Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

# Diplomová práce

# Vizualizace algoritmů pro FJP

Plzeň, 2012

Daniel Hradecký

# Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 17. května 2012

Daniel Hradecký

# Abstract

This work deals with creating teaching aid for subject KIV/FJP at University of West Bohemia. In this subject, students are dealing with formal grammars and languages, finite-state machines, algorithms and tools for this topic. Student, who has successly finished this subject, is able to make a simple language compiler and has professional knowledges of the subject.

The theoretic part of this work is an overview of formal grammars and languages, finite-state machines and transformational algorithms. Than we will continue with description of possible web technologies for implementation of some algorithms and description of current existing solutions.

In the practical part, implementation of four algorithms in Adobe Flash / ActionScript is thoroughly described. These algorithms will be prepared for KIV/FJP lessons. Each algorithm is able to work with user grammar and show process of algorithm in text and visual form. The implementation is available on the CD and it will be used at the Courseware page of KIV/FJP subject.

# Obsah


1 Úvod ... 6
2 O předmětu FJP ... 7
2.1 Náplň předmětu ... 7
2.2 Obdobné předměty na jiných VŠ v ČR ... 8
3 Gramatiky, automaty, algoritmy ... 9
3.1 Základní pojmy ... 9
3.2 Formální gramatiky ... 9
3.2.1 Dělení formálních gramatik ... 11
3.2.2 Frázové gramatiky ... 11
3.2.3 Kontextové gramatiky ... 11
3.2.4 Bezkontextové gramatiky ... 12
3.2.5 Regulární gramatiky ... 12
3.3 Automaty ... 13
3.3.1 Formální definice automatu (matematický model) ... 13
3.3.2 Dělení konečných automatů ... 13
3.4 Algoritmy ... 14
3.4.1 Vlastnosti algoritmů ... 15
3.4.2 Obecné seznámení s algoritmy FJP ... 15
3.4.3 Algoritmus pro zjištění nedostupných symbolů ... 16
3.4.4 Algoritmus pro zjištění prázdného jazyka ... 17
3.4.5 Algoritmus pro zjištění zbytečných symbolů ... 18
3.4.6 Algoritmus pro vyloučení pravidla ... 19
3.4.7 Algoritmus pro odstranění jednoduchých pravidel ... 20
3.4.8 Algoritmus pro odstranění levé rekurze ... 21
3.4.9 Algoritmus first ... 22
3.4.10 Algoritmus follow ... 24
4 Vizualizace algoritmů ... 27
4.1 Technologie vizualizace ve webovém prohlížeči ... 27
4.1.1 Javascript ... 28
4.1.2 Java Applet ... 30
4.1.3 Adobe Flash ... 31
5 Stávající možnosti vizualizace algoritmů ... 33
5.1 Algoritmus first ... 33
5.1.1 Výhody ... 34
5.1.2 Nevýhody ... 34
5.2 Algoritmus follow ... 34
5.2.1 Výhody ... 35
5.2.2 Nevýhody ... 35
5.3 CYK algoritmus ... 36
5.3.1 Výhody ... 37
5.3.2 Nevýhody ... 37
5.4 Ostatní nástroje ... 37
6 Realizace vizualizací vybraných algoritmů ... 39
6.1 Požadavky ... 39
6.2 Analýza grafických objektů ... 41
6.2.1 Vstupní prvky ... 41
6.2.2 Informační textové prvky ... 41

6.2.3 Vizualizační prvky ... 42
6.2.4 Tlačítka ... 42
6.3 Řešení grafické části ... 42
6.3.1 Popis jmenných prvků layoutu ... 43
6.3.2 Popis vstupních a informačních prvků layoutu ... 45
6.3.3 Popis grafických prvků layoutu ... 46
6.3.4 Vizualizace ... 48
6.4 Část společného API ... 51
6.4.1 Datový model ... 51
6.4.2 Načítání gramatiky ... 52
6.4.3 Funkce rozhraní ... 54
6.4.4 Lokalizace ... 55
6.4.5 Generování gramatik ... 58
6.5 Implementace algoritmu pro zjištění nedostupných symbolů ... 59
6.6 Implementace algoritmu pro zjištění prázdnosti jazyka ... 62
6.7 Implementace algoritmu first ... 65
6.8 Implementace algoritmu follow ... 68
7 Testování ... 72
8 Závěr ... 73
9 Literatura a zdroje ... 75
10 Přílohy ... 77

# 1 Úvod

Každý, kdo studoval na Fakultě aplikovaných věd inženýrskou informatiku, si jistě prošel nemalým úskalím v řadě předmětů. Jedním z nich byl jistě i předmět Formální jazyky a překladače (zkráceně FJP). Je zřejmé, že kvalitní studijní materiály mohou výrazně pomoci s výukou. Ne vždy stačí ke správnému pochopení dané látky pouze teoretické texty. Jelikož jsme na technické fakultě, spousta látky je logického charakteru, který je potřeba nejen umět, ale hlavně i pochopit. Proto se poslední dobou klasická textová výuka doplňuje stále častěji nástroji, kde si můžeme danou funkčnost přímo vyzkoušet na příkladu. Jedním z těchto nástrojů jsou i vizualizace pro pochopení složitějších algoritmů. Za jeden z nejznámějších příkladů vizualizace algoritmů můžeme považovat vizualizaci řadících algoritmů. Přestože řadící algoritmy, např. typu Select Sort nebo Insert Sort, nejsou nijak složité algoritmy, jejich textový popis by byl mnohem složitější, než jednoznačná reprezentace, kde si student může daný algoritmus vyzkoušet na svém příkladu a tím i jednoduše pochopit jeho funkčnost. Cílem této diplomové práce je rozšířit toto portfolio o další množinu vizualizovaných algoritmů. Jak již bylo zmíněno, jedná se o algoritmy vyučované v předmětu KIV/FJP.

V této práci se nejdříve seznámíme s problematikou samotného předmětu, tedy s formálními jazyky, gramatikami a jednotlivými algoritmy pro transformaci bezkontextových gramatik. Poté si uděláme přehled o některých možných způsobech realizace těchto algoritmů a o použitelných technologiích. V posledním bodě teoretické práce se podíváme na již hotová řešení některých algoritmů a popíšeme si jejich funkčnost. V praktické části jsem vytvořil 4 funkční algoritmy, popisované v předmětu KIV/FJP. Seznámíme se s technologií, pomocí které byly algoritmy vytvořeny a podrobně si vysvětlíme jejich implementaci. Nakonec této diplomové práce ověříme jejich kvalitu a funkčnost.

# 2 O předmětu FJP

FJP je šestikreditový předmět určený pro studenty navazujícího studijního programu "Inženýrská informatika" na fakultě aplikovaných věd. Cílem předmětu je dát studentům důkladné znalosti o prostředcích a metodách zpracování formálních jazyků a jejich využití při implementaci programovacích jazyků, editorů, příkazových interpretů apod.

## 2.1 Náplň předmětu

Hlavním cílem předmětu je seznámit studenty s formálními metodami konstruování software. Pro správné pochopení této problematiky je důležité dát studentům potřebné znalosti o metodách a prostředcích zpracování formálních jazyků a jejich využití při implementaci programovacích jazyků či příkazových interpretů. To vše je plynule rozloženo do třinácti tříhodinových přednášek, jejichž náplň je následovná:

- Typy překladačů, základní struktura překladače.
- Regulární gramatiky a konečné automaty v lexikální analýze. FLEX.
- Úvod do syntaktické analýzy, metoda rekurzivního sestupu.
- Překlad příkazů.
- Zpracování deklarací.
- Přidělování paměti.
- Interpretační zpracování. Průběžný test.
- Generování cílového kódu.
- Vlastnosti bezkontextových gramatik.
- Deterministická analýza shora dolů.
- LL(1) transformace.
- Deterministická analýza zdola nahoru, LR gramatiky.
- Formální metody konstrukce software, generátory překladačů.

Aby student neměl problémy s absolvováním předmětu, jsou předpokládány základní vstupní znalosti z diskrétní matematiky, teoretické informatiky a programovacích technik. V neposlední řadě se jistě hodí i dobrá znalost vyššího programovacího jazyka, především Javy nebo C.
Student, který úspěšně absolvuje předmět, je nejen schopen úspěšně vytvořit překladač jednoduššího jazyka, ale získá i profesionální znalosti o důsledcích implementace různých konstrukcí programových textů a tím i schopnost jejich efektivnějšího využívání. Všechny tyto vědomosti pomohou studentům používat formální metody pro konstruování softwaru.

## 2.2 Obdobné předměty na jiných VŠ v ČR

Obdobou předmětu KIV/FJP je na ČVUT předmět s názvem Programovací jazyky a překladače (zkratka BI-PJP) [Z1]. Po úspěšném absolvování předmětu budou studenti umět základní metody implementace běžných programovacích jazyků. Získají zkušenost s návrhem a implementací překladu jednotlivých konstruktů programovacích jazyků (datové typy, podprogramy, apod.). Naučí se formálně specifikovat překlad textu, který vyhovuje určité syntaxi, do cílové formy a na základě této specifikace napsat překladač. Překladačem se zde rozumí nejen překladač programovacího jazyka, ale jakýkoliv jiný program analyzující a zpracovávající text zapsaný v jazyku, který je dán LL(1) gramatikou.
Předmět BI-PJP je na rozdíl od předmětu KIV/FJP určen pro studenty bakalářského studia na fakultě informačních technologií. Za úspěšné absolvování předmětu je studentovi přiděleno 5 kreditů.

Na Vysokém učení technickém v Brně se také vyučuje obdobný předmět se shodným názvem Formální jazyky a překladače (zkratka FIT-IFJ). Stejně jako na ČVUT v Praze je i tento předmět zařazen do bakalářského studia na fakultě informačních technologií. Hlavní cíl předmětu je též obdobný a to seznámit studenty s formálními jazyky a jejich modely a na jejich základě objasnit principy konstrukce překladačů.

# 3 Gramatiky, automaty, algoritmy

## 3.1 Základní pojmy

Abychom se mohli seznámit s problematikou této práce, je potřeba si ve stručnosti vysvětlit některé základní pojmy.

**Abeceda**
Libovolná neprázdná konečná množina **V** je abecedou. Její prvky nazýváme znaky nebo symboly.

**Slovo**
Slovo nad abecedou je libovolná posloupnost konečné délky tvořená ze symbolů (znaků) z této abecedy. Speciálním případem je tzv. prázdné slovo. Délka tohoto slova je nulová a značíme ho symbolem $\varepsilon$ .

**Jazyk**
Jazyk nad abecedou je libovolná množina slov, tvořených pomocí symbolů (znaků) abecedy.

## 3.2 Formální gramatiky

**Pojem formální gramatika** označuje v informatice strukturu popisující formální jazyk. Toto pojmenování bylo zvoleno na základě jisté podobnosti s gramatikami, které se používají v přirozených jazycích.
Formální gramatiku tvoří množina pravidel, pomocí kterých může být generováno každé slovo předepsaným způsobem z předem určeného počátečního symbolu. Proto se někdy mluví také o generativní gramatice.
Samotné generování začíná výběrem počátečního symbolu, na který se aplikuje kterékoliv z platných pravidel. Na nově získaný řetězec opět aplikujeme některé z platných pravidel a tento způsob opakujeme, dokud nezískáme požadované slovo [L6].

Pokud má každé slovo pouze jediný postup generování, jedná se o gramatiku jednoznačnou.
Symboly z abecedy, které se již dále nemohou přepisovat (označeny malými písmeny), se nazývají **terminály**. Symboly, pro které existují přepisovací pravidla (označeny velkými písmeny), se nazývají **neterminály**.

Princip generování slov si ukážeme na jednoduchém příkladu. Mějme abecedu obsahující symboly '*a*' a '*b*' a počáteční symbol je '*S*'. Pravidla gramatiky jsou definovány jako:

- $S \rightarrow aSb$
- $S \rightarrow ba$

Pokaždé začínáme startovacím symbolem „S“. Jelikož levá strana obou pravidel obsahuje právě startovací symbol, můžeme si mezi těmito pravidly vybrat to, které chceme aplikovat. Pokud vybereme 1. pravidlo, nahradíme startovací symbol „S“ řetězcem „aSb“. Jelikož se na pravé straně tohoto pravidla nachází neterminální symbol „S“, nemůže být tento krok konečný. Aplikaci prvního pravidla můžeme neomezeně opakovat a tím rozšiřovat vzniklé slovo (*aSb, aaSbb, aaaSbbb* atd.). Pokud chceme generování ukončit, je nutné nakonec aplikovat 2. pravidlo, jehož pravá strana se skládá pouze s terminálů. Pokud tedy aplikujeme na řetězec *aaSbb* 2. pravidlo, vyjde nám výsledné slovo *aababb* a tímto krokem končíme celé generování, jelikož se na pravé straně nenachází žádný neterminální symbol, který bychom mohli dále přepisovat.
Jazykem gramatiky chápeme všechna slova, která dokážeme pomocí dané gramatiky vygenerovat. Pro náš jednoduchý příklad bude jazykem gramatiky množina *{ba, abab, aababb, aaababbb, ...}*.

