UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Prostredie pre podporu výučby predmetu Formálne jazyky a automaty

Bakalárska práca

Peter Havlíček

Odbor: Informatika 9.2.1
Vedúci: RNDr. Dana Pardubská, PhD.
Bratislava, 2007

Čestne prehlasujem, že túto bakalársku prácu som vypracoval samostatne len s použitím uvedenej literatúry.

V Bratislave
14. júna 2007 .....

Ďakujem svojej vedúcej bakalárskej práce RNDr. Dane Pardubskej PhD., za cenné rady a pripomienky pri jej písaní.

Abstrakt

Práca sa zaoberá návrhom a implementáciou softvéru, ktorý má slúžiť ako pomôcka v procese výučby formálnych jazykov. Softvér umožňuje vytvárať regulárne, bezkontextové, kontextové a frázové gramatiky, deterministické turingové stroje a deterministické konečné automaty. Prvky množín, použitých pri konštrukcii, je možné parametrizovať. Softvér následne umožňuje vytvárať odvodenia v skonštruovaných gramatikách a krokovať skonštruované automaty na vstupoch. V jednotlivých častiach práce sa zaoberám návrhom aplikácie, postupom pri implementácii, spracovaním a vyhodnotením vstupu a ukladaním do súboru.

Kľúčové slová: simulácia gramatík, simulácia automatov, pedagogický softvér

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

Tak, ako mnohé iné oblasti informatiky, aj teória formálnych jazykov a automatov je postavená na matematickom základe. Študenti riešia v procese výučby úlohy na papier, bez možnosti vyskúšať si svoje riešenie "na živo". Formálne jazyky a automaty pritom nie sú čisto teoretický predmet, majú široké uplatnenie v praxi, ktoré mnohí študenti podceňujú. Už len taká bežná vec, ako je spracovanie vstupu programu, má základy v tejto teórii. Študenti majú často problém pochopiť jednoduchšie konštrukcie, vidieť za nimi niečo viac, ako len matematickú teóriu.

Rozhodli sme sa preto navrhnúť softvér, ktorý by umožnil študentom simulovať jednotlivé konštrukcie v počítači a vizualizovať a odhaliť tak prípadné chyby riešenia. Softvér by tak umožňoval v zmysle "škola hrou" osvojiť si základnú zručnosť pri konštrukcii gramatík, či automatov.

Táto bakalárska práca bola vypísaná ako tímový projekt pre dvoch študentov. Časti softvéru su preto opísané v práci Dušana Baníka, ktorý so mnou spolupracoval pri vývoji tohto softvéru.

1.1 Cieľ

Cieľom tejto bakalárskej práce bolo navrhnúť softvér, ktorý by slúžil ako pomôcka v procese výučby formálnych jazykov a automatov. Mal by umožniť študentom vyskúšať a overiť si preberanú látku. Softvér by pritom nemal slúžiť ako automatický riešiteľ úloh, ale mal by fungovať ako forma inteligentného elektronického papiera. Mal by overovať formálnu korektnosť vstupu a umožňovať experimentovať s gramatikami, či automatmi. Dôležitá požiadavka bola krokovanie jednotlivých krokov odvodenia (výpočtu).

Kapitola 2

Použité technológie, postupy

V tejto kapitole popíšem technológie, nástroje a postupy, ktoré sme použili počas vývoja softvéru. Z technológií sú to: programovací jazyk Java, vývojové prostredia NetBeans a Eclipse a systém na skupinovú spoluprácu programátorov CVS. Z postupov sú to najmä návrhové vzory.

2.1 Java SE

Java je objektovo orientovaný programovací jazyk vyvinutý spoločnosťou Sun Microsystems. Tento jazyk sme zvolili pre jeho platformovú nezávislosť, dostupnosť vývojových prostredí, rozsiahlu knižnicu objektov vrátane knižnice na tvorbu grafických uživateľských aplikácií (ďalej len GUI), možnosť spúšťať aplikácie cez internet priamo z prehliadača a predošlými skúsenosťami s týmto programovacím jazykom.

SE znamená standard edition a je to verzia určená na vývoj menších aplikácií bežiacich na pracovných staniciach. Okrem tejto edície existujú ešte ME - mobile edition určenej pre mobilne zariadenia a EE - enterprise edition určenej pre vývoj podnikových viacvrstvových aplikácií. Je zrejmé, že tento projekt zapadá najlepšie do určenia edície SE. Pri vývoji sme používali vývojový kit (Java Development Kit, ďalej len JDK) vo verzií 5.

2.2 Vývojové prostredia

Vývojové prostredia (ďalej len IDE - integrated development environment) sú podporné nástroje na vývoj aplikácií. Väčšinou obsahujú editor zvýrazňujúci syntax, prehliadač objektov, spolupracujú s kompilátorom a debuggerom a obsahujú nástroje na tímovú spoluprácu.

2.2.1 NetBeans

NetBeans je IDE vyvinuté firmou Sun na podporu programovania v Jave. Oproti Eclipse obsahuje navyše nástroj na rýchly grafický návrh GUI a uľahčenie písania dokumentácie a tiež nástroje na tvorbu UML diagramov. Toto IDE je šírené pod

licenciou CDDL, ktorá umožňuje jeho bezplatné používanie. Použité NetBeans bolo vo verzii 5.5.

2.2.2 Eclipse

Eclipse je IDE vyvinuté pôvodne firmou IBM, určené pre ich vývojový nástroj na podporu programovania v Jave. Toto prostredie je v kombinácií s Javou veľmi používané. Celé prostredie je tvorené základným aplikačným rámcom (framework), ktorý sa dá rozširovať pomocou tzv. zásuvných modulov. Okrem už spomenutých bežných súčastí, obsahuje aj modul na tímovú spoluprácu cez CVS. Keďže obe spomínané prostredia obsahujú podporu CVS, mohli sme bezproblémovo používať každý svoje preferované IDE.

2.3 UML

UML je štandardizovaný špecifikačný jazyk určený na modelovanie objektových systémov. Umožňuje špecifikovať, vizualizovať, navrhovať a dokumentovať softvérové systémy pomocou grafických diagramov. Tento jazyk sme použili pri prvotnom návrhu aplikácie. UML obsahuje 9 typov diagramov. My sme z nich použili hlavne:

1. Diagram tried - umožňuje vizualizovať triedy a spoluprácu medzi nimi. Tento diagram znázorňuje, ktoré triedy spolupracujú, ale nezachytáva čo sa stane pri-/po výmene správ (je statický).

2. Diagram balíčkov a objektov - Java umožňuje zoskupovanie súvisiacich tried do balíčkov. Tento diagram umožňuje zobraziť zoskupenie tried do balíčkov a ich následnú spoluprácu.

Niektoré nasledujúce diagramy v tejto práci budú práve v UML.

2.4 CVS

CVS znamená Concurrent Versions System a je to open source softvér umožňujúci spolupracovať viacerým vývojárom na rovnakom projekte súčasne. Tento softvér je založený na technológií klient-server.

Aby sa zamedzilo vzájomnému prepisovaniu si kódu, server dovoľuje ukladať len zmeny vykonané na poslednej verzií súboru. Každá zmena vyvolá zvýšenie verzie a programátor musí dokumentovať vykonané zmeny. Klienti si môžu vyžiadať od servera stiahnutie celého projektu (checkout), alebo len zmeny vykonané od poslednej verzie, ktorú majú lokálne prístupnú (update). Okrem toho umožňuje vytvárať bočné vývojové vetvy a tieto následne zlučovať do hlavnej vetvy, čo je výhodne pri dorábaní funkcionality, ak chceme zamedziť znefunkčneniu celej aplikácie. Ďalšia užitočná funkcia je prechádzanie jednotlivých verzií. Táto funkcia sa ukázala byť užitočná aj pre jednotlivého programátora.

Pre vývoj sme zvolili toto riešenie, vzhľadom na jeho dostupnosť, podporu vývojovými prostrediami a dostatočnou znalosťou tejto technológie.

2.4.1 CVSNT

CVSNT[8] je implementácia CVS bežiaca pod OS Windows NT. Použili sme verziu 2.5, čo bola aktuálna verzia v dobe začiatku vývoja. Tento softvér bol zvolený pre jeho ľahkú dostupnosť, pretože je šírený pod GNU GPL, a jeho kompatibilitu s implementáciou CVS vo vývojových prostrediach Eclipse a NetBeans.

2.5 Swing-Layout

Swing-Layout[5] je knižnica implementujúca rozhranie LayoutManager, vhodná najmä pre aplikácie podporujúce grafický návrh GUI. NetBeans potrebuje na grafický návrh GUI práve túto knižnicu. Swing-layout implementuje tri základné triedy:

1. Baseline - umožňuje získať základný riadok komponentu
2. GroupLayout - umožňuje hierarchicky zoskupovať komponenty
3. LayoutStyle - určuje koľko miesta sa má umiestniť medzi jednotlivé komponenty

Spolu tieto triedy umožňujú vytvárať prehľadné okná s odporúčaným rozmiestením komponentov na danej platforme. Ide najmä o veľkosti okrajov, vzdialenosť medzi jednotlivými komponentmi a zarovnanie.

Základné časti tejto knižnice sa stali súčasťou Javy v1.6. Keďže my sme pracovali s verziou 1.5, museli sme použiť samostatnú verziu tejto knižnice. Knižnica je šírená pod LGPL licenciou a použitá bola verzia 1.0.2.

