podobný systém prakticky realizovať. Slabšiu dekompozíciu je možné taktiež realizovať rozdelením generovacieho skriptu na viacero podskriptov a ich následným prepojením pomocou definovania nových pravidiel pre terminálne symboly jednotlivých skriptov.

4.4.5 Agregácia symbolov

Zaujímavou súčasťou jazyka je agregácia symbolov. Vďaka prístupu k derivačnému stromu môže pravidlo derivácia pristupovať ku kontextu derivovaného symbolu.

Tu sa objavuje možnosť agregácie. Pravidlo môže prechádzať pravý kontext symbolu a spracovať hodnoty symbolov. Následne môže požiadať prepisovací systém o preskočenie derivácie N symbolov v pravom kontexte.



Obr. 4.8: Príklad využitia generovacej hiararchie pre vygenerovanie scény s hradom a stromami. Na počiatku skript vygeneruje terén a metaobjekty, reprezenetujúce hrad a stromy v teréne. Následne je spustený iný skript, ktorého vstupom je metaobjekt, reprezentujúci hrad, a ďalší, ktorý implementuje stromy. Nakoniec, výstupy týchto skriptov sa spoja a výsledkom je scéna s hradom.

Agregácia symbolov je možná vďaka zavedeniu typu kolekcia. To umožňuje definovať typ, ktorý by obsahoval kolekciu iných typov a teda agregovať niekoľko symbolov do jedného.

4.4.6 Rozhranie jazyka

Okrem definície symbolov a pravidiel prepisovania musí jazyk umožniť komfortnú manipuláciu so symbolmi, ich členami ako aj s derivačným podsystémom.

Z tohoto dôvodu sú súčasťou jazyka vstavané funkcie (angl. *native functions*). Tie zahrňujú nasledujúce kategórie funkcii:

- **Operácie nad atomickými typmi**
- **Operácie nad kompozitnými typmi**
- **Pridanie symbolov do derivačného reťazca**
- **Dynamické typy a pretypovanie**
- **Prístup ku derivačnému stromu**
- **Generátory náhodných rozložení**

Kapitola 5

Implementácia knižnice

Aby bolo možné nový navrhnutý programovací jazyk s názvom `ProcGen` prakticky využiť, je nutné implementovať systém, ktorý bude realizovať výpočet navrhnutého jazyka a umožní konfiguráciu z textového súboru. Táto kapitola popisuje implementačné detaily tohoto systému z pohľadu architektúry a popísania jednotlivých častí.

5.1 Architektúra procedurálnej knižnice

Architektúra knižnice je znázornená na obrázku 5.1. Knižnica s procedurálnym jazykom je dekomponovaná do nasledujúcich častí:

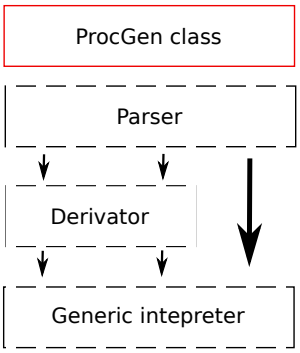
- **generický interpret**
  Zabezpečuje definíciu typov, štruktúr, funkcií a výpočet.
- **derivačný modul**
  Realizuje úkony spojené s prepisovacím mechanizmom a algoritmom výpočtového modelu jazyka.
- **prekladač**
  Spracováva textový vstup a jeho jazyk na reprezentáciu jazyka pomocou interpretu, prípadne derivačného modulu.
- **trieda Procgen**
  Zabezpečuje jednoduché rozhranie pre uživateľov knižnice, ktoré zakrýva zložitosť knižnice.

Implementácia je ukrytá do menného priestoru `ProcGen` z dôvodu predchádzania menných kolízií.

5.2 Generický interpret

Pre realizáciu jazyka bol vytvorený nový interpret s dôrazom na objektovo-orientované programovanie. Interprét je generický z dôvodu možného znovuvyužitia v budúcnosti. Generickosť je zabezpečená presunutím jazykovo-závislej funkcionality do derivačného modulu.

Štruktúra interpretu odpovedá objektovo-orientovanému návrhu zo sekcie 4.3. Implementácia rozširuje návrh o možnosť serializácie inštancií triedy `AbstractType` pomocou



Obr. 5.1: Vizualizácia architektúry knižnice. Diagram znázorňuje jednotlivé moduly a ich závislosti.

virtuálnej metódy `to_json()`. Deserializácia je následne umožnená pomocou metódy `createResourceFromJSON()` v triede `TypeRegister`.

Funkcie a príkazy sú taktiež implementované podľa OOP návrhu. Metóda `do()`, ktorá realizuje v návrhu výpočet samotnej komponenty, je v implementácii realizovaná pomocou C++ operátora `operator()()`, čo umožňuje spustiť výpočet podobne, ako keby komponenta bola funkciou v jazyku C++.

Štandardné funkcie sú definované v súbore `function.h` a ich zaregistrovanie pod fukčným menom spolu s registráciou typov prebieha v súbore `std.cpp`

Pre zmenu výpočetného modelu v prípade použiatia príkazu `return` interpret používa triedu `RunStatus`, ktorej referencia je predávaná ako parameter výpočtu.

Aby bolo možné zabezpečiť korektnú prácu s alokovanými zdrojmi, využíva interpret štandardnú šablónovú triedu `std::shared_ptr`, uvedenú v štandarde C++11, ktorá realizuje funkcionalitu počítania referencií na alokovaný zdroj a automatickú dealokáciu zdroja.

Hlásenie chýb je v interprete implementované dvojakým spôsobom. V čase kompilácie, teda zostavovania a viazania jednotlivých komponent, je použitý klasický spôsob hlásenia chýb pomocou návratových kódov volaní funkcií. V čase výpočtu (behu programu) neočakávaná chyba vedie ku vyvolaniu výnimky jazyka C++, čo umožňuje elegantne riadiť a ukončiť výpočet centrálne.

5.3 Derivačný modul

Derivačný modul implementuje logiku jazyka, definovanú v sekcii 4.4. Centrum funkcionality modulu je sústredené v triede `Derivation`, ktorá poskytuje nasledujúce akcie:

- **výpočet**
  Výpočet pozostáva z iteratívneho prepísania aktuálneho reťazca na nový reťazec symbolov. Výpočet realizuje algoritmus 1.

- **prístup ku derivačnému stromu**
  Vytvorenie, správa a dotazovanie derivačného stromu.

Niektoré akcie derivačného modulu pracujú priamo s interpretom. Napríklad vyhodnotenie predikátu a prevedenie pravidla je implementované ako volanie príslušnej užívateľom definovanej funkcie, ktorú spravuje interpret. Derivačný modul zároveň rozširuje vstavané

funkcie, definované v interprete, o funkcie, potrebné ku prístupu k derivačnému stromu, napríklad `getSymbol()`, či `setMaximumIterations()`.

5.4 Prekladač jazyka

Prekladač jazyka je realizovaný pomocou knižníc `Bison` a `Flex`. Obe knižnice umožňujú vygenerovať syntaktický a lexikálny analyzátor vo forme triedy jazyka `C++` bez vzniku menných kolízii a využitia globálnych symbolov a premenných.

5.4.1 Lexikálna analýza

Lexikálna analýza je realizovaná pomocou knižnice `Flex`. Pre jazyk sú definované vyhradené kľúčové slová jazyka, vypísané na výpisku 5.1.

```
using       struct      rule        parameter       if
else        while       return      typeid          true
convert     insert      at          size            any
del         int         float       bool            false
```

Výpis 5.1: Zoznam kľúčových slov v jazyku ProcGen

Pre jazyk sú definované lexikálne kategórie a operátory, ktorých výčet je na výpisku 5.2.

```
[0-9]+\.[0-9]+                  FLOAT
[0-9]+                          INTEGER
[a-zA-Z0-9]+                    IDENTIFIER
+ - == || && > < * / %
```

Výpis 5.2: Zoznam lexikálnych kategórií

Detekcia typov

Nakoľko sa v navrhnutom jazyku nachádza mechanizmus definície uživateľských typov, nie je možné fixne definovať rozlíšenie medzi identifikátorom a lexémom, reprezentujúcim názov typu.

Riešením tohoto problému je využiť metódu `hasType(std::string)` triedy `TypeRegister` na detekciu existujúceho typu. Pri detekovaní identifikátoru je najprv overené, či sa zodpovedajúci reťazec nenachádza medzi typmi. V prípade, že reťazec nie je typom, je zaradený ako identifikátor.

