Aplikácia pre experimentovanie s formálnymi automatmi a gramatikami

BAKALÁRSKA PRÁCA

Dušan Baník

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

Informatika

Vedúca: RNDr. Dana Pardubská, PhD.

Bratislava, 2007

Čestne prehlasujem, že prácu som vypracoval sám a použil som len literatúru uvedenú v bibliografii

. . . .

// Acknowledgements
Poďakovanie

Ďakujem svojej školiteľke RNDr. Dane Pardubskej PhD. za rady a podnety, ktoré mi poskytla a ktoré výraznou mierou prispeli k skvalitneniu mojej bakalárskej práce.

Ďalej by som chcel poďakovať môjmu kolegovi Petrovi Havlíčkovi, s ktorým bola aplikácia vyvíjaná.

Okrem toho chcem poďakovať všetkým mojim priateľom a kamarátkam za to, že robia môj svet krajším. A samozrejme svojej rodine za úplne všetko.

Abstrakt

Výsledkom aktivít, ktoré boli predmetom tejto práce a práce Petra Havlíčka je aplikácia, ktorá umožňuje vytváranie nových gramatík s parametrizovanými neterminálmi, terminálmi a pravidlami. Je vhodná na experimentovanie s odvodením v jednej zo štyroch typov gramatík. Obsahuje naprogramované algoritmy, ktoré zisťujú príslušnosť slova w do regulárneho, bezkontextového alebo kontextového jazyka. Keďže pre frázové jazyky nie je tento problém rozhodnuteľný, je implementovaná heuristika, ktorá v niektorých prípadoch tento problém rozhodne. Aby sa používateľ mohol vrátiť k už začatým problémom, aplikácia GRASIM umožňuje ukladanie a načítavanie gramatík zo/do súboru.

Cieľom naprogramovanej aplikácie je, aby študenti lepšie porozumeli rôznym úlohám z oblasti formálnych jazykov a automatov.

Obsah

1 Úvod **7**

2 Základné definície a označenia **8**
2.1 Gramatiky 8
2.2 Triedy jazykov 9

3 Postup riešenia **11**

4 Realizácia etáp **13**
4.1 Zisťovanie požiadaviek 13
4.2 Analýza 13
4.3 Postupy, metódy a nástroje 15
4.3.1 Eclipse 15
4.3.2 UML 16
4.3.3 Extrémne programovanie 16
4.4 Návrh systému 17
4.5 Programovanie 20
4.5.1 Rule class 20
4.5.2 Grammar class 21
4.5.3 Regular class 23
4.5.4 Contextfree class 23
4.5.5 Contextsensitive class 24
4.5.6 Phrasal class 24
4.6 Testovanie 25

5 Použité algoritmy **26**
5.1 Cocke-Younger-Kasami (CYK) algoritmus 27
5.2 CYK implementovaný v aplikácii GRASIM 28
5.2.1 Konštruktor 28
5.2.2 go 28
5.2.3 go2 28

6 Záver **31**

7 Príloha **32**

Kapitola 1

Úvod

Teória formálnych jazykov a automatov je jeden z najdôležitejších predmetov v teoretickej informatike, ktorý pomáha študentom vnímať veci abstraktnejšie a tým rozvíjať ich myslenie. Formálne jazyky môžu byť špecifikované rôznymi spôsobmi: slová vygenerované nejakou formálnou gramatikou, slová zhodujúce sa s regulárnym výrazom, slová akceptované automatom ako napr. turingovým strojom alebo konečným automatom, slová rozhodnuté algoritmom s odpoveďou áno/nie. Hlavnou a zároveň najťažšou úlohou je študovať rozhodnuteľnosť problémov na rôznych typoch automatov a gramatík. Mnoho študentov má však problém pochopiť abstrakciu na jednoduchých modeloch a to vedie k zavrhnutiu tohto predmetu.

Téma tejto bakalárskej práce bola vypísaná ako tímová, pričom druhým riešiteľom bol môj spolužiak Peter Havlíček. Zameranie práce je prakticko-implementačné a poslaním nie je skúmanie v teoretickej rovine.

Pod vedením RNDr. Dany Pardubskej PhD., sme sa rozhodli naprogramovať aplikáciu s menom GRASIM, ktorá pomáha predstaviť si formálne konštrukcie, niektoré problémy ako príslušnosť slova do jazyka, alebo či napísaná gramatika generuje rovnaký jazyk. Aplikácia GRASIM má slúžiť ako "dynamický papier", ktorý umožňuje experimentovať, vizualizovať a randomizovať odvodenia v príslušnej gramatike. Bližšie upresnenie požiadaviek na aplikáciu GRASIM je v kapitole 4.

Je dôležité uvedomiť si, že aplikáciu GRASIM sme vyvíjali vo dvojici, a preto bolo nevyhnutné vyriešiť problémy ako efektívna komunikácia, rýchla výmena naprogramovaných častí, zhoda v riešení problémov. Najdôležitejšou časťou bolo rozdelenie úloh medzi dvoch riešiteľov a navrhnutie interakcie medzi nimi. Tento spôsob rozdelenia úloh je podrobnejšie uvedený v kapitole 3.

Kapitola 2

Základné definície a označenia

2.1 Gramatiky

Prostredie, ktoré sme naprogramovali, slúži na podporu výučby formálnych jazykov a automatov, a preto je dôležité zhrnúť základné pojmy, definície a označenia, ktoré sa intenzívne používajú v mojej práci a v aplikácii GRASIM.

Definícia 2.1.1. *Abeceda je konečná, neprázdna množina symbolov. Zvyčajne ju označujeme* Σ.

Definícia 2.1.2. *Slovo nad abecedou* Σ *je konečná postupnosť symbolov. Prázdnu postupnosť písmen nazývame prázdne slovo a označujeme ju* ε.

Definícia 2.1.3. *Dĺžka slova je dĺžka postupnosti, ktorá ho vytvára. Dĺžku slova* w *označujeme* $|w|$.

Definícia 2.1.4. *Podslovo* u *slova* $v = a_1a_2...a_n$ *je ľubovoľná súvislá postupnosť* $a_ia_{i+1}...a_k$ *taká, že platí* $1 \leq i \leq k \leq n$.

Definícia 2.1.5. *Jazyk nad abecedou* Σ *je ľubovoľná množina slov nad abecedou* Σ.

V informatike a lingvistike je gramatika presný opis jazyka. Existujú dva základné spôsoby, ako špecifikovať jazyky.

1. Automaty - všetky typy automatov pracujú na rovnakom princípe: čítajú vstupné slovo, niečo počítajú a prípadne časom vstupné slovo akceptujú. Jazyk akceptovaný takýmto automatom je množina slov, ktoré vyhlási za "dobré".

2. Gramatiky - všetky typy gramatík slová generujú. Jazykom generovaným gramatikou sú všetky vetné formy, ktoré môžu byť odvodené z počiatočného neterminálu, používaním množiny pravidiel, až kým vo vetnej forme nie je žiaden neterminál, to znamená, že tieto vetné formy sú slová.

Hlavný rozdiel medzi automatmi a gramatikami je v tom, že automat má na začiatku slovo w, ktoré buď akceptuje alebo nie. Gramatika má na začiatku len počiatočný neterminál, z ktorého sa snaží odvodiť slovo. Každé slovo, ktoré gramatika odvodí, je potom z jazyka. Základné typy gramatík sú uvedené v nasledujúcich definíciách.

Definícia 2.1.6. *Frázová gramatika je usporiadaná štvorica* $G = (N, T, P, \sigma)$*, kde* N *je množina neterminálov,* T *je množina terminálov* ($N \cap T = \emptyset$)*,* $\sigma \in N$ *počiatočný neterminál a* $P \subseteq ((N \cup T)^* N (N \cup T)^* \times (N \cup T)^*)$ *je konečná množina prepisovacích pravidiel.*

Definícia 2.1.7. *Kontextová gramatika je taká frázová gramatika, v ktorej pre každé pravidlo* $u \rightarrow v$ *platí* $|u| \leq |v|$.

Poznámka 2.1.1. *Keďže v kontextovej gramatike pravá strana každého pravidla musí byť aspoň tak dlhá ako ľavá, nevie žiadna kontextová gramatika vygenerovať prázdne slovo. Preto si zadefinujeme rozšírenú kontextovú gramatiku, ktorá bude mať možnosť v prvom kroku odvodenia vygenerovať prázdne slovo. Avšak musíme zabezpečiť, aby toto pravidlo nemohlo byť použité neskôr v odvodení.*

Definícia 2.1.8. *Rozšírená kontextová gramatika je taká frázová gramatika, v ktorej* $P \subseteq ((N \cup T)^+ \times (N \setminus \{\sigma\}) \cup T)^+) \cup \{\sigma \rightarrow \varepsilon\}$ *a pre každé* $\pi = (u \rightarrow v) \in P, \pi \neq (\sigma \rightarrow \varepsilon)$

Definícia 2.1.9. *Bezkontextová gramatika je taká frázova gramatika, v ktorej platí* $P \subseteq N \times (N \cup T)^*$

Definícia 2.1.10. *Regulárna gramatika je taká frázova gramatika, v ktorej pre pravidlá platí* $P \subseteq N \times T^*(N \cup \{\varepsilon\})$

Definícia 2.1.11. *Jazyk generovaný gramatikou G je množina* $L(G) = \{w \in T^* | \sigma \Rightarrow_G^* w\}$

Definícia 2.1.12. *Krok odvodenia v gramatike G je binárna relácia* $\Rightarrow_G$ *na* $(N \cup T)^*$ *definovaná* $x \Rightarrow_G y$ *práve vtedy, keď existujú slová* w_1*,* w_2 *a pravidlo* $(u \rightarrow v) \in P$ *také, že* $x = w_1 u w_2$ *a* $y = w_1 v w_2$.