### 3.2.1 Dělení formálních gramatik

Gramatiky lze rozdělit podle jednotlivých vlastností. Jednou z nejznámějších hierarchií je **Chomského hierarchie** [L1], která nám gramatiky dělí na 4 druhy:

- Typ 0 : Frázové gramatiky
- Typ 1 : Kontextové gramatiky
- Typ 2 : Bezkontextové gramatiky
- Typ 3 : Regulární gramatiky

### 3.2.2 Frázové gramatiky

Frázové gramatiky jsou velice důležité pro popsání formálních jazyků. Taková gramatika generuje z počátečního symbolu jednotlivá slova pomocí svých pravidel a v každém kroku je možné přepsat určitou část nově vytvořeného slova na slovo jiné. Mezi jazyk, který gramatika generuje, pak můžeme řadit pouze slova složená z terminálních symbolů.

Frázová gramatika obsahuje pravidla ve tvaru:

$x_1ux_2 \rightarrow y_1vy_2$ právě tehdy, když $u \rightarrow v$, kde

$x_1, x_2, y_1, y_2, v$ náleží $(N \cup T)^*$, $u$ náleží $(N \cup T)^* N (N \cup T)^*$

### 3.2.3 Kontextové gramatiky

**Kontextová gramatika** je formální gramatika, která obsahuje pravidla ve tvaru:

$\alpha A\beta \rightarrow \alpha\gamma\beta$

kde $A$ je neterminál a $\alpha$, $\beta$ jsou řetězce neterminálů a terminálů a $\gamma$ je neprázdný řetězec terminálů a neterminálů.

Označení „kontextová“ gramatika znamená, že neterminál A je možno přepisovat na $\gamma$ pouze na základě kontextu, který tvoří řetězce $\alpha$ a $\beta$.

### 3.2.4 Bezkontextové gramatiky

**Bezkontextová gramatika** obsahuje pravidla ve tvaru:

$A \rightarrow \beta$

kde $A$ je neterminál a $\beta$ je řetězec terminálů a/nebo neterminálů.

Označení „bezkontextová“ gramatika znamená, že neterminál je možno přepisovat na řetězec $\beta$, aniž by byl kladen ohled na okolní kontext. Bezkontextová gramatika generuje vždy bezkontextový jazyk. Jako příklad jednoduché bezkontextové gramatiky můžeme brát příklad v kapitole 3.2.

### 3.2.5 Regulární gramatiky

**Regulární gramatika** obsahuje pravidla ve tvaru:

$X \rightarrow wY$ a $X \rightarrow w$

kde $X, Y$ jsou neterminály a $w$ je terminál.

Regulární gramatika může být dvojího typu a to podle toho, z jaké strany rozšiřuje vytvářené slovo. Rozšíření regulární gramatiky o řetězce podle výše zmíněného tvaru se nazývá **pravá lineární gramatika** a generované slovo se vytváří směrem zleva doprava.

Opakem pravé lineární gramatiky je pak **levá lineární gramatika**, která obsahuje pravidla ve tvaru:

$X \rightarrow Yw$ a $X \rightarrow w$

kde $X, Y$ jsou neterminály a $w$ je řetězec terminálů.

Je dokázáno, že pravé i levé lineární gramatiky jsou ekvivalentní. Proto např. slova vytvořená pravou lineární gramatikou jdou vždy vytvořit i pomocí levé lineární gramatiky. Jazyky generované regulárními gramatikami jsou právě jazyky rozpoznatelné konečným automatem [L3].

## 3.3 Automaty

Konečný automat (zkráceně KA) je virtuální systém s omezeným počtem stavů, který pracuje v diskrétním čase. Na vstup KA přicházejí jednotlivé symboly ze vstupní abecedy a KA reaguje na jejich příchod přechodem do dalšího stavu. KA lze tedy jednoduše považovat za abstraktní obraz konkrétního systému, který např. rozpoznává řetězec patřící do nějakého regulárního jazyka [Z2].

### 3.3.1 Formální definice automatu (matematický model)

Deterministickým konečným automatem nazýváme pětici $(\Sigma, S, s_0, \delta, F)$, kde:

- $\Sigma$ je vstupní abeceda (konečná neprázdná množina symbolů).
- $S$ je konečná neprázdná množina stavů.
- $s_0$ je počáteční (iniciální) stav.
- $\delta$ je přechodová funkce: $\delta : S \times \Sigma \rightarrow S$. U nedeterministického automatu může být ve tvaru $\delta : S \times \Sigma \rightarrow P(S)$.
- $F$ je množina koncových stavů.[1]

### 3.3.2 Dělení konečných automatů

Konečné automaty je možné rozdělit do několika skupin podle zvoleného kritéria. První dělení, závislé na jednoznačnosti přechodů z aktuálního stavu, je na:

- Deterministický automat
- Nedeterministický automat

Deterministický konečný automat (zkráceně DKA) je konečný automat, u kterého je po příchodu vstupního symbolu přechod ze současného stavu do stavu nového jednoznačně určen. Přechodová funkce deterministického automatu je znázorněna v kapitole 3.3.1.

[1] Bhattacharyya, Souvik – Sanyal, Gautam. Combining Finite State Machines and PMM method. *International Journal of Electrical and Computer Engineering* [online]. 2010. [cit. 2012-04-16]. Dostupné z <http://www.waset.org/journals/ijece/v5/v5-2-12.pdf>.

Pokud je přechodová funkce definována tak, že přechod z aktuálního stavu do stavu nového není jednoznačně určen (existuje více možností), jedná se o nedeterministický konečný automat (NKA). Takový automat pak přejde do jednoho z možných stavů, ale na základě vstupu nelze jednoznačně určit, do kterého. Vstup je poté přijat tehdy, když alespoň jeden z aktuálních stavů je zároveň stavem koncovým.

Konečné automaty se dále mohou dělit na:

- Konečné automaty bez výstupu
- Konečné automaty s výstupem
    - Moorův automat
    - Mealyho automat

Matematický model automatu, uvedený v kapitole 3.3.1, je řazen mezi konečné automaty bez výstupu. Pokud na posloupnost vstupních symbolů reaguje KA posloupností symbolů výstupní abecedy, jedná se o konečný automat s výstupem. Ten lze dále rozdělovat podle tvaru výstupní funkce na Moorův a Mealyho automat. Mealyho automat generuje výstup na základě příchozího vstupu a momentálního stavu, ve kterém se automat nachází. Moorův automat generuje výstup pouze na základě momentálního stavu. Výstupní funkci Mooreova automatu se někdy říká značkovací funkce a označuje se místo $\lambda$ symbolem $\mu$. Mealyho automat je oproti Moorova automatu obecnějším prostředkem.

## 3.4 Algoritmy

Pojmem algoritmus rozumíme konečný (omezené množství přesně definovaných kroků) schématický postup pro řešení určitého druhu problému. Přestože se tento pojem vyskytuje především v informatice a přírodních vědách, je jeho působnost daleko širší (kuchyňské recepty, návody a postupy atd.) [Z3].
„Samotné slovo algoritmus pochází ze jména perského matematika 9. století Abu Jafar Muhammada ibn Mūsā al-Chwārizmího, který ve svých dílech položil základy algebry (arabské číslice, řešení lineárních a kvadratických rovnic)“[2].

[2] Mička, Pavel. Algoritmus – Algoritmy.net [online]. [cit.2012-04-16]. Dostupné z <http://algoritmy.net>

### 3.4.1 Vlastnosti algoritmů

Každý plně funkční algoritmus by měl splňovat 4 základní vlastnosti [L2]:

1. **Konečnost** – Jak již bylo zmíněno výše, algoritmus musí mít konečné množství kroků, aby byl proveditelný.
2. **Určitost** – Všechny kroky algoritmu musí být přesně definovány tak, aby jejich průběh mohl být vykonáván systematickým strojem.
3. **Korektnost** – Algoritmus musí skončit pro libovolná vstupní data vždy stejným a zároveň správným výsledkem a to v konečném množství kroků.
4. **Obecnost** – Algoritmus musí umět řešit všechny úlohy daného typu. Simulace průběhu pouze jednoho typu příkladu se nenazývá algoritmem.

### 3.4.2 Obecné seznámení s algoritmy FJP

V této kapitole se podíváme na určité tvary bezkontextových gramatik (viz kapitola 3.2.4), na jejich transformace a na algoritmy [L3], které nám k tomu pomohou. Existuje celá řada užitečných transformací gramatik [Z4], které dovolují modifikovat gramatiku, aniž by byl porušen generovaný jazyk.

Např. v některých případech se může stát, že sestrojená gramatika obsahuje zbytečné symboly a nadbytečná přepisovací pravidla. Nehledě k tomu, že tato skutečnost může být důsledkem chyby v konstrukci gramatiky, je velmi pravděpodobné, že syntaktická analýza bude probíhat méně efektivně. Je proto důležité odstranit z gramatiky zbytečné symboly a nadbytečná pravidla. Těchto nedostatků bezkontextových gramatik existuje více. Správnou transformací formou matematických zápisů příslušných algoritmů těchto gramatik se budeme zabývat v následujících kapitolách.

### 3.4.3 Algoritmus pro zjištění nedostupných symbolů

První nedostatek bezkontextových gramatik můžeme zpozorovat tehdy, začneme-li vytvářet ze startovacího symbolu S gramatiky G větné formy. Může se totiž stát, že některé symboly (terminály i neterminály) budou nedostupné, tzn. že k nim vůbec nedojdeme. Princip prvního algoritmu je vyhledat tyto algoritmy a vytvořit novou gramatiku bez jejich použití. Základem tohoto algoritmu je přímočarý postup, kterým procházíme pravidla a sbíráme symboly, na které jsme narazili. Výsledná gramatika bude poté vytvořena pouze ze symbolů nalezených tímto algoritmem.

#### 3.4.3.1 Algoritmus

1. $V_0$ = {S}, i=1
2. $V_i$ = {X: A --> *a* X *b* kde *a* a *b* jsou řetězce symbolů, A náleží $V_{i-1}$} + $V_{i-1}$
3. Je-li $V_i$ != $V_{i-1}$, pak i++ a vrať se na krok 2, jinak
   a) $V_D$ = $V_i$ (jedná se o dostupné symboly)
   b) $V_N$ = N + T - $V_D$ (jedná se o nedostupné symboly)

#### 3.4.3.2 Příklad

Funkci algoritmu pro zjištění nedostupných symbolů si předvedeme na jednoduchém příkladu:

```
S => a | bA
A => b | aA
B => aA | aB
C => b
```

Výpis množiny $V_i$:

$V_0$ = {S}
$V_1$ = {S, A, a, b}
$V_2$ = {S, A, a, b} = $V_1$, tj. $V_D$ = {S, A, a, b}

Potom G' = {N', T', P', S}, kde N' = {S, A}, T' = {a, b} a P:

```
S --> aB | bB
A => b | aA
```

Jak je ve výsledné gramatice vidět, třetí a čtvrté pravidlo bylo zcela zbytečné, jelikož se do nich ze startovacího symbolu S nikdy nedostaneme.

Tomuto algoritmu se budeme později více věnovat, jelikož je jedním z algoritmů, které jsme implementovali v praktické části této diplomové práce.

### 3.4.4 Algoritmus pro zjištění prázdného jazyka

V předchozí kapitole jsme si ukázali, jak z gramatiky vyloučit nedostupné symboly, které v ní jsou naprosto zbytečné. Tyto symboly ovšem nejsou jediné neužitečné, které gramatika může obsahovat. Existují ještě tzv. „zbytečné" symboly, které jsou v bezkontextových gramatikách nežádané. Abychom mohli takové symboly v gramatice nalézt, je potřeba si vysvětlit princip dalšího algoritmu, který je pro tuto operaci nezbytný. Tento algoritmus nám určí, zda jazyk generovaný gramatikou G není prázdný.