Modulárnosť

Táto modulárnosť je zabezpečená dočasnou zmenou výpočtu lexikálnej analýzy, kde prítomnosť špeciálneho reťazca `#include` spôsobí uloženie aktuálneho textového buffera na zásobník vstupov a prepne načítanie vstupu na vkladaný súbor.

Systém zároveň zabraňuje cyklickým závislostiam. V špecialnej množine sú počas prekladu uložené názvy súborov, ktoré už boli načítané a týmto spôsobom je možné detekovať opakované vloženie súboru a ošetriť tak potenciálny problém s redefiníciami.

5.4.2 Syntaktická analýza

Štruktúru zdrojového kódu je možné rozdeliť na deklaráciu typov, a definíciu pravidiel a pomocných funkcií.

Deklarácia typu

Deklarácia nového uživateľského typu je realizovaná ako definovanie štruktúry. Nová štruktúra má povinne aspoň jeden členský typ. Tvar konštrukcie pre deklarovanie štruktúry je vypísaný na výpisku 5.3.

```
using <TYPENAME> = struct <TYPE> {
        <TYPE1> <NAME1>,
        ...
        <TYPEN> <NAMEN>

};
```

Výpis 5.3: Syntax deklarácie štruktúry

Definícia pravidla

Pravidlá sa definujú pre konkrétne symboly (označené ako `<TYPE>`). Definícia pravidla je rozdelená do dvoch blokov - predikátu a procedúry. Predikát je funkcia, ktorej návratová hodnota určuje, či je pravidlo splniteľné. Návratová hodnota je teda typu *bool*. Definícia pravidla je ilustrovaná na výpisku 5.4.

```
using <RULENAME> = rule <TYPE> {
    // rule predicate
    return IS-RULE-VALID;
} {
    // rule procedure
};
```

Výpis 5.4: Syntax definície pravidla

Explicitné pretypovanie

Pre pretypovanie hodnoty je v jazyku vytvorená špeciálna konštrukcia, ktorá umožňuje špecifikovať ľubovoľný atomický typ. Príklad konštrukcie je na výpise 5.5.

```
// (newType) (expression)
int i = (int) (133.7);
```

Výpis 5.5: Príklad explicitného pretypovania

Dynamické pretypovanie

Pri práci s typom `any` nie je možné použiť *explicitné pretypovanie*, pretože zdrojový typ premennej nie je známy v dobe prekladu programu. Z tohoto dôvodu existuje špeciálna konštrukcia pre dynamické pretypovanie. V prípade, že v dobe behu výraz nemá špecifikovanú hodnotu nastane *vyvolanie výnimky* v interprete a ukončenie výpočtu procedurálneho programu. Využitie konštrukcie je znázornené na príklade 5.6.

```
// OK
any randomNumber = 5;
int i = convert<int>(randomNumber);

// Exception during run-time
any randomNumber = true;
int i = convert<int>(randomNumber);
```

Výpis 5.6: Príklad explicitného pretypovania

Získanie typu

Zistenie typu výrazu je možné v dobe behu programu bezpečne previesť pomocou špeciálnej konštrukcie. Tvar konštrukcie má dve podoby – výrazovú a typovú. Výrazová konštrukcia je použiteľná na získanie identifikátora typu výrazu, kdežto typová získa identifikátor konkrétneho typu. Obe varianty sú predstavené vo výpise 5.7.

```
if(typeid<float> == typeid(1.0))
{
    // it's float
}
```

Výpis 5.7: Príklad získania typového identifikátoru

Prístup ku štrukturovaným typom

Pre prístup ku jednotlivým členom štruktúry je definovaná konštrukcia pomocou znaku bodky. Táto konštrukcia nie je aplikovateľná na kolekciu. Pre prístup ku prvkom kolekcie, ako aj vkladanie a získanie počtu prvkov kolekcie je nutné použiť preddefinované konštrukcie `at`, `insert`, `size`, ktorý použitie je znázornené na výpise 5.8.

```
using vec3 = struct {
    float x; float y; float z;
};

vec3 position;
position.x = 10.0;
position.y = position.x + 10;

collection positions;
positions.insert(position);
if(positions.size() > 0)
{
    vec3 theSamePosition = convert<vec3>(positions.at(0));
}
positions.del(0);
```

Výpis 5.8: Príklad práce so štruktúrovanými typmi

Definícia parametra

Jazyk umožňuje definovať globálne premenné, ktoré sú zároveň môžu byť použité pre parametrizáciu skriptov, ktorá je popísaná v podkapitole 4.4.3. Typ parametra nie je obmedzený, avšak voliteľná inicializácia je obmedzená na konštantný výraz. Príkladom definície parametra je výpis 5.9.

```
parameter int startNumber = 10;
parameter rgb backgroundColor;
```

Výpis 5.9: Príklad definície parametra

5.4.3 Sémantická analýza

V rámci sémantickej analýzy jazyka je zabezpečené overenie typovej kompability. Zároveň sú definované rôzne vstavané funkcie, ktoré umožňujú ovládať priebeh generovania alebo zjednodušujú používanie jazyka.

Vstavané funkcie

- `setMaximumIterations(int N)`
  Funkcia nastaví maximálny počet prepisovacích cyklov. Implicitne je tento počet neobmedzený.
- `setRandomSeed(float seed)`
  Funkcia nastaví počiatočnú hodnotu pseudonáhodných generátorov (seed).
- `random()`
  Funkcia, ktorá slúži k výpočtu rovnomerného rozloženia, normalizovaného na interval $< 0, 1 >$
- `uniform(float low, float high)`
  Funkcia, ktorá slúži k výpočtu rovnomerného rozloženia pre interval $< low, high >$
- `sin(float radians)`
  Funkcia, ktorá slúži k výpočtu $sin(x)$, kde x je uhol v radianoch.
- `cos(float radians)`
  Funkcia, ktorá slúži k výpočtu $cos(x)$, kde x je uhol v radianoch.

Navrhovaný jazyk obsahuje mechanizmus pre prácu s kontextom. Každý symbol obsahuje odkazy na symbol, z ktorého vznikol, ako aj symboly, ktoré sa vyskytli pri prepisovaní v jeho okolí (kontext). Pre prístup k týmto symbolom sú dostupné nasledujúce vstavané funkcie:

- `getCurrentPosition()`
  Funkcia získa pozíciu (identifikátor) práve derivovaného symbolu.
- `getCurrentStringId()`
  Funkcia získa číslo aktuálneho reťazca, ktoré zodpovedá číslu iterácie.
- `bool hasSymbol(stringId, symbolPosition)`
  Funkcia vráti hodnotu `true` ak existuje symbol v reťazci `stringId` na pozícii `symbolPosition`.
- `any getSymbol(stringId, symbolPosition)`
  Funkcia vráti štruktúru v reťazci `stringId` na pozícii `symbolPosition`. Ak štruktúra neexistuje, je chovanie nedefinované.
- `getParent(stringId, symbolPosition)`
  Funkcia pre daný symbol vráti pozíciu symbolu v reťazci `stringId -1`. Ak predchodca neexistuje, potom funkcia vráti hodnotu `-1`.

- `skipSymbol(int n)`
  Funkcia spôsobí preskočenie `n` symbolov v aktuálnom prepisovacom cykle.

5.5 Knižničný systém

Implementácia jazyka je šírená ako knižnica s definovaným rozhraním, ktorú je možné ľahko vložiť do projektu. V tejto časti kapitoly je popísané rozhranie knižnice a detaily spojené s prekladom a používaním.

5.5.1 Rozhranie knižnice

Navonok je vnútorné fungovanie knižnice zapúzdrené pomocou triedy `ProcGen`, ktorá túto knižnicu sprístupňuje.

Vnútorná implementácia jazyka je pre uživateľa knižnice zakrytá nakoľko knižnica komunikuje pomocou serializačného formátu `JSON`. Dôvodom je jednoduchosť a zároveň široká podpora toho formátu vo forme slobodných implementácií knižníc realizujúcich serializáciu a deserializáciu.

Trieda `ProcGen` poskytuje nasledujúce metódy:

- `bool parseFile(const std::string& file);`
  Metóda spustí načítanie skriptu zo súboru `file` a preklad. V prípade kompilačnej chyby je chyba vypísaná na štandardný chybový výstup a návratová hodnota metódy zodpovedá `false`.

- `bool runInit();`
  Spôsobí zavolanie funkcie `init()` v skripte. V prípade chýbajúcej funkcie alebo chyby je návratová hodnota `false`.

- `bool run();`
  Spustí prepisovací výpočet. Počet prepisovacích cyklov je implicitne neobmedzený a toto nastavenie je možné ovplyvniť pomocou vstavanej funkcie `setMaximumIterations`. Metóda vráti `false` v prípade, že počas behu nastala jedna z chýb: nedostatok pamäte, neplatná konverzia pomocou `convert<>` a iné.