2.6 Swing

Triedy určené na tvorbu grafických aplikácií sú na platforme Java združené do tzv. JFC (Java Foundation Classes). Časti JFC sú AWT, Swing a Java 2D. Java 2D slúži na kreslenie dvojrozmerných objektov. AWT (Abstract Window Toolkit) je starší toolkit na tvorbu GUI a je platformovo závislý - aj keď rozhranie zostáva rovnaké, na danej platforme musí existovať implementácia. Obsahuje iba základné typy komponentov, a niektoré nemusia fungovať na rôznych platformách rovnako. Oproti tomu novší Swing je celý napísaný v Jave, a preto je platformovo nezávislý. Keďže dostupnosť komponentov nie je závislá na platforme, obsahuje aj viac komponentov. Z týchto príčin som sa rozhodol použiť práve Swing[6].

2.7 Návrhové vzory

Návrhové vzory ponúkajú riešenia často sa vyskytujúcich problémov pri návrhu softvéru. Návrhové vzory pritom nepredstavujú hotové riešenie, ale iba pomáhajú pri vytváraní objektovej štruktúry - čo má byť objekt, aké má mať rozhrania a aká má byť hierarchia objektov, ako objekty spolupracujú. Pri návrhu som použil vzory MVC a visitor.

Kapitola 3

Návrh softvéru

Keďže sme na vývoji softvéru spolupracovali dvaja, rozdelili sme návrh softvéru do niekoľkých disjunktných modulov, aby sme na nich mohli následne pracovať samostatne. Návrh pozostával z týchto fáz:

1. V prvotnej fáze sme špecifikovali v spolupráci s vedúcim práce základné požiadavky na softvér. Softvér má slúžiť ako pomôcka pri výučbe formálnych jazykov.

   Má dohliadať, aby študentom zadaná gramatika mala správny tvar, aby bola zo správnej triedy (správny tvar pravidiel) a má umožniť s gramatikou rôzne experimentovať. Má umožniť aj overovanie príslušnosti slova do jazyka generovaného gramatikou, či už deterministicky alebo heuristicky.

   Podobne má umožniť experimentovať s automatmi, overiť, či automat akceptuje dané slovo a overovať ekvivalenciu medzi gramatikou a automatom.

2. Následne sme sa dohodli na základnej objektovej a dátovej reprezentácii. Pri tomto návrhu vznikli triedy Rule, Grammar, Regular, ContextFree, ContextSensitive, Phrasal. Tiež sme sa dohodli na základných balíčkoch (packages). Pri tejto fázi sme používali najmä UML. Podrobnejší popis spomenutých tried, sa nachádza v práci Dušana Baníka.

3. Navrhol som základné rozhranie aplikácie. Následne sme sa dohodli na rozdelení častí. Ja som implementoval interakciu s uživateľom - grafická časť, spracovanie vstupu, prácu s množinami, implementáciu DTS[1], DKA[2], ukladanie do súboru a kolega Baník už spomenuté triedy na manipuláciu s gramatikami.

3.1 GUI

V tejto časti popíšem grafické uživateľské rozhranie aplikácie (ďalej len GUI - graphics user interface), ktoré som navrhol a implementoval. GUI slúži na interakciu s uživateľom v grafickom prostredí OS. Stará sa o vstup dát do aplikácie a ich zobrazovanie.

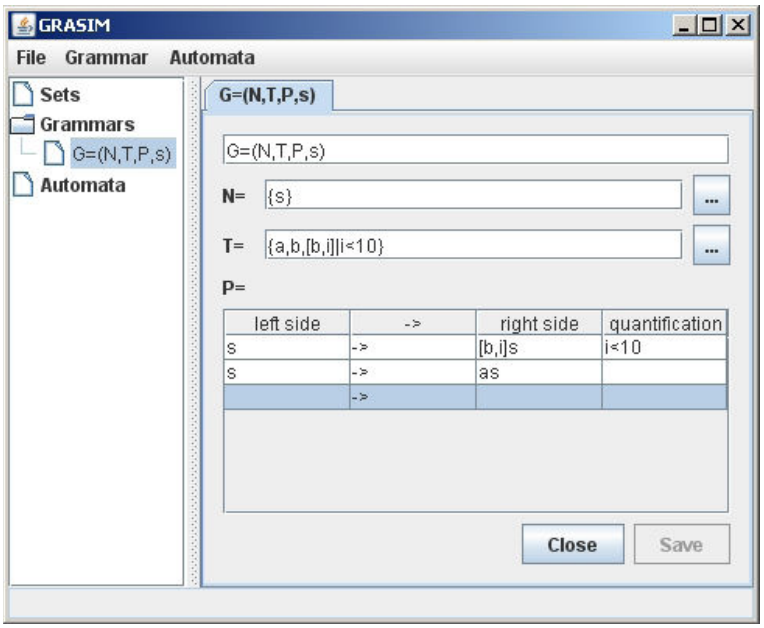
[1] deterministický turingov stroj
[2] deterministický konečný automat

Moderné návrhy nemiešajú logiku aplikácie a prezentáciu dát, toto popíšem v časti MVC (3.2). GUI je práve tá časť aplikácie, ktorú vidí uživateľ ako prvú, nemala by preto pôsobiť odstrašujúcim dojmom, pôsobiť zložito a nezvládnuteľne. Snažil som sa ho preto navrhnúť tak, aby pôsobilo jednoducho a intuitívne pre študenta študujúceho formálne jazyky.

Aplikácia dodržiava štandard SDI[3]. Toto znamená, že každé načítanie súboru vytvorí samostatné nové okno zobrazujúce načítané dáta.

3.1.1 Hlavné okno

Hlavné okno aplikácie môžeme vidieť na obrázku 3.1.



Obr. 3.1: Hlavné okno aplikácie.

Okno je rozdelené na dve časti:

1. Ľavá - strom obsahujúci všetky uživateľom vytvorené množiny, gramatiky a automaty. Tieto radí do priečinkov.

2. Pravá - obsahuje všetky otvorené definície gramatík, množín a automatov, či už nové alebo otvorené existujúce. Tieto radí do záložiek.

Dvojkliknutím na príslušný prvok[4] v strome v ľavej časti sa otvorí nová záložka v pravej časti, alebo sa aktivuje príslušná záložka, ak už je prvok otvorený.

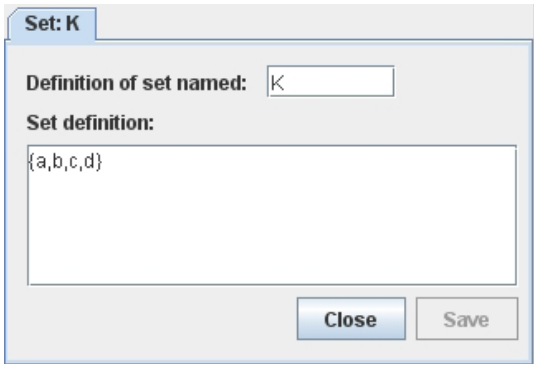
[3]Single Document Interface
[4]gramatika, automat, množina

Množiny

Formulár na zadávanie množín vidieť na obr. 3.2. Nová množina sa dá vytvoriť pomocou menu `File/New/Global set`.

Tento formulár slúži na zadávanie globálnych množín, t.j.. prístupných v definícii inej množiny. Množiny[5] definované v rámci automatov a gramatík nie sú prístupné v iných definíciách automatov a gramatík.

Obsahuje len položky pre názov množiny a pre samotnú definíciu. Definícia množín podporuje jednoduché kvantifikácie. Toto je podrobnejšie popísané v časti 3.3.5. Tlačidlom `Save` sa množina po overení korektnosti vstupu uloží. Tlačidlom `Close` sa záložka uzavrie.



Obr. 3.2: Záložka na zadávanie množiny.

Gramatiky

Na obrázku 3.1 je vidieť otvorenú záložku s jednoduchou gramatikou. Novú gramatiku vytvoríme výberom príslušnej gramatiky pomocou menu `File/New/...`.

Pri návrhu tohto formulára som vychádzal zo zaužívaného poradia zápisu gramatiky: definícia gramatiky, neterminálov, terminálov a samotnej množiny pravidiel. Pre sprehľadnenie sa množina pravidiel zadáva do tabuľky. Korektnosť zápisu sa kontroluje počas písania s výnimkou množiny pravidiel, ktorá sa kontroluje až pri ukladaní. Pri nekorektnom vstupe sa zmení pozadie príslušného riadku na červeno. Množiny terminálov a neterminálov sa zadávajú v zjednodušenom štandardnom množinovom zápise, pričom je možná aj kvantifikovaná indexácia prvkov množiny. Ide o jednoduchý výpočet prvkov, rôzne logické operácie nie sú podporované. Syntax presne popíšem v časti 3.3.5.

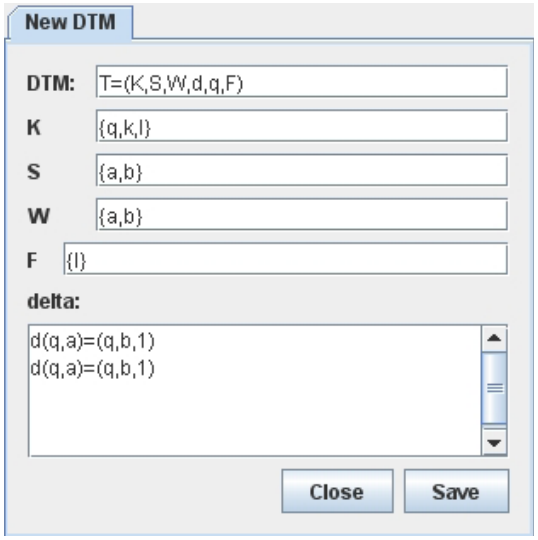
Pre zadávanie definícií množín na viac riadkov slúži tlačidlo s tromi bodkami. Po jeho stlačení sa otvorí okno s textovým vstupom a v prípade, že používateľ skutočne zadal viacriadkový vstup, v pôvodnom jednoriadkovom vstupe sa zamedzí editovanie. Editovanie je opätovne povolené len ak je vstup jednoriadkový (editovanie rozšíreného

[5]terminály, neterminály, symboly vstupnej a pracovnej abecedy

vstupu funguje vždy). Takto sa šetrí miestom v okne a pritom sa zachováva plná funkcionalita.