Jazyk L(G) je v tomto případě množinou řetězců terminálních symbolů. Gramatika může mít ovšem nežádoucí vlastnost, kdy při přepisování nám nebudou neterminální symboly ubývat. Nyní tedy chceme zjistit, zda z gramatiky G vůbec lze takový řetězec generovat.

#### 3.4.4.1 Algoritmus

1. $N_0$ = {}, i = 1
2. $N_i$ = {A: A --> *a*, kde *a* je řetězec složený ze symbolů $N_{i-1}$ a T} + $N_{i-1}$
3. Je-li $N_i$ != $N_{i-1}$, i++ a jdi na (2), jinak $N_t = N_i$
4. Je-li S prvkem $N_t$, je L(G) neprázdný, v opačném případě je prázdný.

Hlavní myšlenka je taková, že pravidla bývají rekurzivní. Abychom mohli generovat řetězec terminálů, musí být rekurze konečná. Proto se nejdříve naleznou neterminály, pro které existuje pravidlo, které je přepíše buď na řetězec terminálů, nebo na prázdný řetězec. Takovým symbolům budeme říkat "dobré". Poté co projdeme všechna pravidla a nalezneme nějaké dobré neterminály, projdeme celou gramatiku znova a hledáme další dobré neterminály, které se přepisují na dobré neterminály, nalezené v předchozím kroku. A tak dále pokračujeme do doby, než se množina dobrých neterminálů přestane zvětšovat. Je-li dobrý i startovací symbol, je dobrá i celá gramatika a generuje neprázdný jazyk.

#### 3.4.4.2 Příklad

Funkci algoritmu pro zjištění toho, že je daný jazyk prázdný, budeme demonstrovat na jednoduchém příkladu.

```
S --> bA | bS
A --> aS | aA
```

Postupujeme podle algoritmu:

$N_0$ = { }
$N_1$ = $N_0$

Množina $N_i$ zůstala prázdná. To znamená, že neexistuje žádné dobré pravidlo, tedy žádný neterminál se zde nepřepisuje na řetězec terminálů nebo prázdný řetězec. Z tohoto důvodu můžeme konstatovat, že je jazyk L(G) prázdný.
Tomuto algoritmu se budeme později více věnovat, jelikož je jedním z algoritmů, které jsme vizualizovali v praktické části této diplomové práce.

### 3.4.5 Algoritmus pro zjištění zbytečných symbolů

Jako první algoritmus jsme si představili algoritmus pro zjištění nedostupných symbolů. Nedostupné symboly jsou v gramatice zbytečné. Dalšími nedostupnými symboly jsou ty, na které je možné přepsat se, ale už neexistuje žádné pravidlo, které by tento symbol dále přepsalo na jiný. Celý problém lze formalizovat a popsat takto:
S -->* *a* X *b* -->* *a c b*

kde *a*, *b* a *c* jsou řetězce terminálních symbolů, X je neterminální symbol a S startovací symbol [5].
Pokud v tomto znázornění neexistuje pro symbol X žádné přepsání, je symbol X zbytečný.
Jinak řečeno, pokud by se na pravé straně některého z pravidel objevil neterminální symbol X, nikdy už nedostaneme výsledek složený pouze z terminálů. Symbol X nám ve výsledném řetězci zůstane.
Oba dva algoritmy, představené v předchozích kapitolách, jsme vysvětlili záměrně jako první, jelikož princip odstranění zbytečných symbolů spočívá právě v nich.

Nejprve najdeme množinu $N_t$. Nová gramatika bude obsahovat jen tyto dobré neterminály. Z nich potom stačí pouze odstranit všechny nedostupné symboly.
Jelikož je tento algoritmus pouze kombinací předchozích dvou algoritmů, které byly předvedeny na příkladech, nebudeme již jeho funkčnost demonstrovat na žádném příkladu.

### 3.4.6 Algoritmus pro vyloučení pravidla

V předchozích kapitolách jsme si vysvětlili, jak pomocí dostupných algoritmů odstranit z gramatiky všechny nežádoucí symboly. Další důležitou operací je vyloučení některého z pravidel. Místo toho, aby se přepisování provedlo při derivaci větné formy, provedeme ho už v zápisu gramatiky.
Pravidly, kterých se nejčastěji potřebujeme zbavit, jsou tzv. prázdná pravidla (e-pravidla). Tyto pravidla způsobí přepsání neterminálního symbolu na prázdný řetězec e. Gramatikou bez těchto pravidel pak rozumíme takovou gramatiku, ve které nejsou žádná e-pravidla kromě pravidla `S => e`, které do gramatiky algoritmus zahrne tehdy, když gramatika generuje prázdný řetězec.

#### 3.4.6.1 Algoritmus

1. Způsobem podobným jako při hledání množiny $N_t$ najdeme tentokrát množinu $N_e$. Tato množina obsahuje symboly, které se dají přepsat na e (prázdné slovo).

2. Pravidlo $A \text{ --> } a_0\ B_1\ a_1\ B_2\ a_2\ \ldots\ B_n\ a_n$ nahradíme pravidly $A \text{ --> } a_0\ X_1\ a_1\ X_2\ a_2\ \ldots\ X_n\ a_n$, kde $X_i$ je buď e nebo $B_i$ (vystřídáme všech $2^i$ možností). Pravidlo typu `A --> e` zapisovat nebudeme, jelikož se právě takových pravidel snažíme zbavit.

3. Patří-li startovací symbol S do množiny $N_e$, zařadíme do gramatiky pravidlo `S' --> S | e`. Novým startovacím symbolem pak bude S'.

#### 3.4.6.2 Příklad

```
S --> aS | aSA | bA
A --> e | bA
```

Naplnění množiny $N_e$:

```
N0 = {}
N1 = {A}        A je dobrý symbol
N2 = {A, S}     S se může přepsat na A
Ne = N2
```

V pravidlech nyní vynecháme symboly, které se přepisují na prázdné pravidlo:

```
S --> aS | a | aSA | aA | bA | A
A --> a | bA
S' --> S | e   jelikož S patří do Ne
```

### 3.4.7 Algoritmus pro odstranění jednoduchých pravidel

Dříve, než si ukážeme princip dalšího algoritmu, si vysvětlíme, co si představit pod pojmem „jednoduché“ pravidlo. Tímto přívlastkem můžeme označit všechna pravidla ve tvaru `A => B`, kde symboly A i B jsou neterminály. Tyto pravidla v gramatice nijak nevadí, ovšem jejich výskyt není žádaný. Hlavní důvod je ten, že takto zapsaná pravidla mohou snadno vytvářet nežádoucí cykly. Odstraněním jednoduchých pravidel zároveň odstraníme i cykly.

Princip algoritmu je jednoduchý. Nejprve zjistíme pro každý neterminál, na jaké neterminály se může za použití jednoduchých pravidel přepsat. Tyto neterminály zapíšeme do množiny $N_X$, kde X jsou procházené neterminály na levé straně gramatiky.

Poté, co naplníme všechny množiny $N_X$, můžeme začít vypisovat nová pravidla taková, kde X se bude přepisovat na všechna nejednoduchá pravidla všech neterminálů v množině $N_X$.

#### 3.4.7.1 Příklad

Průběh algoritmu budeme demonstrovat na jednoduchém příkladu.

```
S => A | bA
A => aA | a
```

Naplníme množiny $N_X$:

$N_S$ = {S, A}
$N_A$ = {A}

Nová gramatika je:

```
S => bA | aA | a
A => aA | a
```

### 3.4.8 Algoritmus pro odstranění levé rekurze

Levorekurzivní pravidlo je ve tvaru `A => Aa`, kde *A* je neterminál a *a* je řetězec terminálů.
Tyto pravidla jsou občas nežádoucí pro některé typy syntaktické analýzy.
Pokud jsou pravidla zapsána v gramatice ve tvaru:

$A \text{ --> } Aa_1 \mid Aa_2 \mid \ldots \mid Aa_m \mid b_1 \mid b_2 \mid \ldots \mid b_n$

kde $a_i$ a $b_i$ jsou řetězce terminálních a neterminálních symbolů a žádné $b_i$ nezačíná symbolem A, můžeme odstranit přímou levou rekurzi.

Pro výše uvedený tvar by bylo odstranění levé rekurze ve tvaru:

$A \text{ --> } b_1 \mid b_2 \mid \ldots \mid b_n \mid b_1A' \mid b_2A' \mid \ldots \mid b_nA'$

$A' \text{ --> } a_1 \mid a_2 \mid \ldots \mid a_m \mid a_1A' \mid a_2A' \mid \ldots \mid a_mA'$

nebo pomocí e-pravidel:

$A \text{ --> } b_1A' \mid b_2A' \mid \ldots \mid b_nA'$

$A' \text{ --> } e \mid a_1A' \mid a_2A' \mid \ldots \mid a_mA'$

Přestože jsme se v předchozích kapitolách snažili e-pravidlům vyhnout, je výskyt tohoto pravidla většinou méně nežádoucí, než výskyt pravidla levorekurzivního.

#### 3.4.8.1 Příklad

Průběh algoritmu budeme reprezentovat na níže uvedeném převzatém příkladu [Z4].

```
E --> E + T | T
T --> T * F | F
F --> (E) | i
```

Podle výše uvedeného vzoru vytvoříme novou gramatiku nejprve bez e-pravidel:

```
E  --> T | TE'
E' --> + T | + TE'
T  --> F | FT'
T' --> * F | * FT'
F  --> (E) | i
```

Nyní si pro vytvoření nové gramatiky pomůžeme pomocí e-pravidel:

```
E  --> TE'
E' --> e | + TE'
T  --> FT'
T' --> e | * FT'
F  --> (E) | i
```

### 3.4.9 Algoritmus first

Množina first se určuje ke každé pravé straně každého pravidla. Jedná se o množinu terminálů, kterými může tato pravá strana začínat. Přesnou definicí se zde nebudeme zaobírat. Speciální význam má prázdné slovo e. Obsahuje-li first pravé strany e, znamená to, že celou tuto pravou stranu lze přepsat na prázdné slovo.

Výpočet funkce first se dá určit třemi způsoby:

1. **metodou "kouknu a vidím"** – pouze pro triviální příklady
2. **rekurzivním algoritmem**
3. **algoritmem s tečkou**

Popíšeme si zde pouze třetí způsob a to algoritmus s tečkou. Ten se řídí pomocí několika základních kroků.

#### 3.4.9.1 Algoritmus

Výpočtem first(α) zjišťujeme, jakým terminálním symbolem začíná řetězec $\alpha = A_1 A_2 A_3 ... A_n$

**Krok 1a:**

Vytvoříme vstupní množinu $F=\{.A_1...A_n\}$

**Krok 1b:**

Je-li v F pravidlo `B --> beta . A gama`, přidáme do F pravidla `A --> . delta`
Tento krok provádíme tak dlouho, dokud to jde.

**Krok 1c:**

Je-li v F pravidlo `B --> delta .`, dáme do F pravidla z F, ve kterých se vyskytovaly symboly `.B`, ale tečku umístíme až za `B`

**Krok 1d:**

Kroky **b** a **c** se opakují tak dlouho, dokud lze do F přidávat.

**Krok 2:**

first(α) = { všechny terminální symboly z F, které jsou bezprostředně za tečkou } + {e: je-li tečka na konci pravidla }

#### 3.4.9.2 Příklad

Průběh algoritmu budeme demonstrovat na jednoduchém příkladu.

```
S => aB | bB
A => a | BA
B => aS | bB | e
```

Na vstupní gramatiku aplikujeme postupně výše popsané kroky

| | | |
|---|---|---|
| Krok 1a: | `F = {.A` | vytvoření množiny |
| Krok 1b: | `A => .a | .BA` | vytvořeno z `.A` |
| Krok 1b: | `B => .aS | .bB | .` | vytvořeno z `A => .a | .BA` |
| Krok 1c: | `A => .a | B.A` | vytvořeno z `B => .aS | .bB | .` |
| Krok 2: | `}` | vytvoření výsledku |

Výsledkem funkce first jsou všechny terminály bezprostředně za tečkou. Je-li tečka na konci pravidla, patří mezi výsledky i prázdný řetězec e. Výsledkem tohoto příkladu jsou tedy terminály a,b,e.

