VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEMONSTRAČNÍ PROGRAM KONSTRUKCE LL TABULKY

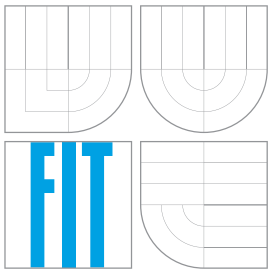
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ONDŘEJ CHOLEVA

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEMONSTRAČNÍ PROGRAM KONSTRUKCE LL TABULKY

DEMONSTRATION PROGRAM OF LL TABLE CONSTRUCTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ONDŘEJ CHOLEVA
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZÁMEČNÍKOVÁ EVA
SUPERVISOR

BRNO 2011

Abstrakt

Tato práce se zabývá návrhem a následnou implementací výukové aplikace pro konstrukci LL tabulky na bázi zadané gramatiky. Tato tabulka je využívaná prediktivním syntaktickým analyzátorem. Součástí této aplikace je i ukázka syntaktické analýzy s využitím zkonstruované tabulky. Aplikace umožňuje krokování všech dílčích algoritmů a jejich zobrazení v grafickém uživatelském rozhraní. Při vývoji byl zohledněn požadavek na přenostitelnost výběrem vhodné multiplatformní knihovny.

Abstract

This work deals with desing and subsequent implementation of educational application for construction of LL table on given grammar basis. This table is used by predictive LL parser. A part of this application is also a demonstration of parsing using table driven parser. Application offers possibility of stepping of all partial algorithms and their visualization in a graphical user interface. By development of application the request for portability was satisfied by choosing an appropriate multiplatform library.

Klíčová slova

LL parsovací tabulka, syntaktická analýza, demonstrace, aplikace, JavaFX, výukový program

Keywords

LL parsing table, syntactic analysis, demonstration, aplication, JavaFX, courseware

Citace

Ondřej Choleva: Demonstrační program konstrukce LL tabulky, bakalářská práce, Brno, FIT VUT v Brně, 2011

Demonstrační program konstrukce LL tabulky

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením slečny Evy Zámečníkové. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Choleva
16. května 2011

Obsah

Kapitola 1

Úvod

Cílem práce bylo vytvořit multiplatformní aplikaci, která bude vhodným způsobem demonstrovat konstrukci LL tabulky ze zadané gramatiky a umožní také ukázku syntaktické analýzy provedené pomocí této tabulky.

Tato aplikace může najít uplatnění jak při výuce předmětu Formální jazyky a překladače na FIT VUT v Brně, tak i pro samostatné zájemce o problematiku syntaktické analýzy.

Znalost principů formálních jazyků, jejich teorie i použití totiž patří bezesporu mezi základní vědomosti jak informatika teoretika, tak informatika programátora. Vždyť poskytují mocné nástroje pro výzkum, popis a následnou interpretaci každého jazyka. Zde samozřejmě máme na mysli jazyky programovací. Bez jejich existence by nemohly být vytvořeny počítačové programy jak je dnes známe. Text napsaný v libovolném programovacím jazyce je pouhým seskupením slov nebo spíše jednotlivých znaků, pokud jej nedokážeme správně rozpoznat a interpretovat. Až poté, co ověříme, že slova textu jsou v jazyce definována a text je podle pravidel daného jazyka přípustný a smysluplný, můžeme jej prohlásit za program a dle něj následně vykonávat činnost, která je jím zadaná.

K rozpoznání jednotlivých slov nám poslouží lexikální analýza, pomocí níž převádíme vstupní text na posloupnost slov, jež jsou v daném jazyce přípustná. V této posloupnosti je následně potřeba ověřit, zda pořadí slov odpovídá syntaxi jazyka. K tomu je nezbytná syntaktická analýza 2.3. K ověření smysluplnosti slouží analýza sémantická, té se však dále věnovat nebudeme. Pravidla určující syntaxi jazyka mohou být definována pomocí různých typů gramatik 2.2 a slouží pro popis všech přípustných kombinací symbolů jazyka.

U určitého typu gramatik je jedním z možných způsobů syntaktické analýzy analýza shora dolů, přičemž čte vstupní posloupnost zleva (Left) a vytváří nejlevější (Leftmost) derivaci. Nazýváme ji tedy LL syntaktická analýza 2.3.1. Klíčovou součástí LL analyzátoru je LL tabulka 2.3.3 určující, které z pravidel odpovídá symbolu právě čtenému ze vstupu. Právě vysvětlením tvorby této tabulky se tato práce zabývá.

Celá konstrukce takové LL tabulky ze zadané gramatiky se skládá z několika dílčích algoritmů. Více o teorii napoví druhá kapitola, která obsahuje stručný úvod do problematiky jazyků, gramatik a principů syntaktické analýzy.

V běžných výukových materiálech (např. knihách, skriptech, přednáškách) bývají tyto algoritmy vysvětleny pomocí příkladů. Pochopení takto prezentovaných algoritmů nemusí být vždy jednoduché. Statická ukázka průběhu algoritmu je totiž náročnější na představivost, je definovaná pouze pro pár konkrétních případů (v lepším případě) a mnohdy pro úsporu místa slučuje více kroků dohromady.

Řešením je tedy demonstrace ve formě interaktivní aplikace. Díky možnosti zobrazit průběh pro libovolný vstup, ručnímu krokování a vizualizaci průběhu, může být významně

usnadněno a urychleno pochopení vysvětlovaného algoritmu.

Návrhem takové aplikace se zabývám ve třetí kapitole. To zahrnuje výběr vhodné multiplatformní knihovny, analýzu možného řešení a návrh grafického uživatelského rozhraní aplikace. Vlastní implementaci se pak věnuje kapitola čtvrtá. Poslední kapitola pak obsahuje závěrečné shrnutí a zhodnocení dosažených výsledků.

Kapitola 2

Teoretický úvod

V této kapitole se budeme věnovat teoretickému popisu, objasnění a definici vzájemných vztahů a pojmů z oblasti formálních jazyků, které jsou nutné k pochopení funkce syntaktické analýzy.

V celé práci dodržuji následující konvenci - index * znamená 0 až nekonečno opakování, index $^+$ znamená 1 až nekonečno opakování.

Všechny uvedené algoritmy jsou pro názornost psány v pseudokódu, bez použití diakritiky. Pro operace s množinami používám znak $=$ (např. $A = B$) pro naplnění množiny A prvky z množiny B a znak $\longleftarrow$ (např. $A \longleftarrow B$) pro přidání obsahu množiny B k obsahu množiny A, tedy $A = A \cup B$.

2.1 Jazyk

Abychom mohli definovat pojem jazyk, je potřeba uvést, co je to abeceda a řetězec (slovo). Abecedou myslíme konečnou množinu symbolů. Příkladem mohou být arabské číslice {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Řetězcem potom nazveme libovolně dlouhou posloupnost symbolů (kde pořadí symbolů je podstatné). Prázdný řetězec, který neobsahuje žádný symbol, značíme ε.

Nyní si představme množinu obsahující všechny řetězce, které lze nad danou abecedou vytvořit. Jazyk je podmnožinou takovéto množiny.

Definice 2.1.1 Buď Σ abeceda a nechť $L \subseteq \Sigma^*$. Pak L je jazyk nad abecedou Σ. Prázdná množina $\varnothing$ a $\{\varepsilon\}$ jsou jazyky nad každou abecedou. Přitom ovšem platí

$$\varnothing \neq \{\varepsilon\}$$

jelikož $\{\varepsilon\}$ obsahuje jedno (prázdné) slovo, zatímco jazyk $\varnothing$ neobsahuje žádné slovo. Protože je jazyk definován jako množina, může být konečný (potom lze počet jeho prvků vyjádřit číslem n, kde $n \geq 0$) nebo nekonečný [7].

Příklad 2.1.1 Množina $\{1, 11, 111\}$ je konečný jazyk nad abecedou $\{1\}$ jehož počet prvků $n = 3$. Kdežto $\{1\}^*$ je nekonečný jazyk nad abecedou $\{1\}$.

2.1.1 Reprezentace jazyků

Abychom mohli s jazykem pracovat, je potřeba mít nástroje pro jeho specifikaci. Konečný jazyk lze reprezentovat výčtem jeho slov tak, jak je uvedeno v příkladu 2.1.1. Ovšem toto je pro praktické využití nevhodné a pro nekonečné jazyky je takovýto postup dokonce

nemožný. Proto byly zavedeny jiné nástroje pro reprezentaci jazyka, aby bylo možno i nekonečný jazyk popsat konečnou množinou prostředků. Základní dva takové nástroje jsou automaty a gramatiky [8].

Gramatika je založena na konečné množině pravidel, umožňujících generování řetězců jazyka postupným uplatňováním těchto pravidel.

Automat je formální model, na základě kterého lze rozhodnout, zda dané slovo do jazyka patří nebo nepatří. Definice automatu je z pohledu této práce zbytečná, a proto se jí nebudeme věnovat.

2.2 Gramatika

Gramatika je formální prostředek k reprezentaci jazyka. Využívá dvou vzájemně disjunktních abeced a to:

- Nonterminální symboly N, jež mají roli pomocných proměnných, označujících syntaktické celky. Symboly této abecedy značíme zpravidla velkými písmeny.
- Terminální symboly T, tato abeceda je totožná s abecedou popisovaného jazyka. Symboly této abecedy značíme zpravidla malými písmeny.

Jádrem gramatiky je konečná množina P obsahující pravidla, což jsou uspořádané dvojice (α, β) řetězců, určujících možnou náhradu řetězce α za řetězec β, který se vyskytuje jako podřetězec generovaného řetězce. Řetězec α musí obsahovat alespoň jeden nonterminální symbol, řetězec β je prvek množiny $(N \cup T)^*$. Pomocí aplikace posloupnosti pravidel lze z počátečního nonterminálu dojít ke kterémukoliv řetězci [2].

Nyní můžeme úplně definovat gramatiku, prostřednictvím níž lze generovat jazyk touto gramatikou reprezentovaný.

Definice 2.2.1 *Gramatika* je čtveřice $G = (N, T, P, S)$

- N je konečná množina nonterminálních symbolů
- T je konečná množina terminálních symbolů, $N \cap T = \varnothing$
- P je konečná podmnožina kartézského součinu $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$
- $S \in N$ je počáteční nonterminální symbol gramatiky

Prvek (α, β) množiny P nazýváme pravidlem a pro naše účely jej budeme zapisovat ve tvaru $\alpha \to \beta$. Řetězec α nazveme levou stranou a řetězec β pravou stranou přepisovacího pravidla[2]. Více pravidel lišících se pouze pravou stranou lze zapsat do jednoho řádku a pravé strany oddělit pomocí symbolu | (ve významu nebo).