Automaty

Z automatov softvér podporuje len deterministické turingové stroje (DTS) a deterministické konečné automaty (DKA). Formulár na zadávanie DTS vidíme na obrázku 3.3. Nový DTS vytvoríme pomocou menu `File/New/DTM`, podobne DKA pomocou `File/New/DFA`.



Obr. 3.3: Záložka na zadávanie DTS.

Prvky formulára sledujú definíciu DTS podľa [2]. Sú na ňom položky v tomto poradí: samotná definícia DTS, definícia množiny stavov, abecedy vstupných symbolov, abecedy pracovných symbolov, množiny akceptačných stavov a definícia δ funkcie. Táto sa zadáva do textového komponentu po riadkoch, na každom riadku musí byť jeden riadok δ funkcie.

Kontrola vstupu definície DTS sa vykonáva počas písania, ak nie je korektná, riadok má červené pozadie, inak biele. Ostatné prvky sa kontrolujú až pri ukladaní.

Tento formulár, na rozdiel od formulára pre zadávanie gramatík, neobsahuje tlačidlá na zadávanie viacriadkového vstupu. Pre toto som sa rozhodol z dôvodu možnosti zadávať samostatné množiny, kde sa toto dá.

Formulár na zadávanie DKA vyzerá rovnako ako formulár na zadávanie DTS, len je v ňom vynechaný riadok na zadávanie pracovných symbolov. Pre zadávanie δ funkcie platí to isté ako pre zadávanie δ funkcie DTS.

3.1.2 Menu

Menu aplikácie obsahuje položky **File**, **Grammar**, **Automata**.
File obsahuje:

1. New - umožňuje vytvoriť novú gramatiku, množinu a automat. Obsahuje podmenu s položkami Regular Grammar, Context Sensitive Grammar, Context Free Grammar, Phrasal Grammar, DTM, DFA, Global set. Vytvorené typy gramatík zodpovedajú slovenskému ekvivalentu príslušného názvu položky, DTM vytvorí nový deterministický turingov stroj, DFA nový det. konečný automat a global set novú množinu.

2. Open - načíta súbor. Tento súbor musí byť vopred uložený aplikáciou. Jeho formát bude popísaný v časti 3.6. Po kliknutí na túto položku sa otvorí dialóg na výber súboru, následne sa načíta a ak ma správny formát, vytvorí sa nové okno aplikácie. Aplikácia neumožňuje zlúčenie už vytvorených prvkov (gramatík, automatov) s vopred uloženým dokumentom. Otvorenie súboru a následné pridávanie a uberanie prvkov samozrejme funguje podľa očakávaní.

3. Save - uloží vytvorené gramatiky, automaty a množiny do súboru. Po kliknutí na túto položku sa otvorí dialóg na výber umiestnenia a názvu súboru a následne sa uloží.

4. Quit - ukončí aplikáciu.

Grammar sa vzťahuje vždy na aktuálne otvorenú gramatiku - aktívnu záložku, pričom pri otvorení záložky obsahujúcej iný element (napr. automat) sa nesúvisiace položky zneprístupnia. Obsahuje:

1. Derive - otvorí okno na odvodzovanie vetných foriem. Toto bude popísané v časti Okno Derive.

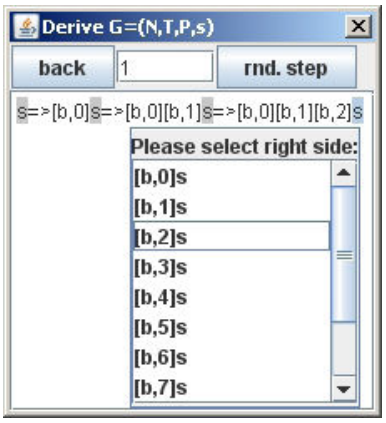
2. Test word is from G - otvorí dialóg na zadanie slova a následne sa pokúsi overiť jeho príslušnosť do jazyka generovaného gramatikou.

Automata sa podobne, ako v prípade gramatík, vzťahuje vždy na aktuálne aktívnu záložku, nesúvisiace záložky sa zneprístupnia. Momentálne obsahuje len položky Run DTM on input a Run DFA on input, ktoré vyvolajú okno umožňujúce krokovanie daného DTS(DFA) na zadanom vstupe.

3.1.3 Okno Derive

Toto okno sa otvorí po kliknutí na položku Derive v hlavnom menu a slúži na manuálne alebo automatické generovanie krokov odvodenia. Okno je vidieť na obrázku 3.1.3. Nachádza sa v ňom:

1. Tlačidlo back - umožňuje vrátiť sa o krok späť. Dá sa vrátiť až k počiatočnému neterminálu.



Obr. 3.4: Okno Derive

2. Pole textového vstupu - slúži na zadanie počtu opakovaní náhodne urobeného kroku odvodenia.

3. Tlačidlo rnd. step - vytvorí n nasledujúcich krokov odvodenia náhodne výberom v každom kroku aplikovateľných pravidiel, kde n je počet zadaný v predošlom poli.

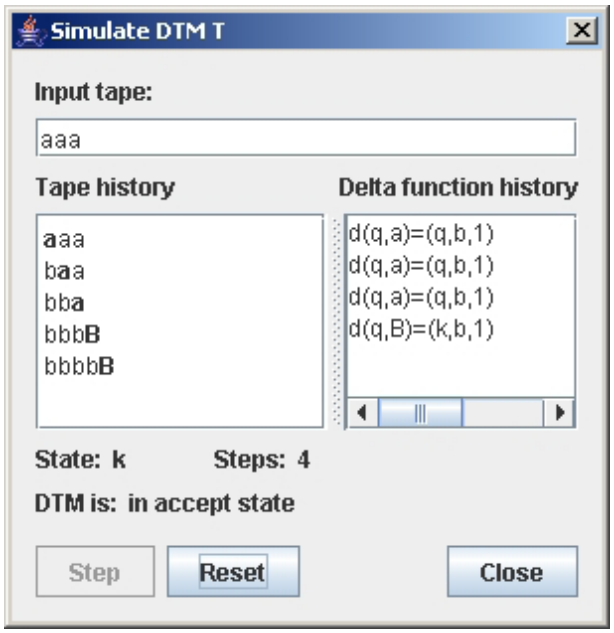
4. Samotná plocha s odvodením - na začiatku obsahuje iba počiatočný neterminál. Ľavú stranu pravidla vyberieme výberom textu do bloku. Po kliknutí pravým tlačidlom myši sa vyvolá menu, ktoré zobrazí aplikovateľné pravé strany pravidiel. Toto menu je vidieť na obr. 3.1.3. Pri manuálnom odvodení sa zvýrazňuje vybraná časť predošlých vetných foriem. Vyberať do bloku možno len text, ktorý tvorí poslednú vetnú formu v odvodení.

3.1.4 Okno Simulate DTM

Toto okno sa vyvolá z menu `Automata/Run DTM on input` a slúži na simuláciu DTS na zadanom vstupe. Toto okno môžeme vidieť na obr. 3.5. Nachádza sa v ňom po rade:

1. Riadok na zadanie vstupu. Po stlačení tlačidla `Reset` sa overí, či riadok obsahuje iba symboly zo vstupnej abecedy a začne sa simulácia s týmto vstupom. Jednotlivé kroky výpočtu sa simulujú tlačidlom `Step`. Ak už simulácia prebieha a zmeníme vstup, preruší sa a začne sa nová simulácia. Ak prepíšeme tento riadok a nový vstup je rovnaký ako starý, simulácia pokračuje ďalej. Po stlačení tlačidla `Reset` sa zruší prebiehajúca simulácia a začne sa nová.

2. Časť s pravou a ľavou stranou. Posledný riadok ľavej strany obsahuje momentálny obsah pásky, predošlé zaznamenávajú históriu, pričom pozícia hlavy je zvýraznená tučným písmom. Pravá strana obsahuje históriu použitých riadkov delta funkcie.



Obr. 3.5: Okno Simulate DTM

3. Spodná časť obsahuje aktuálny stav, počet krokov a stav automatu, teda či je zaseknutý (`lock`), krokuje (`step`), alebo je v akceptačnom stave (`in accept state`).

3.1.5 Okno Simulate DFA

Toto okno vyzerá a správa sa rovnako ako okno Simulate DTM.

3.2 Swing a MVC

MVC znamená model-view-controller a je to návrhový vzor popisujúci architektúru aplikácie prezentujúcej nejaké dáta. Základom tohoto návrhového vzoru je myšlienka oddelenia dát (aplikačnej logiky) a uživateľského rozhrania. Model reprezentuje dáta aplikácie, view vizualizovanie dát - v našom prípade[6] okná aplikácie a controller spracováva uživateľský vstup a robí prípadné zmeny v modeli.

Swing vychádza z designu MVC, pričom poskytuje vlastné implementácie modelov pre jednotlivé komponenty a tak nenúti programátora používať tento návrhový vzor.