Označenie 2.1.1. *Budeme často písať* $\Rightarrow$ *namiesto* $\Rightarrow_G$*, ak bude zrejmé, o ktorú gramatiku sa jedná.*
$\Rightarrow^+$ *bude označovať tranzitívny uzáver relácie.*
$\Rightarrow^*$ *bude označovať reflexívno-tranzitívny uzáver relácie* $\Rightarrow$.

Definícia 2.1.13. *Odvodenie* $\sigma \Rightarrow u_1 \Rightarrow u_2 \Rightarrow ... \Rightarrow u_n$ *je postupnosť krokov odvodení. Dĺžka odvodenia je počet týchto krokov.*

Definícia 2.1.14. *Vetná forma je slovo z* $(N \cup T)^*$*, ktoré vieme odvodiť z počiatočného neterminálu používaním pravidiel.*

2.2 Triedy jazykov

V aplikácii GRASIM sme implementovali všetky štyri typy gramatík. Každý z týchto typov gramatík, ktorý sme už zadefinovali a naprogramovali, nám určuje jednu triedu jazykov.

1. Frázové $\mathcal{L}_{RE} = \{$L | existuje frázová gramatika G, taká že $L = L(G)\}$

2. Kontextové $\mathcal{L}_{ECS} = \{L \cup \{\varepsilon\} |$ existuje kontextová gramatika G, taká že $L = L(G)\}$

3. Bezkontextové $\mathcal{L}_{CF} = \{L \mid$ existuje bezkontextová gramatika G, taká že $L = L(G)\}$

4. Regulárne $\mathcal{R} = \{L \mid$ existuje regulárna gramatika G, taká že $L = L(G)\}$

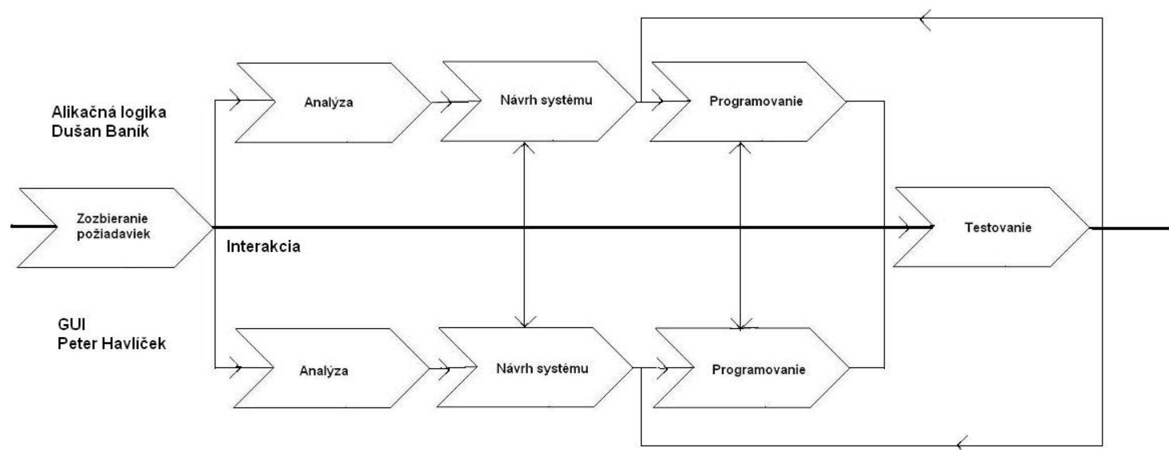
Tieto triedy tvoria Chomského hierarchiu. Je dôležité uvedomiť si, aké vzťahy platia medzi jednotlivými triedami jazykov.

$$\mathcal{L}_{RE} \supseteq \mathcal{L}_{ECS} \supseteq \mathcal{L}_{CF} \supseteq \mathcal{R}$$

Kapitola 3

Postup riešenia

Pri riešení problémov vývojového procesu sme použili metódu GRAPPLE (Guidelines for Rapid APPlication Engineering pokyny pre rýchly návrh a vývoj aplikácií). Táto metóda bola vytvorená z myšlienok mnohých iných metód. Základnými myšlienkami sú: prispôsobiteľnosť, pružné pokyny a zjednodušená kostra vývojového procesu. Bližšie informácie o metóde GRAPPLE sa môžete dozvedieť [6].
Vývoj aplikácie GRASIM pozostával z viacerých fáz, ktoré sú popísané na diagrame 3.1. V tejto sekcii si nevysvetlíme, ako sú jednotlivé etapy nami realizované, ale zameriame sa predovšetkým na procesy, ktoré sú realizované v jednotlivých fázach.



Obr. 3.1: Etapy pri vývoji aplikácie

1. Zozbieranie požiadaviek. Ak by sme priradili každej etape vývoja aplikácie relatívnu dôležitosť, tak zozbieranie požiadaviek by malo prioritu číslo jedna. Ak totiž vývojári neporozumejú, čo zadávateľ požaduje, nemôžu vyvinúť správny systém. Najlepšou myšlienkou je pochopiť jednotlivé procesy, ktoré vyžaduje

zadávateľ. Dôležité je porozumieť problematike a získať potrebnú slovnú zásobu z oblasti terminológie zadávateľa. Extrémne dôležitým krokom je identifikovať základné vlastnosti, funkčnosti a kompetencie, požadované od aplikácie. Cieľom tejto etapy je transformovať neformálne požiadavky do štruktúrovaného opisu, podľa ktorého sa budú tieto požiadavky analyzovať.

2. Analýza. V tejto etape tím podrobne analyzuje zozbierané požiadavky. Zámer analýzy je pochopenie systému a jeho štruktúry v zmysle problémov, ktoré má riešiť. Prvý krok pozostáva z identifikovania prípadov použitia [4] vytváraného systému a následne navrhnutie prvotného modelu. Dôležitou časťou analýzy je aj premyslenie metodík a softvérových produktov, ktoré sa budú podieľať na vývoji aplikácie.

3. Návrh systému. V tejto časti sa pracuje na architektúre systému, identifikujú sa základné časti ako sú triedy, rozhrania a spôsob interakcie medzi nimi. V ďalšej úrovni návrhu sa ide hlbšie do detailov a jasne sa špecifikujú metódy, ich parametre a návratové hodnoty. Dobrým návrhom architektúry je možné realizovať programovaciu fázu efektívnejšie a tým ušetriť čas. Výsledkom tejto fázy sú UML modely, podľa ktorých sa začína programovať samotný systém.

4. Programovanie. Po dostatočnej analýze a návrhu, by sa táto fáza mala realizovať ľahko a rýchlo. Pomocou diagramov tried a objektov programátori transformujú podrobnú špecifikáciu jednotlivých modulov a ich vzájomných vzťahov do programovacieho jazyka.

5. Testovanie. Veľmi dôležitá fáza pri vývoji softvéru, ktorá overuje, či naprogramované vlastnosti spĺňajú používateľské požiadavky.

Fázy 4 a 5 sa stále opakujú, a tým sa zabezpečuje korektné napredovanie vo vývoji.