Příklad 2.2.1 Gramatika G generující jazyk $L = \{a^n b^n | n > 0\}$:

$$\begin{aligned} G &= (N, T, P, S) \\ N &= \{S\} \\ T &= \{a, b\} \\ P &= \{S \to aSb | ab\} \\ S &= \{S\} \end{aligned}$$

2.2.1 Bezkontextová gramatika

Obecnou definici gramatiky již známe. Gramatiky lze však dále dělit podle tvaru přepisovacích pravidel, jež obsahuje množina P. Toto dělení je známo jako Chomského hierarchie.

Nás zajímají bezkontextové gramatiky[1], jelikož LL syntaktický analyzátor pracuje s tímto typem gramatik. Podle Chomského hierarchie jde o gramatiky typu 2, kompletní dělení lze najít například v [2] na straně 10.

Bezkontextová gramatika má pravidla zadaná ve tvaru $A \to \gamma$, kde $A \in N$ a $\gamma \in (N \cup T)^*$. Přídavné jméno *bezkontextová* v názvu gramatiky označuje fakt, že substituci pravé strany pravidla za nonterminál A lze provést bez ohledu na kontext (okolní symboly), ve kterém se nonterminál A nachází. Tato gramatika může obsahovat pravidla typu $A \to \varepsilon$, které připouští substituci nonterminálu A za prázdný řetězec. Gramatika z příkladu 2.2.1 je bezkontextová.

2.2.2 Redukování gramatiky

Zadaná gramatika nemusí vždy nutně obsahovat pouze pravidla, která jsou nezbytná. Toto může nastat buď chybou v návrhu nebo v důsledku převodů a úprav gramatiky z jiných forem. Správnost zadané gramatiky je vhodné zkontrolovat ihned po jejím nadefinování, lze tak odhalit případné chyby a upozornit na zbytečná pravidla již před započetím konstrukce LL tabulky. V bezkontextové gramatice mohou existovat tři typy zbytečných pravidel: obsahující nedefinované nonterminály, nedosažitelná a neproduktivní pravidla. K jejich nalezení poslouží algoritmy 2.2.1 a 2.2.2 popsané níže. Doprovodné texty a algoritmy převzaty a upraveny z [4].

Příklad 2.2.2 Ukázka zdánlivě bezproblémové gramatiky, na níž lze demonstrovat odstranění zbytečných pravidel:

$$\begin{aligned} S &\to AB|DE \\ A &\to a \\ B &\to bC \\ C &\to c \\ D &\to dF \\ E &\to e \\ F &\to fD \end{aligned}$$

Neproduktivní nonterminály a pravidla

Pokud dosažitelné pravidlo obsahuje na pravé straně nonterminál, pro který neexistuje posloupnost pravidel, jejichž uplatněním by došlo k expanzi na posloupnost terminálů nebo ε, jde o neproduktivní nonterminál.

K nalezení neproduktivních nonterminálů slouží následující algoritmus 2.2.1. Spočívá v nalezení produktivních pravidel, a označení zbytku jako neproduktivní. Produktivní pravidlo obsahuje na pravé straně pouze produktivní symboly. Terminály a ε přirozeně považujeme za produktivní. Nonterminály jsou produktivní tehdy, pokud pravidlo ve kterém figurují na levé straně, je označeno za produktivní. Na začátku označíme všechna pravidla

[1] Jazyky generované tímto typem gramatik se nazývají bezkontextové

a nonterminály jako neznámé[2]. Potom postupně procházíme všechna neznámá pravidla a zjišťujeme, zda je celá jejich pravá strana produktivní. Pokud ano, označíme pravidlo spolu s nonterminálem na jeho levé straně za produktivní. Opakujeme dokud nacházíme produktivní pravidla. Potom označíme všechna zbylá neznámá pravidla a nonterminály za neproduktivní. Překvapivě, jako vedlejší efekt, nalezneme i nedefinované nonterminály, jelikož nedefinovaný nonterminál nemůže být produktivní.

Algoritmus 2.2.1: Nalezení neproduktivních pravidel

Vstup: mnozina nonterminalu N, seznam pravidel P ve tvaru $A \rightarrow X_1X_2..X_n$
Výstup: mnozina neproduktivnich nonterminalu, seznam neproduktivnich pravidel
Metoda:

```
begin
    neznameN = N
    neznameP = P
    repeat
        zmena = false
        foreach pravidlo P_i z neznameP do
            foreach symbol X_i z P_i do
                if X_i je v neznameN then                    // neznamy symbol -
                    prejdi na pravidlo P_{i+1}                // nezname pravido
                end
            end
            odeber nonterminal A z neznameN
            odeber pravidlo P_i z neznameP
            zmena = true
        end
    until zmena = false
    neznameN jsou neproduktivni
    neznameP jsou neproduktivni
end
```

Při prvním průchodu jsou jako produktivní pravidla $A \rightarrow a$, $C \rightarrow c$ a $E \rightarrow e$, v druhém průchodu potom pravidlo $B \rightarrow bC$ a nakonec $S \rightarrow AB$. Tím už se nic nezmění a zbylá pravidla jsou tedy označena za neproduktivní. Ukázková gramatika bude po odstranění neproduktivních pravidel vypadat takto:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \\ B &\rightarrow bC \\ C &\rightarrow c \\ E &\rightarrow e \end{aligned}$$

[2]Protože nakonec všechna pravidla a nonterminály, které zůstanou *neznámé* jsou označeny za neproduktivní, lze při implementaci rovnou použít inicializační hodnotu *neproduktivní*. V textu by ovšem toto mohlo být matoucí.

Nedefinované nonterminály

Pokud se na pravé straně kteréhokoli pravidla vyskytuje nonterminál, jež se nevyskytuje na levé straně žádného pravidla, potom bychom při dalším nahrazování takového pravidla museli skončit chybou, protože pro tento nonterminál žádné další pravidlo nemáme. Takovému nonterminálu říkáme nedefinovaný.

Nedefinované nonterminály nalezneme při hledání neproduktivních pravidel (viz algoritmus 2.2.1).

Nedosažitelný nonterminál

Pokud existuje nonterminál na levé straně pravidla, ale neexistuje posloupnost pravidel, kterou bychom byli schopni dostat se ze startovního symbolu k tomuto nonterminálu, potom jde o nedosažitelný nonterminál. Pravidla obsahující tento nonterminál nebudou v gramatice nikdy využita.

Při nacházení těchto pravidel postupujeme obdobně jako při hledání neproduktivních pravidel. Na začátku opět označíme všechna pravidla a nonterminály jako neznámé. Startovní nonterminál označíme jako dosažitelný. Potom procházíme všechna neznámá pravidla a pokud je nonterminál na jejich levé straně dosažitelný, potom celé pravidlo a všechny nonterminály na pravé straně označíme jako dosažitelné. Opakujeme dokud nacházíme dosažitelná pravidla. Zbylá pravidla a nonterminály označíme za nedosažitelné.

Algoritmus 2.2.2: Nalezení nedosažitelých pravidel

Vstup: mnozina nonterminalu N, pocatecni nonterminal S, seznam pravidel P ve tvaru $A \rightarrow X_1X_2..X_n$
Výstup: mnozina nedosazitelnych nonterminalu, seznam nedosazitelnych pravidel
Metoda:

```
begin
    neznameN = N
    neznameP = P
    odeber S z neznameN
    repeat
        zmena = false
        foreach pravidlo P_i z neznameP do
            if nonterminal A z P_i neni v neznameN then
                foreach nonterminal X_i z P_i do
                    odeber X_i z neznameN
                end
                odeber P z neznameP
                zmena = true
            end
        end
    until zmena = false
    neznameN jsou nedosazitelne
    neznameP jsou nedosazitelne
end
```

Jak můžeme vidět, bylo odstraněno pouze pravidlo $E \rightarrow e$, které je sice produktivní, ale nonterminál E se nevyskytuje na pravé straně žádného dosažitelného providla. Gramatika tedy ve výsledku vypadá takto:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \\ B &\rightarrow bC \\ C &\rightarrow c \end{aligned}$$

2.3 Syntaktická analýza

Syntaktická analýza je proces, zjišťující zda zdrojový text (resp. posloupnost symbolů) odpovídá gramatice jazyka, v němž má být text napsán. Výsledkem úspěšné syntaktické analýzy je derivační strom reprezentující zdrojový text. Při neúspěchu je zpravidla výsledkem oznámení chyby ve zdrojovém textu.

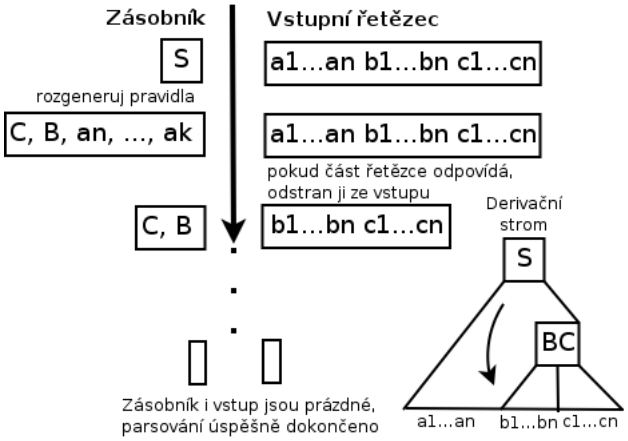
2.3.1 Metody syntaktické analýzy

Rozlišujeme dvě základní metody syntaktické analýzy: shora dolů a zdola nahoru. Jak názvy napovídají, jde o postup při vytváření derivačního stromu.

Při analýze shora dolů probíhá proces generování derivačního stromu postupným rozvíjením nonterminálních symbolů pravidel, počínaje startovním symbolem (kořenem stromu) ve snaze dospět k posloupnosti terminálních symbolů shodné se zdrojovým textem. Jelikož při tomto postupu nahrazujeme vždy nejlevější nonterminál, lze derivační strom reprezentovat posloupností pravidel, které říkáme levá derivace. Příkladem analyzátoru používajícího tuto metodu je LL analyzátor (viz. 2.3.1).

Při analýze zdola nahoru postupujeme opačně. Snažíme se zredukovat posloupnost terminálů na startovní nonterminál. Při tomto postupu redukujeme zprava, proto je výsledkem derivační strom odpovídající pravé derivaci. Tuto metodu využívá LR analyzátor.