- `json serialize() const;`
  Vráti JSON objekt, reprezentujúci finálny (aktuálny) reťazec prepisovacieho systému. Každá štruktúra obsahuje špeciálny atribút `_type`, ktorý zodpovedá štruktúre, definovanej v skripte.

- `bool setUniform(std::string uniformName, T value)`
  Táto metóda umožňuje nastaviť hodnotu parametra `uniformName`. Metóda podporuje len atomické typy `float`, `int` a `bool`.

- `bool getUniform(std::string uniformName, T* value)`
  Analogicky ku `setUniform()`, metóda umožňuje získať hodnotu parametra `uniformName`.

- `void reinitialize();`
  Vymazanie aktuálneho derivačného stromu a znovu spustenie inicializačnej funkcie `init()`. Týmto spôsobom je možné viacnásobné generovanie pomocou toho istého objektu a skriptu.

- `bool appendSymbol(json symbol);`
  Pomocou tejto metódy je možné pridať spracovať symbol zapísaný v jazyku JSON do reťazca symbolov prepisovacieho mechanizmu. V prípade, že poskytnutý symbol nie je možné vyjadriť štruktúrami, definovanými v skripte, vráti metóda hodnotu `false`.
- `bool hasAnyErrorMessage()`
  Metóda vráti `true` v prípade, že po spustení metódy `run()` nastala behová chyba.
- `const std::string& getLastErrorMessage() const;`

5.5.2 Prekladový systém CMake

Kompilácia jazyka C/C++ je z pohľadu prenositeľnosti relatívne komplikovaný proces nakoľko neexistuje jeden rozšírený prekladač, ktorý by bol masívne používaný na všetkých platformách ako je to napr. v jazyku Java.

Pre dosiahnutie jednoduchšej prenositeľnosti je preto implementácia knižnice vybavená konfiguráciou systému `CMake`, ktorý umožňuje z jednotnej konfigurácie projektu generovať platformne závislé prekladové skripty.

Systém CMake umožňuje napríklad generovať Makefile na systémoch rodiny `*nix` alebo generovať projekt pre program `Microsoft Visual Studio C++` na platforme Windows.

5.5.3 Dokumentácia a distribúcia knižnice

Vytvorená knižnica je zverejnená na verejnom repozitári portálu `GitHub` [1]. Súčasťou repozitára je dokumentácia knižnice, zapísaná vo formáte `Markdown`, obsahujúca referenciu k jazyku ProcGen, jednoduchý tutoriál, úplnú gramatiku jazyka a návod na kompiláciu.

Samotný kód je zdokumentovaný pomocou metaznačiek nástroja `Doxygen`.

5.6 Vyhliadky do budúcna

Nasledujúca časť textu popisuje prípadné zefektívnenie prevedenia jazyka.

5.6.1 Vizualizátor derivačného stromu jazyka

Nakoľko jazyk kombinuje imperatívny štýl programovania s teóriou prepisovacích systémov, nemusí byť jeho činnosť zrejmá ani programátorom s imperatívnou skúsenosťou ale nedostatkom formálnych znalostí.

Z tohoto dôvodu by bol vytvorený grafický program, ktorého rozhranie pararelne ukazuje kód programu a na druhej strane reprezentáciu symbolov, ktoré postupne vznikajú v pamäti vykonávaním jednotlívych príkazov programu.

Program by zároveň bolo možné použiť na vizualizáciu rastu jednoduchých L-systémov.

5.6.2 Pokročilá sémantická analýza

- **Upozornenie na pravidlo, ktoré nepridáva žiaden symbol**
  V prípade, že pravidlo pre daný symbol nepridáva žiaden nový symbol do ďalšieho kroku, spôsobuje pravidlo strátu doterajšieho symbolu a teda efektívne ruší účinok generovania. V istých prípadoch môže byť takéto rozhodnutie žiadúce (napr. neskorá selekcia typu generovania), avšak obecne je takéto chovanie možno považovať za chybu.

[1] https://github.com/Romop5/procgen

5.6.3 Optimalizácia

- **Obmedzenie pamäťových presunov** Pridanie kľúčového slova `const` pre formálny parameter funkcie, ktorý by umožnil predávanie referencie namiesto kopírovania argumentu.

- **Optimalizácia prepisovania symbolov**
  V prípade, že predikátová funkcia je deterministicky závislá výhradne na hodnote symbolu, je možné vynechať opätovné overenie platnosti pre nezmenený symbol.

- **Pararelné generovanie**
  Symboly je možné generovať pararelne a v prípade agregácie (preskočenia následníka generovaného symbolu) odstrániť nadbytočné symboly.

- **Optimalizácia číselných výrazov**
  Nahradenie konštantných výrazov literálom.

5.6.4 Bindingy pre jazyky

Referenčná implementácia je naprogramovaná v jazyku `C/C++`. Tieto jazyky ale dnes už nie sú výhradne dominantné, naopak čoraz častejšie sa preferuje písanie krátkych, nesystémových programov v skriptovacích jazykoch akými sú `Python`, `PHP`, `JavaScript` alebo v moderných programovacích jazykoch typu `Go`, `D` alebo `Rust`.

Pre využitie knižnice v týchto jazykoch by bolo nutné vytvoriť programovú medzivrstvu (angl. *binding*).

Kapitola 6

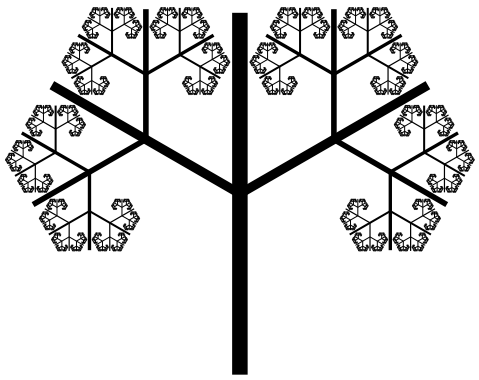
Príklady využitia knižnice

V rámci tejto kapitoly sú popísané príklady využitia knižnice, ktoré boli vypracované spoločne s touto prácou.

6.1 SVG Creator

Cieľom programu SVG Creator je umožniť uživateľovi vytvárať vektorové obrázky na základe vkladania grafických primitív, ktoré sú definované formátom SVG. Proces vytvorenia obrazu teda pozostáva v sekvenčnom spracovaní symbolov, ktoré vytvoril uživateľom definovaný skript a ich interpretáciou.

Symboly, ktoré rozpoznáva program, ako aj ich spracovanie je pevnou súčasťou kódu programu. Zlepšenie uživateľského komfortu zabezpečuje štandardný skript `svg.procgen`, ktorý definuje SVG štruktúry v procedurálnom jazyku.



Obr. 6.1: Obraz stromu, vytvorený pomocou programu SVG Creator, bez použitia korytnačej grafiky. Pozíciu, farbu a šírku čiar definuje uživateľ priamo pri vytvorení štruktúry, ktorá ich reprezentuje.

```
1 #include "svg.procgen"
2
3 parameter float ANGLE = 10.0;
```

```
parameter float RATE = 0.60;

using seed = struct { point start;  point way; float thickness; };
//vzdy platne pravidlo, ktore prepise aktualny konar na dva nove
using transform = rule seed { return true; }
{
   float PI = 3.1415;
   collection STYLE = collection(strokeWidthStyle(this.thickness),
                     strokeColorStyle(REDCOLOR));
   // pridaj ciaru, ktora bude reprezentovat aktualny konar
   appendSymbol(line(this.start.x, this.start.y, this.start.x+this.way.x,
         this.start.y+this.way.y, STYLE));
   // vypocet bodu, ktory je v~strede aktualneho konaru a skratenie noveho smeroveho vektoru
   point middlePoint = point(this.start.x+(this.way.x)/2.0,
         this.start.y+(this.way.y)/2.0);
   this.way.x = this.way.x*RATE; this.way.y = this.way.y*RATE;

   // pridanie dvoch novych konarov, rotovanych o~uhol ANGLE
   point leftVector = point(cos(PI*ANGLE/180.0)*this.way.x - sin(PI*ANGLE/180.0)*this.way.y,
                        cos(PI*ANGLE/180.0)*this.way.y + sin(PI*ANGLE/180.0)*this.way.x);
   point rightVector = point(cos(PI*ANGLE/180.0)*this.way.x + sin(PI*ANGLE/180.0)*this.way.y,
                         cos(PI*ANGLE/180.0)*this.way.y - sin(PI*ANGLE/180.0)*this.way.x);

   appendSymbol(seed(middlePoint, leftVector,this.thickness*RATE));
   appendSymbol(seed(middlePoint, rightVector, this.thickness*RATE));
};
int init() {
   // pociatok (koren) stromu, na pozicii 500.0, 900.0, so smerom rastu Y = -700
   appendSymbol(seed(point(500.0,900.0), point(0.0,-700.0),30.0));
   // nastavenie maximalneho poctu prepisovacich cyklov na 7
   setMaximumIterations(7);
}
```