Model pre dokument aplikácie je reprezentovaný triedou `Main`. Táto trieda vzniká nová pre každý otvorený súbor. Pre jednoduché dialógové okná som sa rozhodol používať hotové modely a na tabuľku s pravidlami som vytvoril vlastný model, ktorý je implementovaný v triede `RuleTableModel`. Okrem toho som vytvoril modely pre udržovanie definícií množín, automatov a gramatík. Tieto slúžia na uloženie uživateľom

[6]Napr. pri web aplikácii tvorí view HTML stránka v klientskom prehliadači

zadaných dát. Obsahujú aj metódy na kontrolu týchto dát a spolupracujú s triedou `Main` na ich ukladaní. Všetky nasledujúce triedy sa nachádzajú v balíku `sk.fmfi.gui`:

1. `Main` - Táto trieda tvorí hlavnú triedu aplikácie. Obsahuje hlavnú metódu `main()`. Okrem toho sa v nej nachádzajú zoznamy vytvorených gramatík, automatov a množín a metódy na ukladanie a načítavanie stavu aplikácie. Hlavný formulár sa zobrazí metódou `exec()`.

2. `MainFrame` - Trieda implementujúca hlavné okno aplikácie. V tejto triede sa nachádzajú obslužné metódy udalostí vyvolané z menu, ďalej metódy na pridávanie uzlov do ľavej časti okna (stromu) a vytváranie záložiek v pravej časti.

3. `JPanelGrammarDef` - Trieda implementujúca panel zadávania gramatiky (obr. 3.1). Keďže je táto trieda odvodená od triedy `JPanel`, nie je problém vytvoriť na zadávanie samostatné okno, ktorého obsahom bude tento panel. V súčasnej verzii je však implementované iba otváranie v záložkách.

   Editáciu gramatiky, terminálov a neterminálov som implementoval tak, že vyhodnocuje syntaktickú správnosť vstupu hneď, ako užívateľ píše. Nájdené chyby sa zobrazujú v stavovom riadku. Modelom pre toto okno je trieda `GrammarModel`.

4. `GrammarModel` – V tejto triede sú uložené vstupy presne tak, ako ich zadal užívateľ. Predstavuje model pre `JPanelGrammarDef`. V prípade, že sa otvorí už existujúca gramatika, dostane trieda `JPanelGrammarDef` na vstup konštruktora túto triedu, z ktorej potom inicializuje príslušné komponenty. Inštancie týchto tried sú uložené v triede `Main`.

5. `RuleTableModel` – Trieda implementujúca model pre tabuľku (trieda `JTable`). Túto triedu som implementoval z dôvodu automatického pridávania posledného prázdneho riadku do tabuľky, rušenia prázdnych riadkov v prípade zrušenia pravidla používateľom a automatické zobrazovanie šípky medzi ľavou a pravou stranou pravidla.

6. `JPanelSetDef` – Trieda implementujúca formulár pre zadanie globálnej množiny (obr. 3.2). Korektnosť zadaného vstupu sa overuje až pri ukladaní. Táto trieda je odvodená od `JPanel`. Modelom pre tento formulár je `SetModel`.

7. `SetModel` – Trieda implementujúca model pre množinu. Nachádzajú sa v nej dáta pre formulár `JPanelSetDef`.

8. `JPanelDTMDef` – Trieda implementujúca formulár na zadávanie DTS. Korektnosť zadaného vstupu sa overí podobne ako v prípade `JPanelSetDef` až pri ukladaní. Modelom pre tento formulár je `DTMAutomataModel`.

9. `DTMAutomataModel` – Trieda implementujúca model pre DTS. Pretože jednotlivé definície automatov sa líšia, musel som pre DTS implementovať konkrétny model. Nachádzajú sa v nej dáta pre formulár `JPanelDTMDef`.

10. `JPanelDFADef` – Trieda implementujúca formulár na zadávanie DFA. Správa sa rovnako ako `JPanelDTMDef`.

11. `DFAAutomataModel` – Trieda implementujúca model pre DFA. Táto trieda reprezentuje dáta pre formulár `JPanelDFADef`.

12. `InputSetDialog` - Implementácia dialógu viacriadkového zadávania množín terminálov a neterminálov.

13. `AppletStarter` - Táto trieda je odvodená od triedy `javax.swing.JApplet`, ktorá implementuje rozhranie Applet. Toto zabezpečuje, že aplikácia sa dá pustiť z internetového prehliadača. V jej hlavnej metóde `init()` je vytvorený hlavný objekt `Main`, ktorý zobrazí hlavný formulár.

3.3 Spracovanie vstupu

V tejto kapitole budú popísané špecifikácie a spracovanie uživateľského vstupu. Keďže vstup je potrebné aj vyhodnocovať, bolo potrebné použiť pokročilejšie techniky, ponúkajúce systematický prístup. Použil som známe techniky používané pri vývoji kompilátorov. V nasledujúcich častiach sa pokúsim stručne zhrnúť metódy ktoré som použil ja, podrobnejšie sa čitateľ môže dočítať v [10].

3.3.1 Špecifikácia vstupu

Pri spracovaní syntakticky pevne daného vstupu potrebujeme mať najprv definované, aká táto syntax je. Na toto sa najčastejšie používajú práve prostriedky z teórie formálnych jazykov. V tejto časti preto formálne špecifikujem formát vstupu, aký sa od používateľa očakáva.

EBNF

Na špecifikáciu formálneho vstupu sa používa viacero prístupov. Medzi najznámejšie patrí XML, BNF a EBNF. Ja som sa rozhodol pre EBNF[7], ktorý je štandardizovaný organizáciou ISO v norme ISO 14977 [3]. Syntaxe definované gramatikami v EBNF generujú bezkontextové jazyky [10, p.14].

Pretože popis tohto metajazyka presahuje rámec tejto práce, popíšem len prvky ktoré budeme potrebovať.

Syntaktický metajazyk je zápis definície syntaxe jazyka za použitia gramatických pravidiel.
Neterminál je prvok jazyka, ktorý definujeme. Môžu byť nahradené pravou stranou, ak sa zhoduje príslušná ľavá strana pravidla.
Terminál je reťazec jedného, alebo viac znakov tvoriacich nedeliteľný prvok jazyka, tzn. že nemôžu byť nahradené žiadnou inou postupnosťou.
Syntax jazyka pozostáva z jedného, alebo viacerých pravidiel.

[7]Extended Backus-Naur form

Pravidlo pozostáva z neterminálneho symbolu, ktorý definujeme, nasledované "=", nasledované zoznamom definícií, oddelených "|". Zoznam definícií je ukončený ";". Pravidlo teda definuje možné náhrady neterminálnych symbolov.
Voliteľná postupnosť pozostáva z "[", zoznamu definícií, ukončená "]".
Opakovaná postupnosť pozostáva z "{", zoznamu definícií, ukončená "}".
Zoskupenie symbolov pozostáva z "(", zoznamu definícií, ukončená ")".

Definícia syntaxe EBNF v EBNF vyzerá potom takto[8] [10]:

```
syntax = {production};
production = identifier "=" expression ";";
expression = term {"|" term};
term = factor {factor};
factor = identifier | string | "(" expression ")"
       | "[" expression "]" | "{" expression "}";
identifier = letter {letter | digit};
string = """ {character} """;
letter = "A" | ... | "Z" | "a" | ... "z";
digit = "0" | ... | "9";
```

3.3.2 Syntax zápisu gramatiky

Gramatika sa zapisuje podľa [2, p.9]. Zjednodušene je to štvorica $G = (N, T, P, s)$.

```
gramatika = string,"=(",string,",",string,",",string,","
    ,element-starting,")";
string = letter,[{letter}];
element-starting = letter|digit|"[",elbr-starting,"]";
elbr-starting = oneelbr,[{",",oneelbr}];
oneelbr = (letter,{(letter|digit)});
strana-pravidla = element-string,{element-string};
kvantifikacia-pravidiel = kvantifikacia,{"|",kvantifikacia};
```

Kvantifikácia a `element-string` sú popísané v časti 3.3.5. Elnumber predstavuje max. 5 ciferné číslo. Toto som zvolil ako primerané optimum medzi použiteľnosťou a rozsahom reprezentovateľným celočíselným typom int. Pri definícii gramatiky nie sú povolené biele znaky. Príklad vstupu:

```
G=(N,T,P,[s,1])
N={[s,a]|a<10}
T={a}
P={
[s,i]->a[s,i+1] i<5
[s,6]->a
[s,6]->[s,1]
}
```

[8] Formálna definícia EBNF definovaná pomocou EBNF v norme ISO zaberá 2 strany A4.

Sémantika

Jednotlivé neterminály `string` znamenajú názvy, priama definícia anonymnej množiny nie je povolená. V súčasnej verzii nie je možné zadať množinu priamo pomocou globálnej množiny jej zápisom do definície. Dá sa to nahradiť pomocou $N = \{[a]|a\ in\ M\}$ kde M v príklade je globálna množina (v tomto prípade zrejme neterminálov). Prvky množín tvaru $[cislo * a + cislo]$ sa vyhodnocujú podľa očakávania. V prípade, že a nie je kvantifikované, výraz ostáva nezmenený.

Spracovanie vstupu

Vstup je rozpoznaný regulárnym výrazom pomocou triedy `Matcher`. Táto trieda je štandardnou súčasťou Java API a umožňuje aj zistiť jednotlivé matchujúce reťazce, čo využívam pri zisťovaní jednotlivých názvov (gramatiky, množín, počiatočného neterminálu). Uvádzam príslušný regulárny výraz zapísaný v jazyku Java:

```
String str = "([a-zA-Z]+)";
String CS = "[a-zA-Z0-9]+(?:,[a-zA-Z0-9]+)*";
String El = "([a-zA-Z0-9]|\\["+CS+"\\])";
String gramRegexp = "^"+str+"=\\("
          +str+","+str+","+str+","+El+"\\)$";
```

Implementácia je v triede `InputGrammarParser`. Táto trieda poskytuje okrem metódy na rozpoznanie syntaxe (metóda `checkSyntax()`) aj metódy na zistenie jednotlivých reťazcov.