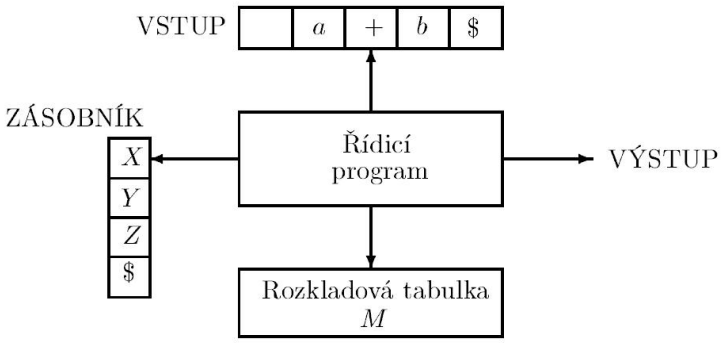
O metodách syntaktické analýzy se lze více dočíst v [3], [4] nebo [7].



Obrázek 2.1: Znázornění syntaktické analýzy shora dolů, převzato z [12]

2.3.2 LL syntaktický analyzátor

Jedná se o tabulkou řízený syntaktický analyzátor využívající metodu shora dolů. LL[3] v názvu říká, že čte vstupní posloupnost zleva doprava a tvoří nejlevější derivaci věty. Podmnožina bezkontextových gramatik, které lze analyzovat tímto analyzátorem se nazývají LL gramatiky. LL(k) syntaktický analyzátor potřebuje znát k symbolů ze vstupu, aby mohl rozhodnout, které pravidlo použít. Gramatiky, které jsou analyzovatelné LL(k) syntaktickým analyzátorem bez nutnosti zpětných kroků, se nazývají LL(k) gramatiky. Nejpopulárnější jsou LL(1) gramatiky, které lze analyzovat LL(1) syntaktickým analyzátorem - ten se totiž dokáže rozhodnout na základě jednoho přečteného symbolu ze vstupu [13]. Pomocné algoritmy sloužící k tvorbě LL tabulky, stejně jako algoritmus LL syntaktické analýzy jsou převzaty a upraveny z [6] a [8] a odpovídají tak metodám vyučovaným na fakultě Informatiky VUT v Brně. Některé jiné zdroje uvádějí modifikované metody, například nepoužívají množinu Empty() nebo uvádějí LL tabulku v jiném tvaru.



Obrázek 2.2: Model LL syntaktického analyzátoru, převzato z [3]

LL syntaktický analyzátor postupuje dle algoritmu 2.3.1. Ke své funkci potřebuje ještě další tři komponenty (viz také obr. 2.2):

- vstupní sekvenci tokenů zakončených speciálním znakem $
- zásobník, který umožňuje vkládání a vybírání nonterminálních a terminálních symbolů gramatiky
- rozkladovou (rozhodovací) tabulku, obsahující pravidla gramatiky na některých průsečících terminálů a nonterminálů. Nazýváme ji LL tabulka. Více o ní v podsekci 2.3.3.

[3] LL pochází ze zkratky anglického Left to right a Leftmost

Algoritmus 2.3.1: Syntaktický analyzátor

Vstup: mnozina terminalu T, mnozina nonterminalu N, startovni nonterminal S, vstupni sekvence I, LLtabulka obsahujici pravidla ve tvaru $A \rightarrow X_1X_2..X_n$
Výstup: levy rozbor L
Metoda:

```
begin
    uspech,chyba = false
    vlozNaZasobnik($)
    vlozNaZasobnik(S)
    repeat
        a=ctiVstup(I)
        V=vrcholZasobniku()
        if V = $ and a = $ then
            uspech = true
        else if V je v T and X = a then
            odstranZeZasobniku(V)
            a = ctiVstup(I)
        else if V je v N and LLtabulka(V, a)! = NULL then
            P = LLtabulka(V, a)        // Vrati pravidlo na pruseciku [V,a]
            uloz P do L
            odstranZeZasobniku(V)
            foreach symbol X_i z X_n..X_2X_1 do            // V obracenem poradi!
                vlozNaZasobnik(X_i)
            end
        else
            chyba = true
        end
    until chyba or uspech
    if uspech then
        kompletni levy rozbor v L
    end
end
```

Množina First a Empty

Množina First(X) určuje pro řetězec $X \in (N \cap T)^+$ všechny terminální symboly, jimiž mohou začínat řetězce derivované z řetězce X. Množina Empty(X) potom určuje, zda lze z řetězce X derivovat prázdný řetězec ε [3]. Pro určení množin First a Empty pro celé řetězce je potřeba nejprve nalézt množiny First a Empty pro každý jednotlivý symbol.

Algoritmus 2.3.2: Nalezení množin First a Empty pro jednotlivé symboly

Vstup: mnozina terminalu T, mnozina nonterminalu N, seznam pravidel P ve tvaru $A \rightarrow X_1X_2..X_n$
Výstup: mnozina First() a Empty() pro kazdy ze symbolu
Metoda:

```
begin
    First(ε) = ∅
    Empty(ε) = {ε}
    foreach X z T do
        First(X) = {X}
        Empty(X) = ∅
    end
    foreach X z N do
        First(X) = {∅}
        if v P existuje pravidlo X → ε then
            Empty(X) = {ε}
        end
        Empty(X) = ∅
    end
    repeat
        zmena=false
        foreach P_i z P do
            i=0
            repeat                                  // dokud lze X_i nahrazovat ε
                i++
                First(A) ⟵ (First(X_i))
                zmena=true
            until Empty(X_i) = ∅
            if Empty(X_1X_2..X_n) = {ε} then
                Empty(A) ⟵ {ε}
                zmena=true
            end
        end
    until zmena=false
end
```

Algoritmus 2.3.3: First a Empty pro posloupnosti symbolů

Vstup: posloupnost U symbolu $X_1X_2..X_n$
Výstup: mnozina First(U) a Empty(U)
Metoda:

```
begin
    i=0
    repeat                                   // dokud lze X_i nahrazovat ε
        i++
        First(U) ⟵ (First(X_i))
    until Empty(X_i) = ∅
    vseEmpty = true
    foreach X_i do
        if Empty(X_i)={∅} then
            vseEmpty = false
        end
    end
    if vseEmpty = true then
        Empty(U) ⟵ {ε}
    end
    Empty(U) ⟵ {∅}
end
```

Množina Follow

Množinu Follow(N) pro nonterminál N definujeme jako množinu všech terminálních symbolů T, které se mohou vyskytovat bezprostředně vpravo od N[3].

Algoritmus 2.3.4: Nalezení množiny Follow pro nonterminály

Vstup: mnozina nonterminalu N, seznam pravidel P ve tvaru $A \rightarrow X_1X_2..X_n$, mnoziny First a Empty
Výstup: mnozina Follow() pro kazdy z nonterminalu
Metoda:

```
begin
    Follow(S) ⟵ {$}
    Follow(N) ⟵ ∅
    repeat
        zmena=false
        foreach P_i z P do
            foreach X_i z P_i do
                if X_i je v N and i <> n then
                    Follow(X_i) ⟵ (First(X_{i+1}..X_n))
                    zmena = true
                end
            end
            if Empty(X_{i+1}..X_n)={ε} then
                Follow(X_i) ⟵ (Follow(A))
                zmena = true
            end
        end
    until zmena=false
end
```

Množina Predict

Množina Predict(r) pro pravidlo r je množina všech terminálních symbolů, které mohou vzniknout expanzí pravidla r.

Algoritmus 2.3.5: Výpočet množiny Predict pro pravidla

Vstup: mnozina nonterminalu N, seznam pravidel P ve tvaru $A \rightarrow X_1X_2..X_n$, mnoziny First, Empty a Follow
Výstup: mnozina Predict() pro kazde z pravidel
Metoda:

```
begin
    foreach P_i z P do
        Predict(P_i) = First(X_1X_2..X_n)
        if Empty(X_1X_2..X_n)={ε} then
            Predict(P_i) ⟵ Follow(A)
        end
    end
end
```

2.3.3 LL tabulka

LL tabulka je struktura, do níž ukládáme pravidla gramatiky. Hlavičky jejích sloupců tvoří terminální symboly a speciální symbol $, hlavičky řádků tvoří nonterminální smyboly. Pravidla ukládáme do buněk na průsečíku řádku a sloupce odpovídající dvojici [terminál, nonterminál] tak, že pravidlo potom určuje příští očekávanou expanzi při syntaktické analýze daného řetězce. Konstrukci samotné tabulky musí předcházet výpočet pomocných množin First(X), Empty(X) a Follow(X). Na základě těchto množin lze teprve vypočítat množinu Predict(r) pro každé z pravidel a pomocí množiny Predict vyplnit rozhodovací LL tabulku.

Algoritmus 2.3.6: Naplnění LL tabulky

Vstup: mnozina nonterminalu N, mnozina terminalu T, seznam pravidel P ve tvaru $A \rightarrow X_1X_2..X_n$, mnoziny First, Empty, Follow a Predict
Výstup: Vyplnena LL tabulka
Metoda:

```
begin
    LLtabulka.hlavickySloupcu = T + $
    LLtabulka.hlavickyRadku = N
    foreach P_i z P do
        foreach T_i z Predict(P_i) do
            if LLtabulka.bunka[A, T_i] = null then
                LLtabulka.bunka[A, T_i] = P_i          // A je z N, T_i je z T ∪ $
            else
                Chyba, gramatika neni LL.
            end
        end
    end
end
```

Kapitola 3

Návrh aplikace

3.1 Didaktičnost aplikace

Při návrhu jsem se snažil vyhovět následujícím požadavkům na didaktičnost aplikace a zvýšit tak její celkový přínos a možné uplatnění. Dobrou učební pomůcku lze charakterizovat např. těmito kritérii vybranými z [1]:

- tam, kde je nutno zdůraznit složitost jevů, genezí, vzájemné vztahy a souvislosti, dynamiku procesu, necháváme možnost pro dotváření pomůcky během výkladu
- přenosový kanál volíme podle požadavku na efektivnost přenosu informací a interakce mezi vyučujícím a žákem, ale i z hlediska hygieny duševní práce.
- pomůcky simulují objektivní skutečnost
- vyžadují (až na výjimky) realizaci pomocí vyučovací techniky
- předpokládá se, že budou do výuky včleněny ve formě demonstrace, nebo budou objekty bezprostřední manipulace žáků
- vysoce aktivizují, vytvářejí přechod od neúmyslné k úmyslné pozornosti
- pomáhají překonávat útlum, který je často přirozenou obranou organizmu proti nadměrnému vyčerpání anebo reakcí na jednotvárnost a nudu

Z toho vyplývá, že dobrá interaktivní pomůcka by měla umožňovat jak učiteli, tak samotnému žákovi určitý stupeň manipulace a experimentování metodou "pokus-omyl" nebo "co se stane když…". Tím může napomoci zaujmout studenty nejen během výkladu, ale také ve volném čase a lépe si tak procvičit a zapamatovat probíranou látku.