### 3.4.10 Algoritmus follow

Množina follow se zjišťuje ke každému neterminálu. Obsahuje terminály, které se mohou nacházet bezprostředně za daným neterminálem. Speciální význam má opět prázdné slovo e, které symbolizuje konec vstupu. Množina follow daného neterminálu tento symbol obsahuje jen tehdy, pokud za tímto neterminálem nemusí být žádný další symbol. Dá-li se startovací symbol přepsat na větu, ve které je daný neterminální symbol až na konci, patří do množiny follow i prázdný řetězec.

Stejně jako u výpočtu first se i výpočet funkce follow dá určit třemi způsoby:

1. **metodou "kouknu a vidím"** – pouze pro triviální příklady
2. **rekurzivním algoritmem** – formalizace metody "kouknu a vidím"
3. **algoritmem s tečkou** – nejlepší řešení pro implementaci

I zde si popíšeme pouze třetí způsob a to algoritmus s tečkou z důvodu nejvhodnějšího řešení pro implementaci. Ten se řídí pomocí několika základních kroků.

#### 3.4.10.1 Algoritmus

Výpočtem follow(α) zjišťujeme, jaký terminální symbol pokračuje za řetězcem $\alpha = A_1 A_2 A_3 ... A_n$

**Krok 1:**

Položíme $N_e$ rovno množině všech prvků, které se dají přepsat (i rekurzivně) na prázdný řetězec.

**Krok 2a:**

F = { A => A. }, do F umístíme fiktivní pravidlo, ze kterého vyjdeme.

**Krok 2b:**

Je-li v F pravidlo B => gama . , kde gama je neprázdný řetězec, dáme do F všechna pravidla, ve kterých je na pravé straně B a tečku umístíme za B. V podstatě budeme dále určovat first(řetězec, který následuje za B).

**Krok 2c:**

Je-li v F pravidlo C => alfa . B beta, přidáme do F všechna pravidla s B na levé straně, tečku umístíme na začátek.

**Krok 2d:**

Je-li v F pravidlo, kde je za tečkou neterminální symbol, který patří do $N_e$, do F vložíme toto pravidlo ještě jednou, ale tečku posuneme o jeden symbol doprava.

**Krok 2e:**

Kroky 2b, 2c, 2d opakujeme, dokud do F můžeme přidávat další položky.

**Krok 3:**

Do follow(α) dáme všechny terminální symboly, před kterými je tečka. Je-li v F pravidlo S => α ., kde S je startovací symbol, přidáme do follow(α) i symbol *e*.

#### 3.4.10.2 Příklad

Průběh algoritmu budeme demonstrovat na jednoduchém příkladu:

```
S => aSAb | AB
A => Bb | aA
B => bB | e
```

Na vstupní gramatiku aplikujeme postupně výše popsané kroky

| Krok | Množina | Popis |
|---|---|---|
| Krok 1: | `Množina Ne = (B)` | zjištění dobrých symbolů |
| Krok 2a: | `F = {A => A.` | vytvoření množiny |
| Krok 2b: | `S => aSA.b \| A.B` | vytvořeno z `A => A.` |
| Krok 2b: | `A => Bb \| aA.` | vytvořeno z `A => A.` |
| Krok 2c: | `B => .bB \| .` | vytvořeno z `S => A.B` |
| Krok 2d: | `S => aSA.b \| AB.` | vytvořeno z `S => A.B` |
| Krok 2b: | `S => aS.Ab \| AB` | vytvořeno z `S => AB.` |
| Krok 2c: | `A => .Bb \| .aA` | vytvořeno z `S => aS.Ab` |
| Krok 2d: | `A => B.b \| .aA` | vytvořeno z `A => .Bb` |
| Krok 3: | `}` | vytvoření výsledku |

Výsledkem funkce follow jsou všechny terminály bezprostředně za tečkou. Je-li v množině také pravidlo `S => alfa .` , patří mezi výsledky i prázdný řetězec e. Výsledkem tohoto příkladu jsou tedy terminály *a, b, e*.

# 4 Vizualizace algoritmů

Pojmem vizualizace se rozumí zobrazování nějaké skutečnosti, jejichž výsledky jsou vnímány prostřednictvím zrakových receptorů. Za český ekvivalent můžeme považovat slova jako náhled (např. na navržený produkt), názorná ukázka nebo zobrazení[3]. Vizualizace úzce souvisí s uplatňováním zásady názornosti.

S vizualizací se setkáváme tehdy, je-li textový či slovní popis daného problému komplikovaný. Jedná se většinou o technické záležitosti, jejichž názorná ukázka poskytne lepší vysvětlení dané problematiky.

Vhodným příkladem pro požití vizualizace jsou také některé algoritmy. V kapitole 3.4 jsme si řekli, že algoritmus je konečný schématický postup pro řešení určitého druhu problému. V informatice existuje řada problémů, které lze řešit algoritmicky. Vzpomenout můžeme známé řadící algoritmy jako např. quicksort nebo kódovací algoritmy jako např. Boyer Moore algoritmus.

## 4.1 Technologie vizualizace ve webovém prohlížeči

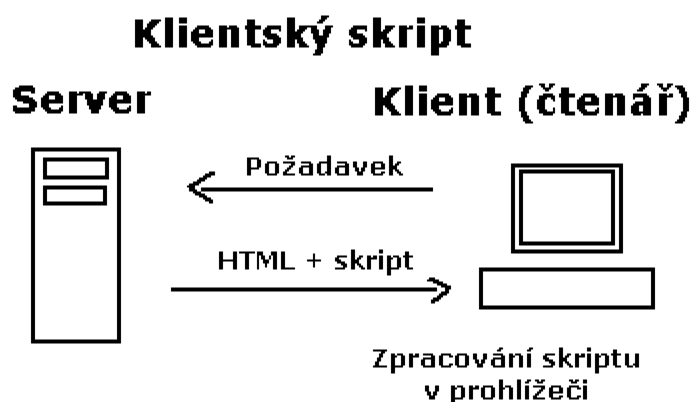
Dnes asi nejrozšířenějším způsobem vizualizace algoritmů je jejich webová podoba. Vizualizace je v tomto případě součástí webových stránek a pro její zobrazení nepotřebuje uživatel většinou nic jiného, než internetový prohlížeč. Technologie, které se pro tuto práci nabízí, jsou např. Javascript, Flash nebo dnes již méně rozšířený Java Applet. Pokud by se jednalo pouze o zjištění výsledků ze zadání bez jakékoliv vizuální stránky, postačilo by nejspíš pro méně náročné algoritmy samotné PHP. V poslední době pomalu vstupuje do hry také poměrně silný hráč v podobě technologie HTML 5, kterou zde ale nebudeme podrobně rozvádět, jelikož se jedná o mladou záležitost.

[3] ABZ.cz – Slovník cizích slov [online]. *Význam slova vizualizace*. Dostupné z <http://slovnik-cizich-slov.abz.cz/web.php/slovo/vizualizace>

### 4.1.1 Javascript

Javascript byl původně obchodní název pro implementaci společnosti Netscape, kde byl nejprve vyvíjen pod názvem Mocha, později LiveScript a následně byl ohlášen jako doplněk k jazykům HTML a Java. Javascript je jedním z programovacích jazyků, který se používá přímo v internetových stránkách. Jeho zápis probíhá přímo do HTML kódu, což je vzhledem k jednoduchosti veliká výhoda [L5].

Javascript patří mezi tzv. klientské skripty. Znamená to, že program se odesílá se stránkou klienta do prohlížeče a až teprve tam je vykonáván. Opakem klientských skriptů jsou pak skripty serverové, jejichž výpočetní část probíhá na specializovaném serveru a klient pouze sbírá příchozí výsledky. Princip klientského skriptu budeme demonstrovat na jednoduchém schématu:




Obr 1 : Demonstrace použití klientského skriptu

Javascript je dnes nejrozšířenějším klientským skriptem. To ale neznamená, že jiné klientské skripty neexistují. Známý je např. i VBScript. Jeho použití je ale pouze ojedinělé. JavaScript je vzhledem k názvu často zaměňován s Javou. Oproti Javascriptu je Java samostatný programovací jazyk. Jediné, co má s Javascriptem společného je podobná syntaxe.

#### 4.1.1.1 Vlastnosti

Javascript je:

- **objektový** - využívá především objektů prohlížeče
- **interpretovaný** - bez kompilace
- **case sensitive** - záleží na velikosti písem v zápisu
- **závislý na prohlížeči** - funguje pouze v prohlížeči

Funkčnost javascriptu lze jednoduše rozšířit o případné frameworky. Jedním z nejznámějších frameworků je např. jQuery, který značně rozšiřuje interakci mezi javascriptem a HTML. Známou modifikací javascriptu je také technologie AJAX (Asynchronous JavaScript and XML), která umožňuje měnit obsah webových stránek bez nutnosti jejich znovunačítání. Oproti klasickým webovým aplikacím poskytují AJAXové stránky uživatelsky příjemnější rozhraní, které je ale kompenzováno nutností použití moderních webových prohlížečů.

#### 4.1.1.2 Výhody

- **Klientský skript** - šetří šířku pásma.
- **Jednoduchost** - dá se poměrně snadno naučit a syntaxe je podobná Javě nebo C.
- **Součást webové stránky** - není potřeba žádného vývojového prostředí a kompilátoru. Snadná úprava kódu.
- **Rychlost pro koncového uživatel** - přestože je Javascript poměrně pomalý, jeho rychlost je oproti serverovým skriptům značně vyšší.

#### 4.1.1.3 Nevýhody

- **Bezpečnost** - jakožto klientský skript trpí slabou bezpečností
- **Závislost na webovém prohlížeči** - v různých prohlížečích nemusí být kompatibilní

### 4.1.2 Java Applet

**Applet** je programová komponenta běžící v kontextu jiného programu, nejčastěji webového prohlížeče. Applet nebývá používán k plnění obecných funkcí, nýbrž jako prvek pro řešení konkrétních problémů. Nepředpokládá se tedy, že applet bude používán jako samostatná aplikace bez doprovodu např. webové stránky.
Stejně jako např. Javascript je i Applet klientskou aplikací. Opakem appletu je pak servlet, který je jeho serverovou alternativou. Applet může přistupovat k některým funkcím hostitelského programu (webové stránky), ne však k tolika, jako např. Javascript.

Applet může být napsán v jiném jazyce než hostitelský program. Nejčastěji si pod pojmem Applet představíme Java Applet, tedy applet psaný v jazyce Java. Přestože je Java Applet mezi applety nejrozšířenější, v poslední době tato technologie pomalu vymírá a nahrazuje jí již zmíněný Javascript.

#### 4.1.2.1 Výhody a nevýhody

Pokud máte v plánu vytvářet Java Applet, budete potřebovat i patřičné vývojové prostředí Javy nebo alespoň její kompilátor. Pro funkčnost již hotového appletu budete následně potřebovat webový prohlížeč a Java Runtime Environment (JRE). Java applet je často poměrně velký a jeho ladění není nijak jednoduché. Další nevýhodou je také to, že applet tvoří samostatný soubor a do html souboru se vkládá pouze odkaz na tento soubor, popřípadě některé další parametry, které budou potřebné při konečné kompilaci.
Oproti Javascriptu také nemá tak velké možnosti přístupu k funkcím kontejneru.

Mezi výhody Java Appletu řadíme to, že není závislý na operačním systému ani webovém prohlížeči. Java applet může běžet na většině verzích JRE. Některé funkce mohou vyžadovat novější verze, ale instalace nové verze prostředí není nijak obtížná. Opětovné spouštění appletů je mnohem rychlejší než spuštění prvotní, kdy se zároveň musí spouštět i JVM. Stejně jako Javascript i Java Applet běží na straně klienta, tudíž odlehčuje práci serveru. Oproti Javascriptu je bezpečný. Applet neumožňuje spouštění aplikací na počítači uživatele, má přístup jen do části paměti a smí se připojovat pouze na server, ze kterého byl spuštěn.

### 4.1.3 Adobe Flash

Flash je, jednoduše řečeno, program pro tvorbu prezentací, animací či webových stránek. Pracuje s vektorovou grafikou, což má za důsledek, že flashové aplikace bývají paměťově nenáročné, avšak o to více zaměstnávají procesor, na kterém běží.

Flash byl od počátku ve vlastnictví společnosti Macromedia, dnes je již delší dobu vlastníkem společnost Adobe. Flash je vhodný pro tvorbu převážně internetových interaktivních animací, prezentací a her.

Flash ovšem není pouze program pro kreslení vektorových obrazů. Jeho součástí je i vlastní implementovaný programovací jazyk s názvem ActionScript, který slouží k rozvinutí všech možností nejen interaktivních animací, ale dnes i k vývoji robustních aplikací. ActionScript si po čase prošel inovacemi a v aktuálních verzích je poměrně vyspělým objektově orientovaným programovacím jazykem schopným konkurovat o dost starším jazykům.