3.3.3 Syntax zápisu DTS

DTS sa zapisuje podobne ako gramatika podľa štandardnej definície. S využitím už definovaných neterminálov je to:

```
DTS = string,"=(",string,",",string,",",string,","
    ,string,",",element-starting,",",string)";
delta = {delta-riadok,newline};
delta-riadok = string,"(",element-string,",",element-string,")=("
  ,element-string,element-string,"0"|["-"]"1"),[{"|",kvantifikacia}];
```

Jednotlivé množiny v gramatikách a automatoch sa definujú podľa 3.3.5.

Sémantika

Sémantika je podobná ako pri gramatike. Pre úplnosť uvádzam príklad:
$T = (K, S, W, d, q, F)$
$K = \{q, k\}$
$S = \{a, b\}$
$W = \{a, b, [a, b]|b < 3\}$
$d(q, a) = (q, a, 0)$
$d(q, [a, i]) = (q, [a, i + 1], 1)|i < 2$

V riadku $W = \{a, b, [a, b] | b < 3\}$ definujeme, že abeceda pracovných symbolov bude mať prvky a,b,[a,0],[a,1],[a,2]. V riadku $d(q, [a, i]) = (q, [a, i + 1], 1) | i < 2$ definujeme 2 riadky delta funkcie:
$d(q, [a, 0]) = (q, [a, 1], 1)$
$d(q, [a, 1]) = (q, [a, 2], 1)$

Spracovanie vstupu

Podobne ako v prípade gramatík, vstup je rozpoznaný regulárnym výrazom. V tomto prípade sú takto rozpoznané aj riadky δ funkcie. Regulárny výraz na rozpoznanie δ funkcie zapísaný v jazyku Java:

```
String N = "^[-]?[0-9]{1,5}$";
String N1 = "(?:[-]?[0-9]{1,5}\\*)?";
String N2 = "(?:[+-][0-9]{1,5})?";
String CS = "(?:"+N1+"(?:[a-zA-Z0-9][a-zA-Z0-9]*)"+N2+")";
String El = "((?:[a-zA-Z])|(?:\\["+CS+"(?:,"+CS+")*\\]))";
String DIR = "((?:[-]?1)|0)";
String QUANTS = "(\\|.*)?";
String delta = "\\("+El+","+El+"\\)=\\("+El+","+El+","
        +DIR+"\\)"+QUANTS+"$";
```

Kvantifikácie (podvýraz `QUANTS`) sú rozpoznané a súčastne vyhodnotené dodatočne samostatným parserom.

Implementácia kontroly vstupu pre definíciu TS je v súbore `InputDTMParser`, implementácia parsovania delta funkcie je v triede `DTMDeltaParser`. Rozpoznanie syntaxe a sparsovanie sa robí metódou `parse()`, ktorá má ako výstup zoznam riadkov delta funkcie (časť 3.4).

3.3.4 Syntax zápisu DFA

Pre tento zápis platí to isté ako pre DTM, mení sa len formát definície a tvar delta funkcie:

```
DFA = string,"=(",string,",",string,","
    ,string,",",element-starting,",",string)";
delta = {delta-riadok,newline};
delta-riadok = string,"(",element-string,",",element-string,")="
    ,element-string,[{"|",kvantifikacia}];
```

3.3.5 Syntax množinového zápisu

V nasledujúcich definíciách som pre prehľadnosť vynechal biele znaky. Nejde teda o formálne presné zápisy.

```
mnozina = "{",obsah-mnoziny,"}";
obsah-mnoziny = element-string|kvant-element,
```

```
              [{",",(element-string|kvant-element)}]|"";

kvantifikacia = cislo,relacia,retazecc
              | cislo,relacia,retazecc,relacia,cislo
              | retazecc," in ",retazec
relacia = "<"|"<=";
retazec = letter,[{letter}];
retazecc = letter|digit,[{cislo|retazec}]
cislo = ["-"],digit,[digit],[digit],[digit],[digit];

kvant-element = brelement,{"|",kvantifikacia};
element-string = letter|digit|brelement;
brelement = "[",elbr,"]";
elbr = oneelbrexpr,[{",",oneelbrexpr}];
oneelbrexpr = [["-"],elnumber,"*"],(letter,{(letter|digit)})
    ,[("+"|"-"),elnumber];
elnumber = digit,[digit],[digit],[digit],[digit]
```

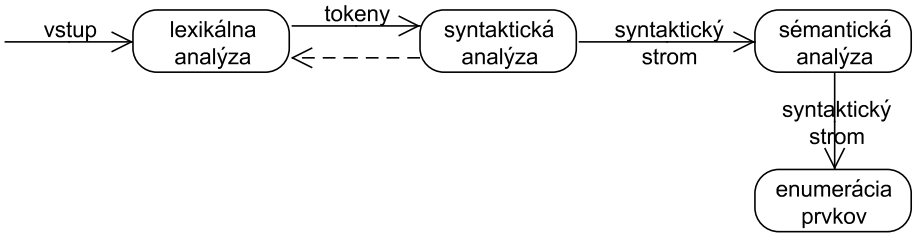
V uvedenej syntaxi sú významné biele znaky len v prípade `obsah-mnoziny`. Prvok obsahujúci biele znaky nebude v tomto prípade považovaný za korektný. Príklad zápisu:
$\{a, [aa], [a, b] | b < 10 | a < 10, [a] | a\ in\ L\}$

Sémantika

Množinový zápis som sa snažil navrhnúť tak, aby sa čo najviac podobal známemu zápisu množín z matematiky. Bolo pri tom treba umožniť kvantifikovanie prvkov množiny. Keďže v textovom zápise nie je možné písať jeden viacznakový prvok (neterminál/terminál) ako jeden znak, teda tak ako vo formálnych jazykoch napr. nové neterminály vytvorené z kartézského súčinu neterminálov, viacznakové prvky sa píšu do hranatých zátvoriek. Kvantifikovať je možné len prvky v hranatých zátvorkách a kvantifikácia sa vzťahuje len na predošlý prvok. Kvantifikované sú potom prvky pre ktoré existuje kvantifikovaná premenná v kvantifikácii zhodujúca sa s niektorou súradnicou. Príklad:
$\{[a, c], a, [a, b] | 2 < a < 5\}$
V tomto príklade kvantifikujeme len prvok $[a, b]$. Zápis znamená $\{[a, c], a, [3, b], [4, b]\}$. Prvky b a $[b]$ nie sú ten istý prvok. Vstup
$L = \{[a, b], c\}$
$M = \{[c, d] | d\ in\ L\}$
Vygeneruje prvky $\{[c, a, b], [c, c]\}$ pre množinu M, kde L je zadefinovaná ako globálna množina.

3.3.6 Fázy spracovania vstupu

Spracovanie vstupu je rozdelené do niekoľkých fáz, ktoré sú zachytené na obr. 3.6. V nasledujúcich častiach popíšem jednotlivé fázy.



Obr. 3.6: Fázy spracovania vstupu

3.3.7 Lexikálna analýza

Prvou fázou spracovania vstupu je lexikálna analýza. Úlohou lexikálneho analyzátora je vytvoriť zo vstupu zodpovedajúcu postupnosť tokenov, ktoré dostane na vstup syntaktický analyzátor. V minulosti sa program rozdeľoval na takéto časti pre nedostatok operačnej pamäte [10]. V súčastnosti sú hlavné dôvody takéhoto rozdelenia [7]:

1. zjednodušenie a sprehľadnenie návrhu
2. efektívnosť (môžme napr. urýchľovať načítanie vstupu bufferovaním a pod.)
3. portabilita a nezávislosť na kódovaní vstupu

Pozn.: **Lexéma** je reťazec znakov, ktorý predstavuje najnižšie postavenú syntaktickú jednotku v danom jazyku. **Token** označuje triedu lexém. Úlohou lexikálneho analyzátora je potom vytvoriť dvojice $< token, lexema >$ (napr. $< retazec,$ "$nejakyretazec$" $>$).

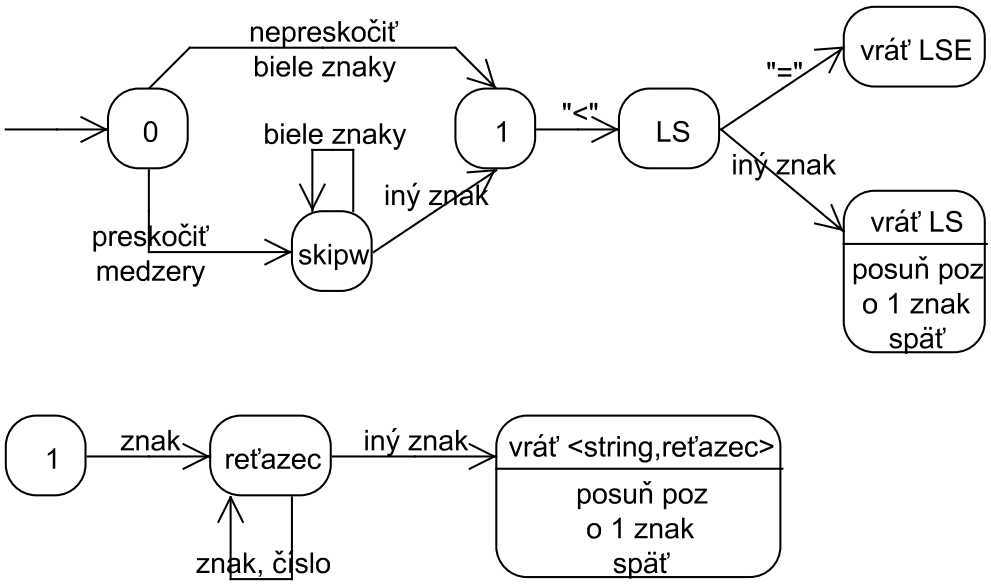
Zjednodušený stavový diagram lexikálneho analyzátora, ktorý rozpoznáva ostrú a neostrú nerovnosť a reťazce, ktoré môžu obsahovať číslo, ale začínajú znakom je na obr 3.7.