Demonstrační aplikace popisovaná v této práci by se dle [14] dala zařadit do kategorie Počítačem podporovaná výuka (angl. CAI - computer assisted instruction). Taková pomůcka umožňuje učiteli individualizovat výuku (například při cvičení) a přebírá od něj řadu rutinních úkolů v průběhu přípravy a samotné výuky, uvolňuje tak učiteli ruce k jiným výukovým aktivitám.

3.2 Výběr platformy a vývojového prostředí

Cílem bylo vytvořit interaktivní aplikaci, obsahující grafické uživatelské rozhraní, která je navíc multiplatformní. Tento požadavek splňují například platforma Java, Adobe Flash nebo různé nadstavbové knihovny pro jazyk C++ (např. Qt, wxWidgets).

Protože jsem absolvoval seminář Javy, zvažoval jsem použití této platformy. Odrazovalo mě však příliš pracné vytváření uživatelského rozhraní a určitá strohost výsledné aplikace. Podařilo se mi ovšem najít kompromis ve formě nedávno[1] vzniklé platformy JavaFX od společnosti Sun Microsystems (tedy tvůrců jazyka Java). JavaFX je podrobněji popsána v podkapitole 3.3.

Aplikaci jsem začal vyvíjet v JavaFX Scriptu verze 1.2, ale v průběhu práce přišla netrpělivě očekávaná verze 1.3, která přinesla nové prvky, řeší chyby předchozí verze a hlavně slibuje větší podporu operačního systému Linux. To s sebou přineslo menší potíže, dané dílčí zpětnou nekompatibilitou nové verze, které se však podařilo vyřešit.

Vývojová prostředí pro JavaFX jsou v zásadě dvě, a to Eclipse a NetBeans. Vybral jsem si NetBeans, protože jsem s ním již v minulosti pracoval. Prostředí NetBeans navíc pochází přímo od firmy Sun Microsystems a obsahuje tudíž nástrojové lišty pro přímou práci s jazykem JavaFX. S novou verzí JavaFX Scriptu vyšlo i NetBeans 6.9, které navázalo na předchozí verzi 6.8. Vývoj probíhal na operačním systému Windows XP a Windows 7, s následným ověřením funkčnosti na různých prohlížečích pod MacOS a Linuxem.

3.3 JavaFX

Platforma JavaFX je zaměřena na vývoj grafiky bohatých, platformně nezávislých programů. Vznikla jako konkurence k již zavedeným platformám jako je Adobe Flash a Microsoft Silverlight, nabízí tedy podobné možnosti. Navíc je výsledná aplikace schopna existovat jak ve formě appletu ve webovém prohlížeči, tak ve formě offline desktopové aplikace. Zároveň je JavaFX zaměřena i na jiná média než počítač a web. Aplikace běží i na podporovaných mobilních telefonech a televizích. Naplňuje tak motto "Jednou napiš, všude spusť!"[2].

JavaFX využívá ke svému běhu Java Virtual Machine. Vývojový jazyk pro platformu JavaFX se nazývá JavaFX Script. Jde o staticky typovaný, deklarativní, skriptovací jazyk. Při psaní záleží na velikosti písmen (podobně jako v Javě). Z celého jazyka lez vypozorovat snahu o intuitivnost a pro anglicky mluvícího vývojáře srozumitelnost a jednoduchost na první pohled. Tím se JavaFX zpřístupňuje širšímu okruhu zájemců. Ovšem programátor zvyklý na některé konstrukce běžné v Javě nebo C++ si musí chvíli zvykat. V některých ohledech se spíše podobá skriptovacímu jazyku Python.

3.3.1 Vytváření instancí třídy

Pro vytvoření instance třídy slouží následující konstrukce: `JmenoTridy {promenna: hodnota, dalsiPromenna: hodnota}` Narozdíl od Javy tedy chybí konstruktor `new` a veřejné proměnné třídy lze jednoduše inicializovat již při vytvoření instance. Hlavní přednost tohoto způsobu tkví v možnosti použití níže popsaných datových vazeb.

3.3.2 Datové vazby

Datové vazby (angl. data binding) slouží ke svázání proměnných, resp. navázání jedné proměnné na druhou (nebo na hodnotu výrazu). To umožňuje velmi jednoduše vytvářet závislosti, kdy se změna jedné proměnné dynamicky projeví na jiné, navázané proměnné,

[1] JavaFX v1.0 zveřejněna 8. prosince 2008, viz `http://blogs.sun.com/javafx/entry/javafx_1_0_is_live`

[2] V originále "Write once, run everywhere!"

bez potřeby použití jakékoliv další konstrukce typu `listener` známé z Javy. K tomu slouží klíčové slovo `bind`, které naváže levou stranu na výraz na pravé straně. Například `width: bind height*2` mění automaticky šířku v závislosti na dvojnásobku výšky. Lze takto snadno realizovat konzistentní napojení model-pohled. Toto je relativně důležitá a hodně používaná vlastnost JavaFX Scriptu. Od JavaFX verze 1.3 jsou vazby tzv. líné, tj. aktualizují se až v případě požadavku na hodnotu navázané proměnné[9]. To s sebou přineslo nárůst výkonu.

3.3.3 Sekvence

Další zvláštností je sekvence, jež nahrazuje pole známé z Javy i jiných programovacích jazyků. Nabízí zjednodušený a optimalizovaný systém práce s položkami. Práci se sekvencemi podrobně popisuje [10]. Proměnná typu sekvence se vytváří pomocí uvedení hranatých závorek `[]` za typem proměnné a pomocí hranatých závorek ji lze také naplnit hodnotami při inicializaci. Pro práci se sekvencemi slouží klíčová slova: `insert polozka into sekvence` pro přidání položky do sekvence, `delete polozka from sekvence` pro odebrání položky, `sizeof sekvence` pro počet položek, `sort` pro setřízení a `reverse` pro otočení. Sekvence v JavaFX jsou pouze jednorozměrné. Lze je sice do sebe vkládat, ale výsledná sekvence je vždy zploštěna na jednorozměrnou. Vícerozměrné sekvence lze tvořit pomocí sekvence objektů obsahující další sekvence.

Zajímavá a praktická je také možnost výřezu (angl. slice) některých položek, ze kterých je pak automaticky vytvořena nová sekvence. Například: `[3..5], [..5], [5..<]` vybere 3.-5. položku, první až 5., a 5. až předposlední.

Rozdílné je také procházení sekvence. JavaFX nezná klasický cyklus `for`, místo něj se používá konstrukce `for nazev_polozky in sekvence` {`akce`} což vykoná požadovanou akci pro každou položku v sekvenci. Pro zjištění indexu aktuálně zpracovávané položky slouží klíčové slovo `indexof nazev_polozky`.

3.3.4 Grafické prvky

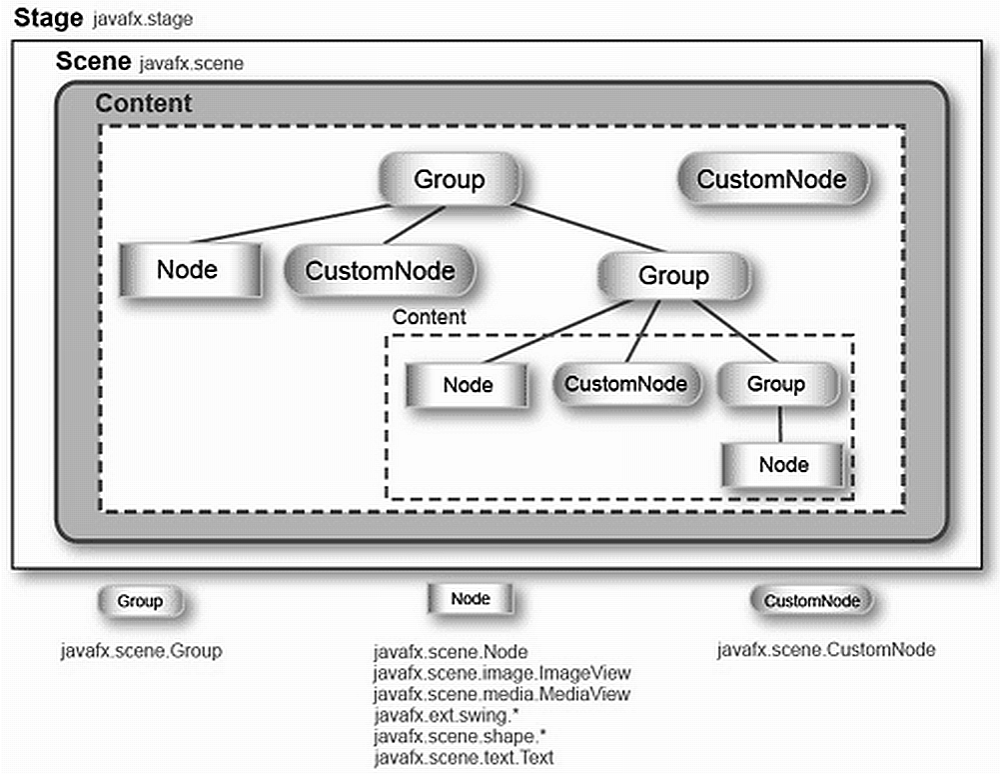
JavaFX je zaměřena na tvorbu GUI aplikací, a proto obsahuje širokou podporu pro tvorbu grafických prvků. Jednak jde o paletu základních geometrických tvarů, již hotových klasických ovládacích prvků (typu tlačítka, textová pole, atd.). Navíc umožňuje poměrně snadnou tvorbu vlastních složitějších prvků (pomocí třídy `CustomNode`) skládáním prvků nižší úrovně.

Grafické prvky aplikace jsou uspořádány do stromu, jehož kořenem je okno (třída `Stage`), jež obsahuje scénu (třída `Scene`), do níž umisťujeme grafické prvky (třída `Node`) nebo skupiny prvků (např. třída `Group`). Výsledné uspořádání lze vidět na obrázku 3.1. Scén můžeme mít obecně několik a podle potřeby je lze v okně přehazovat.