Kapitola 4

Realizácia etáp

4.1 Zisťovanie požiadaviek

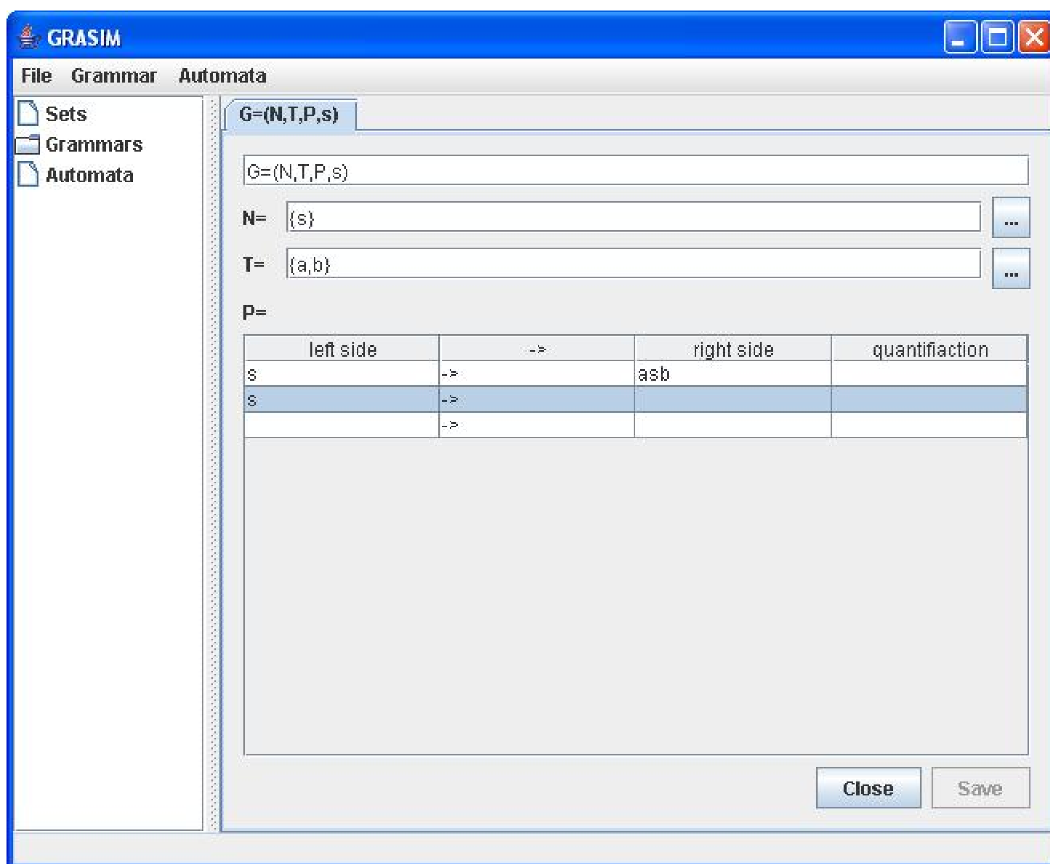
Táto etapa bola realizovaná spoločne s mojím kolegom Havlíčkom. Pred samotným návrhom sme sa dohodli na funkcionalite, ktorú budeme vyžadovať od aplikácie GRASIM. Aplikácia má slúžiť ako "dynamický papier", ktorý má študentom zjednodušiť prácu s možnosťou vizualizácie odvodenia. Požiadavky, ktoré sme zozbierali sú:

1. vytvárať nové gramatiky štyroch typov (regulárne, bezkontextové, kontextové, frázové),
2. experimentovať s odvodením v jednej z už vytvorených gramatík,
3. možnosť randomizovať odvodenie v gramatike,
4. schopnosť parametrizovať terminály, neterminály, pravidlá,
5. načítavať a ukladať gramatiky z/do súboru,
6. rozhodovať príslušnosť slova do (regulárneho, bezkontextového, kontextového) jazyka,
7. jednoduchá heuristika, ktorá niekedy rozhodne, že či slovo je z frázového jazyka,
8. jednoduché a prehľadné grafické prostredie,
9. generovanie slov zo zadaného jazyka,
10. porovnávanie jazykov generovaných dvomi gramatikami.

4.2 Analýza

Po dôkladnom zozbieraní požiadaviek sme si prácu rozdelili do ortogonálne disjunktných častí, ktoré boli implementované paralelne, čo podstatne zrýchlilo a zlepšilo kvalitu našej práce. Tieto dve časti sú:

1. Grafic user interface (GUI). Najdôležitejšou úlohou GUI je sprehľadniť, zjednodušiť, ale pritom zachovať estetické rozmiestnenie grafických komponentov, z ktorých je prostredie zložené. Používateľské rozhranie slúži na zadávanie vstupu, rozpoznávanie korektne zadaných vstupov, experimentovanie s odvodeniami slov v gramatike, zadávanie a testovanie príslušnosti slov do daného jazyka. Treba si uvedomiť, že rozparsovať vstup nie je vôbec jednoduché pri možnosti zadávania parametrizovaných neterminálov, terminálov ba dokonca i pravidiel, ktoré môžu byť kvantifikované. Po skontrolovaní syntaktických vlastností vstupu sa kontrolujú logické vlastnosti vstupu, ako napr. či pravidlá sú v požadovanom tvare. Podrobnejšie informácie o používateľskom rozhraní, ktoré naimplementoval môj kolega Peter Havlíček, sa dozviete v jeho práci, kde budú vysvetlené základné algoritmy na parsovanie vstupu, prostredie, v ktorom bolo používateľské rozhranie vyvinuté. Na diagrame 4.1 môžeme vidieť, ako vyzerá grafické prostredie na zadávanie vstupu. Na tomto diagrame je uložená bezkontextová gramatika.

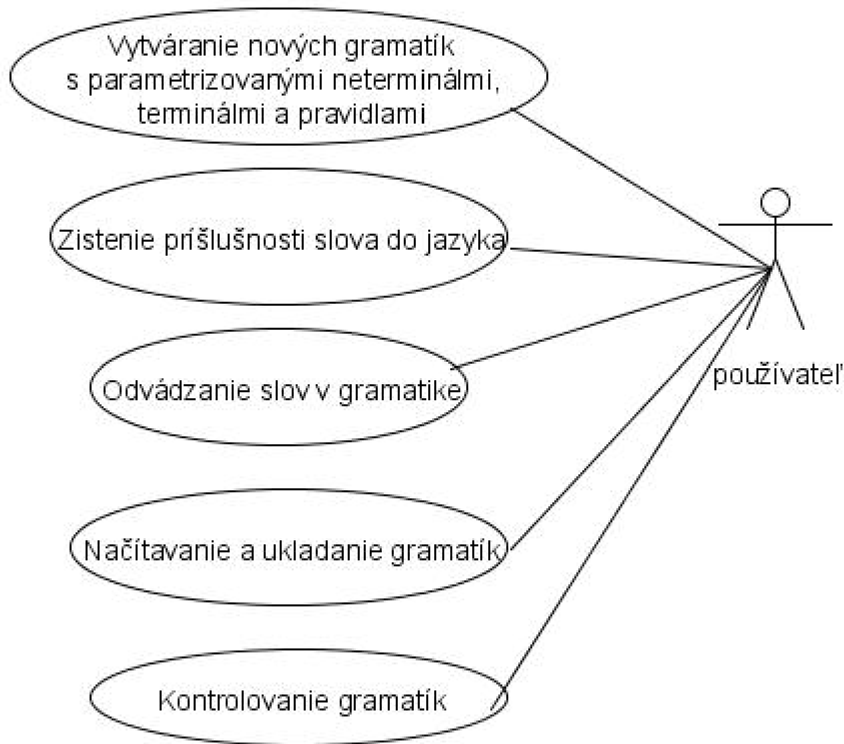


Obr. 4.1: Používateľské rozhranie

2. Aplikačná logika. Túto časť aplikácie som navrhol a naprogramoval ja. V nasledujúcej sekcii si opíšeme, ako táto logika bola navrhnutá a v ďalšej kapitole sa bližšie zoznámime s funkcionalitou. Takisto si povieme o triedach, ktoré tvoria

jadro aplikácie GRASIM.

Po rozdelení celku na hore uvedené časti, každý z nás začal analyzovať svoju podčasť. V nasledujúcom texte sa zameriam len na aplikačnú logiku. Dôležitou časťou analýzy je zostrojenie usecase diagramu [4]. Tento diagram 4.2 znázorňuje interakciu medzi používateľom a aplikáciou.



Obr. 4.2: UseCase Diagram

4.3 Postupy, metódy a nástroje

Táto časť je venovaná nástrojom a postupom, ktoré sme použili pri vývoji aplikácie GRASIM. Nejde nám len o vymenovanie jednotlivých produktov a metód, ktoré sme použili, ale najmä o vysvetlenie dôvodov, ktoré nás viedli k ich použitiu. Je potrebné si uvedomiť, že aplikáciu sme vyvíjali dvaja, a preto sme hľadali spôsoby, ktoré maximálne zefektívnia, skvalitnia a hlavne skrátia čas na výmenu častí, ktoré sme naprogramovali osobitne. Pri vyberaní použitých produktov sme kládli veľký dôraz na tímovú prácu.

4.3.1 Eclipse

Vývojové prostredie Eclipse je jedným z mnohých IDE (Integrated Development Environment). IDE je softvér, ktorý pomáha programátorom vyvíjať softvér a zvyčajne

pozostáva z editora, zdrojového kódu, kompilátora, nástrojov na automatizovanie zostavovania a debuggeru. Ďalšie prostredia, ktoré sa pri vývoji softvéru dajú použiť, sú napr. JBuilder alebo NetBeans. V tejto práci bolo použité prostredie Eclipse, pretože má veľmi veľa príjemných výhod, ako sú napr. rôzne typy perspektív, prehľadnosť a doterajšia prax s týmto prostredím. Perspektíva je schéma rozmiestnenia pracovného prostredia Eclipsu. Pri vývoji aplikácie sme používali tieto perspektívy:

1. Java (východisková perspektíva): vhodná najmä pre písanie Java aplikácií.

2. CVS Repository Exploring: pre tímovú prácu a pre synchronizovanie sa s Concurent Version System (CVS).

3. Team Synchronizing: ďalšia perspektíva pre tímovú prácu.

4. Debug: Táto perspektíva je veľmi vhodná na hľadanie chýb a kontrolovanie medzivýsledkov. Zvyčajne otvorená, keď sa v aplikácii hľadajú chyby.

Viac o vývojovom prostredí Eclipse je v [7].