Analýza začína v stave 0, v príslušnom stave analyzátor vráti výsledok, stav rozdelený na dve časti znamená, že treba vrátiť jeden znak na vstup, pretože už nie je súčasťou tokenu. Po zavolaní metódy `nextToken()` analyzátor pokračuje v stave 0. Po dočítaní vstupu je každý ďalší token EOF. Po každom rozpoznanom tokene preskočí biele znaky a začne rozpoznávať ďalší.

Implementácia

Lexikálny analyzátor pre vstup "`mnozina`" je implementovaný v triede `Tokenizer`. Ako vstup konštruktora má táto trieda celý rozpoznávaný reťazec. Ide o jednoduchý stavový automat, ktorý sa môže pohybovať po vstupe (LBA). Toto je z toho dôvodu, že potrebujeme rozpoznávať aj "dlhšie" lexémy ako 1 znak, napr. reťazec a v prípade konca lexémy sa vrátiť o jeden znak naspäť.

Kľúčová je metóda `nextToken()`. Na začiatku sa preskočia biele znaky. Toto sa dá však vypnúť z dôvodu jemnejšej kontroly v syntaktickom analyzátore. Potom sa



Obr. 3.7: Lexikálna analýza

načíta jeden znak zo vstupu. Aktuálna pozícia je zaznamenaná jednoducho celočíselnou premennou. Následne sa podľa načítaného znaku rozhodne, o aký token ide jednoduchým príkazom switch. Načítanie reťazca je v samostatnej metóde a tak je možné napr. odvodiť nový lexikálny analyzátor, ktorý nepripúšťa čísla v reťazci. Na obr. 3.6 prerušovaná čiara medzi lexikálnym analyzátorom a syntaktickým analyzátorom znamená, že syntaktický analyzátor si "pýta" ďalšie tokeny postupne, teda sa nespracuje celý vstup naraz.

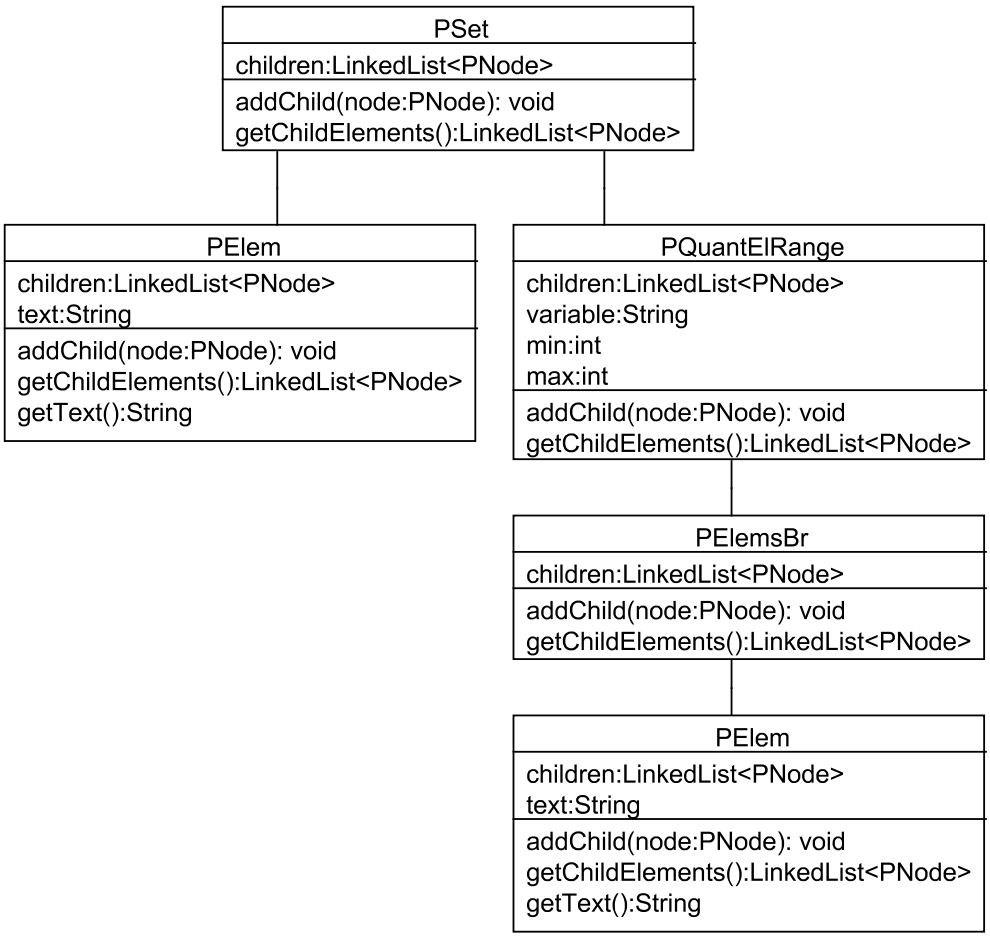
Pre zjednodušenie pri takto jednoduchom vstupe analyzátor vytvára samostatné tokeny len pre reťazce a čísla. Pre ostatné operátory a znaky je výsledkom názov príslušného operátora alebo znaku.

3.3.8 Syntaktická analýza

Úlohou syntaktickej analýzy je spracovať tokeny a zistiť, či daný vstup patrí do jazyka generovaného gramatikou. Syntaktický analyzátor pritom vytvára syntaktický strom. Jeden taký syntaktický strom je na obr.3.8.

Je viacero spôsobov, ako zistiť príslušnosť slova do jazyka, napr. zdola nahor, zhora nadol. Ja som si vybral metódu rekurzívne zostupného prehľadávania (recursive descent parsing). Hlavnou výhodou tejto metódy je jej jednoduchosť a názornosť, nevýhodou je, že nerozpoznáva všetky bezkontextové jazyky [7], gramatika generujúca jazyk nesmie obsahovať ľavú rekurziu a musí byť LL(1) gramatika. Pri rozpoznávaní slova z jazyka generovaného takouto gramatikou sa musí dať urobiť rozhodnutie pre nasledujúci krok iba na základe nasledujúceho tokenu.

Gramatika pre množinový zápis obsahuje rekurziu iba pri opakovaní. Rozšírenie v EBNF pre opakovanie sa dá zostrojiť pravidlom $\alpha \rightarrow \alpha A \mid \alpha$, čo nie je ľavá rekurzia. Z gramatiky vidieť, že rozhodnutie pre nasledujúci neterminál sa dá urobiť na základe



Obr. 3.8: Syntaktický strom

oddeľujúceho symbolu (čiarka, | ,hranaté zátvorky, kučeravé zátvorky). V prípade kvantifikácie sa dá rozhodnutie medzi hornou hranicou, rozsahom a príslušnosťou do množiny urobiť na základe názvu premennej resp. číslom ako prvým tokenom, následne podľa operátora.

Parsovací algoritmus sa zostrojí tak, že sa pre každý neterminál zostrojí samostatná metóda, ktorá rozpoznáva vstup a nesie názov neterminálu. Výskyt neterminálu v syntaxe (podľa gramatiky) sa nahradí volaním zodpovedajúcej metódy. Je vidieť, že toto sa dá zostrojiť priamo z danej gramatiky [9],[10].

Implementácia

Parser je implementovaný uvedeným postupom v triede `SetParser` v balíku `sk.fmfi.parsers`. Jednotlivé uzly stromu sú definované v balíku `sk.fmfi.parsers.PTree`. Všetky uzly sú odvodené od triedy `PNode`. Významné metódy triedy `SetParser` sú najmä:

1. `boolean parse(String setName,Graph graph)` – táto metóda dostane na vstup názov množiny a graf závislosti zatiaľ definovaných množín. Graf závislosti je orientovaný graf, kde ak množina A využíva prvky množiny B, potom smeruje z A do B orientovaná hrana. Tento graf sa v súčasnej verzii nevyužíva. Ak parsovanie prebehlo v poriadku, výsledok získame metódou `getPTree()`.

2. `boolean parseQuantified(LinkedList<String>,String q,Graph graph)` – táto metóda dostane na vstup zoznam prvok, kvantifikáciu a graf závislostí. Používa sa na generovanie kvantifikovaných pravidiel a riadkov δ funkcie. Výsledok získame metódou `getPTree()`.

3.3.9 Návrhový vzor visitor

V tejto časti popíšem návrhový vzor visitor, ktorý sa využíva v nasledujúcich fázach spracovania vstupu.

Návrhový vzor visitor je spôsob, ako oddeliť algoritmus od objektovej štruktúry. Dovoľuje nám definovať nové operácie na objektovej štruktúre, bez zmeny tried elementov, na ktorých majú byť vykonané.

Motivácia

Majme nejaký syntaktický strom ako výsledok syntaktickej analýzy a chceme na ňom vykonávať nejaké operácie, napr. overovanie platnosti rozsahu kvantifikačných premenných, vypisovanie jednotlivých prvkov množiny a pod. Visitor nám dovoľuje pridávať nové operácie bez toho, aby sa robili zmeny v triede `PNode` a v jej potomkoch. Jednotlivé operácie zapuzdrime do samostatných tried implementujúcich Visitor a potom inštanciu takejto triedy budeme posielať špeciálnej metóde `accept(Visitor v)` triedy `PNode`. Táto trieda následne zavolá metódu `visit(PNode n)` objektu v triedy `Visitor` s parametrom `this`. Visitor teda musí implementovať metódu `visit` so správnym typom parametra pre každého potomka triedy `PNode`. Pri volaní `visit()` pritom nemusíme vytvárať žiadne podmienky, správna metóda sa zavolá automaticky podľa typu volajúcej metódy.