3.4 Návrh řešení

Jelikož podstata aplikace je demonstrace několika algoritmů založená na jejich krokování, bylo potřeba vymyslet vhodný způsob jak toto krokování realizovat. Zastavovat přímo průběh prováděného algoritmu ve stylu debuggeru by bylo sice možné[3], ale velmi neefektivní, navíc by nebylo možné krokovat dozadu. Proto je lepší nechat algoritmus doběhnout,

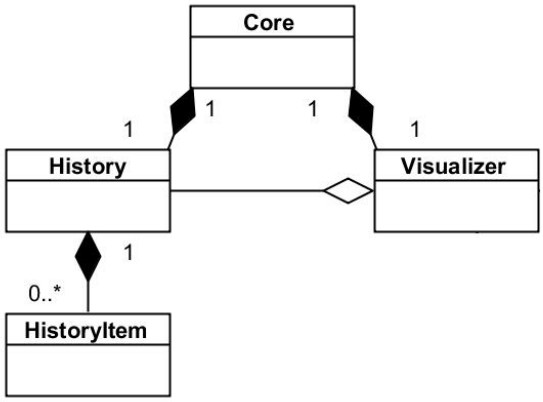
[3] Zřejmě s využitím vláken, nekonečné smyčky a volání `pause()`.



Obrázek 3.1: Schéma uspořádání prvků GUI pomocí JavaFX skriptu. Převzato z [11].

ale veškeré prováděné kroky ukládat do historie. Krokování je potom založeno na procházení historie a rekonstrukci běhu z uložených záznamů.

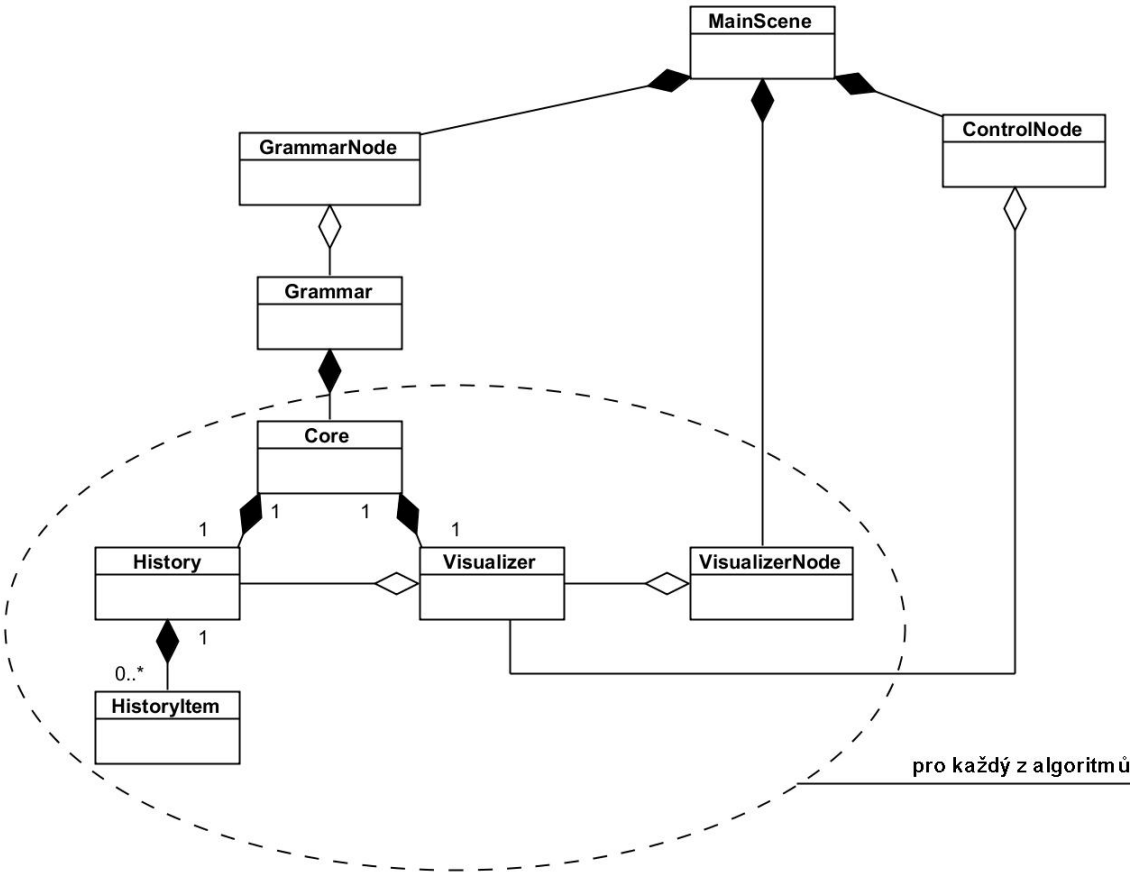
Pro každý z algoritmů je tedy obecně nutné mít jádro provádějící vlastní algoritmus a k němu připojenou historii a vizualizér, který drží stav klíčových proměnných jádra a nastavuje je podle záznamů z historie. To vede k návrhu na obrázku 3.2.



Obrázek 3.2: Diagram tříd umožňující krokování algoritmu

Vstupem těchto algoritmů mohou být prvky gramatiky a také výsledky ostatních (předešlých) algoritmů. To lze řešit tak, že gramatika je algoritmům nadřazena a zároveň přes ni mohou komunikovat jednotlivé algoritmy mezi sebou.

Dále je potřeba každý z vizualizérů těchto algoritmů zapojit do grafického uživatelského rozhraní. Aby se mohl stav vizualizéru projevit v grafickém rozhraní, je potřeba vytvořit korespondující grafický prvek typu `Node` (viz 3.3.4). Totéž je potřeba udělat i pro gramatiku, aby bylo možné zobrazit ji a editovat. Dále potřebujeme zobrazit ovládací panel, s popisem použitého algoritmu.



Obrázek 3.3: Celkový diagram tříd

Protože krokovat každý algoritmus nezávisle by bylo zbytečné a dokonce nežádoucí, jelikož na sebe navazují, vystačíme si s jedním centrálním grafickým ovládacím prvkem, který bude ovládat vždy jeden z vizualizérů a dokáže se mezi nimi přepínat. Zároveň by měl být schopen zvýraznit v grafickém rozhraní, který z vizualizérů je právě aktivní.

Na základě předcházejícího návrhu lze sestavit celkový diagram tříd, který je na obrázku 3.3. Část diagramu označená elipsou je obecná a každý ze zobrazovaných algoritmů tyto obecné třídy s využitím polymorfismu implementuje obecné prototypy funkcí, případně rozšiřuje o vlastní funkce.

3.5 Návrh grafického uživatelského rozhraní

Uživatelské rozhraní by mělo uživateli umožnit:

- Definování gramatiky zadáním pravidel.
- Zobrazení aktuálního stavu konstruovaného prvku.
- Ruční krokování algoritmů.
- Pohyb mezi jednotlivými částmi programu.
- Zadání vstupu pro syntaktickou analýzu a její provedení.

Zadávání vstupů probíhá pomocí textových polí. Krokování bude spojeno s pohybem po programu v jeden celek, umožňující procházet jednotlivé části programu představující jednotlivé algoritmy.

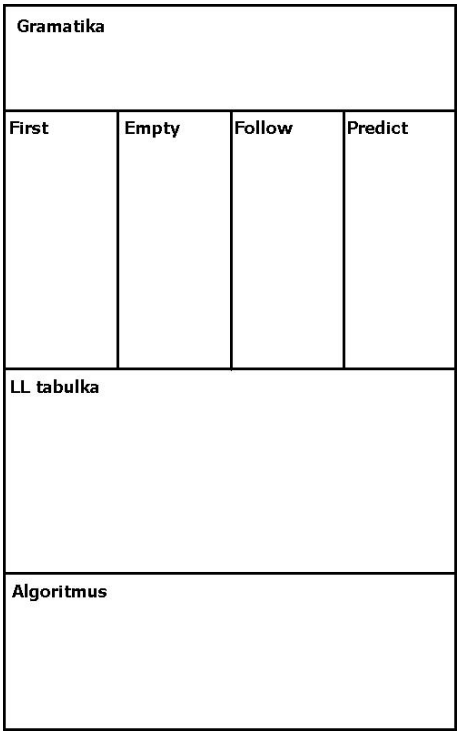
Pro každou z výše vyjmenovaných činností je potřeba zobrazit vhodné uživatelské rozhraní. Nejprve je nutné umožnit uživateli zadat vstupní data pro definici gramatiky. Gramatiku je třeba ošetřit od neproduktivních a nedosažitelných pravidel. Dále konstrukci samotné tabulky předchází tvorba pomocných množin, které na sebe navazují. Je tedy potřeba uživateli postupně zobrazit demonstraci hned několika algoritmů. Poté lze sestavit a zobrazit výslednou LL tabulku. Na závěr zbývá ještě ukázka syntaktické analýzy. Pro každý z těchto algoritmů musí být k dispozici obecný popis a aktuálně prováděný krok.

Má původní představa vycházela z konceptu "vše na jednom místě" aby bylo jasně vidět všechny souvislosti, bez nutnosti přepínat pohledy nebo záložky. Nakonec se přece jen jedno přepnutí pohledu ukázalo jako vhodné. A to při přechodu na ukázku provedení syntaktické analýzy. V tomto okamžiku je již LL tabulka sestavená, a proto ji lze posunout na místo, které zabíraly pomocné množiny First, Empty, Follow i Predict a syntaktickou analýzu provést uprostřed okna.

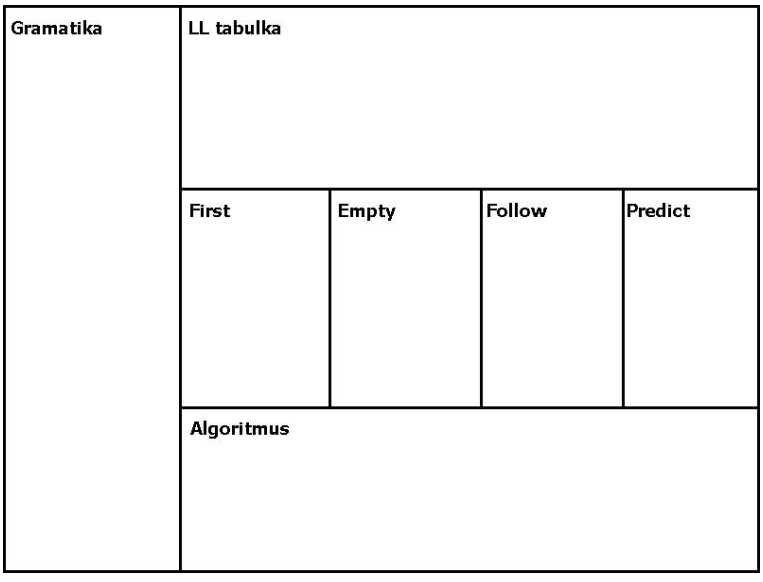
Následující obrázky ukazují dva z návrhů rozmístění prvků v okně.