4.3.2 UML

Unified Modeling Language alebo UML je v softvérovom inžinierstve univerzálny grafický jazyk na vizualizáciu, špecifikáciu, navrhovanie a dokumentáciu programových systémov. UML ponúka štandardný spôsob zápisu návrhov systémov vrátane konceptuálnych prvkov, ako sú business procesy a systémové funkcie, ale aj konkrétnych prvkov, ako sú príkazy programovacieho jazyka, databázové schémy a znovupoužiteľné programové komponenty. Jazyk UML bol navrhnutý, aby podporoval objektovo orientovaný prístup k analýze, avšak má oveľa širšie využitie, ktoré vyplýva z jeho už zabudovaných mechanizmov [5]. UML neobsahuje exaktný spôsob používania, ani neobsahuje metodiku, ako analyzovať, špecifikovať alebo navrhovať programové systémy. V UML sa väčšinou používajú grafické zápisy, ktoré opisujú dizajn softvérového produktu. Používanie tohto jazyka pomáha ľahko porozumieť dizajnu, skúmať potenciálne lepšie návrhy a dokáže zhodnotiť výsledný dizajn. V aplikácii GRASIM bol tento jazyk použitý na dizajn hlavnej a zároveň najdôležitejšej časti, ktorá reprezentuje celú aplikačnú logiku. UML model aplikácie je bližšie vysvetlený v kapitole návrh systému, kde sú aj pridané diagramy z analýzy, ktorú som spravil.

4.3.3 Extrémne programovanie

Extrémne programovanie je prístup k tvorbe softvéru, pri ktorom sa využíva jednoduchosť, komunikatívnosť a spätná väzba. Tento postup je hlavne určený pre malé tímy, pre ktoré je nevyhnutné vyvinúť softvér v krátkom časovom horizonte pri dynamicky meniacich sa požiadavkách. Keďže pri vývoji aplikácie GRASIM sme aj my využili prvky extrémneho programovania, uvedieme si stručnú charakteristiku niektorých odporúčaní, tak ako sú charakterizované v literatúre [3].

1. Plánovanie - v procese plánovania treba jasne ohodnotiť požadované vlastnosti softvéru. Treba si stanoviť priority a rozhodnúť, ktoré časti je potrebné vyriešiť, a ktoré nie sú až tak podstatné. Význam takéhoto plánovania je v tom, že umožňuje efektívne vedenie projektu k úspechu.

2. Rýchly výstup - programátorský tím zabezpečí čo najskôr prvý funkčný kód, ktorý sa v krátkych cykloch rozširuje, zlepšuje so zohľadnenými pripomienkami používateľa.

3. Metafora - programátorský tím používa spoločnú terminológiu. Je to základ efektívnej komunikácie v tíme.

4. Jednoduchosť - výsledný program by mal byť čo najjednoduchší a najprehľadnejší, ale pritom musí spĺňať aktuálne požiadavky používateľa.

5. Testovanie - programátorský tím po celý čas vývoja venuje zvláštnu pozornosť overovaniu vytvoreného kódu. Testy sa pripravujú dokonca skôr, než vlastný program. Testy zaručujú, že výstup programu bude správny a používateľské požiadavky budú splnené.

6. Prebudovávanie - počas celého vývoja sa programové riešenie vylepšuje. Dosahuje sa to "vyčisťovaním" programu - odstraňujú sa duplicity, zjednodušuje sa riešenie, ale zachováva sa kompletnosť. Využíva sa k tomu nepretržitá komunikácia v rámci riešiteľského tímu, ale aj komunikácia s používateľom.

7. Párové programovanie - programátori píšu kódy programov v dvojiciach pri jednom stroji. Mnohé experimenty potvrdili, že v dvojiciach sa vytvára softvér lepšie, rýchlejšie a s menej chybami, než v prípade samostatnej práce jednotlivca.

8. Kolektívne vlastníctvo - všetky zdrojové kódy patria všetkým programátorom. To umožňuje tímu rýchlo napredovať v riešení, pretože úpravy a zmeny je možné robiť bez zbytočných problémov.

9. Neustála integrácia - softvérový produkt sa kompiluje a spája veľakrát počas jedného dňa. To udržiava najnovšiu funkčnú verziu a rýchle napredovanie v riešeniach.

10. Štandardy kódovania - pre efektívnu prácu v tíme je dôležité písať zdrojový kód rovnakým spôsobom podľa dopredu stanovených zásad a pravidiel.

Poznámka 4.3.1. *Ďalšie použité vývojové nástroje ako CVS, Netbeans, grafic layout, . . . , ktoré boli tiež použité pri vývoji, opísal kolega Peter Havlíček.*

4.4 Návrh systému

Pri návrhu oboch častí aplikácie GRASIM sme sa dohodli, ako budú naše časti kolaborovať. Ak by sme tieto časti robili bez vzájomnej koordinácie, tak po spojení

naprogramovaných častí by sa aplikácia nedala používať. Kvôli tomu sme sa snažili navrhnúť obe časti tak, aby komunikácia medzi aplikačnou logikou a GUI bola čo najjednoduchšia. Dizajn bol navrhnutý ešte pred zahájením programovacej fázy. V dôsledku výskytu chýb sa skutočnosť jemne odlišuje od pôvodne navrhnutých modelov. Tento rozdiel je v tom, že v pôvodnom modeli je pravidlo reprezentované dvojicou stringov. V skutočnosti sa však pravidlo reprezentuje aj dátovou štruktúrou vector, ktorá umožňuje rýchly prístup k neterminálom a terminálom. Dôvodom vzniku tejto chyby bolo zlé zhodnotenie vlastností triedy `Rule`. UML model (diagram 4.3) reprezentuje vzťahy medzi balíkmi. Ku každému balíku si uvedieme stručnú charakteristiku jeho účelu, aby bolo jasné, čo reprezentuje.

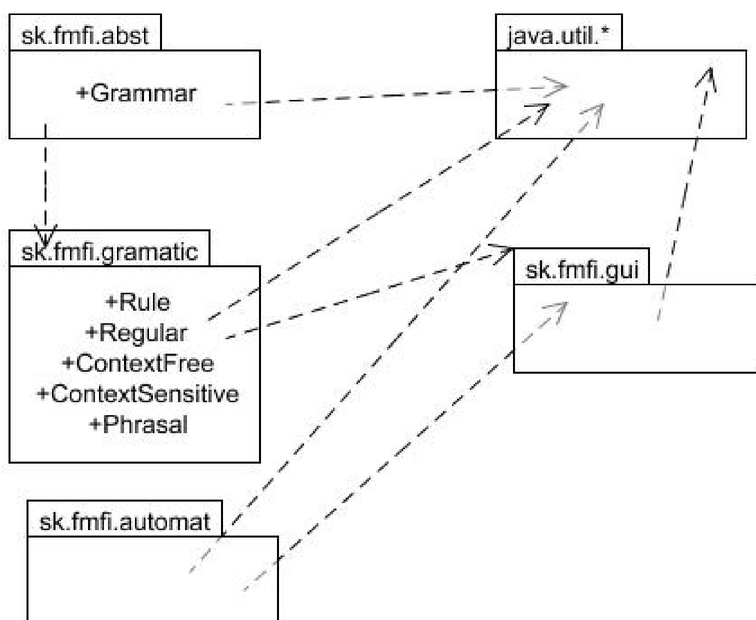
1. `java.util.*` Tento balík slúži na importovanie zložitejších dátových štruktúr a algoritmov, ktoré sú v aplikácii využité.

2. `sk.fmfi.abst` V tomto balíku je abstraktná trieda, ktorá reprezentuje všetky typy gramatík, ktoré sú v aplikácii implementované.

3. `sk.fmfi.gui` Slúži na vykresľovanie grafických prvkov a komponentov, z ktorých je používateľské rozhranie zostavené.

4. `sk.fmfi.gramatic` Obsahuje všetky štyri typy gramatík a triedu `Rule`, ktorá reprezentuje pravidlo.