#### 4.1.3.1 ActionScript

Jak již bylo řečeno, ActionScript (zkráceně AS) je objektově orientovaný programovací jazyk pro aplikace vyvíjené především pomocí technologie Flash. Pomocí ActionScriptu se dají vytvářet vysoce interaktivní programy, animace a jiné prezentace. ActionScript je, stejně jako JavaScript, dialektem ECMAScriptu. Syntaxe obou jazyků je tak velmi podobná, pochopitelně se ale liší výchozí objektovou výbavou [L4].

ActionScript byl vyvinut v roce 2000 a do dnešní doby prošel třemi velkými inovacemi.

- **ActionScript 1.0** - vše začalo verzí 1.0, což byla nejjednodušší forma ActionScriptu, která se dnes stále používá v některých verzích přehrávače Flash Lite Player.
- **ActionScript 2.0** - verze 2.0 prošla mnoha inovacemi a nabízí tak širší základnu funkcí. ActionScript 2.0 je poměrně dobrý jazyk pro mnoho druhů projektů, které nejsou výpočetně náročné, např. pro vzhledově orientovaný obsah. Na učení je verze 2.0 podstatně jednodušší než ActionScript 3.0.

- **ActionScript 3.0** - poslední verze ActionScriptu vyžaduje větší znalosti objektově orientovaného programování, než verze starší. Oproti nim se také vykonává několikrát rychleji a zároveň plně vyhovuje specifikaci ECMAScript. Mezi inovace patří např. lepší zpracování XML, vylepšený model událostí nebo vylepšená architektura pro práci s obrazovkovými elementy.

#### 4.1.3.2 Výhody

- **Jednoduchost** – to, co lze jednoduše udělat ve Flashi pomocí mnoha nástrojů, se např. přes Javascript a kaskádové styly provádí mnohem složitěji. HTML (myšleno včetně Javascriptu a css) nabízí oproti Flashi méně prostředků, jak vytvořit animovanou stránku nebo jak ji rozpohybovat.
- **Bez nutnosti kompletního načtení obsahu** - Flashové stránky se mohou spustit i když ještě nejsou zcela načtené v paměti počítače.
- **Kompatibilita ve všech prohlížečích** - u flashových stránek se vám nestane, že v jednom prohlížeči se stránka zobrazí tak a v jiném jinak. Flash totiž není závislý (např. jako Javascript) na prohlížeči.
- **Reklama** – flash je momentálně nejvhodnějším pomocníkem pro vytváření reklamních bannerů. Postupně nahradil zastaralé GIF animace.
- **Univerzálnost -** ve Flashi lze vytvořit prakticky cokoliv, na co si vzpomenete.

#### 4.1.3.3 Nevýhody

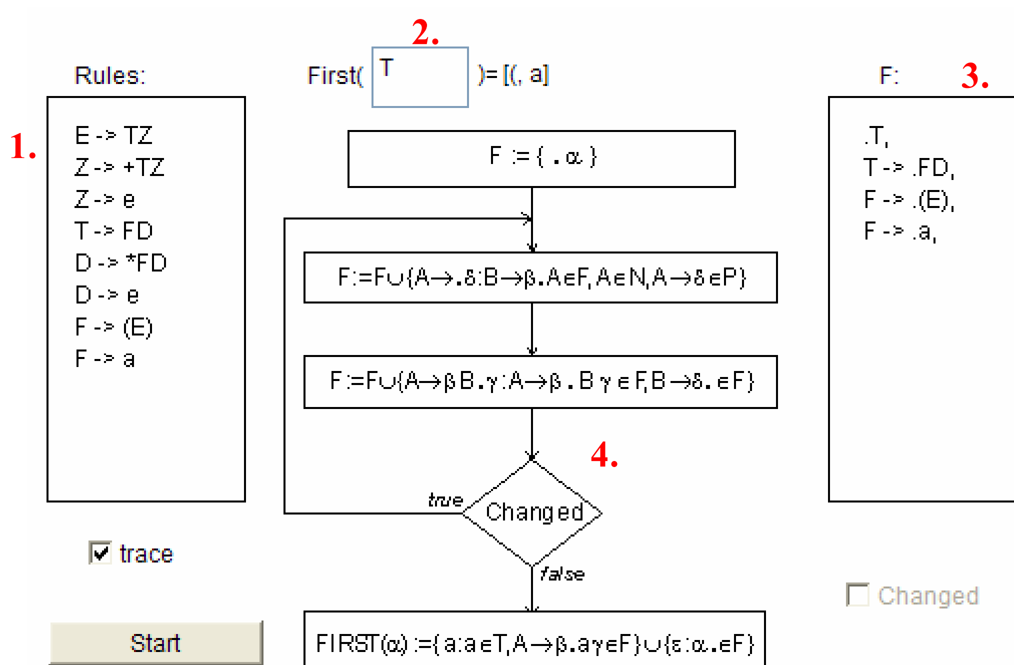
- **Náročnost** – flash je poměrně náročný na výpočetní výkon CPU. Starší počítačové sestavy mohou mít s některými flashovými prezentacemi značné problémy.
- **Nutnost instalace pluginu -** pokud uživatel nemá nainstalován požadovaný plugin, Flash prvek se mu jednoduše nezobrazí.
- **Závislost na verzi Flash Playeru -** mezi jednu z výhod patřila i nezávislost na webovém prohlížeči. Tato skutečnost je ovšem degradována závislostí na verzi Adobe Flash Playeru.
- **Mobilní nedostupnost** – trvalý bojkot této technologie ze strany společnosti Apple. Ostatní mobilní OS (Android, BlackBerry nebo Windows Mobile) obsahují minimálně částečnou podporu této technologie.

# 5 Stávající možnosti vizualizace algoritmů

V této kapitole si představíme již hotová řešení vizualizace algoritmů, se kterými se můžeme sejít na internetu. Vyjmenujeme si zde některá řešení a ukážeme jejich funkčnost.

## 5.1 Algoritmus first

První povedenou vizualizací algoritmu first je applet na stránkách ČVUT. Applet umožňuje zadat vlastní gramatiku nebo použít předem vyplněnou. Pokud zadáme korektní prvek do políčka „first“ a potvrdíme tlačítkem „Start“, můžeme vidět průchod daného algoritmu na velice povedeném diagramu. Současně se nám naplňuje i množina F napravo od diagramu. Jakmile je průchod algoritmu ukončen, vypíše se nám množina terminálních symbolů, které reprezentují výsledek algoritmu [Z5].




Obr 2 : Vizualizace algoritmu first

1. Vstupní gramatika
2. Symbol, pro který hledáme first
3. Výsledná množina F
4. Diagram průchodu algoritmu

### 5.1.1 Výhody

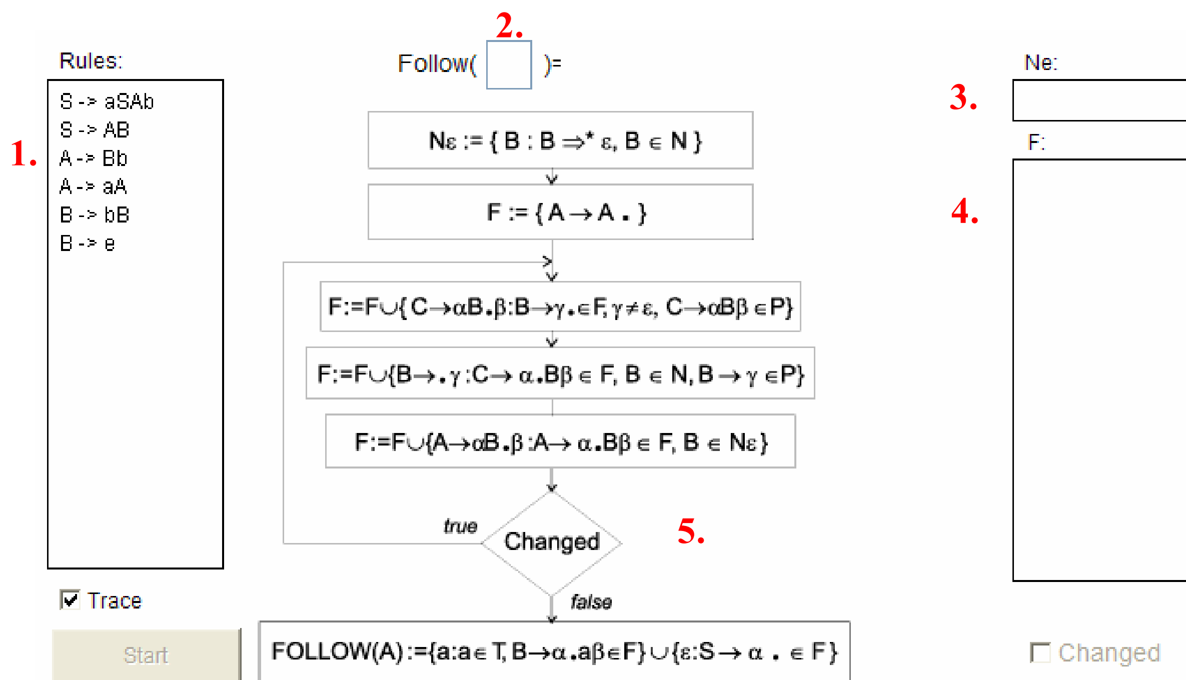
- Jednotlivá pravidla jsou na stránkách s algoritmem popsána
- Průběžné naplňování množiny F
- Průchod algoritmu reprezentován na diagramu
- Vstupní prvek funkce first je omezen pouze na korektní znaky

### 5.1.2 Nevýhody

- Chybí podrobnější informace o použití pravidel
- Chybí log pro textové vysvětlení průběhu algoritmu

## 5.2 Algoritmus follow

Na stránkách ČVUT jsme stejně jako v případě algoritmu first nalezli i povedenou vizualizací algoritmu follow. Applet opět umožňuje zadat vlastní gramatiku nebo použít předem vyplněnou. Pokud zadáme korektní prvek do políčka „follow“ a potvrdíme tlačítkem „Start“, můžeme vidět průchod daného algoritmu na velice povedeném diagramu. Současně se nám naplňuje i množina F napravo od diagramu. Jakmile je průchod algoritmu ukončen, vypíše se nám množina terminálních symbolů, které reprezentují výsledek algoritmu [Z6].




Obr 3 : Vizualizace algoritmu follow

1. Vstupní gramatika
2. Symbol, u kterého hledáme funkci first
3. Množina Ne, jejíž neterminály se mohou přepsat na prázdné slovo
4. Výsledná množina F
5. Diagram průchodu algoritmu

### 5.2.1 Výhody

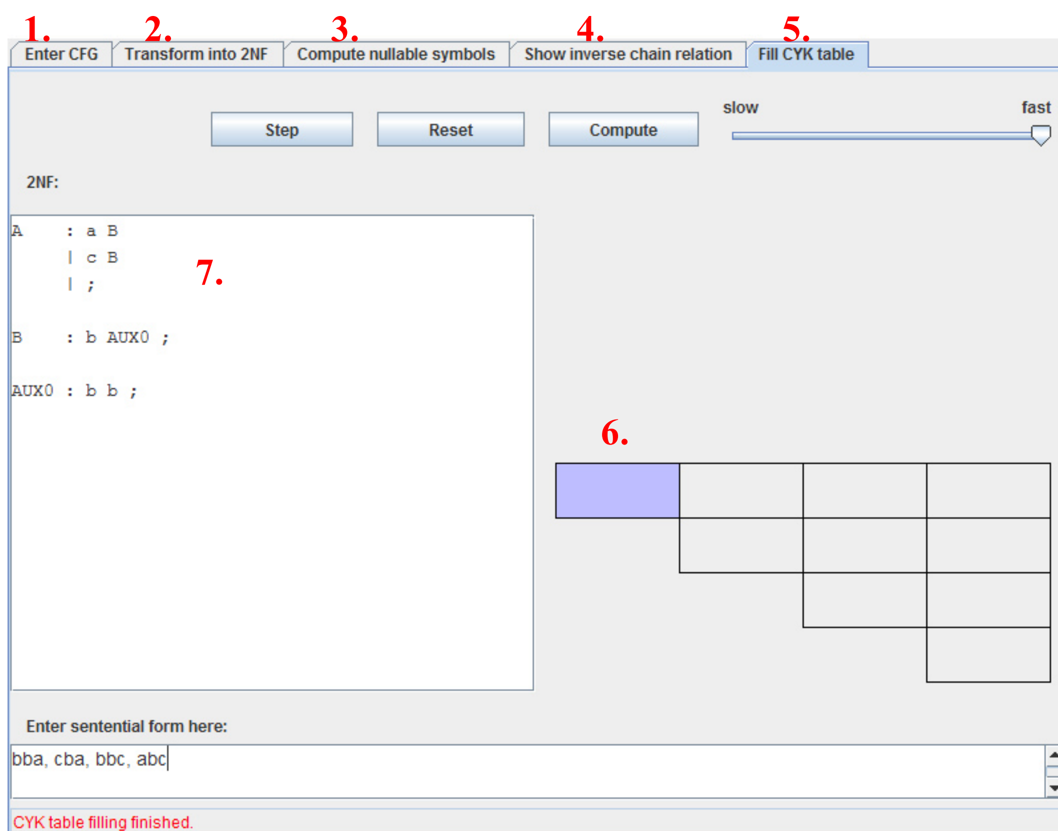
- Jednotlivá pravidla jsou na stránkách s algoritmem popsána
- Průběžné naplňování množiny F
- Výpis množiny Ne
- Průchod algoritmu reprezentován na diagramu
- Vstupní prvek funkce first je omezen pouze na korektní znaky