Výpis 6.1: Zdrojový kód pre obrázok 6.1

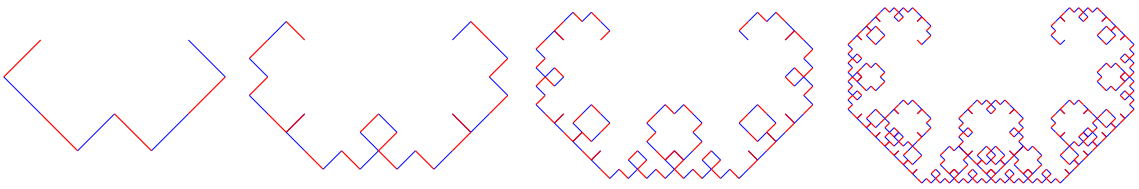
Vo formáte SVG majú grafické primitíva pevný tvar, čo umožňuje elegantne mapovať ich XML štruktúru na štruktúry v procedurálnom jazyku. Problém nastáva pri štylizovaní týchto primitív, pretože výraz v jazyku CSS, ktorý je k tomuto účelu v SVG využitý, nemá fixnú podobu a umožňuje variabilný počet príkazov. Tento problém je elegantne riešený zavedení voliteľnéhej kolekcie `style`, ktorá môže obsahovať inštancie jednotlivých CSS príkazov, ktoré už sami o sebe sú fixné. Príkazy sú podobne ako primitíva mapované na uživateľskú štruktúru v procedurálnom jazyku.

Príklad generovaného obrázku týmto spôsobom môžeme vidieť na obrázku 6.1. Výpis 6.1 predstavuje kód, ktorým bol obrázok vygenerovaný.

6.1.1 Tvorba animovaných GIF obrázkov

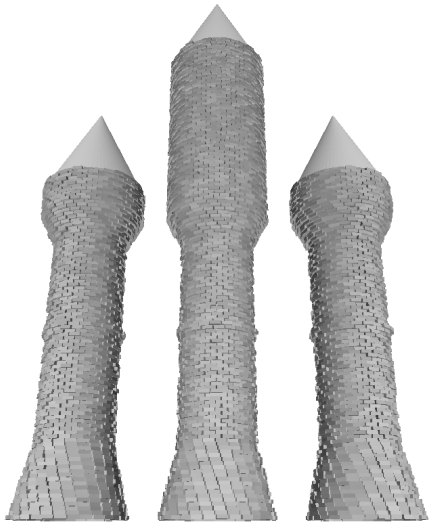
Výstupom `SVGCreatora` je jediný obrázok vo formáte SVG. Výsledný obraz nemusí závisieť len na samotnom skripte, ale vďaka mechanizmu parametrov jazyka `ProcGen` je možné predať do generovacieho scriptu hodnoty priamo z programu, prípadne uživateľa. Tento princíp je možné využiť pri generovaní animovaných obrazov – definuje sa parameter, ktorý zodpovedá číslu snímku animovaného obrázku. Následne je tento parameter využitý pre ovplyvnenie chodu generovania. Napríklad, číslo snímky môže indikovať maximálny počet iterácií prepisovacieho systému a takto je možné získať animáciu rastu systému. Podobne je možné ovplyvniť iné parametre, od uhlu rastu až po farbu primitív.

Obrázok 6.2 znázorňuje jednotlivé obrázky, z ktorých vznikne animovaný GIF obrázok.



Obr. 6.2: Jednotlivé obrazy, ktorých spojením vznikne pohyblivý obrázok vo formáte GIF. Každý obrázok bol vygenerovaný tým istým skriptom, ktorého parametrom bol počet iterácií.

6.2 Vizualizácia 3D modelov pomocou knižnice OpenGL



Obr. 6.3: Veža, vytvorená pomocou rotovania tehiel a stochastického posunutia tehiel pre efekt nerovnosti. Tvar veže je definovaných krivkou v skripte.

Pre vizualizáciu 3D modelov bol vytvorený jednoduchý vykreslovací engine pre knižnicu `OpenGL`, založený na ukladaní objektov do grafu scény (*ang. scene graph*). Pre zabezpečenie prenositeľnosti pri vytváraní grafického okna v operačných systémov bola využitá knižica `SDL`. Nakoniec, pre komfortné ovládanie programu bola použitá voľne dostupná grafická knižnica `Dear ImGUI`.

Vytvorený program umožňuje vykreslovať jednotlivé skripty (program `viewer`) alebo zobraziť zoznam demo skriptov a dynamicky ich načítať (program `demo`).

Zaujímavý príklad prestavuje generovanie veže s tematikou stredoveku, ktorá je zobrazená na obrázku 6.3. Veža je generovaná pridaním jedného symbolu typu `tower`, s počiatočnou pozíciou veže v priestore, maximálnym polomerom veže a kolekciou bodov, ktoré definujú krivku. Tento spôsob generovania by v prípade prepojenia na uživateľské rozhranie umožnil generovať grafický objekt podľa predstavy umelca pomocou explicitného stanovenia požadovaného tvaru. Následne by boli body krivky predané do aplikácie pomocou JSON importu.

Výsledok generovania veže je znázornený na obrázku 6.3.

Obr. 6.4: Jednoduchý terén s hradom a stromami. Každý strom je unikátne generovaný. Hrad je postavený zo 4 veží a 4 hradieb.

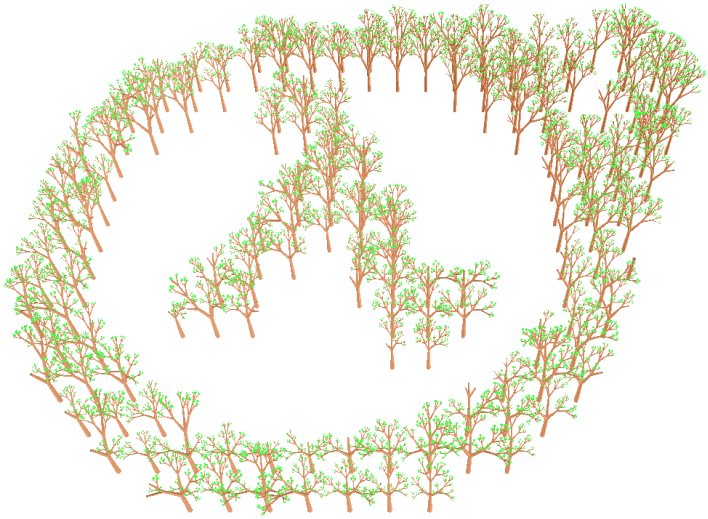
Rozšírením generovania veže o hradby a stromy vznikla jednoduchá scéna, ktorá je znázornená na obrázku 6.4.

6.2.1 Import textúry do jazyka ProcGen

Pre demonštrovanie možností práce s importovaním JSON reťazcov do jazyka ProcGen bol upravený program pre vykreslenie generovanej scény. Program pred spustením generovania načíta bitmapu zo súboru a jednotlivé body (pixeli) serializuje do JSON reťazca, ktoré forma zodpovedá preddefinovanej štruktúre pre 2D textúru zo súbora `stdtexture.gen`. Skript, ktorý ďalej pracuje s textúrou, definuje pravidlo pre symbol textúry, ktoré následne môže spracovať importovanú textúru a vytvoriť ďalšie pomocné štruktúry. Týmto spôsobom je napríklad možné simulovať rast lesa, ktorého výška bude obmedzená hodnotami bitmapy, ako je možné vidieť na obrázku 6.5.

6.3 Algoritmus BSP

Vďaka podpore obecných príkazov a štruktúr je možné pomocou implementovaného procedurálneho jazyka definovať aj obecné algoritmy, ktoré nevychádzajú z L-systémov. Príkladom je metóda BSP, popísaná v podkapitole 2.2.4. Jej implementácia využíva prístup k derivačnému stromu ku spojeniu vzniknutých uzlov spätne do jednej štruktúry. Výsledkom behu algoritmu je jediná štruktúra, ktorá obsahuje kolekciu oblastí a ciest medzi oblasťami. Následne je možné použiť tieto oblasti k ďalšiemu generovaniu, napríklad ku vytvoreniu budov. Príklad behu algoritmu je vizualizovaný na obrázku 6.6.