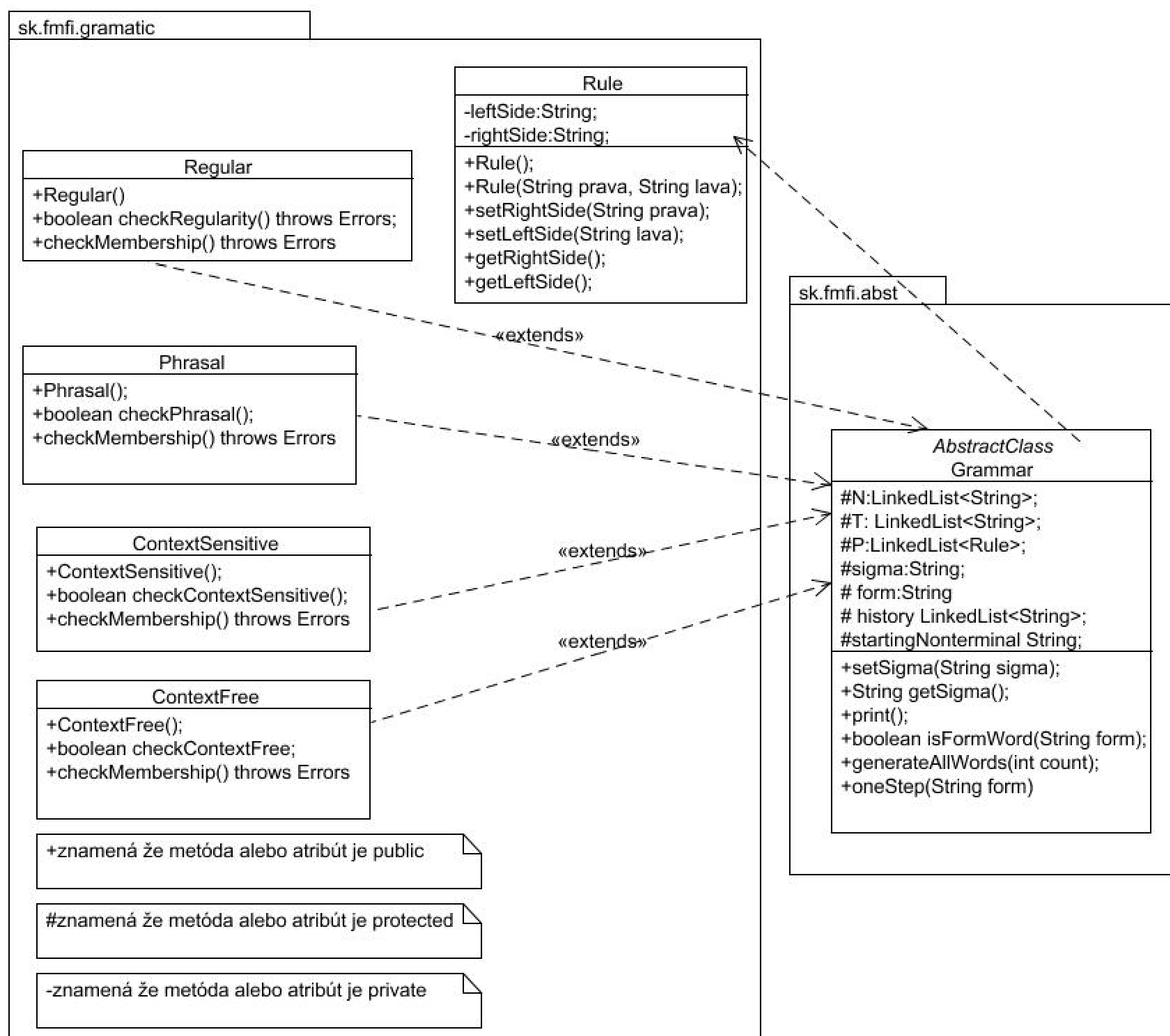
5. `sk.fmfi.automata` Obsahuje dva typy automatov, ktoré sú naimplementované v našej práci. Podrobnejšie o týchto automatoch vysvetlí Peter Havlíček.



Obr. 4.3: Vzťahy medzi balíkmi

Podľa tejto analýzy som navrhol podrobnejšie jednotlivé triedy ako sú `Rule`, `Grammar`, `Regular`, `ContextFree`, `ContextSensitive`, `Phrasal`. Nevyhnutnou súčasťou bolo tiež vybrať vhodné dátové štruktúry na reprezentovanie potrebných dát ako sú napr. pravidlá, neterminály, terminály, vetná forma, ...

UML model (diagram 4.4) sa zvyčajne nazýva class diagram a reprezentuje podrobnú architektúru aplikačnej logiky. Z obrázka by malo byť zrejmé, aké sú vzťahy medzi triedami napr. `Regular` extends `Grammar` znamená, že regulárna gramatika má všetky funkcie abstraktnej triedy `Grammar`. Medzi najdôležitejšie triedy patria `Grammar` a `Rule`. Trieda `Rule` reprezentuje všetky druhy pravidiel. Štyri triedy `Regular`, `ContextFree`, `ContextSensitive`, `Phrasal` reprezentujú štyri základné typy gramatík. Pri programovaní týchto tried som vychádzal z modelu, ktorý reprezentuje nasledujúci obrázok.



Obr. 4.4: Vzťahy medzi triedami

4.5 Programovanie

Podľa modelov sme začali programovať každý svoju časť. V tejto časti si podrobne povieme, ako sú jednotlivé triedy z aplikačnej logiky naprogramované a akú funkcionalitu majú metódy v týchto triedach.

4.5.1 Rule class

Táto trieda je použitá na reprezentáciu všetkých typov pravidiel. Z návrhu vidíme, ako je táto trieda implementovaná, ktoré sú jej atribúty a metódy. Pravidlo je reprezentované dvojicou `leftSide, rightSide`. `leftSide` je string, ktorý predstavuje ľavú časť pravidla, tak isto `rightSide` je string reprezentujúci pravú stranu pravidla. Keďže tieto atribúty sú privátne (nie sú dostupné z inej triedy), zadefinovali sme si public metódy na prácu s privátnymi atribútmi. Tieto metódy sa volajú getters a setters:

1. `public void setLeftSide(String leftSide)` nastaví ľavú stranu pravidla podľa parametra `leftSide`, ktorý dostane metóda na vstup.
2. `public void setRightSide(String rightSide)` nastaví pravú stranu pravidla podľa parametra `rightSide`, ktorý dostane metóda na vstup.
3. `public String getLeftSide()` vráti ľavú stranu pravidla.
4. `public String getRightSide()` vráti pravú stranu pravidla.

Nové pravidlo sa v objektovom programovaní vytvára pomocou konštruktora `public Rule(String leftSide, String rightSide)`, ktorého meno sa zhoduje s menom triedy. Tento konštruktor má dva parametre - ľavú a pravú stranu pravidla a volá sa pri vytváraní nového objektu. Mnou navrhnutý model nebol úplne dostatočný, pretože v pravidlách nebolo možné jednoducho zistiť, kde začína a kde končí neterminál alebo terminál. Preto som doprogramoval ďalšie metódy, ktoré umožňujú ľahký a prehľadný spôsob k prístupu pravej alebo ľavej časti pravidla.

Poznámka 4.5.1. *Tento problém nastal preto, lebo v pôvodnom modeli som predpokladal, že terminál ako aj neterminál reprezentovaný vo formálnych jazykoch a automatoch je len jeden symbol, lenže v počítači nie je možné reprezentovať neterminál* $N_{i,k}$ *jedným symbolom, a preto sú indexované neterminály alebo terminály v upravenom návrhu reprezentované v hranatých zátvorkách, napr. neterminál* $N_{i,k}$ *bude reprezentovaný [N,i,k], čo však už nie je len jeden symbol v stringu, a preto vznikli menšie komplikácie, ktoré boli rýchlo rozanalyzované a vyriešené doprogramovaním metód, ktoré sú nižšie charakterizované.*

1. `public static Vector<String> decode(String w)` táto metóda dostane na vstupe string a rozloží ho na neterminály a terminály. Táto metóda sa volá v konštruktore a dekóduje obe strany pravidla, ktoré sú potom reprezentované v `public Vector<String> leftVector`,`public Vector<String> rightVector`. Pretože dátová štruktúra `Vector` je indexovaná, vieme získať ľahký prístup k neterminálom a terminálom.

2. `public int leftCount()` zistí počet terminálov a neterminálov v ľavej strane pravidla.

3. `public int rightCount()` zistí počet terminálov a neterminálov v pravej strane pravidla.

4.5.2 Grammar class

Táto abstraktná trieda tvorí jadro pre všetky typy gramatík, ktoré sú implementované v aplikácii. Ako vidno z UML class diagramu 4.4, `Grammar` class rozširuje zvyšné typy gramatík `Regular, Contextfree, Contextsensitive, Phrasal`. V gramatike sú tieto atribúty:

1. `protected LinkedList<String> nonterminal` spájaný zoznam neterminálov, každý prvok zoznamu je jeden neterminál. Túto dátovú štruktúru som zvolil, pretože sa používa ľahko, je indexovaná a má už predprogramované funkcie, ktoré používam. `<String>` znamená, že do zoznamu neterminálov sa dajú pridávať len Stringy a taktiež tento typ zabezpečuje, že pri vyberaní prvku zo zoznamu netreba prvok pretypovávať, čím sa zjednoduší práca.

2. `protected LinkedList<String> terminal` spájaný zoznam terminálov, reprezentuje všetky terminály v danej gramatike, každý prvok zoznamu je jeden terminál.

3. `protected LinkedList<Rule> rule` spájaný zoznam pravidiel, ktorý reprezentuje celú množinu pravidiel v gramatike pričom jeden prvok zoznamu je jedno pravidlo.

4. `protected String startingNonterminal` reprezentuje počiatočný neterminál.

5. `protected String form` reprezentuje aktuálnu vetnú formu pri odvodení slova v gramatike.

6. `protected LinkedList<String> history` reprezentuje históriu odvodenia v danej gramatike.