Původní návrh uspořádání "na výšku" (viz obr. 3.4) se ukázal jako nevhodný, protože dostatečně nevyužívá plochy monitoru či promítacího plátna, orientovaného zpravidla "na šířku". Navíc je přehlednější mít pravidla gramatiky napsaná pod sebou, avšak v tomto uspořádání pro ně není místo. Proto jsem začal uvažovat o jiné, vhodnější kompozici.

Návrh na obrázku 3.5 se mi jevil přijatelný jak z hlediska přehlednosti, tak z hlediska optimálního využití plochy. Sloupcový tvar části pro gramatiku umožňuje zadávání všech pravidel najednou pomocí víceřádkového textového pole. Ve spodní části je zase dostatek místa pro vypsání jednotlivých algoritmů.



Obrázek 3.4: Uspořádání “na výšku”



Obrázek 3.5: Uspořádání “na šířku”, které je vhodnější

Kapitola 4

Implementace

V této kapitole postupně popíšu konkrétní implementaci nejdůležitějších tříd odpovídajících navrženému diagramu 3.3.

Samotná implementace aplikace byla zpočátku komplikována jednak neznalostí programovacího jazyka, tedy experimentováním s kusy kódu a také tím, že platforma JavaFX není na trhu dlouho, proto je dostupná literatura omezená na dokumentaci od výrobce a různé ukázky kódu na webových stránkách příznivců a nadšenců. Navíc JavaFX stále prochází relativně podstatnými změnami, mnohdy bez zpětné kompatibility. Tyto změny byly nejvíce patrné při přechodu z verze 1.2 na verzi 1.3, ale také při studiu mnohých ukázek napsaných pro dřívější verze jazyka (např. v1.1), používajících značně odlišné konstrukce.

4.1 Gramatika

Základní logická část programu reprezentující gramatiku, pro kterou chceme sestavit LL tabulku, je realizována třídou `Grammar`. Tato třída obsahuje implementaci funkcí, které zpracují vstup od uživatele.

První z těchto funkcí je `splitRule()`, která s pomocí funkcí `insertRule()` a `createRule()` rozpozná správně zadaná pravidla ze vstupu, a postupně vytvoří sekvenci pravidel zvanou `rawrules` obsahující všechna správně i špatně zadaná pravidla. Účel uchování všech pravidel je ve vytvoření prostoru pro pozdější zneplatnění některých pravidel při ošetřování gramatiky. Správně zadané pravidlo lze definovat regulárním výrazem:
`[whitespace]*[uppercase]+[alphanum]*[whitespace]+->[whitespace]+.*`

Musí tedy začínat velkým písmenem, za nímž mohou následovat další písmena která jsou pak alespoň jedním bílým znakem oddělena od šipky, za kterou opět následuje alespoň jeden bílý znak a za ním následuje libovolná posloupnost znaků představující jednotlivé symboly oddělené bílými znaky. Tedy např. `A -> B c D` nebo `Item -> Item1`.

Epsilon pravidlo je definováno buď prázdným řetezcem za šipkou, nebo slovem `eps`, případně znakem ε. Například `A ->` nebo `A -> eps`.

Špatně zadané pravidlo je označeno symbolem # a samozřejmě není k tvorbě gramatiky využito. V seznamu pravidel jsem se rozhodl ponechat i špatně zadaná pravidla, protože většinu chyb (typu překlepy) lze jednoduše opravit bez opětovného psaní pravidla. Navíc lze využít symbolu # také k uvození komentářů.

Funkce `insertRule()` vloží správně zadané pravidlo do seznamu pravidel, špatně zadanému přiřadí smybol #.

Po vytvoření nebo po změnění seznamu pravidel je použita funkce `checkTokens()` pro

nalezení na terminálů a nonterminálů vyskytujících se v pravidlech a následně je pomocí `insertToken()` vloží do patřičné množiny.

Startovním nonterminálem je určen první načtený nonterminál (tj. levá strana aktuálně prvního zadaného pravidla). Tímto je definována čtveřice (N, T, S, P) definující gramatiku.

Třída `Grammar` vytváří instance tříd `Productive, Reachable, FirstSet, EmptySet, FollowSet, PredictSet, Table, Analyzer` reprezentují odpovídající algoritmy použité pro výpočty nad gramatikou.

O grafické zobrazení množin N, T, S a textového vstupu pro zadání a zobrazení pravidel P gramatiky se stará třída `GrammarNode`. Textový vstup je řešen pomocí víceřádkového prvku `TextBox`, jednou z "novinek"JavaFX verze 1.3 [1]. Tlačítko Ok/Editovat zobrazené pod textovým vstupem slouží pro potvrzení resp. úpravu zadaných pravidel. Dále jsou zobrazena tlačítka pro nahrání a uložení zadaných pravidel z/do souboru.

Dialog pro výběr souboru není v JavaFX zatím implementován, rozhodl jsem se tedy "vypůjčit" si dialog `JFileChooser` z grafického prostředí Java Swing. Načtení vstupu ze souboru jsem původně řešil pomocí funkce `readLine()` ze standardní třídy `java.io.FileReader`, ovšem toto řešení se ukázalo jako nevhodné kvůli vlastnosti jazyka JavaFX, který vyhodnocuje prázdný řetězec a hodnotu `null` jako totéž. Jelikož funkce `readLine()` vrací hodnotu `null` při dosažení konce souboru, je zřejmé, že čtení probíhá právě do návratu této hodnoty. Při přečtení prázdného řádku je však vrácený prázdný řetězec intepretován také jako `null` a čtení je tudíž ukončeno. Toto řešení neumožňuje čtení souborů s prázdnými řádky.

Naštěstí existuje řešení v použití třídy `java.util.Scanner` a jejích funkcí `hasNextLine()` a `nextLine`, pomocí nichž lze číst i soubory s prázdnými řádky, protože čtení probíhá dokud platí podmínka `hasNextLine() == true`[5]. Uložení do souboru je řešeno jednoduše pomocí funkce `write` z třídy `java.io.FileWriter`. Do souboru se ukládá aktuální obsah textového pole pro zadání pravidel gramatiky, což je vzhledem k automatickému rozpoznání ostatních složek dostatečné.

4.2 Abstraktní model algoritmu

Následující podsekce popisují detailněji jednotlivé abstraktní třídy z diagramu 3.2. Součástí všech těchto tříd jsou mimo jiné inicializační funkce zajišťující shodný výchozí stav.

4.2.1 Jádro

Jádro je třída, která implementuje funkce nutné pro běh daného algoritmu. Pro získaní potřebných vstupních hodnot do těchto funkcí (např. terminály a nonterminály, výsledky předchozích algoritmů) je definováno propojení s gramatikou. Aby mohl být proces provedení algoritmu postupně uložen a později reprodukován pro potřeby krokování, definuje tato třída historii jako instanci třídy `History` a zobrazovací mechanismus jako instanci třídy `Visualizer`, který je navázán na odpovídající instanci historie.

V průběhu provádění algoritmu jsou tedy klíčové změny hodnot ukládány do historie.

4.2.2 Historie

Abstraktní třídy `History` a `HistoryItem` slouží jako vzor pro třídy, které mají za úkol archivovat změny provedené průběhem určitého algoritmu. Informace o změnách se ukládá

[1] Předchozí verze podporovala jen jednořádkové textové vstupy.

do sekvence položek typu `HistoryItem`, z nichž každá obsahuje informaci o změně hodnoty některé z proměnných algoritmu oproti předchozímu stavu. Pomocí indexu do této sekvence a funkcí `backward` a `forward` jež tento index ovládají, je možné získávat jednotlivé položky tak jak jdou po sobě v čase a tím umožnit krokování pro následnou vizualizaci.

Potomci těchto tříd jsou `ProductiveHistory`, `ReachableHistory`, `AnalyzerHistory`, `SetHistory`, `TableHistory`
resp. `ProductiveHistoryItem`, `ReachableHistoryItem`, `AnalyzerHistoryItem`, `SetHistoryItem`, `TableHistoryItem`.

4.2.3 Vizualizace

Abstraktní třída `Visualizer` slouží jako vzor pro třídy které zobrazují aktuální stav určitého algoritmu v čase. K tomu slouží položky historie vracející hodnoty sledovaných proměnných algoritmu, pohyb mezi nimi je zajištěn voláním odpovídajících funkcí `backward()` a `forward()`. Jednotlivé proměnné třídy `Visualizer` jsou nastaveny na stav odpovídající aktuální položce. Tím je reprezentován stav proměnných v určitém kroku daného algoritmu.

K zobrazení aktuálního stavu třídy `Visualizer` v grafickém uživatelském rozhraní slouží abstraktní třída `VisualizerNode` pomocí níž jsou proměnné daného algoritmu převedeny do podoby složeného grafického prvku. Tím je zajištěno oddělení dat od jejich reprezentace.

Potomci tříd `Visualizer` a `VisualizerNode` jsou `ProductiveVisualizer`, `ReachableVisualizer`, `AnalyzerVisualizer`, `SetVisualizer`, `TableVisualizer` resp. `ProductiveNode`, `ReachableNode`, `AnalyzerNode`, `SetNode`, `TableNode`.

4.3 Konkrétní algoritmy

Popis implementace jednotlivých algoritmů.

4.3.1 Produktivní pravidla

Třída `Productive` reprezentující jádro pro nalezení produktivních pravidel. Funkce `checkProductive()` implementuje algoritmus 2.2.1. Zajišťuje ukládání historie při jeho provádění. Vstupem funkce `checkProductive()` je seznam pravidel `rawrule`, který v upravené podobě také vrací. V něm jsou pravidla, která jsou neproduktivní, vyřazena a označena znakem ! a červenou barvou.

4.3.2 Dosažitelná pravidla

Třída `Reachable` reprezentující jádro pro nalezení dosažitelných pravidel velmi se podobá třídě `Productive`. Funkce `checkReachable()` implementuje algoritmus 2.2.2. Zajišťuje ukládání historie při jeho provádění. Vstupem i výstupem funkce `checkReachable()` je opět seznam pravidel `rawrule`. I zde jsou pravidla, která jsou nedosažitelná, vyřazena a označena znakem ! a červenou barvou.