Obr. 6.5: Les v tvare loga, ktorého rast bol ovplyvnený výškovou mapou, ktorá vznikla importovaním textúry z formátu BMP do jazyka skriptu pomocou serializačného formátu JSON.

Obr. 6.6: Ukážka algoritmu BSP, implementovanom v procedurálnom jazyku. Jednotlivé oblasti vznikajú iteratívne, ako samostatné symboly. Nakoniec sú vždy dvojice oblastí spojené pomocou agregácie a výsledkom algoritmu je štruktúra s kolekciou oblastí a ciest medzi nimi.

Kapitola 7

Záver

Vrámci tejto práce boli popísané obvyklé metódy procedurálneho generovania. Na základe jednej z techník bol postavený nový jazyk za účelom obecnosti, dynamickosti a komfortu. Spomenuté vlastnosti boli dosiahnuté špecifickými jazykovými konštrukciami vhodnými pre generovanie, naprogramovaním generovacieho systému v času spustenia a nutnými aj rozšírenými konštrukciami jazyka, ktoré umožňujú obecné operácie so symbolmi.

Okrem návrhu bola vytvorená referenčná, prenositeľná implementácia jazyka, dekomponovaná do trojvrstvovej architektúry.

Interpret bol navrhnutý ako typovaný objektovo-orientovaný systém, ktorý obecne umožňuje realizovať obecný výpočet založený na selekcii, sekvencii a iterácii. Umožňuje definovať zložitejšie typy a funkcie z existujúcich, prípadne atomických častí. Vrámci interprétu sú definované základné typy a operácie nad nimi. *Derivačná časť* realizuje prepisovací systém podobný L-systémom s podporou pre prístup ku histórii prepisovania a kontextu prepisovaného symbolu.

Nakoniec *prekladač* zabezpečuje spracovanie vstupného textového jazyka na štruktúry interprétu a príkazy pre derivačný modul. Z dôvodu zlepšenia udržateľnosti kódu využité voľne dostupné knižnice `Flex` a `Bison`, ktorý umožňujú generovať prekladač z gramatiky v `BNF`. Zároveň bola navrhnutá gramatika jazyka, ktorá má konštrukcie podobné jazyku C++, čo umožňuje ľahšie zorientovanie s jazykom.

Ako aplikačné rozhranie knižnice bola navrhnutá trieda, ktorá oddeluje spomenuté moduly od vonkajšieho sveta. Trieda umožňuje načítať súbory so skriptami, spúšťať generovací proces, načítať serializovaný vstup do knižnice ako aj získať serializovaný výstup štruktúr. Pre komfortnú prácu s knižnicou bol zvolený kompaktný serializačný jazyk `JSON`, pre ktorý existuje mnoho implementácii v rôznych jazykoch.

Výsledkom práce je implementovaná knižnica v jazyku `C/C++` s jazykom pre procedurálne generovanie. Súčasťou práce sú aj príklady využitia knižnice pri praktickom generovaní. Jedným z príkladov je program *generujúci obrázky vo formáte* `SVG`. Druhým príkladom využitia je aplikácia využívajúca knižnicu `OpenGL`, ktorá umožňuje uživateľovi načítať skripty, generujúce trojrozmerné modely a následne prezerať tieto modely s rôznymi parametrami generovania.

V práci je taktiež načrtnutý potencionálny vývoj jazyka, ktorý by smeroval ku optimalizácii interprétu a prípadnej edukačnej aplikácii pre vizualizáciu činnosti jazyka.

Literatúra

[1] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, & Tools with Gradiance*. USA: Addison-Wesley Publishing Company, druhé vydanie, 2007, ISBN 0321547985, 9780321547989.

[2] Aurenhammer, F.: Voronoi Diagrams&Mdash;a Survey of a Fundamental Geometric Data Structure. *ACM Comput. Surv.*, ročník 23, č. 3, September 1991: s. 345–405, ISSN 0360-0300, doi:10.1145/116873.116880.
URL http://doi.acm.org/10.1145/116873.116880

[3] Donnelly, C.; Stallman, R. M.: *Bison - The Yacc-compatible Parser Generator*. Free Software Foundation, 2015, ISBN 1882114442, bison Version 3.0.4.
URL https://www.gnu.org/software/bison/manual/bison.pdf

[4] Fišer, M.: *L-systems online*. Bakalárska práca, Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra softwaru a výuky informatiky, Praha, 2012.

[5] Fuller, A. R.; Krishnan, H.; Mahrous, K.; aj.: Real-time Procedural Volumetric Fire. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-628-8, s. 175–180, doi:10.1145/1230100.1230131.
URL http://doi.acm.org/10.1145/1230100.1230131

[6] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN 0-201-63361-2.

[7] Glass, K. R.; Morkel, C.; Bangay, S. D.: Duplicating Road Patterns in South African Informal Settlements Using Procedural Techniques. In *Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-288-7, s. 161–169, doi:10.1145/1108590.1108616.
URL http://doi.acm.org/10.1145/1108590.1108616

[8] Green, D.: *Procedural Content Generation for C++ Game Development*. Community experience distilled, Packt Publishing, Limited, 2016, ISBN 9781785886713.
URL https://books.google.sk/books?id=7PAvjwEACAAJ

[9] Gustavson, S.: Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report*, 2005.

[10] Hanan, J. S.: *Parametric L-systems and Their Application to the Modelling and Visualization of Plants*. Dizertačná práca, The University of Regina (Canada), 1992, aAINN83871.

[11] Jäger, G.; Rogers, J.: Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, ročník 367, č. 1598, 2012: s. 1956–1970, ISSN 0962-8436, doi:10.1098/rstb.2012.0077.

[12] Johnson, L.; Yannakakis, G. N.; Togelius, J.: Cellular Automata for Real-time Generation of Infinite Cave Levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, New York, NY, USA: ACM, 2010, ISBN 978-1-4503-0023-0, s. 10:1–10:4, doi:10.1145/1814256.1814266.
URL http://doi.acm.org/10.1145/1814256.1814266

[13] Levine, J.; John, L.: *Flex & Bison*. O'Reilly Media, Inc., prvé vydanie, 2009, ISBN 0596155972, 9780596155971.

[14] Linz, P.: *An Introduction to Formal Languages and Automata*. Sudbury: Jones, 6 vydanie, 2016, ISBN 978-1-284-07724-7.

[15] Parish, Y. I. H.; Müller, P.: Procedural Modeling of Cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, New York, NY, USA: ACM, 2001, ISBN 1-58113-374-X, s. 301–308, doi:10.1145/383259.383292.
URL http://doi.acm.org/10.1145/383259.383292

[16] Pirk, S.; Stava, O.; Kratt, J.; aj.: Plastic Trees: Interactive Self-adapting Botanical Tree Models. *ACM Trans. Graph.*, ročník 31, č. 4, Júl 2012: s. 50:1–50:10, ISSN 0730-0301, doi:10.1145/2185520.2185546.
URL http://doi.acm.org/10.1145/2185520.2185546

[17] Prusinkiewicz, P.; Lindenmayer, A.: *The algorithmic beauty of plants*. Springer, 1996, ISBN 978-0-387-94676-4.

[18] Shaker, N.; Togelius, J.; Nelson, M. J.: *Procedural Content Generation in Games*. Computational Synthesis and Creative Systems, Springer, 2016, ISBN 978-3-319-42714-0, doi:10.1007/978-3-319-42716-4.
URL https://doi.org/10.1007/978-3-319-42716-4

[19] Wolfram, S.: Universality and complexity in cellular automata. *Physica D: Nonlinear Phenomena*, ročník 10, č. 1-2, 1984: s. 1–35.

Príloha A

Obsah přiloženého paměťového média

Priložený CD nosič obsahuje nasledujúcu súborovú štruktúru:

- **bachelorthesis**
  Repozitár, obsahujúci túto správu v súboroch formátu LaTeX. Dokument je dostupný po preložení príkazom `make` pod názvom `projekt.pdf`.
- **procgen**
  Repozitár knižnice ProcGen. Obsahuje dokumentáciu, zdrojové kódy, unit testy.
- **procgen-svg**
  Repozitár praktickej aplikácie SVGCreator, popísanej v kapitole 6. V priečinku `build` obsahuje príklady generovania SVG obrázkov vo forme skriptu `.procgen` a výsledného obrázku `.svg`.
- **noob-engine**
  Repozitár jednoduchého 3D vykreslovacieho engine-u s integrovanou ProcGen knižnicou. V priečinku `build` obsahuje príklady skriptov pre 3D modely. Zároveň je v priečinku štandardný skript `stdlinear.gen`, ktorý obsahuje definované operácie pre prácu s trojrozmerným vektorom, násobenie 3x3 matíc, rotáciupriestoru, výpočet normálového vektoru a podobne.
- **video**
  Demonstračné video, ukazujúce dosiahnuté výsledky.
- **libraries**
  Kolekcia knižníc pre OS Windows, potrebná pre kompiláciu projektov.