Poznámka 4.5.2. *Protected znamená, že tieto atribúty sa dajú dediť z tejto triedy, ale len priami potomkovia triedy majú vyššie uvedené atribúty.*

Na prácu s horeuvedenými atribútmi boli zadefinované public metódy:

1. `public Grammar(LinkedList<String> N, LinkedList<String> T, LinkedList<Rule> R, String o)` -konštruktor, ktorý nám umožňuje vytvárať gramatiku. Konštruktor má 4 parametre a nastaví neterminály, terminály, pravidlá a začiatočný neterminál.

2. `public LinkedList<String> getNonterminal()` vráti zoznam neterminálov v danej gramatike.

3. public LinkedList<String> getTerminal() vráti zoznam terminálov v danej gramatike.

4. public String getStarting() vráti začiatočný neterminál.

5. public LinkedList<Rule> getP() vráti zoznam pravidiel v danej gramatike.

6. public void setNonterminal(LinkedList<String> nonterminal) nastaví zoznam neterminálov pre danú gramatiku, podľa vstupu metódy.

7. public void setTerminal(LinkedList<String> terminal) nastaví zoznam terminálov pre danú gramatiku, podľa vstupu metódy.

8. public void setP(LinkedList<Rule> rule) nastaví zoznam pravidiel pre gramatiku, podľa vstupu metódy.

9. public void setStarting(String startingNonterminal) nastaví počiatočný neterminál na string, ktorý dostane metóda.

10. private void setFromtoHistory(String form) pridá vetnú formu do histórie vetných foriem.

Niektoré ďalšie metódy, ktoré sú implementované tak, aby sa dali použiť pre ľubovoľný typ gramatiky.

1. public LinkedList<String> useRule(String form, Rule r) metóda dostane na vstupe vetnú formu form a pravidlo r, ktoré sa má použiť. Na výstup je vrátený spájaný zoznam vetných foriem, ktoré môžu byť odvodené použitím pravidla r. Ak pravidlo r nie je možné použiť na vetnú formu form, tak metóda pošle na výstup null.

2. public LinkedList<Rule> ruleCanBeUsed() vzťahuje sa na aktuálnu vetnú formu v gramatike. Metóda vráti na výstup všetky pravidlá, ktoré môžu byť použité v ďalšom kroku odvodenia.

3. public LinkedList<String> allRightSides(String marked) používaná, keď používateľ experimentuje s odvodením v gramatike. Označí si nejakú časť vetnej formy, ktorá je potom poslaná tejto metóde. Metóda vráti všetky pravé strany, na ktoré môže byť označená časť prepísaná. Samozrejme, ak neexistuje pravidlo, ktorým by sa označená časť dala prepísať, tak metóda vracia null.

4. public LinkedList<String> oneStep(String form) táto metóda dostane na vstupe vetnú formu form a vráti spájaný zoznam vetných foriem, ktoré sa dajú odvodiť na jeden krok použitím ľubovoľného pravidla. Využíva sa hlavne pri generovaní všetkých slov.

5. public LinkedList<String> generateAllWords(int count) metóda dostane na vstupe číslo count a vygeneruje všetky slová z počiatočného neterminálu na count krokov, ktoré pošle na výstup v spájanom zozname, v ktorom nie sú duplicitné slová.

6. public boolean isFormWord(String form) skontroluje, či všetky symboly vo vetnej forme na vstupe form sú terminály. Ak áno, tak vráti true, ak vetná forma nie je terminálne slovo, tak vráti false.
7. public abstract boolean checkMembership() throws Errors abstraktná metóda, ktorá je implementovaná v triedach, ktoré sú rozšírené o triedu Grammar.

4.5.3 Regular class

Táto trieda reprezentuje všetky regulárne gramatiky. Z UML class diagramu 4.4 vidíme, že abstraktná trieda rozširuje triedu Regular, to znamená, že táto trieda dedí celú funkcionalitu z abstraktnej Grammar triedy. Najdôležitejšími metódami v tejto triede sú:

1. public boolean checkRegularity() throws Errors táto metóda pre každé pravidlo v zozname pravidiel zistí, či ľavá strana pravidla je práve jeden neterminál. Ak je, overí, či tento neterminál je v množine neterminálov. Ak neterminál nebol nájdený v množine neterminálov, tak je vyvolaná výnimka, ktorá sa vypíše používateľovi a nedovolí uložiť regulárnu gramatiku. Pre pravú stranu pravidla je kontrola trochu zložitejšia, pretože sú dve možnosti. Buď pravá strana je slovo, a potom treba skontrolovať, či každý terminál je v množine terminálov. Alebo pravá strana sa skladá z terminálov, pričom posledný symbol pravej strany je neterminál. V tomto prípade treba skontrolovať, či terminály sú z množiny terminálov a posledný neterminál je z množiny neterminálov. Ak to neplatí, tak metóda vyvolá výnimku. Ak úspešne prejdeme všetkými pravidlami bez výnimky, tak sa gramatika uloží. Inak používateľ musí zmeniť niektoré pravidlá, ktoré sa mu zobrazia, keď nevyhovujú regularite vo vytváranej gramatike.
2. public Regular(LinkedList<String> N, LinkedList<String> T, LinkedList<Rule> R, String o) konštruktor, pomocou ktorého sa vytvorí regulárna gramatika a nastavia sa neterminály na N, terminály na T, pravidlá na R a začiatočný neterminál na o.
3. public boolean checkMembership() throws Errors táto metóda overí, či novovytvorená gramatika je regulárna, ak nie je, metóda vyvolá výnimku.

4.5.4 Contextfree class

Trieda Contextfree reprezentuje všetky bezkontextové gramatiky. Z UML class diagramu 4.4 vidíme, že abstraktná trieda rozširuje triedu Contextfree, to znamená, že táto trieda dedí celú funkcionalitu z abstraktnej Grammar triedy. Najdôležitejšími metódami v tejto triede sú:

1. public boolean checkContextFree() throws Errors táto metóda pre každé pravidlo v zozname pravidiel zistí, či ľavá strana pravidla je len jeden neterminál, ak je, tak treba overiť, či tento neterminál je v množine neterminálov. Ak

na ľavej strane nie je neterminál, tak metóda vyvolá výnimku. Pre pravú stranu pravidla musí platiť, že všetky symboly sú buď z množiny neterminálov alebo terminálov. Ak to neplatí, tak metóda vyvolá výnimku a používateľ musí opraviť ukladanú gramatiku. Ak úspešne overíme, či všetky pravidlá sú v požadovanom tvare, tak gramatika je úspešne uložená a pripravená na experimentovanie.

2. public ContextFree(LinkedList<String> N, LinkedList<String> T, LinkedList<Rule> R, String o) konštruktor, pomocou ktorého sa vytvorí bezkontextová gramatika a nastavia sa neterminály na N, terminály na T, pravidlá na R a začiatočný neterminál na o.

3. public boolean checkMembership() throws Errors táto metóda overí, či novovytvorená gramatika je bezkontextová.

4.5.5 Contextsensitive class

Trieda Contextsensitive reprezentuje všetky kontextové gramatiky. Z UML class diagramu 4.4 vidíme, že trieda Contextsensitive dedí z abstraktnej triedy celú funkcionalitu. Najdôležitejšími metódami v tejto triede sú:

1. public boolean checkContextSensitive() throws Errors táto metóda skontroluje, či pravá strana každého pravidla je dlhšia alebo aspoň tak dlhá ako ľavá, ak nie je, tak nie je splnená podmienka pre kontextovú gramatiku. Ďalej pre každé pravidlo v zozname pravidiel skontrolujeme, či všetky symboly, ktoré sú použité na pravej aj ľavej strane pravidla, sú z množiny terminálov alebo neterminálov. Ak existuje symbol v pravidle, ktorý nie je v termináloch alebo neterminaloch, tak metóda vyvolá výnimku a gramatika nie je uložená. Používateľ tento problém može vyriešiť buď pridaním požadovaného symbolu medzi terminály, neterminály alebo zmení pravidlo, ktoré vlastnosť porušuje. Pri úspešnom preverení všetkých pravidiel je gramatika uložená a je pripravená na experimentovanie.

2. public ContextSensitive(LinkedList<String> N, LinkedList<String> T, LinkedList<Rule> R, String o) konštruktor, pomocou ktorého sa vytvorí kontextová gramatika a nastavia sa neterminály na N, terminály na T, pravidlá na R a začiatočný neterminál na o.

3. public boolean checkMembership() throws Errors táto metóda overí, či novovytvorená gramatika je kontextová.

4.5.6 Phrasal class

Trieda Phrasal reprezentuje všetky frázové gramatiky, ktoré môžu byť vytvorené v aplikácii GRASIM. Z UML class diagramu 4.4 vidíme, že abstraktná trieda rozširuje triedu Phrasal, to znamená, že táto trieda dedí celú funkcionalitu z abstraktnej Grammar triedy. Najdôležitejšími metódami v tejto triede sú:

1. `public boolean checkPhrasal() throws Errors` táto metóda pre každé pravidlo v zozname pravidiel zistí, či na ľavej a pravej strane sú všetky symboly len z množín terminálov alebo neterminálov. Ak nejaký symbol, ktorý bol použitý v pravidle, nie je z množiny neterminálov a terminálov, tak táto metóda vyvolá výnimku a používateľ si môže dané pravidlo opraviť, alebo pridať symbol do jednej z množiny terminálov alebo neterminálov. Ak úspešne overíme všetky pravidlá a nevyskytla sa žiadna výnimka, tak gramatika je uložená a dá sa s ňou experimentovať.