### 5.2.2 Nevýhody

- Chybí podrobnější informace o použití pravidel
- Chybí log pro textové vysvětlení průběhu algoritmu

## 5.3 CYK algoritmus

Jedná se o algoritmus, který určuje, zda slovo náleží do jazyka zadané gramatiky. Přestože se tento typ algoritmu úplně netýká naší problematiky, zmiňujeme ho kvůli jeho rozšíření. Poměrně povedená reprezentace toho algoritmu s názvem SeeYK je dostupná na stránkách Ludwig-Maximilians Universität München. Jedná se o Java Applet, jehož vstupem je uživatelem definovaná bezkontextová gramatika. Následuje její převod do druhé normální formy (2NF), vyhledání neterminálních symbolů, které mohou končit prázdným slovem a ve výsledku applet vykreslí CYK tabulku [Z7].




Obr 4 : Nástroj SeeYK

1. Záložka pro vložení vstupní gramatiky
2. Záložka s převedením na druhou normální formu
3. Záložka pro zjištění neterminálních symbolů, které mohou být prázdným slovem
4. Záložka pro zjištění předchůdců jednotlivých neterminálních prvků
5. Záložka pro generování výsledné CYK tabulky
6. Vizualizace CYK tabulky
7. Druhá normální forma vstupní gramatiky

### 5.3.1 Výhody

- Přehledný nástroj
- Automatický převod do druhé normální formy
- Zjištění neterminálních symbolů, které mohou být prázdným slovem
- Vizualizace CYK tabulky
- Ošetření vstupní gramatiky vůči nežádoucím symbolům

### 5.3.2 Nevýhody

- Jedná se o beta verzi (prototyp)
- Vizualizace tabulky by mohla zobrazovat v buňkách i samotné prvky
- Chybí textový výpis pro zjištění, co právě algoritmus dělá
- Při hledání předchůdců se obtížně kliká na jednotlivé symboly (nepřesné)

## 5.4 Ostatní nástroje

V předchozích kapitolách jsme se seznámili s již hotovými řešeními některých algoritmů pro zpracování gramatik. Všechny výše popsané nástroje slouží především jako doplněk ke studiu na vysoké škole a jsou příbuzné problematice této práce. Na internetu však existuje spousta dalších nástrojů, které pomohou přiblížit problematiku formálních gramatik a automatů. Tato řešení jsou však vzhledem k této práci velice obecná, proto si vystačíme pouze s vyjmenováním těch nejpovedenějších.

**JFLAP** (Formal Languages and Automata Package) – balíček grafických nástrojů, které slouží pro snadné pochopení základních konceptů v oblasti formálních jazyků a teorie automatů [Z8].
Jedná se o nástroj pro práci s konečnými a zásobníkovými automaty a s Turingovými stroji. JFLAP vznikl na univerzitě v Durhamu v Severní Karolíně v USA. Autorem je skupina profesorky Susan H. Rodger. Tato skupina představila program JFLAP na několika konferencích.

První verze programu byla napsána v jazyce C++ a byla určena pro UNIX systémy s grafickým rozšířením X-Windows. Později (srpen 1996) byl program přepsán do platformově nezávislého jazyka Java. Na počátku roku 1999 byla zveřejněna třetí verze, která byla obohacena.

**VISA** – komerční nástroj pro vizualizaci formálních jazyků a automatů. Je napsán v programovacím jazyce Java a je dostupný jak v podobě internetové aplikace, tak i jako samostatný program [Z9].

**Qfsm** – jedná se o grafický editor pro vytváření konečných automatů. Je napsán v jazyce C++ pomocí grafické nástavby Trolltech [Z10].
Qfsm umožňuje zkontrolovat, jestli je vytvořený automat deterministický, bez slepých větví a exportovat ho (do obrázku nebo textových popisů, které lze přeložit do programovacích jazyků). Kromě zdrojových textů si můžete stáhnout hotovou distribuci pro Windows. Program je i v repozitářích openSUSE.

**ANTLR (ANother Tool for Language Recognition)** – jméno nástroje vzniklo jako zkratka z anglického "ANother Tool for Language Recognition". Dříve byl znám pod názvem PCCTS. Celý je napsán v jazyce Java a díky použité Public Domain licenci jej lze bez potíží integrovat do dalších produktů.
Je to nástroj pro tvorbu syntaktických analyzátorů, kompilátorů a překladačů gramatiky pro jazyk Java, C#, C++ a Python. Napsán je v jazyce Java. Podporuje generování kódu průchodem syntaktickým stromem a některé sémantické akce. Další výhodou je zcela jistě generování kódu pro syntaktickou analýzu rekurzivním sestupem [Z11].

**Automata tools** – seznam dalších nástrojů z problematiky formálních jazyků a automatů [Z12].

# 6 Realizace vizualizací vybraných algoritmů

Po představení teoretické části této práce přistupujeme k praxi. Praktickým úkolem této práce bylo vytvořit řešení vybraných algoritmů, probíraných v hodinách předmětu KIV/FJP. Hlavním významem celého řešení je tedy vytvoření učební pomůcky pro snazší pochopení dané problematiky. Podmínkou zadání bylo vytvořit v internetové podobě alespoň 3 různé algoritmy pomocí libovolné technologie. Po důkladném uvážení jsme se tedy rozhodli pro implementaci algoritmu pro zjištění nedostupných symbolů, algoritmu pro zjištění prázdnosti jazyka a algoritmů funkcí first a follow. Význam a použití těchto algoritmů byly důkladně popsány v teoretické části této práce. Po výběru algoritmů následoval výběr technologie použité pro jejich vytvoření. Prakticky se nabízely tři možnosti a to použití Java Scriptu, Java Appletu nebo Adobe Flashe. Výhody a nevýhody těchto řešení jsou opět popsány v teoretické části této práce. My jsme si pro implementaci algoritmů vybrali technologii Adobe Flash. Hlavním důvodem použití tohoto řešení byla především snadná tvorba grafického prostředí v nástroji Adobe Flash Professional a v neposlední řadě i zvědavost vyzkoušet něco nového. Společně s tímto výběrem nás čekal i výběr verze programovacího jazyka ActionScript, který Adobe Flash používá. Po důkladném uvážení jsme zvolili starší ActionScript ve verzi 2. Důvod byl opět prostý. Novější verze ActionScript 3 je pro náš problém zbytečně složitá a nové funkce oproti verzi 2 nemají pro naše algoritmy uplatnění. ActionScript ve verzi 2 je snazší na pochopení a stále masivně používaný pro realizaci snazších webových nástrojů a prezentací. Nyní máme vše potřebné vybráno a nezbývá nic jiného, než začít implementovat vybrané algoritmy.

## 6.1 Požadavky

Jak už bylo řečeno, hlavním požadavkem bylo vytvořit alespoň 3 algoritmy, probírané v hodinách předmětu KIV/FJP pomocí libovolné technologie. Následuje však několik dalších požadavků na samotnou implementaci algoritmů. Požadováno bylo:

**Vytvoření jednotného API** – tedy vytvoření určitého základu, který by byl společný pro všechny vybrané algoritmy. Jedná se tedy o jakousi knihovnu tříd, která poskytuje určité

služby přímo pro implementaci každého algoritmu. Hlavním důvodem tohoto požadavku je především snazší dodatečná implementace nových algoritmů v budoucnu.

**Vytvoření intuitivní grafické vizualizace** – nejde tedy pouze o samotný běh algoritmu, jako jsme poznali již v některých existujících řešeních, ale především o jeho vysvětlení a názornou ukázku. Je tedy nutné vytvořit grafickou vizualizaci, která by přehledně reprezentovala, co v aktuální chvíli daný algoritmus vykonává. Hlavním požadavkem je především přehlednost použitého řešení.

**Vytvoření textového výpisu** – kromě grafické vizualizace bylo jedním z požadavků také vytvoření textového výpisu, ve kterém bude popsán každý krok algoritmu. Samotný výpis pak musí být schopen poskytnout kompletní informace o funkčnosti algoritmu, tedy student by si měl umět vystačit pouze s tímto řešením.

**Ošetření vstupní gramatiky** – dané řešení by mělo alespoň částečně ošetřit vstupní gramatiku zadanou uživatelem. Tedy minimálně se omezit na určitou množinu symbolů, vyloučit ty nežádoucí a vyvarovat se možným překlepům. Pokud přece jen uživatel zadá nekorektní vstupní gramatiku, je žádoucí, aby nástroj na chybu upozornil co nejkonkrétnějším způsobem.

**Možnost lokalizace řešení** – dalším zajímavým požadavkem byla možnost snadné změny textů v algoritmu bez zásahu do programovacího kódu. To např. umožňuje snadnou modifikaci nevyhovujících textů nebo jednoduché vytvoření více jazykových variant. Představa řešení je taková, že každý algoritmus bude svázán s textovým souborem, který bude obsahovat veškeré texty algoritmu. Algoritmus poté bude veškeré texty čerpat právě z tohoto textového souboru. Bude ovšem nutné dodržet striktní strukturu lokalizačního souboru.

## 6.2 Analýza grafických objektů

Dříve, než jsme přistoupili k tvorbě samotného grafického rozhraní naší práce, bylo potřeba si uvědomit, co vše toto řešení bude vyžadovat. Proto si v této kapitole stroze shrneme, co je od grafického rozhraní požadováno a v pozdějších kapitolách naše řešení více rozebereme.

### 6.2.1 Vstupní prvky

- **Pole pro vstupní gramatiku** – naše řešení musí kromě samotné ukázky průchodu algoritmu umět správně zpracovat uživatelem zadanou vstupní gramatiku. Proto je nutné vytvořit vstupní pole pro zadání této gramatiky
- **Pole pro dodatkovou funkci** – jedná se o volitelné vstupní pole pro zadání doplňující funkce. Toto pole budeme využívat u algoritmů funkcí first a follow.

### 6.2.2 Informační textové prvky

- **Pole pro zobrazení načtené gramatiky** – prvním informačním prvkem bude pole pro zobrazení vstupní gramatiky tak, jak jí pochopil mechanismus načtení. Jelikož každý uživatel bude zadávat gramatiku jiným stylem (s mezerami x bez mezer, ve více řádcích x v jednom řádku), je potřeba tuto gramatiku ještě jednou zobrazit, tentokrát s určitou a jednotnou štábní kulturou.
- **Pole pro zobrazení průběhu algoritmu** – velice důležité pole, které bude ukazovat průběh samotného algoritmu. Tento průběh bude záviset pouze na samotném algoritmu (např. naplňování určité množiny).
- **Pole pro výsledek** – je potřeba v nějakém poli zobrazovat výsledek, případně konečný verdikt. Scénář tohoto pole bude záviset opět pouze na samotném algoritmu.
- **Pole pro textový výpis** – další důležité pole, kde bude vypisován ve větách každý krok algoritmu. Toto pole bude sloužit jako hlavní pomůcka pro pochopení samotného algoritmu.