Príloha B

Manuál knižnice ProcGen

B.1 Kompilácia

Pred kompiláciou knižnice `ProcGen` je nutné nainštalovať do systému závislosti, konkrétne knižnicu `Bison` a `Flex`. V prípade kompilácie na systéme Microsoft Windows sú dané knižnice dostupné pod názvom `winbison` a `winflex` na priloženom CD nosiči a cestu ku ním je nutné manuálne nastaviť v premenných prostredia CMake.

Kompiláciu je možné vykonať nasledujúcimi príkazmi v priečinku `/procgen/build`:

```
$ cmake ../
$ make
```

Po úspešnej kompilácii je možné knižnicu nainštalovať do systému pomocou `make install`.

B.2 Dokumentácia knižnice ProcGen

Jednotlivé zdrojové súbory knižnice sú zdokumentované pomocou knižnice `Doxygen`. Pre vygenerovanie uživateľskej knižnice je nutné v prienčinku `/procgen/` spustiť nasledujúci príkaz:

```
$ doxygen Doxygen
```

Výsledkom generovanie je HTML stránka, dostupná v priečinku `/doxygen/html/`. Stránka obsahuje nasledujúce štruktúry:

- **kompilácia**
  Odkazy na kompiláciu pod systémom Windows či Linux.
- **tutorial**
  Jednoduchý tutoriál, popisujúci fungovanie knižnice a generovanie na praktickom príklade.
- **referenčnú príručku jazyka**
  Kompletný prehlad konštrukcií.

Príloha C

Manuál programu SVGCreator

C.1 Kompilácia

Kompilácia programu `SVGCreator` vyžaduje tie isté požiadavky ako kompilácia samotnej knižnice `ProcGen`.

Po skompilovaní sú k dispozícii dva programy:

- **svgcreator**
  Program umožňuje spustiť skript, ktorý generuje SVG primitíva a následne tieto primitíva uloží do výstupného súboru. Program tak isto umožňuje voliteľne parametrizovať toto generovanie pomocou nastavenia hodnôt parametrov. Hodnoty parametrov sú predané do programu vo forme `JSON` štruktúry.

  ```
  $ svgcreator <inputScriptPath> <outputSVGname> [ "JSON literal" ]
  ```

  Príkladom literálu môže byť `{ 'iterations': 5, 'stochasticity': 0.3}`, ktorý nastaví parametru `iterations` hodnotu 5 typu `int` a parametru `stochasticity` hodnotu 0.3 typu `float`.

- **turtle**
  Program umožňuje vygenerovať korytnačiu grafiku pomocou zátvorkového jazyka, ktorý používa preddefinované terminálne štruktúry. Pre tento program sú v zložke `/procgen-svg/build/` príklady, ktoré začínajú prefixom `turtle`.

- **growAnimation.sh**
  Jednoduchý skript v `Bash`, ktorý umožňuje vytvárať GIF obrázky vďaka parametrizácii programu `svgcreator`.

Príloha D

Manuál Noob-enginu

`Noob-engine` je jednoduchý vykreslovací engine, ktorý integruje `ProcGen` knižnicu. Pre úspešnú kompiláciu engineu je nutné mať nainštalované knižnice `GLM`, `GLEW` a `SDL2`. Predpokladom je taktiež úspešne skompilovateľná knižnica `ProcGen`.

Postup a príkazy pre kompiláciu sú identické s knižnicou `ProcGen`. Kompiláciou knižnice vzniknú nasledujúce spustiteľné programy:

viewer Viewer umožňuje prehliadať základné skripty. Spustenie programu vyžaduje argument, ktorý zodpovedá názvu skriptu.

viewer-texture Viewer umožňuje prehliadať základné skripty a skripty, ktoré začínajú prefixom `texture`. Spustenie programu taktiež vyžaduje argument, ktorý zodpovedá názvu skriptu. Po kompilácii skriptu program načíta súbor `texture.bmp` a následne predá skriptu serializované body obrázku vo formáte `JSON`.

editor Editor sa spúšťa bez argumentov príkazového riadku. Po spustení sa zobrazí užívateľské rozhranie, ktoré umožňuje spustiť existujúci skript a nastaviť rôzne parametre generovania – počet iterácii a mieru stochastickosti. Rozhranie programu je znázornené na obrázku D.1.

D.1 Ovládanie aplikácie

Aplikácia umožňuje manipulovať so scénou. K dispozícii sú nasledujúce úkony:

W Pohyb kamery vpred.

S Pohyb kamery dozadu.

A Pohyb kamery doľava.

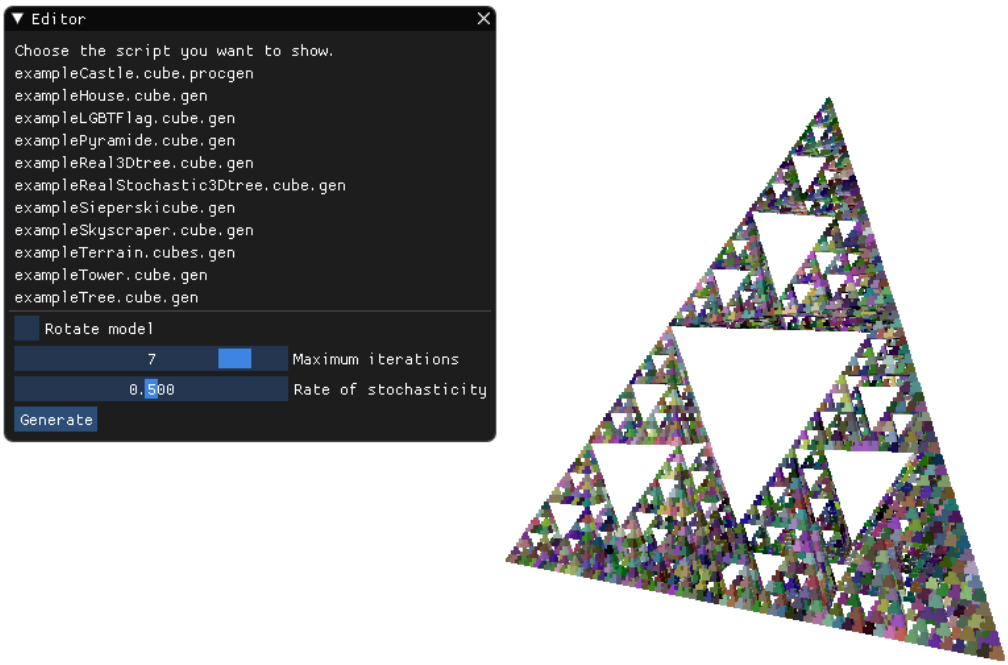
D Pohyb kamery napravo.

Up arrow Rotácia kamery smerom hore.

Down arrow Rotácia kamery smerom dole.

Left arrow Rotácia kamery doľava.

Right arrow Rotácia kamery napravo.



Obr. D.1: Rozhranie programu `editor`. Program umožňuje načítať skripty z priečinku `build`. Rozhranie programu umožňuje voliť rôzne parametre generovania.