2. `public Phrasal(LinkedList<String> N, LinkedList<String> T, LinkedList<Rule> R, String o)` konštruktor, pomocou ktorého sa vytvorí frázová gramatika a nastavia sa neterminály na `N`, terminály na `T`, pravidlá na `R` a začiatočný neterminál na `o`.

3. `public boolean checkMembership() throws Errors` táto metóda overí, či novovytvorená gramatika je frázová.

4.6 Testovanie

Pre účely testovania som si naprogramoval triedu Test, ktorá slúžila na simuláciu používateľského rozhrania. Preto som mohol začať testovať aplikačnú logiku po častiach aj bez časti GUI. Programovacia a testovacia fáza sú úzko spojené a treba dávať veľký pozor, aby sa neprogramovalo na chybných základoch. Po dokončení oboch častí sme začali testovať aplikáciu spoločne. Pri testovaní sme odhalili chyby týchto typov:

1. Zlé parsovanie vstupu.
2. Problémy pri vytváraní parametrizovaných neterminálov a terminálov.
3. Chyba pri kontrolovaní, či pravidlá spĺňajú podmienky regulárnej, bezkontextovej, kontextovej a frázovej gramatiky.
4. Chyba v algoritme CYK.
5. Zlé generovanie slov zo zadaného jazyka.
6. Chyba pri načítavaní a ukladaní gramatiky z/do súboru.
7. Problém pri experimentovaním s odvodením.
8. Chyba v backtraku pri zisťovaní príslušnosti slova do kontextového jazyka.
9. Najvážnejšia chyba, ktorá vznikla bolo zlé reprezentovanie pravidla. Pravidlo je reprezentované ako dvojica stringov a z tohto dôvodu bolo ťažké hľadať začiatok neterminálu alebo terminálu. Táto chyba vznikla kvôli zlému návrhu reprezentácie pravidla. Chyba bola opravená doprogramovaním metód, ktoré pravidlo reprezentujú ako vector pričom i-ty znak vo vectore je i-ty symbol v pravidle.

Keďže po naprogramovaní malého celku sme vždy testovali, či naprogramovaná časť je korektná, chyby sa podarilo nájsť a eliminovať.

Kapitola 5

Použité algoritmy

V tejto kapitole si vysvetlíme, ako fungujú jednotlivé algoritmy, ktoré sú v aplikácii implementované. Na zistenie príslušnosti slova do jazyka použijeme rôzne typy algoritmov v závislosti na type gramatiky. Metóda `isWordInGrammar`, ktorá je implementovaná v aplikácii GRASIM, dostane na vstupe slovo w a gramatiku G, a ak sa dá, rozhodne o príslušnosti slova w do jazyka L(G). Vieme, že pre frázovú gramatiku G je tento problém nerozhodnuteľný, preto v tomto prípade je implementovaná heuristika. Ak je gramatika G kontextová, overujeme všetky odvodenia slova w. Tento algoritmus má však exponenciálnu časovú zložitosť, pretože musíme vygenerovať všetky slová dĺžky $|w|$ a následne overiť, či je slovo w v tejto množine. Na zistenie príslušnosti slova w do bezkontextového jazyka použijeme algoritmus CYK, keďže každý regulárny jazyk je bezkontextový, tak algoritmus CYK bude fungovať aj pre regulárne jazyky.

Poznámka 5.0.1. *Exponenciálna časová zložitosť pre kontextovú gramatiku vychádza z maximálneho počtu krokov, ktorým slovo w mohlo vzniknúť a to je počet rôznych vetných foriem dĺžky nanajvýš $|w|$ a tých je maximálne $|N \cup T|^{|w|}$.*

Poznámka 5.0.2. *Aby sa CYK dal použiť na regulárnu gramatiku, musíme previesť regulárnu gramatiku do Chomského normálneho tvaru.*

Definícia 5.0.1. *Gramatika $G = (N, T, P, \sigma)$ je v Chomského normálnom tvare, ak $P \subseteq N \times (NN \cup T \cup \{\varepsilon\})$.*

Tvrdenie 5.0.1. *Ku každej bezkontextovej gramatike G existuje bezkontextová gramatika G' taká, že $L(G) = L(G')$, ktorá je v Chomského normálnom tvare.*

Poznámka 5.0.3. *Pre záujemcov je dôkaz tejto vety podrobne vysvetlený v [1].*

Poznámka 5.0.4. *Chomského normálny tvar je výhodný, lebo pravidlá sú dvoch typov:*
1. $N \to NN$
2. $N \to \{T \cup \{\varepsilon\}\}$
Tento tvar nám umožní realizovať algoritmus CYK prehľadnejšie a efektívnejšie.

5.1 Cocke-Younger-Kasami (CYK) algoritmus

Hlavnou myšlienkou algoritmu je dynamické programovanie. Ak $w = \varepsilon$, zistíme, či počiatočný neterminál je v množine vymazávajúcich neterminálov. Ak je, tak slovo w je z jazyka L, ak nie je, tak slovo w nie je z jazyka. Predpokladajme, že $w = u_1 u_2 ... u_n$, kde $n \geq 0$. Gramatiku G prevedieme do Chomského normálneho tvaru a potom budeme vytvárať množiny $N_{i,j}$(kde $1 \leq i \leq j \leq n$) v množine $N_{i,j}$ budú všetky tie neterminály, z ktorých sa dá vygenerovať podslovo $u_i ... u_j$, formálne:

$$\varphi \in N_{i,j} \iff \varphi \Rightarrow^* u_i ... u_j$$

Vďaka tomu, že gramatika je v Chomského normálnom tvare, vieme ľahko konštruovať každú množinu $N_{i,i}$, je to množina tých neterminálov φ, pre ktoré v G existuje pravidlo $\varphi \to u_i$.
Zamyslime sa nad tým, ako teraz zostrojíme množinu $N_{i,j}$, ak poznáme množiny, ktoré zodpovedajú kratším podslovám. Predovšetkým chceme nájsť všetky neterminály φ, z ktorých sa dá odvodiť $u_i ... u_j$ V prvom kroku sa snažíme použiť pravidlo, ktoré nám rozbije φ na nejaké dva neterminály $\alpha\beta$, napr. pravidlom $\varphi \to \alpha\beta$. Následne sa pre všetky k snažíme z prvého neterminálu α odvodiť $u_i ... u_k$ a z druhého neterminálu β odvodiť $u_{k+1} ... u_j$. Formálne:

$$N_{i,j} = \{\varphi | \ \exists k \in \{i, ..., j-1\}, \alpha \in N_{i,k}, \beta \in N_{k+1,j}; (\varphi \to \alpha\beta) \in P\}$$

Slovo patrí do jazyka, ak množina $N_{1,n}$ obsahuje počiatočný neterminál. Podobne je algoritmus CYK opísaný v [1]. Asymptoticky najhorší čas pre tento program je $O(n^3)$, kde n je dĺžka vstupného slova. Takáto časová zložitosť robí algoritmus CYK najefektívnejším algoritmom na zisťovanie príslušnosti slova do bezkontextového jazyka. Pseudokód, ktorý môžeme nájsť na wikipédii [2]:

Algorithm 1 CYK

1: Nech vstupný string sa skladá z n symbolov $a_1 ... a_n$
2: Nech gramatika obsahuje r neterminálov R1 … Rr.
3: Nech $P[n, n, r]$ je pole booleanov, ktorého sú všetky jeho elementy nastavené na false.
4: **for** i:=1 to n **do**
5: For each pravidlo $R_j \to a_i$, nastavíme P[i,1,j] = true.
6: **for** i:=2 to n **do** – Length of span
7: **for** j:=1 to n-i+1 **do** – Start of span
8: **for** k:=1 to i-1 **do** – Partition of span
9: For each pravidlo $R_A \to R_B R_C$
10: P[j,k,B] and P[j+k,i-k,C] tak nastavíme P[j,i,A]=true
11: **if** pre nejaké $P[1, n, x] = true$ (x je iterované cez všetky s, kde s sú všetky indexy v R_s) **then**string $a_1 ... a_n$ je z jazyka
12: **else**string $a_1 ... a_n$ nie je z jazyka

Poznámka 5.1.1. *Tento pseudokód predpokladá, že gramatika je v Chomského normálnom tvare. Ku každej bezkontextovej však existuje gramatika v Chomského normálnom tvare, ktorá má však omnoho viac nových neterminálov. Preto som algoritmus CYK upravil tak, aby sme gramatiku nemuseli prevádzať do Chomského normálneho tvaru. Prikladám aj algoritmus, ktorý som naprogramoval, aby si čitateľ mohol porovnať zložitosť upraveného algoritmu CYK, ktorý nepotrebuje, aby boli pravidlá v špeciálnom tvare.*

5.2 CYK implementovaný v aplikácii GRASIM

Algoritmus CYK, ktorý je naimplementovaný v aplikácii GRASIM, sa skladá z týchto troch základných častí: konštruktora, go, go2 metód.