### 6.2.3 Vizualizační prvky

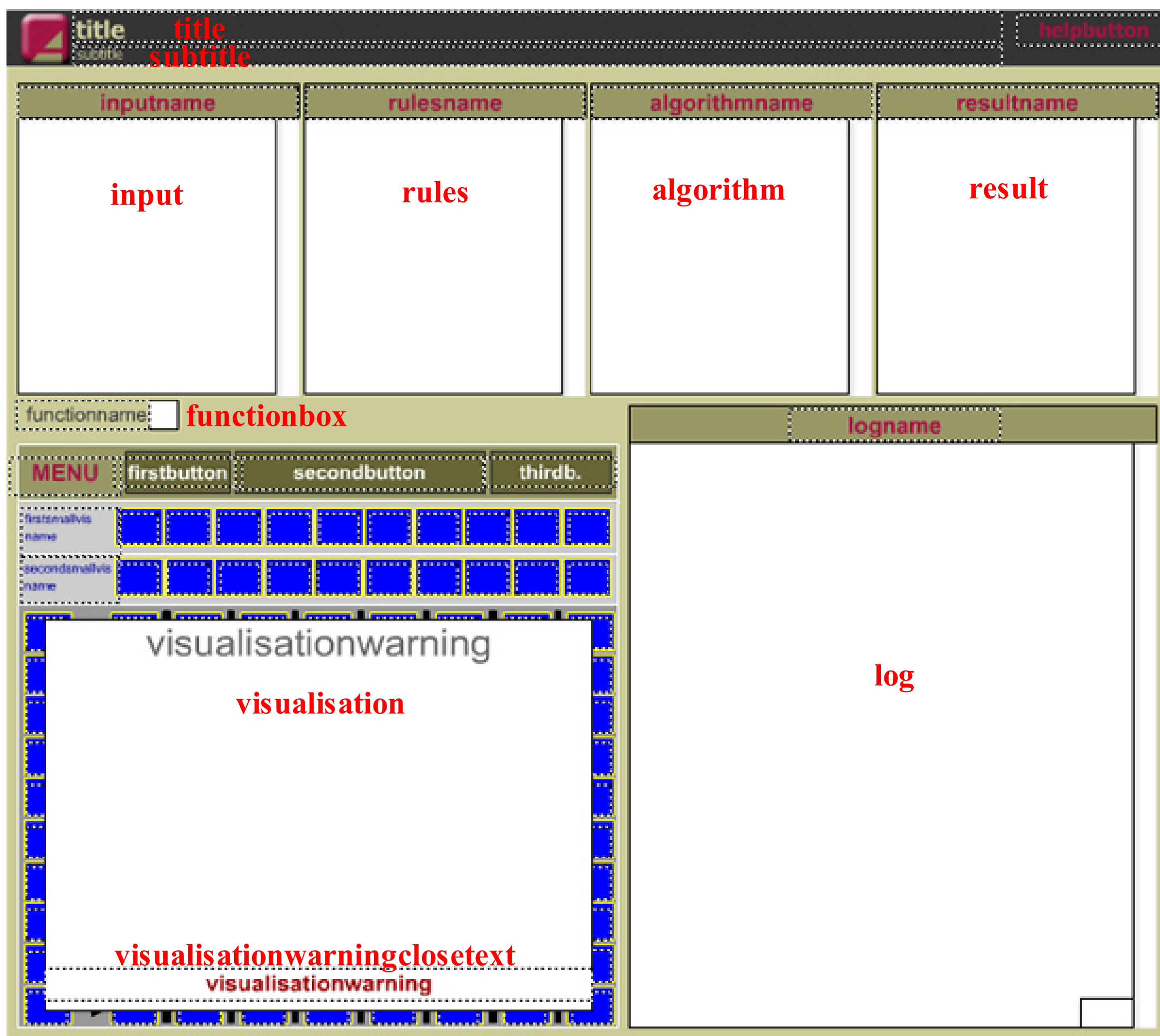
- **Hlavní vizualizace** – důležitou částí tohoto řešení bude i vizualizace algoritmu. Bude tedy potřeba vyhradit určitý prostor layoutu pro grafické znázornění průběhu algoritmu.
- **Vedlejší vizualizace** – kromě hlavní vizualizace je žádoucí vytvořit i pomocnou menší vizualizaci, která bude znázorňovat méně náročné operace (např. naplňování množiny prvky).

### 6.2.4 Tlačítka

- **Načtení gramatiky** – jak již název napovídá, tímto tlačítkem se bude načítat vstupní gramatika. Toto tlačítko lze sjednotit s tlačítkem, které bude reprezentovat krok algoritmu.
- **Další krok algoritmu** – stisknutím tohoto tlačítka bude proveden algoritmus po určitých krocích.
- **Reset** – tlačítko, které vrátí algoritmus do vstupního stavu.
- **Nápověda** – každý algoritmus by měl obsahovat nápovědu, ve které bude vysvětleno jeho použití.
- **Generování gramatiky** – v případě, že bude řešení obsahovat i generování vstupní gramatiky, je nutné vytvořit pro tuto akci tlačítko.

## 6.3 Řešení grafické části

Jedním ze základních požadavků bylo vytvoření intuitivní grafické části algoritmu. Pomocí technologie Adobe Flash nebyl tento úkol nijak obtížný, jelikož veškerá grafika je zde pouze klikací záležitost. Veškeré grafické práce jsou vytvářeny v rámci jednoho FLA souboru. Bylo tedy nejdříve nutné navrhnout samotný layout celé vizualizace. Pro ten jsme zvolili rozměry 840x740 pixelů. Tento rozměr plně vyhovuje pro zobrazení všech důležitých prvků algoritmu. Nyní přejdeme k samotnému layoutu, jehož podobu můžeme vidět na následujícím obrázku:




Obr 5 : Základní podoba layoutu

### 6.3.1 Popis jmenných prvků layoutu

- **title** – hlavní nadpis v hlavičce algoritmu
- **subtitle** – jednořádkový text pod nadpisem v hlavičce algoritmu
- **inputname** – nadpis prvního z textových polí. Pole slouží pro vstupní gramatiku zadanou uživatelem
- **rulesname** – nadpis textového pole s výpisem vstupní gramatiky pro kontrolu správného zadání
- **algorithmname** – nadpis textového pole pro kontrolní výpis průběhu algoritmu.
- **resultname** – nadpis textového pole pro výpis konečného verdiktu
- **logname** – nadpis textového pole pro výpis chování algoritmu.

- **menu** – pojmenování hlavní nabídky
- **firstbuttonname** – text prvního tlačítka nabídky
- **secondbuttonname** – text druhého tlačítka nabídky
- **thirdbuttonname** – text třetího tlačítka nabídky
- **helpname** – text tlačítka pro nápovědu
- **functionname** – nadpis textového pole pro případný výpočet funkce algoritmu
- **firstsmallvisname** – nadpis první malé vizualizace
- **secondsmallvisname** – nadpis druhé malé vizualizace
- **visualisationwarningclosetext** – text v tlačítku pro zavření hlášení vizualizace

Výše zmíněný seznam prvků je zároveň i seznamem názvů jejich proměnných, pomocí kterých jsou prvky v programu jednoznačně identifikovatelné. Pro vepsání určitého textu do těchto prvků je nutné použít externí lokalizační soubor nebo text pevně zvolit uvnitř programu. Všechny výše zmíněné prvky jsou pro prostředí Adobe Flash označeny jako dynamické textové pole, nad kterým ActionScript definuje několik funkcí. V dalším seznamu si vyjmenujeme pouze ty nejdůležitější:

- **_root.promenna.text = „“** – do pole s názvem promenna vepíše daný text. Ten má následně formát, který byl předurčen textovému poli při jeho vytváření. Tato funkce se může během vykonávání programu opakovat a tím měnit hodnoty pole.

- **_root.promenna.htmlText = „“** - pole s názvem promenna vepíše daný text. Na rozdíl od předchozího příkazu je tento text formátován jako HTML (ořezaná verze). Lze tedy text v tomto prvku formátovat nezávisle na pevném formátu tlačítka.

- **_root.promenna._visible = boolean** – příkaz o viditelnosti prvku s názvem proměnná. Hodnota true zviditelní prvek, hodnota false ho zneviditelní. Stejně jako u předchozího příkazu lze i tento v programu několikanásobně opakovat.

Možností editace těchto prvků je samozřejmě mnohem více, ale při implementaci algoritmů si bohatě vystačíme s těmito třemi.

### 6.3.2 Popis vstupních a informačních prvků layoutu

V předešlé kapitole jsme si vysvětlili funkci jmenných prvků. Tyto prvky slouží pouze jako textové nadpisy a jsou vždy asociovány s některým vstupním nebo informačním prvkem. Jako vstupní prvek si můžeme představit textové pole pro zadání vstupní gramatiky nebo u některých algoritmů dodatečné vstupní pole pro určení funkce (např. funkce first). Informační prvky jsou pak další textová pole, do kterých nelze nic vepsat a slouží např. pro výpis průchodu algoritmu nebo pro výpis konečné gramatiky. Tato pole mají tedy informační charakter a pomáhají ke snazšímu pochopení dané gramatiky. V následujícím seznamu si popíšeme funkce těchto prvků:

- **input** – vstupní prvek, který slouží pro zadání uživatelské gramatiky.

- **rules** – informační prvek. Po načtení uživatelem zadané gramatiky vypíše prvek tuto gramatiku ve formátu takovém, jak ji algoritmus pochopil.

- **algorithm** – velice důležitý informační prvek. Jedná se o výpis hlavní funkce algoritmu. Pomocí tohoto prvku lze kontrolovat průběh celého algoritmu. Na stejném principu funguje i hlavní grafická vizualizace.

- **result** – informační prvek pro výpis finálního verdiktu algoritmu. Některé algoritmy odpovídají pouze formou odpovědi ano/ne, některé vypíšou množinu prvků a některé formulují novou opravenou gramatiku. Výpis tohoto prvku je zpravidla aktivován na samotném konci algoritmu.

- **log** – velice důležitý informační prvek, který podrobně popisuje, co v určité chvíli algoritmus dělá. Log byl navrhnut tak, aby pouze s jeho pomocí byl uživatel schopen pochopit průběh celého algoritmu. Sekundární funkcí tohoto prvku je také výpis potenciálních chyb, kterých se uživatel dopustí při zadání vstupní gramatiky.

- **visualisation** – informační prvek, který informuje o stavu vizualizace. Před spuštěním algoritmu definuje, co má vizualizace uživateli ukázat. Tento prvek se také aktivuje při překročení limitů vizualizace. Pokud k tomu dojde, je vizualizace deaktivována a následuje výpis informační zprávy.

- **functionbox** – druhý vstupní prvek použitý pouze u některých algoritmů. Definuje dodatečnou informaci ke vstupní gramatice. Jako tento dodatek můžeme považovat například definici symbolu ve funkci first.

Jelikož jsou tyto prvky v prostředí Adobe Flash definovány jako vstupní nebo dynamické pole, definuje nad nimi ActionScript stejné funkce, jako v kapitole 6.3.1.

### 6.3.3 Popis grafických prvků layoutu

Poslední skupinou prvků, vyskytujících se v layoutu algoritmu, jsou grafické prvky. Jedná se o všechny obrazce a tvary. Existují dva základní typy těchto prvků. Prvním typem je klasický pevně definovaný objekt. Tímto stylem jsou definovány veškeré rámečky v našem layoutu. Jsou to prvky, které stačí pevně definovat při vytváření layoutu a nepředpokládá se další jejich využití přímo v programu algoritmu. Takovéto prvky nemají žádný jednoznačně identifikovatelný název a po vytvoření s nimi nelze nijak pracovat. Druhou skupinou objektů jsou tzv. symboly. Pod pojmem symbol si můžeme představit alespoň jeden objekt (lze i skupina objektů), který je jednoznačně definován a uložen pod unikátním názvem. Se symbolem lze oproti obyčejným prvkům v ActionScriptu přímo pracovat. Symboly se dále dělí podle funkčnosti na 3 typy:

- Tlačítko
- Filmový klip
- Grafika

V našem layoutu můžeme symboly nalézt jako pozadí tlačítek či jako prvky vizualizace (o vizualizaci více v kapitole 6.3.4). V našem layoutu se nachází pozadí tlačítek ve čtyřech případech:

- **firstbutton** – pozadí prvního funkčního tlačítka v menu. Jeho funkce je implicitně definována jako generování předem uložených vstupních gramatik. Tato funkce je určená pro rychlejší demonstraci algoritmu bez nutnosti uživatele vyplňovat vstupní gramatiku.