Príloha E

BNF gramatika jazyka ProcGen

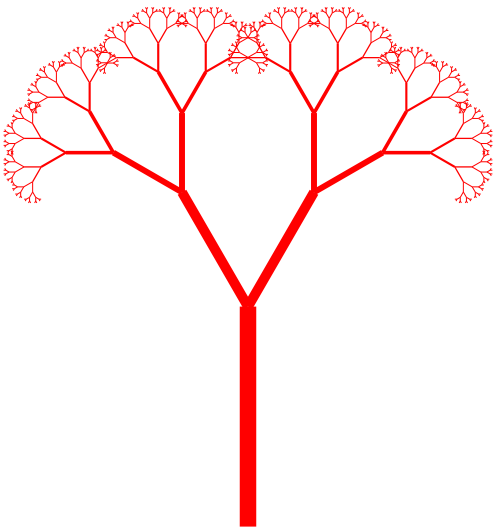
```
<program>                 ::= <declarations>
<declarations>            ::= <declaration> <declarations>
                              | <empty>
<declaration>             ::= <using-declaration>
                              | <parameter-declaration>
                              | <function-declaration>
<using-declaration>       ::= USING NAME "=" <using-variant> ";"
<using-variant>           ::= STRUCT "{" <structure-declaration> "}"
                              | RULE TYPE  <compound-statement>
<parameter-declaration>   ::= PARAMETER TYPE NAME <assign>
<assign>                  ::= "=" <literal> ";"
                              | ";"
<function-declaration>    ::= TYPE NAME  "(" <type-list> ")" <compound-statement>
<structure-declaration>   ::= <type-declaration> ";"
                              | <structure-declaration> <type-declaration> ";"
<type-list>               ::= <type-declaration>
                              | <type-list> "," <type-declaration>
                              | <empty>
<type-declaration>        ::= TYPE NAME
<compound-statement>      ::= "{" <statements> "}"
<statements>              ::= <statement> <statements>
                              | <empty>
<statement>               ::= <call-statement> ";"
                              | <declaration> ";"
                              | <assignment> ";"
                              | <if-statement>
                              | <while-statement>
                              | <return> ";"
<declaration>             ::= TYPE NAME <declaration-end>
<declaration-end>         ::= <empty>
                              |   "=" <expression>
<call-statement>          ::= <function-call>
<function-call>           ::= NAME  "(" <argument-list> ")"
<argument-list>           ::= <empty>
                              | <argument>
                              | <argument-list> "," <argument>
<argument>                ::= <expression>
<assignment>              ::= <structured-member>  <assignment-end>
<assignment-end>          ::= <empty>
                              | "=" <expression>
                              | "+=" <expression>
                              | "-=" <expression>
```

```
                                | "*=" <expression>
                                | "/=" <expression>
<if-statement>              ::= IF "(" <expression> ")" <compound-statement> <else-clause>
<else-clause>               ::= ELSE <compound-statement>
                                | <empty>
<while-statement>           ::= WHILE "(" <expression> ")" <compound-statement>
<return>                    ::= RETURN <return-end>
<return-end>                ::= <empty>
                                |   <expression>
<typeid>                    ::= "<" type ">"
                                |   "(" NAME ")"
<expression>                ::= <literal>
                                |   TYPEID <typeid>
                                |   TYPE
                                |   "(" TYPE ")" "(" <expression> ")"
                                |   CONVERT "<" type ">" "(" <expression> ")"
                                |   <function-call>
                                |   <expression> "&&" <expression>
                                |   <expression> "||" <expression>
                                |   <expression> "<" <expression>
                                |   <expression> ">" <expression>
                                |   <expression> "==" <expression>
                                |   <expression> "!=" <expression>
                                |   <expression> "-" <expression>
                                |   <expression> "+" <expression>
                                |   <expression> "/" <expression>
                                |   <expression> "*" <expression>
                                |   <expression> "%" <expression>
                                |   "(" <expression> ")"
                                |   "-" <expression>
                                |   "+" <expression>
                                |   "!" <expression>
<structured-member>         ::= <structured-member> "." <structured-member-end>
                                |    NAME
<structured-member-end>     ::= NAME
                                |    INSERT "(" <expression> ")"
                                |    AT "(" <expression> ")"
                                |    SIZE "(" ")"
                                |    DEL "(" <expression> ")"
<literal>                   ::= INTEGER
                                | FLOAT
                                | STRING
                                | BOOL
                                | <structured-member>
```

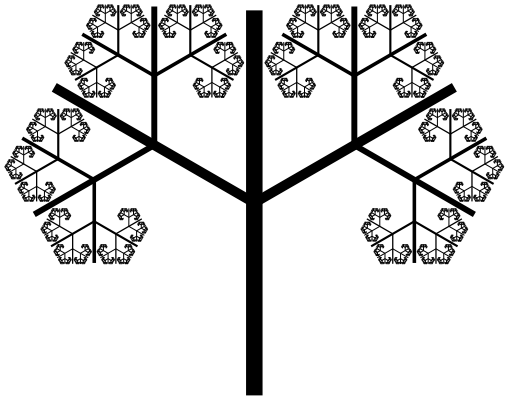
Výpis E.1: BNF gramatika

Príloha F

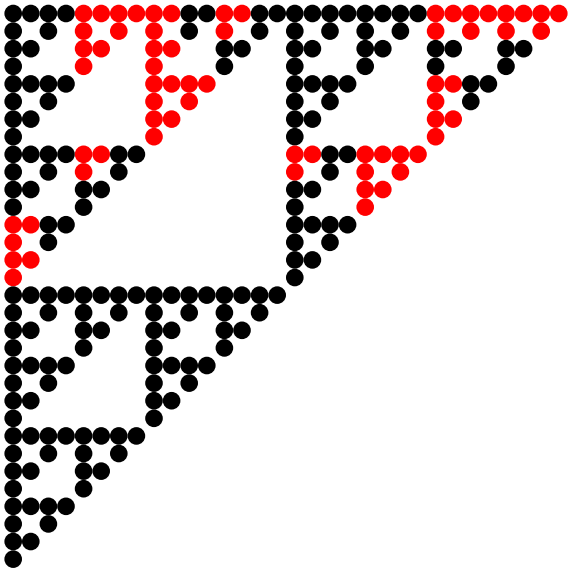
Príklady generovania



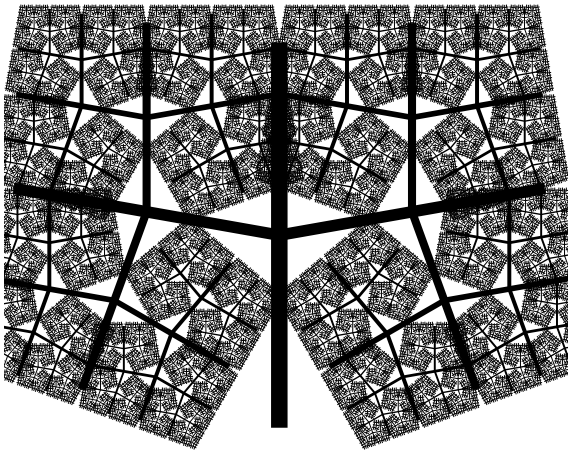
(a) prilohy-obrazky/basicTree-procgen-svg.pdf