4.3.3 Množina First

Třída `FirstSet` rozšiřující třídu `Set` implementuje funkce pro vytvoření množiny First(X) podle algoritmu 2.3.2. Funkce `create()` provede inicializaci množiny tj. First(N)={} pro

nonterminály, First(T)={T} pro terminály (výjimka: First(T)={T} neukládáme do historie, jde o konstantní přiřazení). Funkce `fill()` vyplní množinu First(X). Funkce `firstOf()` vrací množinu First(X) určenou pro obecný řetězec.

4.3.4 Množina Empty

Třída `EmptySet` rozšiřující třídu `Set` implementuje funkce pro vytvoření množiny Empty(X) podle algoritmu 2.3.2. Funkce `create()` provede inicializaci množiny tj. Empty(N)={} pro nonterminály, Empty(T)={} pro terminály (výjimka: Empty(T)={} neukládáme do historie, jde o konstantní přiřazení). Funkce `isEmpty()` vrací množinu Empty(X) určenou pro obecný řetězec.

4.3.5 Množina Follow

Třída `EmptySet` rozšiřující třídu `Set` implementuje funkce pro vytvoření množiny Follow(N) pro každý z nonterminálů podle algoritmu 2.3.4. Funkce `create()` provede inicializaci množiny tj. Follow(N)={}. Funkce `fill()` vyplní všechny množiny Follow(N).

4.3.6 Množina Predict

Třída `PredictSet` rozšiřuje třídu `Set` a implementuje funkce pro vytvoření množiny Predict podle algoritmu 2.3.5. Funkce `fill()` vyplní Predict(r) pro všechna pravidla.

4.3.7 Společné třídy množin

Protože jsou si všechyn čtyři množiny a jejich reprezentace v podstatě velmi podobné, využil jsem pro ně společné schéma. Konstrukce každé z množin je ukládána do historie `SetHistory`, která je součástí třídy `Set` a zobrazována pomocí vizualizéru `SetVisualizer`, který je též součástí třídy `Set` resp. zobrazována ve formě grafiky pomocí třídy `SetNode`.

4.3.8 LL tabulka

Protože JavaFX nepodporuje vícerozměrná pole (viz 3.3.3), je moje abstrakce tabulky tvořena polem hlaviček sloupců a polem řádků třídy `TableRow` obsahující hlavičky řádku a pole zbývajících buněk řádku. Výhoda takového řešení tkví v jednoduchém vyhledání prvku na dané pozici, protože požadovaný prvek leží na dvojici indexů odpovídající řádku a sloupci, které lze vyhledat přímo podle hlaviček.

Třída `Table` kromě samotné struktury tabulky implementuje i funkce pro konstrukci LL tabulky podle algoritmu 2.3.6. Funkce `create()` má na starost vytvoření prázdné tabulky. Ta je tvořena polem terminálů a symbolem $ představujícím hlavičky sloupců a jednotlivými řádky začínajícími nonterminálem a dále obsahujícími pole pravidel. Funkce `fill()` vyplní vytvořenou tabulku odpovídajícími pravidly, k tomuto účelu volá funkci `insertRule()`. Aby mohla LL tabulku následně využít syntaktická analýza, vrací funkce `getRule()` pravidlo na požadované pozici. Třída `Table` vytváří instanci historie `TableHistory` a vizualizér `TableVisualizer`. Položky historie jsou ve tvaru `[column, row, rule, error]` určující kam bylo vloženo pravidlo, případně chybu v podobě pokusu o vložení pravidla do již obsazené buňky.

K zobrazení vizualizéru slouží třída `TableNode` graficky reprezentující tabulku. Protože JavaFX API nenabízí hotový grafický prvek tabulky[2], musel jsem implementovat vlastní, sestávající z grafických primitiv. To s sebou přineslo několik komplikací.

V prvé řadě bylo potřeba vytvořit prvek reprezentující buňku tabulky - rozhodl jsem se pro tvar připomínající obrácené písmeno "L", pokud se takovéto elementy poskládají do mřížky, vytvoří tabulku s otevřenou levou a horní stranou. Tento element je vytvořen pomocí dvou kolmých čar a textového pole umístěného doprostřed. Výška elementu je konstantní, šířka je vypočtena předem tak, aby se do buňky tabulky vešel nejdelší ze symbolů gramatiky. Pravidlo je na dané pozici reprezntováno pouze číslem, jelikož dlouhá pravidla příliš roztahovala výslednou tabulku, na úkor přehlednosti.

Dále bylo potřeba elementy uspořádat do formy tabulky. Prvotní nápad použít struktury horizontálních prvků `HBox` tvořící řádky tabulky ve vertikálním prvku `VBox` ovšem nebylo možné realizovat. Důvodem byla nutnost vytvořit řádek složením hlavičky řádku a jednotlivých prvků řádku. Toto se mi bohužel nedařilo realizovat tak, aby bylo možné použít dynamické napojení hodnot z vizualizéru pomocí `bind`. Přišel jsem ovšem na způsob jak tuto nepříjemnost obejít. Povedlo se mi to pomocí prvku `Panel`, jež umožňuje definovat vlastní rozvržení vnitřního obsahu. Tabulku jsem pak poskládal ze tří samostatných částí: vrchní řádek hlaviček, levý sloupec hlaviček a vnitřní mřížka buněk.

Aby se tabulka vždy vešla do jí určeného prostoru, má proměnné měřítko, nastavené pomocí parametrů `scaleX` a `scaleY`, jejichž hodnota je vypočtena podle vzorce:

```
min((width / localWidth), (height / localHeight))
```

Kde `width` a `height` je požadovaná maximální šířka resp. výška tabulky, `localWidth` a `localHeight` jsou původní rozměry tabulky. Výběrem minima z těchto hodnot docílíme požadovaného efektu.

4.3.9 Syntaktická analýza

Jádro syntaktické analýzy tvoří třída `Analyzer` a implementuje funkci `analyze()` pro provedení LL syntaktické analýzy podle algoritmu 2.3.1. Tato funkce nejprve pošle vstupní text funkci `tokenize()`, která funguje jako jednoduchý lexikální analyzátor, rozdelí tedy vstupní posloupnost na tokeny, a ověří zda jde o prvky z množiny terminálů. Třída `Analyzer` vytváří instanci historie třídy `AnalyzerHistory`, instanci vizualizéru třídy `AnalyzerVisualizer` spravující stav zásobníku, zbyvajícího vstupu a výsledného levého rozboru. Položky historie jsou ve tvaru `[stackRemove, stack, input, rule, result]` vyjadřující změnu zásobníku, čtení vstupu, použité pravidlo a výsledek analýzy. Ke grafickému zobrazení vizualizéru slouží třída `AnalyzerNode`, která obsahuje textové pole na zadání vstupu, grafickou reprezentaci stavu zásobníku, nepřečteného vstupu a výsledný levý rozbor.

4.4 Krokování a zobrazení algoritmu

Třída `ControlNode` sdružuje a ovládá vizualizéry všech jednotlivých algoritmů. Umí v aktuálně kontrolovaném vizualizéru vyvolat provedení kroku vpřed a vzad a načíst z něj popis aktuálně prováděné akce. Také nastavuje proměnnou `focus` aktuálně kontrolovaného vizualizéru na hodnotu `true`, což se projeví jeho zvýrazněním.

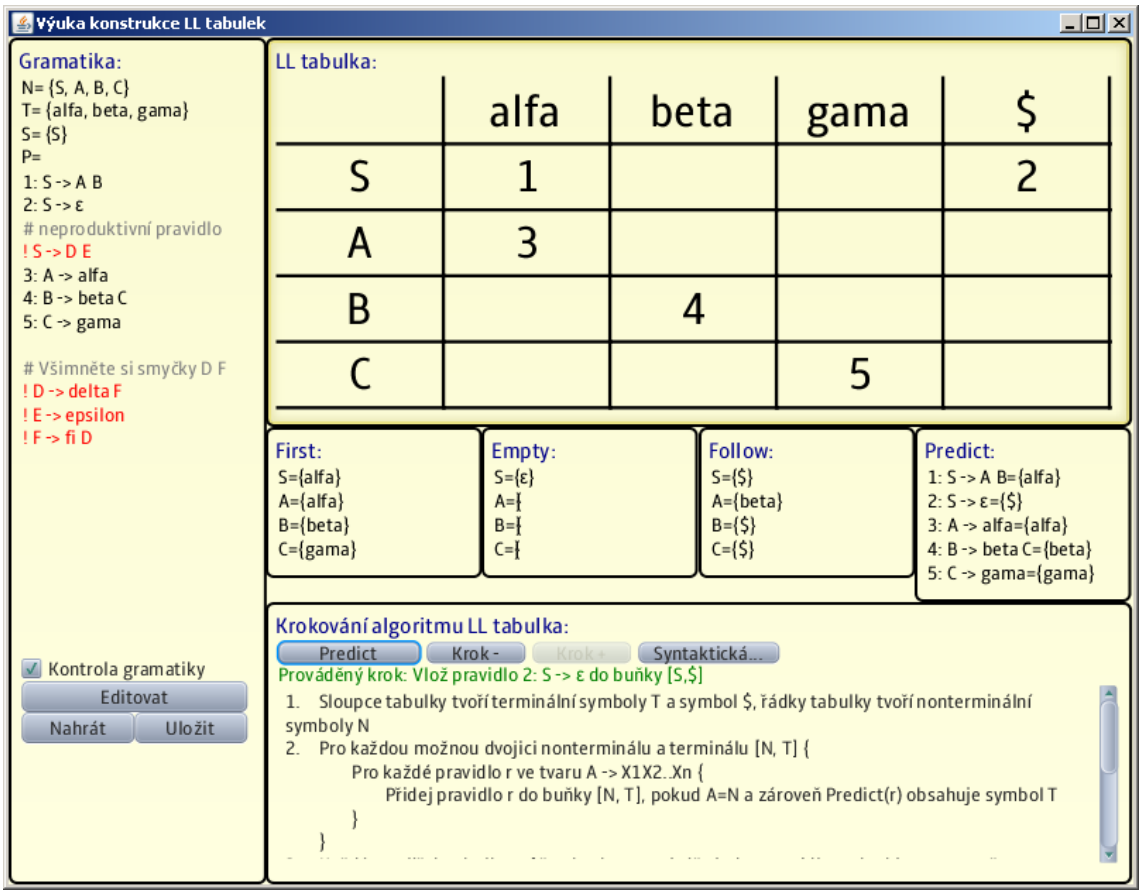
[2] V budoucí verzi 2.0 by se podle `http://javafx.com/roadmap/`[stav ke dni 14.5.2011] měla podpora tabulky objevit.