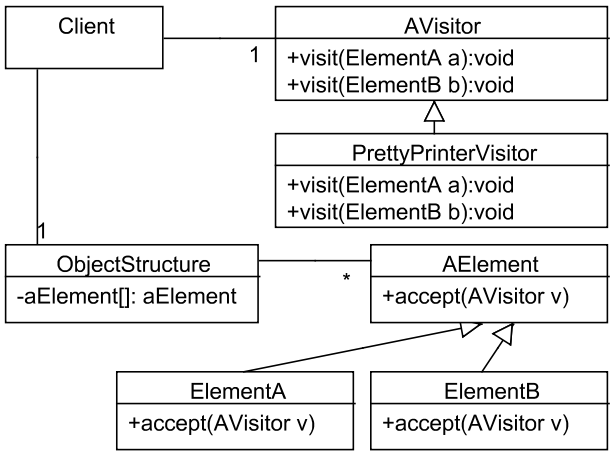
Štruktúru objektov vidíme na obr. 3.9
Pre tento návrhový vzor som sa rozhodol z nasledujúcich dôvodov:

1. Prehľadnosť. Jednotlivé operácie sú od seba oddelené a rovnako sú oddelené od objektovej štruktúry na ktorej ich vykonávame.

2. Oddelenie operácií. Toto mi umožnilo pracovať na jednej operácii a pritom neovplyvniť funkčnosť už implementovanej operácie.

3. Ľahké pridávanie nových operácií a experimentovanie s nimi.

Tento vzor však nie je bez chýb, za najväčší nedostatok považujem nutnosť pridávať metódu `visit` do rozhrania visitor a teda aj jeho implementujúcich tried po pridaní každého nového potomka triedy `PNode`. Je zrejmé, že stav momentálneho spracovania štruktúry si bude musieť visitor pamätať ako súčasť svojho stavu. Toto je na jednej



Obr. 3.9: Návrhový vzor visitor - štruktúra

strane výhoda, pretože objekty, ktoré visitor prechádza nemusia definovať stavové premenné pre každú operáciu, na druhej strane musia poskytnúť dostatok informácií objektom visitor, čo v prípade absencie spriatelených tried, čo je prípad Javy, môže viesť k narušeniu princípu zapuzdrenia.

Nasledujúce spracovanie syntaktického stromu je vykonávané práve pomocou visitorov.

3.3.10 Ďalšia kontrola vstupu

Po sparsovaní vstupu a vytvorení syntaktického stromu sa kontrolujú rôzne sémantické vlastnosti. Toto je implementované v triede `SetValidateVisitor`. Kontroluje sa, či je každá kvantifikačná premenná prítomná v prvku množiny, ktorý kvantifikuje a či sa premenná nevyskytuje v kvantifikácii dvakrát. Aj keď toto nie sú chyby, pretože v prvom prípade by bol prvok vypísaný bez dosadenia, avšak zrejme to nie to, čo používateľ zamýšľal. V druhom prípade to nie je chyba v prípade disjunktných rozsahov kvantifikačnej premennej, toto však momentálne nie je implementované a preto takýto vstup vylúčime.

3.3.11 Enumerácia prvkov množiny

V definícii syntaxe množinového zápisu bolo povedané, že je možné kvantifikovať jednotlivé prvky množiny. Pri nasledujúcom spracovaní triedami, ktoré implementoval kolega Dušan Baník, sme potrebovali mať zoznam jednotlivých prvkov po dosadení hodnôt za jednotlivé kvantifikačné premenné.

Pod enumeráciou prvkov množiny budem označovať postupné dosadenie hodnôt z rozsahu kvantifikačných premenných do príslušného prvku, vyhodnotenie výrazu a jeho vypísanie na výstup.

Toto je implementované ako visitor v triede `SetEvaluatorVisitor`. Táto trieda

môže ukladať jednotlivé prvky ako prvky množín pomocou kolekcie `HashSet`, alebo pomocou spájaného zoznamu `LinkedList` [9]. O spôsobe ukladania rozhoduje parameter konštruktora `isSet`. Ak je true, výsledok bude množina, inak zoznam. Toto je potrebné pri enumerácii kvantifikovaných pravidiel, keď môžeme mať zhodnú jednu stranu pravidla, ale rôznu druhú. Pri spájanom zozname a dodržaní vhodného poradia vypisovania prvkov množiny som sa vyhol implementácii ďalších tried. Stačí dohoda, že párne prvky spájaného zoznamu sú pravé strany a nepárne sú ľavé, postupne od prvého. Teda prvky sú enumerované v poradí ako boli zadané na vstupe. Toto riešenie sa dá zovšeobecniť na ľubovoľný počet prvkov. Nasledujúce objekty budem potrebovať pri popise činnosti visitora v nasledujúcom texte, preto stručne popíšem ich význam:

1. `PQuantifiedElement` – predstavuje kvantifikovaný prvok množiny. Sú v ňom uložené zoznam prvkov – ako jeden prvok, alebo viac prvkov ako `PElems` – a zoznam kvantifikácií – ako `PQuantification`.

2. `PQuantification` – zoznam kvantifikácií v `PQuantifiedElement`.

3. `PQuantElRange` – predstavuje celočíselný rozsah kvantifikovanej premennej.

Hodnoty premenných pri enumerácii sa získavajú pomocou iterátorov. Každá premenná, ktorá môže nadobudnúť nejaké hodnoty musí mať implementovaný príslušný iterátor.

Toto riešenie som zvolil z dôvodu enumerácie a dosadzovania aj iných ako číselných hodnôt, napríklad prvkov iných množín.

Pre číselné rozsahy hodnôt je iterátor implementovaný v triede `RangeIterator`, ktorá je súčasťou triedy `SetEvaluatorVisitor` ako vnorená trieda.

Pre samotnú enumeráciu sa udržuje v objekte `SetEvaluatorVisitor` tabuľka premenných, pre každý iterátor príslušný objekt z ktorého ho získame, usporiadaná n-tica iterátorov a tabuľka hodnôt. Udržovať tabuľku objektov je nutné z dôvodu absencie metódy na nastavenie iterátora na začiatok v rozhraní `iterator`. Pre optimálny výkon sú objekty v týchto tabuľkách hashované pomocou triedy `HashMap`.
Enumerácia prebieha podľa nasledujúceho postupu:
Vo `visit(PQuantifiedElement)`:

1. Vynuluj tabuľky premenných, hodnôt, objektov a iterátorov.

2. Zavolaj `accept(this)` na `PQuantification`.
   Toto volanie spôsobí naplnenie tabuliek hodnôt, premenných a iterátorov.

3. Opakovane volaj `accept` na podstrome s prvkami množiny, dokiaľ existujú ďalšie hodnoty iterátorov.

Vo `visit(PElem)`: dosaď hodnoty z aktuálnej tabuľky hodnôt za zhodujúce sa súradnice prvkov.

[9] Viac o kolekciách a iterátoroch napr. v [1]

Samotné prechádzanie cez n-ticu iterátorov prebieha postupným "zvyšovaním" hodnoty iterátora a prenosu do vyššieho rádu v prípade dosiahnutia konca. Následne sa všetky iterátory v nižšom ráde "vynulujú". Enumerácia končí, ak iterátor v najvyššom ráde dosiahol poslednú hodnotu. Toto teda kopíruje klasické počítanie s prenosom do vyššieho rádu. Takto sa postupne prejde cez všetky možné kombinácie hodnôt.

3.4 Návrh a implementácia simulátora DTS a DFA

V tejto časti popíšem návrh a implementáciu simulátora deterministického turingovho stroja (DTS) a deterministického konečného automatu (DKA).

Návrh

DTS som sa rozhodol navrhnúť ako prvý. Pomocou neho vieme jednoducho simulovať DKA a LBA, preto som pri návrhu bral do úvahy jednoduché vytvorenie odvodených tried na takúto simuláciu. V prípade DKA je to hlavne deštruktívne čítanie vstupu, v prípade LBA a DTS je to zmena políčka na páske a pohyb po páske o jedno políčko, v prípade DTS je to "predlžovanie" pásky v oboch smeroch (simulácia nekonečnej pásky). Na toto sa najlepšie hodí spájaný zoznam, všetky tieto operácie na ňom vieme vykonať v konštantnom čase, pretože hlava automatu sa pohybuje po vstupnej páske maximálne o jedno políčko.

Vytvoril som jednu triedu pre DTS ako základ a DFA som potom odvodil od tejto triedy. DFA bude vo výpočtovom kroku kontrolovať, či už nedočítal vstup a rušiť prázdne políčka pásky. V DTS si budeme musieť pamätať, či sa má zastaviť v akceptačnom stave. DTS dostane na vstup zoznamy prvkov jednotlivých množín a zoznam riadkov δ funkcie. V jednom kroku simulácie potom vyhľadá zhodujúci sa riadok δ funkcie s aktuálnym stavom a obsahom pásky, prepíše obsah pásky, nastaví nový stav a posunie hlavu.

Pretože jednotlivé δ funkcie sa mierne líšia rozhodol som sa zvoliť δ funkciu DTS ako základ. Pomocou nej potom vieme vytvoriť (simulovať) δ funkciu pre DKA. Táto funkcia bude zapisovať prázdny reťazec a vždy sa posunie o jedno políčko doprava.