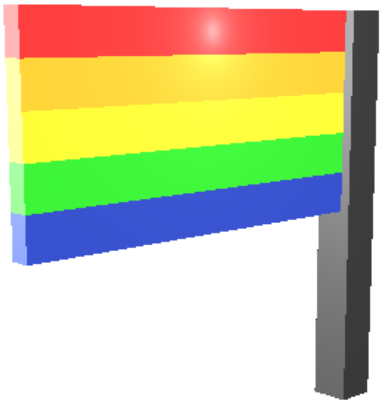
(b) prilohy-obrazky/bracket-svg.pdf



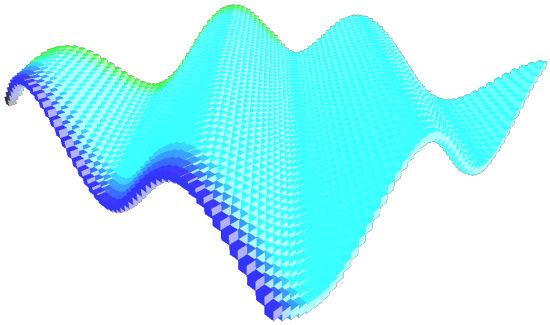
(a) prilohy-obrazky/bullet-svg.pdf



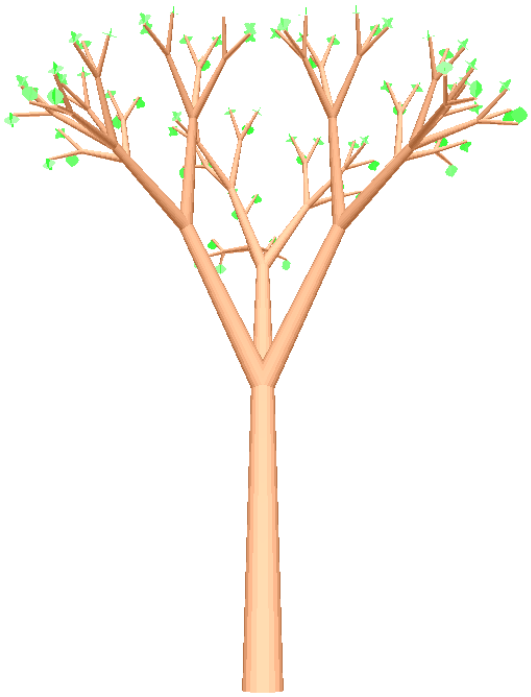
(b) prilohy-obrazky/densetree-svg.pdf



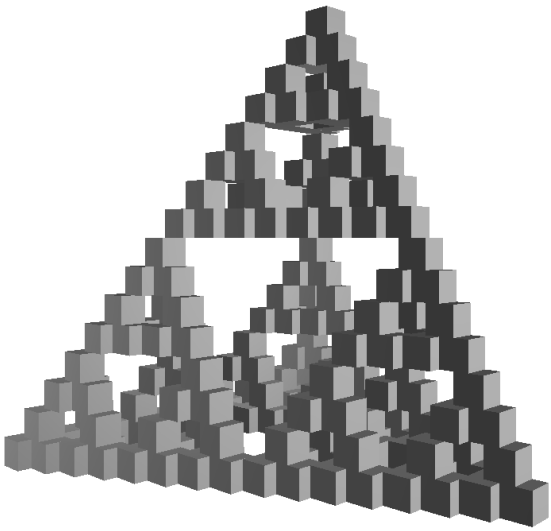
(a) prilohy-obrazky/exampleLGBT-png.pdf



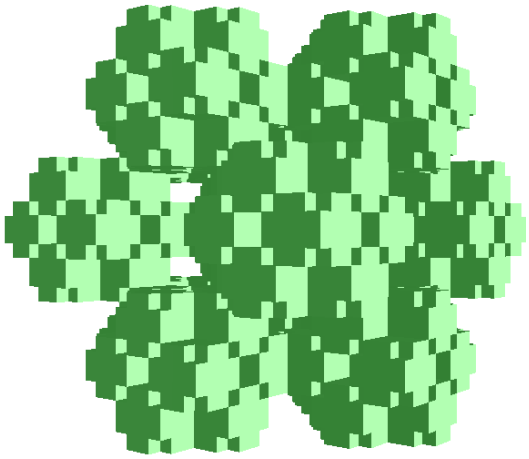
(b) prilohy-obrazky/exampleProceduralPlane-png.pdf



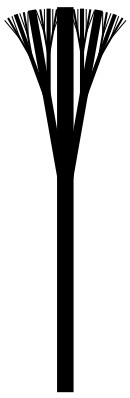
(a) prilohy-obrazky/exampleRealTree-png.pdf



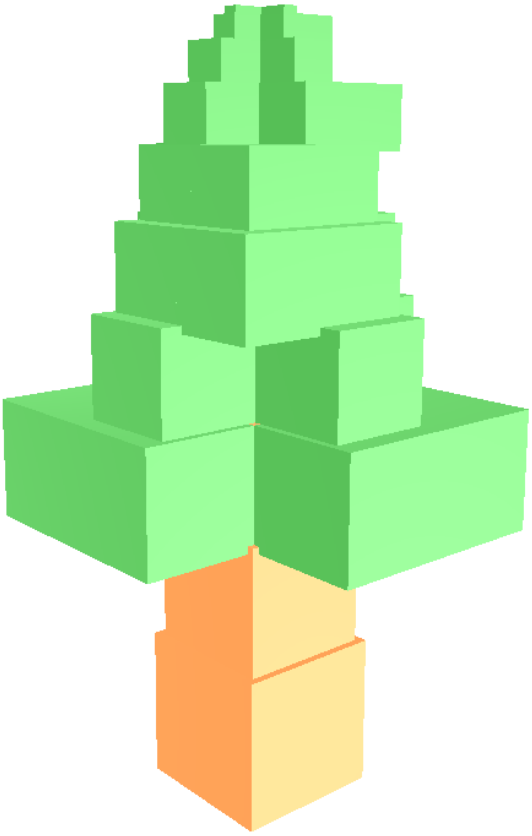
(b) prilohy-obrazky/examplePyramide-png.pdf



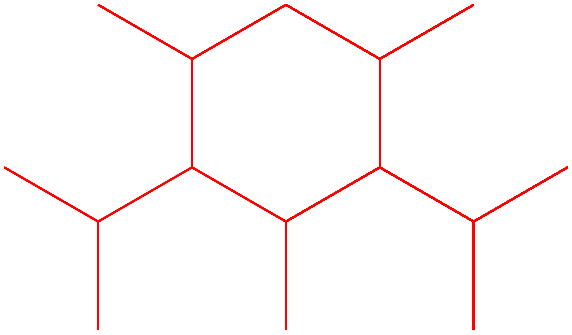
(a) prilohy-obrazky/exampleSiepiersky-png.pdf



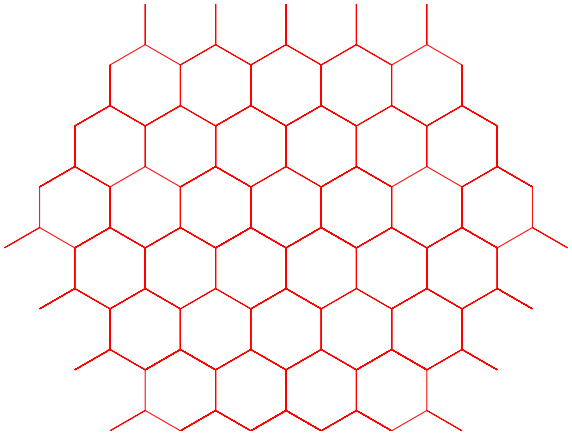
(b) prilohy-obrazky/ker-10angle-svg.pdf



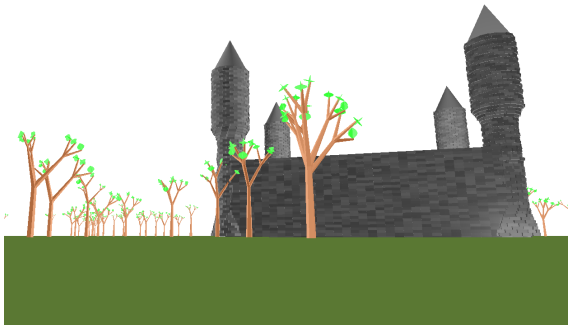
(a) prilohy-obrazky/exampleTree-png.pdf



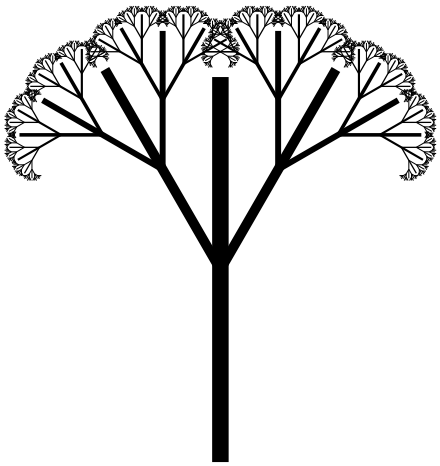
(b) prilohy-obrazky/turtle-basicTree-procgen-svg.pdf



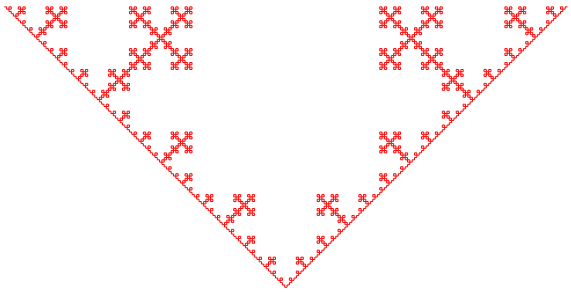
(a) prilohy-obrazky/ooo-svg.pdf



(b) prilohy-obrazky/castleBig.pdf

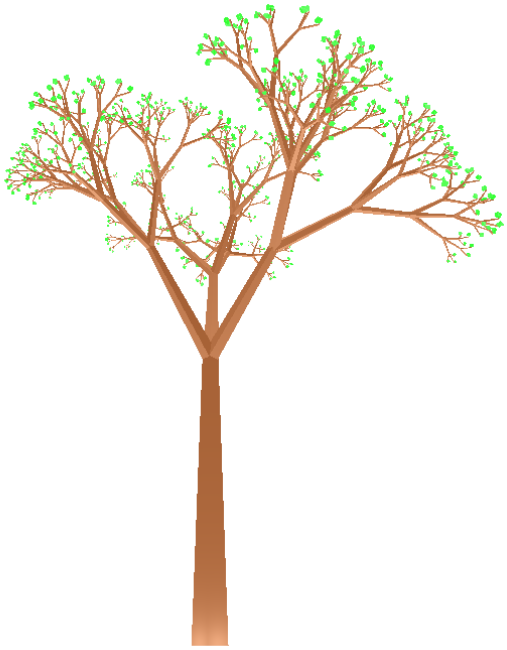


(a) prilohy-obrazky/strom-30-06-svg.pdf



(b) prilohy-obrazky/result-svg.pdf

(a) prilohy-obrazky/hlred2.pdf



(b) prilohy-obrazky/treStochastic.pdf