5.2.1 Konštruktor

V konštruktore sa dekóduje vstupné slovo `word` a získa sa množina pravidiel pre vstupnú gramatiku. V riadkoch 30-33 na diagrame 5.1 sa hľadá najdlhšia pravá strana pravidla. Potom sa nainicializuje pole `h` na samé -1. Do `TreeMap<String Integer>` sa priradia neterminály, ktoré obsahuje vstupná gramatika na konci konštruktora riadok 46, sa zavolá metóda go2 s parametrami int, int, int. Kde prvý parameter určuje začiatočnú pozíciu vo vstupnom slove, druhý parameter určuje koncovú pozíciu vo vstupnom slove a tretí parameter je identifikátor počiatočného neterminálu. Metóda go2 sa zavolá s parametrami 0, n-1, a s identifikátorom počiatočného neterminálu, ktorý je uložený v TreeMap. Zdrojový kód je na diagrame 5.1

5.2.2 go

Táto metóda dostane na vstupe štyri parametre. Prvý parameter `a` určuje začiatočnú pozíciu podreťazca, druhý parameter `b` určuje koncovú pozíciu podreťazca vo vstupnom slove. Tretí parameter `r` je identifikátor pravidla a štvrtý parameter `pos` je pozícia v pravej strane pravidla `r`. Táto metóda vracia hodnotu väčšiu ako nula v prípade, že sa podslovo určené parametrami `a, b` sa dá odvodiť zo sufixu pravej strany pravidla `r`. Sufix pravidla `r` začína od pozície `pos`. V prípade, keď výpočet vedie k zacykleniu metóda vracia -2. Inak metóda vracia -1. Zdrojový kód metódy je na diagrame 5.2.

5.2.3 go2

Métóda dostane na vstup tri parametre. Kde prvý parameter `a` určuje začiatočnú pozíciu podreťazca, druhý parameter `b` určuje koncovú pozíciu podreťazca a tretí parameter `nt` je identifikátor neterminálu. Táto metóda skúša, či sa z neterminálu `nt` dá odvodiť podslovo, ktorého začiatok určuje parameter `a` a koniec určuje parameter `b`. Táto metóda vracia hodnotu väčšiu ako nula v prípade, že sa podslovo určené parametrami `a, b` dá odvodiť zo sufixu pravej strany pravidla `r`. Sufix pravidla `r`

začína od pozície pos. V prípade, keď výpočet vedie k zacykleniu metóda vracia -2. Inak metóda vracia -1. Zdrojový kód metódy je na diagrame 5.3.

```
 1 package sk.fmfi.algorithm;
 2⊕import java.util.*;
 6
 8⊕ * @author dbanik
14 public class CYK {
15     Grammar g;
16     boolean vysledok;
17     Vector<String> s;
18     TreeMap<String, Integer> M;
19     LinkedList<Rule> rules;
20     int k;
21     int h[][][][];
22⊖    CYK(Grammar g, String word){
23         this.g=g;
24         this.s=Rule.decode(word);
25         rules=g.getP();
26         M=new TreeMap<String, Integer>();
27         k=0;
28         int max=0;
29         int n=s.size();
30         for (Iterator iter = rules.iterator(); iter.hasNext();) {
31             Rule element = (Rule) iter.next();
32             if (element.rightVector.size()>max) max=element.rightVector.size();
33         }
34         h= new int[n+2][n+2][rules.size()][max];
35         for(int i1=0; i1<n+2; ++i1)
36             for(int i2=0; i2<n+2; ++i2)
37                 for(int i3=0; i3<rules.size(); ++i3)
38                     for(int i4=0; i4<max; ++i4)
39                         h[i1][i2][i3][i4]=-1;
40
41         for (Iterator iter = g.getNonterminal().iterator(); iter.hasNext();) {
42             String element = (String) iter.next();
43             M.put(element, new Integer(k++));
44         }
45         vysledok=true;
46         if (go2(0,n-1, f(g.getStarting()) )==-2){
47           vysledok=false;
48         }
49     }
```

Obr. 5.1: Konštruktor

```
54     int go(int a, int b, int r, int pos){
55         if(a>b) {
56             a=1;b=0;
57             if(rules.get(r).rightVector.size()<=pos) return 1;
58             //return -2;
59         }
60         if(rules.get(r).rightVector.size()<=pos){
61             if(a>b) return 1;
62             return -2;
63         }
64         if (h[a][b][r][pos]!=-1) return h[a][b][r][pos];
65          h[a][b][r][pos]=-2;
66         String x=rules.get(r).rightVector.get(pos);
67         if (isTerminal(x)){
68             if(a>b) return -2;
69             if(x.equals(s.get(a))) {
70                 int tmp= go(a+1,b,r,pos+1);
71                 h[a][b][r][pos]=tmp;
72                 return tmp;
73             }
74             return -2;
75 
76         }else{
77             for(int i=a; i<=b+1; ++i){
78             //zistime ci x=>* w[a..i-1] a zvysok pravidla r na w[i..b]
79                 int t1=go2(a,i-1,f(x));
80                 int t2;
81                 if(pos+1==rules.get(r).rightVector.size()) t2= i==b+1?1:-2;
82                 t2=go(i,b,r,pos+1);
83 
84                 if(t1!=-2 && t2!=-2){
85                     h[a][b][r][pos]=1;
86                     return 1;
87                 }
88             }
89         }
90 
91         return -2;
92     }
```

Obr. 5.2: go

```
93     int go2(int a, int b, int nt){
94         for(int i=0;i<rules.size();++i)
95             if(rules.get(i).getLeftSide().equals(g.getNonterminal().get(nt) ) ) {
96                 if (go(a,b,i,0)!=-2) {
97                     return 1;
98 
99                 }
100 
101             }
102         return -2;
103     }
104     boolean isTerminal(String s){
105         if (M.get(s)==null) return true;
106         return false;
107     }
108 
109 }
```

Obr. 5.3: go2

Kapitola 6

Záver

Výsledkom našej práce je aplikácia GRASIM, ktorá umožňuje vytváranie gramatík s parametrizovanými neterminálmi, terminálmi a pravidlami. Je vhodná na experimentovanie s odvádzaním v jednej zo štyroch typov gramatík. Obsahuje naprogramované algoritmy, ktoré zisťujú príslušnosť slova w do regulárneho, bezkontextového alebo kontextového jazyka. Keďže pre frázové jazyky nie je tento problém rozhodnuteľný, je implementovaná heuristika, ktorá v niektorých prípadoch tento problém rozhodne. Aby sa mohol používateľ vrátiť k už začatým problémom, aplikácia umožňuje ukladanie a načítavanie gramatík z/do súboru.

Po upevnení vedomostí z formálnych jazykov a automatov, ktoré prispeli k lepšiemu porozumeniu problematike, som začal analyzovať požiadavky. Pri analýze som sa naučil používať UML, ktorý podstatne zjednodušil návrh a dizajn aplikačnej logiky, ktorá tvorí jadro aplikácie GRASIM. Pozitívne hodnotím získanie nových skúseností s programovaním, zoznámenie sa s novými technológiami, osvojenie postupov pri tvorbe aplikácie a zlepšenie tímovej práce, ktorá hrala pozoruhodnú úlohu pri vývoji.

Aplikácia je určená pre študentov, ktorí chcú lepšie porozumieť formálnym jazykom a automatom, ale takisto aj pre profesorov, ktorí majú možnosť vytvárať úlohy pre žiakov. Pevne verím, že aplikácia GRASIM bude prínosná aj pre samotných používateľov, ktorým pomôže upevniť si vedomosti z formálnych jazykov a automatov.

Kapitola 7

Príloha

K tejto práci je priložené CD-médium, na ktorom sa nachádza aplikácia GRASIM, zdrojové kódy a dokumentácia vo formáte JavaDoc.

Literatúra

[1] Michal Forišek: 19.5.2005, *Formálne jazyky a automaty*

[2] *CYK algoritmus,* http://en.wikipedia.org/wiki/CYK_algorithm

[3] Imrich Buranský: *30.3.2001, Extrémne programovanie,* http://programovanie.pc.sk/jazyky/clanok.php?ID=145

[4] Jim Arlow, IIa Neustadt: 2005, *UML a unifikovaný proces vývoje aplikací,* Brno

[5] *Unified Modeling Language,* http://sk.wikipedia.org/wiki/Unified_Modeling_Language

[6] *UML Unified Modelling Language,* http://66.102.9.104/search?q=cache:B_oJN1zoGZwJ:s.ics.upjs.sk/~soki/HomeIbe/Ibe%2520-%2520Softwarov%25E9%2520in%259Einierstvo/dokumenty/UML.doc+UML+asoci%C3%A1cia&hl=sk&ct=clnk&cd=5&gl=sk

[7] Martin Kováčik *1.3.2006, Tutoriál k Eclipse 3.1,* http://www2.fiit.stuba.sk/oop/c/eclipse_tutorial/index.html

[8] Mária Bieliková *2000, Softvérové inžinierstvo, Princípy a manažment,* Slovenská technická univerzita v Bratislave

[9] *Formal language,* http://en.wikipedia.org/wiki/Formal_language

[10] *Regular,* http://en.wikipedia.org/wiki/Regular_grammar