Implementácia

Triedy implementujúce funkcie simulátora sa nachádzajú v balíku `sk.fmfi.automata`. Ide o triedy:

1. `TM` – trieda implementujúca DTS. Obsahuje metódy `setInput(String input)`, `boolean isCorrect()`, `int oneStep()`. Okrem toho sú tu pomocné metódy na zistenie aktuálnej pozície hlavy a obsahu pásky, ktoré sa využívajú pri vizualizácii simulácie.

   Simulácia sa nastavuje v troch krokoch:
   Najprv sa metódou `isCorrect` overí, či sú všetky množiny definované správne, overí sa podľa δ funkcie, či je TS deterministický.

V ďalšom kroku sa metódou `setInput` nastaví obsah vstupnej pásky. Táto metóda rozdelí vstup na jednotlivé políčka a tieto si zapamätá v spájanom zozname. Overuje pritom, či symbol na vstupnej páske je zo vstupnej abecedy. Na aktuálnu pozíciu hlavy si trieda `TM` uchováva obojsmerný iterátor. Tento sa nastaví na začiatku na prvé políčko.

Následne sa metódou `oneStep()` vykoná jeden krok výpočtu TS. Ak je v akceptačnom stave a nemusí dočítať pásku, nerobí nič, inak podľa δ funkcie prepíše pásku a nastaví iterátor. Ak je na začiatku alebo konci zoznamu, pridá blank (reťazec "B"). Metóda vráti stav automatu, teda či počíta, je v akceptačnom stave, alebo sa zasekol.

2. `Delta` – trieda predstavujúca jeden riadok δ funkcie DTS.

3. `DFA` – trieda odvodená od DTS simulujúca DKA. V konštruktore nastaví, že sa musí pred zastavením sa v akceptačnom stave dočítať celý vstup. Okrem toho prekrýva metódu `oneStep`, kde volá prekrytú metódu a následne vymaže z pásky prázdne reťazce. Ak je na páske len blank, znamená to, že sa celá dočítala a simulácia sa skončí.

4. `DFADelta` – trieda odvodená od `Delta`, predstavujúca jeden riadok simulácie DKA. Konštruktor predka volá so zápisom prázdneho reťazca a posunom hlavy o jeden krok vpravo.

3.5 Príklady použitia

Príklad: Zostrojte regulárnu gramatiku G, kde $L(G) = \{w|w \in \{a,b\}^+, \#_a(w)\ mod\ 10 = 0\}$.
Gramatika si bude v neterminále pamätať počet a a v každom kroku môže okrem a vygenerovať b bez zmeny počtu a.
Vytvoríme novú regulárnu gramatiku pomocou `File/New/Regular grammar`. Vyplníme príslušné textové polia:

```
G=(N,T,P,s)
N={s,[a,i]|1<=i<=10}
T={a,b}
P={
s->a[a,1]
s->bs
[a,i]->a[a,i+1]  1<=i<=8
[a,i]->b[a,i] 1<=i<=10
[a,9]->a
[a,9]->as
}
```

Príklad: Zostrojte DKA akceptujúci jazyk $L = \{w|w \in \{a,b\}^+$, vo w je druhé písmeno od konca b$\}$. Automat si bude v stave pamätať posledné prečítané 2 písmená,

akceptačné stavy budu tie, kde je toto písmeno *b*.
Budeme potrebovať pomocnú množinu vstupnej abecedy, vytvoríme preto novú globálnu množinu pomocou `File/New/Global set` a vyplníme:

```
Definition of set named: A
Set definition: {a,b}
```

Vytvoríme nový DKA pomocou `File/New/DFA` a vyplníme príslušné textové polia:

```
DFA: T=(K,S,d,q,F)
K={q,[q,a],[q,b],[a,b]|a in K|b in K}
S={[a],[b]}
F={[b,a]|a in K}
delta:
d(q,[a])=[q,a]
d(q,[b])=[q,b]
d([q,i],[j])=[i,j]|i in A|j in A
d([i,j],[k])=[j,k]|i in A|j in A|k in A
```

3.6 Formát súboru

V tejto časti popíšem formát súboru s uloženými dátami. Pri navrhovaní tohto formátu som sa snažil zabezpečiť aspoň minimálnu kontrolu korektnosti. Pritom som chcel dosiahnuť čitateľnosť bežným textovým editorom a možnú editáciu bez potreby púšťať program. Preto je súbor s dátami bežný ASCII textový súbor, kde jednotlivé zadefinované prvky[10] sú vypísané pod seba v textovej podobe. V pseudo[11] EBNF vyzerá štruktúra takto:

```
file = {sets},{grammars},{automata-dtm},{automata-dfa};
sets = "set",newline,numbern,stringn,mnozinan;
grammars = "grammar",newline,grtype,newline,numbern,gramatika
  ,newline,mnozinan-N,mnozinan-T,elbr,newline,"rules",newline
  ,numbern,{rules};
automata-dtm = "DTM",newline,numbern,numbern,dtm,newline,mnozinan-K
  ,mnozinan-S,mnozinan-W,numbern,string-delta,newline,mnozinan-F;
automata-dfa = "DFA",newline,numbern,numbern,dtm,newline,mnozinan-K
  ,mnozinan-S,numbern,string-delta,newline,mnozinan;
set_length = number;
grtype = "REGULAR"|"CONTEXT_FREE"|"CONTEXT_SENSITIVE"|"PHRASAL";
rules = stringn-left,stringn-right,numbern,[string-quant],newline;
numbern = number,newline;
mnozinan = numbern,mnozina,newline;
stringn = numbern,string,newline;
```

[10] gramatika, množina, automat
[11] Niektoré neterminály sú nedefinované, ale ich význam je zrejmý.

```
number = digit,{digit};
string = ASCII-STRING;
```

Treba poznamenať, že táto gramatika nemá generovať (a ani negeneruje) syntakticky správne súbory, má slúžiť len ako pomôcka pri popise formátu súboru. Využíva niektoré neterminály z predošlých definícií. Súbor pozostáva z troch častí: množiny, gramatiky, automaty. Najprv sú zapísané množiny v opakujúcich sa blokoch, každý začína reťazcom "set", potom gramatiky, začínajúce "grammar" a nakoniec automaty, TS začínajúce "DTM" a DKA začínajúce "DFA". Po úvodnom reťazci bloku nasleduje vždy celková dĺžka bloku, v prípade gramatík jej predchádza typ gramatiky. Každému reťazcu predchádza jeho dĺžka (neterminály končiace na n). Takto sú postupne vypísané všetky reťazce definície, v prípade pravidiel je to počet pravidiel, ľavá strana, pravá strana a kvantifikácia.

Kapitola 4

Záver

Cieľom práce bolo naprogramovanie softvéru na experimentovanie s gramatikami a automatmi, ktorý by slúžil ako pomôcka pri výučbe formálnych jazykov.

Výsledkom našej práce je softvér, ktorý umožňuje vytvárať všetky základné typy gramatík chomského hierarchie a skúšať vytvárať odvodenia vetných foriem, skúšať príslušnosť daného slova do jazyka generovaného regulárnou, kontextovou a bezkontextovou gramatikou. Z automatov softvér umožňuje vytvárať deterministické konečné automaty a deterministické turingové stroje a ich krokovanie na vstupe.

Na všetkých konštrukciách softvér umožňuje kvantifikáciu vstupu, a tak vytvárať zložitejšie gramatiky a automaty, ktorých ručné zadávanie by bolo veľmi zdĺhavé.

V práci bol prezentovaný opis použitých technológií pri vývoji a tímovej spolupráci, postup pri spoločnom návrhu aplikácie, následoval návrh a konečná implementácia grafického používateľského rozhrania a architektúry aplikácie. Samostatná časť bola venovaná spracovaniu vstupu a vyhodnocovaniu kvantifikovaných prvkov množín. V nasledujúcich častiach bol uvedený popis návrhu a implementácie simulácie DTS a DKA, jednoduché príklady použitia a opísaný formát súboru aplikácie.

Pri vývoji tohto softvéru som sa oboznámil s prácou s frameworkom Swing, prehĺbil som si znalosti programovacieho jazyka Java, no najväčší prínos mala pre mňa práca v prehĺbení si vedomostí z oblasti parsovania a následného spracovania vstupu a vyskúšania si tímovej spolupráce.

Literatúra

[1] Bruce Eckel. Myslíme v jazyku java, knihovna programátora. 2000. ISBN 80-247-9010-6.

[2] Michal Forišek. Formálne jazyky a automaty (skriptá). 2007.

[3] International Organization for Standardization. ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF. Technical report, Geneva, Switzerland, 1996. 12 pp.

[4] Sun Microsystems. Java jre. `http://java.sun.com/javase/downloads/index.jsp`.

[5] Sun Microsystems. swing-layout, 2007. `https://swing-layout.dev.java.net/`.

[6] Sun Microsystems. *The Swing Tutorial*, 2007. `http://java.sun.com/docs/books/tutorial/uiswing/`.

[7] CSc. RNDr. Ján Šturc. Kompilátory (skriptá). 1998.

[8] March Hare Software. Cvsnt. `http://www.cvsnt.org/`.

[9] wikipedia. Recursive descent parser, 2007. `http://en.wikipedia.org/wiki/Recursive_descent_parser`.

[10] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996. ISBN 0-201-40353-6.

Príloha na CD

K tejto práci je priložené CD, na ktorom sa nachádza preložený softvér, zdrojové kódy a dokumentácia vo formáte JavaDoc. Na spustenie softvéru treba mať nainštalovaný Sun Java JRE v. 1.5.0 [4]. Adresárová štruktúra na CD je nasledovná:

1. adresár bin obsahuje preloženú verziu aplikácie spolu s dávkovym súborom (run.bat) na jej spustenie
2. adresár doc obsahuje vygenerovaný javadoc projektu
3. adresár src obsahuje zdrojové súbory aplikácie