Instance `ControlNode` v grafickém uživatelském rozhraní slouží pro zobrazení krokovacích tlačítek, tlačítek pro pohyb mezi jednotlivými algoritmy, popisu aktuálně prováděné akce a obecného popisu právě prováděného algoritmu. Při přepínání mezi jednotlivými algoritmy je ve směru kupředu vždy dokončeno provádění současného a poté se teprve přepne na následující. Při přepínání směrem vzad je analogicky nejdříve současný algoritmus uveden do výchozího stavu a předchozí nastaven do stavu posledního kroku.

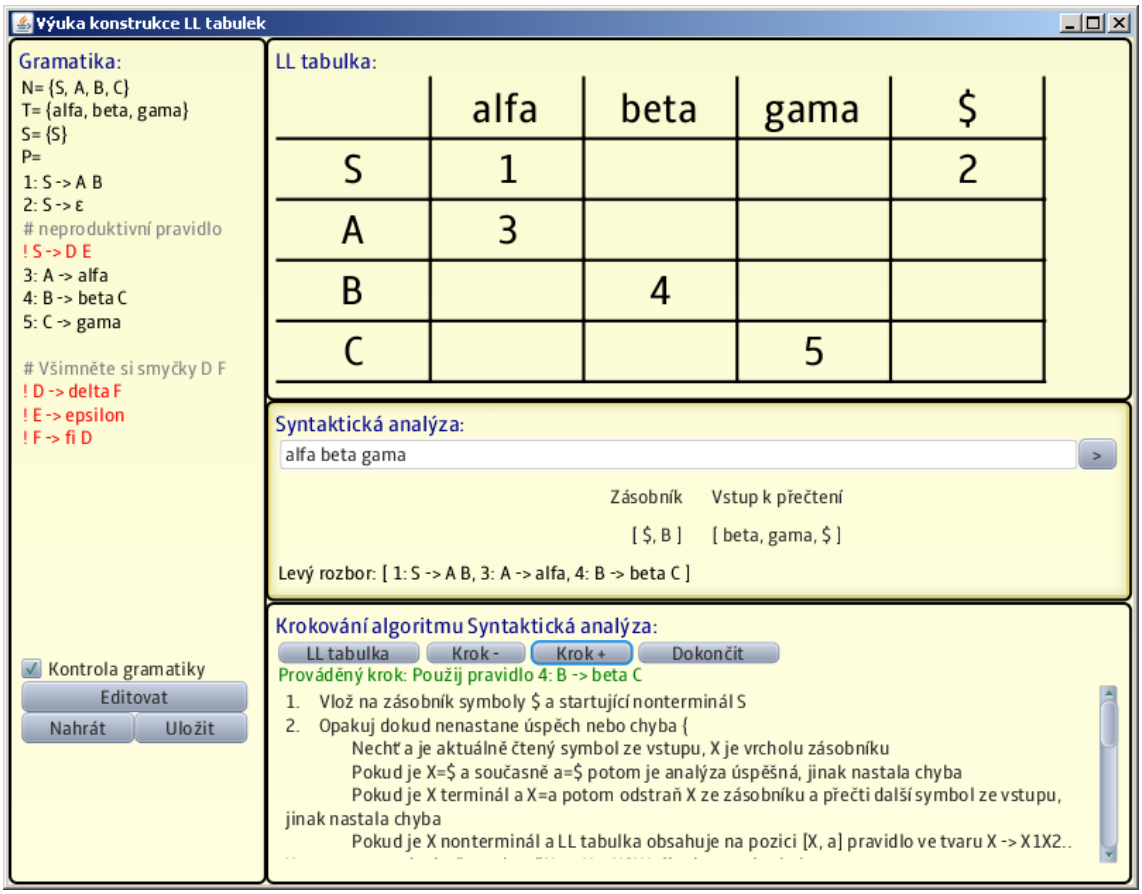
4.5 Grafické rozhraní

Protože knihovny JavaFX zahrnují pouze grafická primitiva a standardní ovládací prvky, musel jsem řešit problém optického oddělení jednotlivých částí, které obsahují grafickou reprezentaci hlavních částí aplikace. Rozhodl jsem se pro grafické ohraničení evokující představu samostatných panelů. Panel je vytvořen orámováním obdélníkem daných rozměrů, nadpisem funkčního celku a vnitřním prostorem pro zobrazenou komponentu. Dále panel umožňuje zvýraznění sebe sama pomocí dalšího obdélníku jiné barvy a dává tak uživateli zřetelně vědět, ve které části aplikace se právě nachází. Panel je řešen pomocí třídy `FrameNode`.

Jednotlivé panely jsou v okně rozmístěny podle schématu vzniklém ve fázi návrhu (viz obr. 3.5) tak, aby pružně reagovaly na případnou změnu rozměrů okna. To je zajištěno konstrukcí sestávající z prvků `VBox` a `HBox`, které představují svislé resp. vodorovné kontejnery. Okno programu běžícího na systému Windows 7 je zachyceno na obrázcích 4.1 a 4.2.



Obrázek 4.1: Výsledné grafické uživatelské rozhraní aplikace v průběhu tvorby LL tabulky



Obrázek 4.2: Grafické uživatelské rozhraní po přepnutí na ukázku syntaktické analýzy

Kapitola 5

Závěr

Samostatná práce na projektu takového rozsahu mi přinesla mnoho zkušeností, zejména prohloubila mé znalosti objektového návrhu a programování. Jistou výzvu představovalo naučit se používat programovací jazyk JavaFX, který se na fakultě Informatiky VUT v Brně nevyučuje, přičemž jde o moderní technologii, u které lze předpokládat uplatnění do budoucna.

Podařilo se mi splnit všechny kroky specifikované v zadání. Nad rámec zadání jsem navrhl a implementoval rozšíření ukazující kontrolu vstupní gramatiky. Výsledná aplikace vytvořená v rámci této práce je díky použité technologii plně multiplatformní, poskytující příjemné grafické uživatelské rozhraní. Avšak byť tvořena se značným úsilím, zcela jistě není dokonalá a do budoucna by bylo vhodné ji rozšířit například o následující funkce: vícejazyčné uživatelské rozhraní, možnost zvětšení samostatných panelů tak, aby zabíraly téměř celou plochu aplikace, u syntaktické analýzy například vykreslit odpovídající derivační strom.

Z didaktického hlediska je zvolená forma demonstrace pasivní, založená na pozorování průběhu konstrukce tabulky krok po kroku. Jako možné rozšíření si dovedu představit i koncepci založenou na aktivní formě učení - student by měl za úkol simulovat průběh daného algoritmu ručně a vyplnit jak pomocné množiny tak zkonstruovat tabulku (případně doplnit chybějící části), program by řešení průběžně ověřoval a upozornil na případné chyby.

Podobná aplikace již vznikla v rámci projektu DIDEFOM v roce 2006[1], a slouží pro podporu výuky v předmětu Formální jazyky a překladače na fakultě Informatiky VUT v Brně. Jelikož jsou algoritmy pro konstrukci LL tabulky jednoznačně definovány, lze stěží očekávat velkou odlišnost ve funkčnosti. Hlavní odlišnost tedy nastává v mnou implementovaném rozšíření a odlišné koncepci grafického uživatelského rozhraní. Nalezl jsem také několik dalších aplikací[2] [3] [4], které se zabývají konstrukcí LL tabulky či ukázkou LL sytaktické analýzy. Zobrazují ovšem pouze hotové výsledky a neumožní znázornění průběžného postupu metody (krokování) a student tak nemusí plně pochopit obtížná místa.

[1] [online] `http://www.fit.vutbr.cz/~meduna/work/didefom/doku.php?id=start`

[2] [online] `http://www.cs.binghamton.edu/~zdu/parsdemo/ll1frame.html`

[3] [online] `http://www.supereasyfree.com/software/simulators/compilers/principles-techniques-and-tools/parsing-simulator/parsing-simulator.php`

[4] [online] `http://smlweb.cpsc.ucalgary.ca/start.html`

Literatura

[1] Dostál, J.: UČEBNÍ POMŮCKY: Role učebních pomůcek ve výuce [online]. http://www.elektrotechnickestavebnice.xf.cz/ucebni_pomucky.htm, [cit. 2011-05-14].

[2] Češka, M.: Teoretická informatika - Učební texty vysokých škol [online]. http://www.fit.vutbr.cz/study/courses/TI1/public/Texty/ti.pdf, 2002 [cit. 2010-05-15].

[3] Češka, M.; Hruška, T.; Beneš, M.: Překladače - Učební texty vysokých škol [online]. http://www.fit.vutbr.cz/∼meduna/fjp/skripta.pdf, [cit. 2010-05-13].

[4] Grune, D.; Jacobs, C. J.: *Parsing techniques: A Practical Guide, Second Edition.* Springer, 2008, iSBN 978-0-387-20248-8.

[5] Herout, P.: *Učebnice jazyka Java: rozšířené vydání.* Kopp, 2007, iSBN 978-80-7232-323-4.

[6] Lukáš, R.: Příklady pro cvičení 5. z IFJ: Syntaktická analýza shora dolů [online]. https://www.fit.vutbr.cz/study/courses/IFJ/private/cviceni/Ifj05-cvic.pdf, [cit. 2010-05-10].

[7] Meduna, A.: *Automata and languages: theory and application.* Springer, 1999, iSBN 1-85223-074-0.

[8] Meduna, A.; Lukáš, R.: Formální jazyky a překladače - Studijní opora [online]. https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IFJ-IT/texts/OporaIFJ.pdf, 2006 [cit. 2010-05-13].

[9] Oracle: JavaFX 1.3 Migration Guide [online]. http://download.oracle.com/docs/cd/E17411_01/javafx/1.3/tutorials/porting-guide-javafx1-3.html, [cit. 2010-07-25].

[10] Oracle: Learning the JavaFX Script Programming Language [online]. http://download.oracle.com/javafx/1.3/tutorials/core/sequences/index.html, [cit. 2011-04-23].

[11] Oracle: Module 1 Task 1: Loading and Displaying an Image [online]. http://download.oracle.com/javafx/1.2/tutorials/mediabrowser/module1_task1.html, [cit. 2011-05-12].

[12] Vyskočil, M.: Jazyky a překladače - 4 (syntaxe 2) [online]. http://www.abclinuxu.cz/clanky/programovani/jazyky-a-prekladace-4-syntaxe-2, [cit. 2011-05-12].

[13] WWW stránky: LL parser [online].
`http://www.fact-index.com/l/ll/ll_parser.html`, [cit. 2010-05-15].

[14] Zounek, J.: *Elearning - jedna z podob učení v moderní společnosti.* MUNI press, 2009, iSBN 978-80-210-5123-2.