- **secondbutton** – pozadí druhého funkčního tlačítka v menu. Jeho funkce je implicitně definována jako další krok algoritmu. Postupným stiskáváním tohoto tlačítka můžeme sledovat vývoj algoritmu krok po kroku. Pokud dorazíme na konec algoritmu, je toto tlačítko zneviditelněno.

- **thirdbutton** – pozadí třetího funkčního tlačítka v menu. Jeho funkce je implicitně definována jako reset celého algoritmu. Po stisknutí tohoto tlačítka jsou smazány všechny vstupní i informační prvky a algoritmus je uveden do původního stavu a je připraven pro zadání vstupu nové gramatiky.

- **helpbutton** – pozadí posledního funkčního tlačítka. Jeho funkce je pevně nastavena jako vyvolávač nápovědy algoritmu. Více informací o nápovědě v kapitole X.

ActionScript definuje nad každým symbolem standardní množinu funkcí. My si zde opět vypíšeme pouze ty v našem případě užitečné:

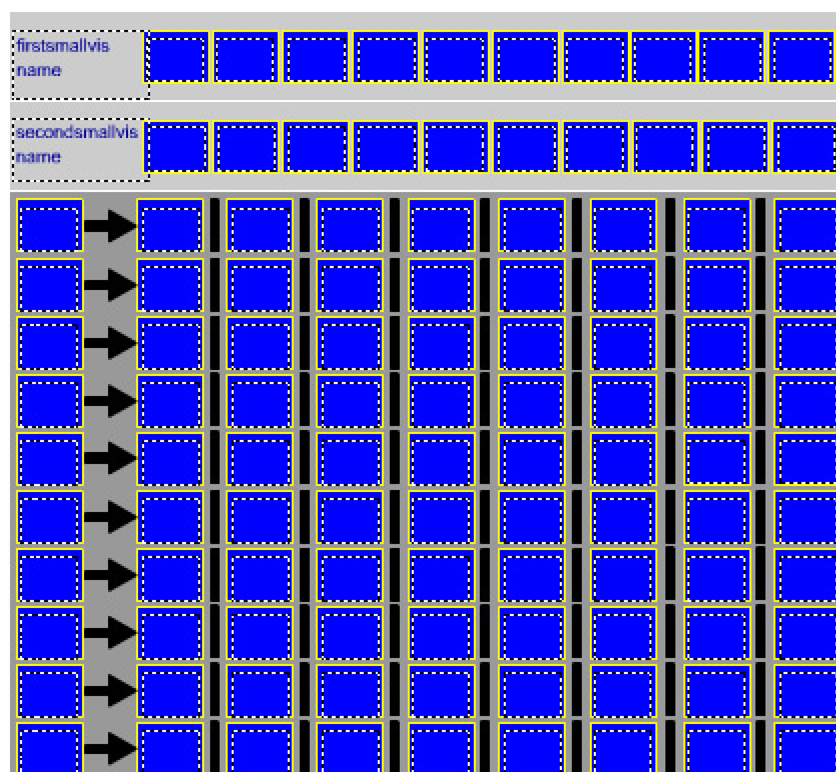
- **_root.promenna._visible = boolean** – příkaz o viditelnosti symbolu s názvem proměnná. Hodnota true zviditelní symbol, hodnota false ho zneviditelní.

- **barva = new Color(promenna) –** definice prvku typu Color s názvem barva a propojení s prvkem s názvem *promenna*. Po této definici lze jednoduše změnit barvu daného prvku pomocí příkazu **barva.setRGB(0x000000)**, kde v závorce jsou hodnoty barvy ve formát RGB.

Grafické funkce typu viditelnost prvku a změna barvy nám v našem případě vystačí. Symboly mají ovšem kromě těchto funkcí další velkou výhodu. Jedná se o nastavení akcí, které jsou spjaté s daným prvkem. Tímto způsobem lze jednoduše definovat akci, která se vykoná např. stisknutím daného tlačítka. Tímto způsobem jsou v našem případě definované akce k funkčním tlačítkům.

### 6.3.4 Vizualizace

Jedním z hlavních požadavků práce bylo vytvoření přehledné grafické vizualizace průběhu algoritmu. V kapitole 6.3.3 jsme si řekli něco o symbolech a zmínili jsme se, že je pomocí nich vytvořená i celá vizualizace. V našem případě dělíme celkovou vizualizaci naší práce na 3 části.

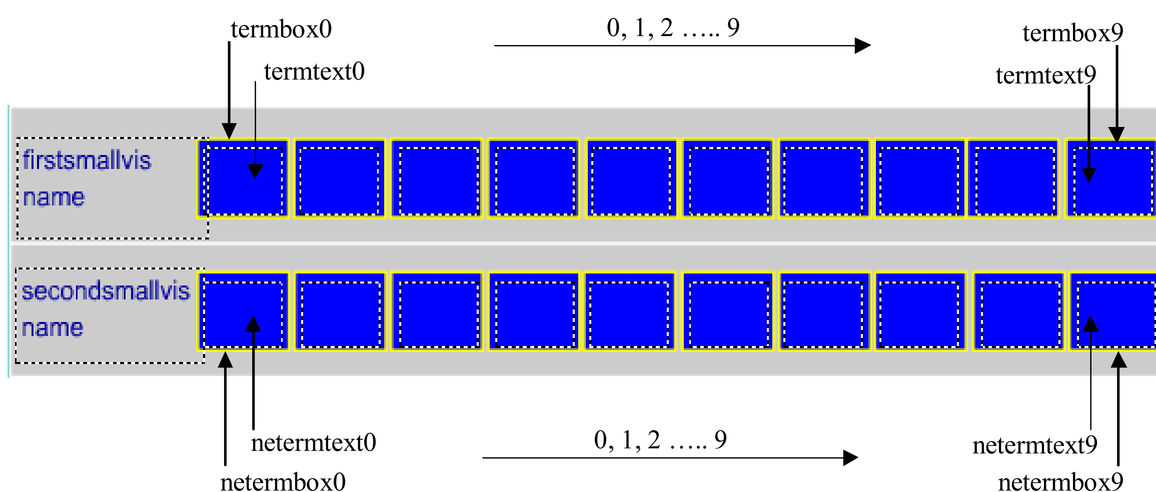



Obr 6 : Základní podoba vizualizace

#### 6.3.4.1 Malá vizualizace

První částí vizualizace je sada deseti prvků v řadě s označením „firstsmallvisname". Na stejném principu funguje i druhá část, která se vyskytuje pod částí první a má popisek secondsmallvisname. Obě tyto části slouží např. k výpisu objevených prvků či nalezených dobrých neterminálních symbolů (viz teoretická kapitola 3.4.5). U algoritmů funkce first a follow lze tuto vizualizaci využít např. jako znázornění naplňování množiny F nebo jako výpis posloupnosti použitých pravidel.

Stejně jako v kapitole 6.3.3 jsou i zde hlavními funkcemi viditelnost prvků a změna jejich barvy. Abychom mohli provádět tyto operace, musíme znát názvy všech prvků. Na následujícím obrázku můžeme vidět hierarchii pojmenování jednotlivých prvků.

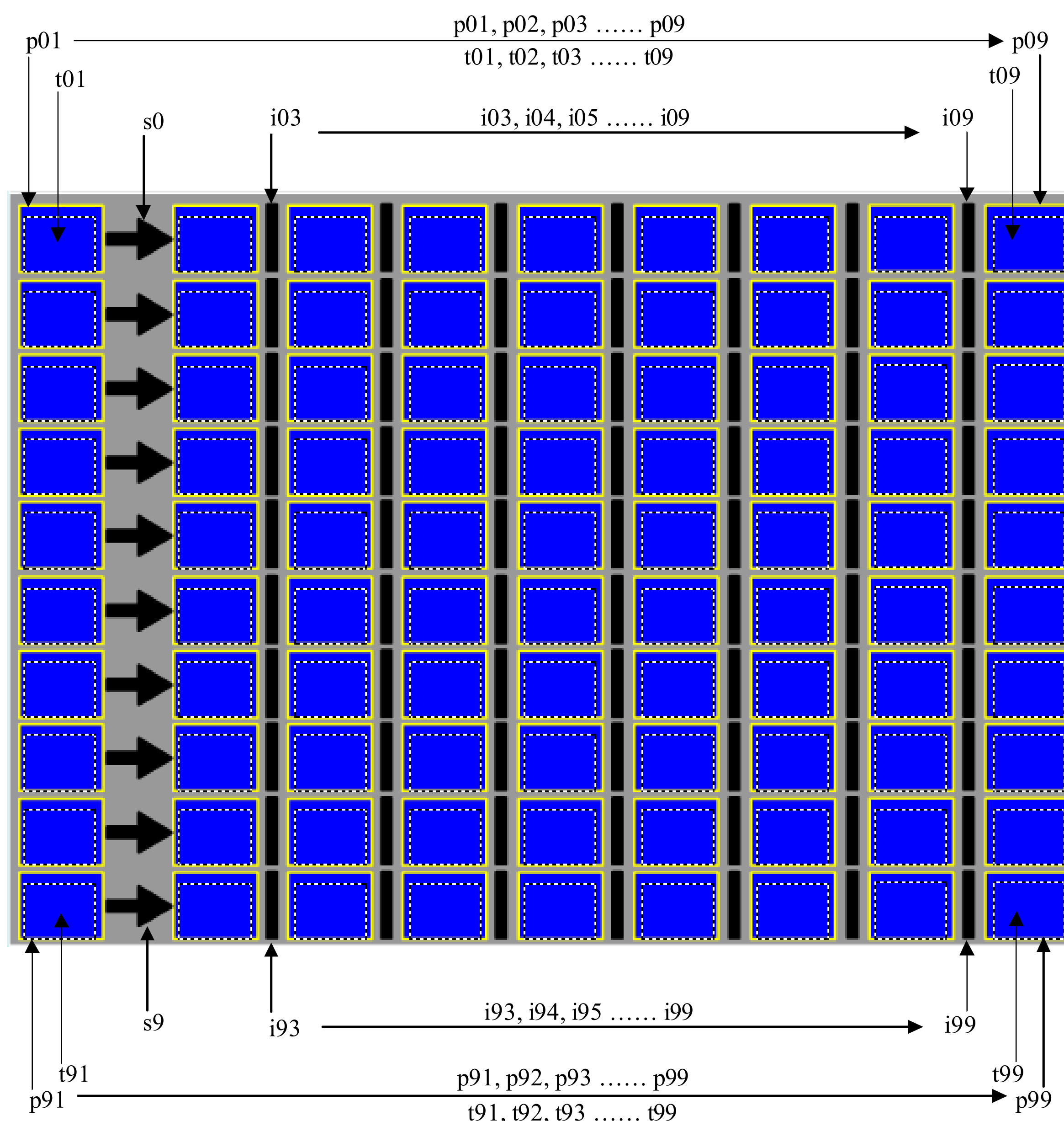



Obr 7 : Malá vizualizace

Jak je na obrázku 7 vidět, jeden prvek malé vizualizace se skládá ze 2 komponent. První komponentou je modrý obrazec, který symbolizuje hranice prvku (označení box). Druhou komponentou je pak text prvku, jehož hodnota se může měnit (označení text). Operace nad těmito prvky byly vysvětleny již v kapitolách 6.3.1 a 6.3.3.

#### 6.3.4.2 Velká vizualizace

Hlavní částí celé vizualizace je tzv. velká vizualizace, která obsahuje matici prvků o rozměru 9 x 10. Zde je vizualizován průběh celého algoritmu. Vzhledem k tomu, že je zde tento průběh často znázorněn na samotné gramatice, jsou prvky vizualizace doplněné i o pomocné prvky ve formě přepisovacího symbolu (->) a symbolu „nebo“ (|). Schéma rozložení prvků velké vizualizace a hierarchie jejich názvů můžeme vidět na následujícím obrázku.




Obr 8 : Velká vizualizace

Jak je vidět na obrázku 8, velká vizualizace se skládá s již většího počtu prvků a k jejich hierarchii je důležité říct více informací.

Základní tvar názvu prvku je ***aXY*** (s výjimkou prvku pro přepisovací symbol ve tvaru ***aX***), kde *a* je symbol typu prvku, *X* je vertikální souřadnice a *Y* je souřadnice horizontální.

**Typ prvku** – p (tvar prvku), t (text prvku), s (přepisovací symbol), i (symbol „nebo“)
**Rozsah vertikální souřadnice** – 0…9
**Rozsah horizontální souřadnice** – 1…9
Stejně jako u předchozích prvků platí i ve velké vizualizaci stejné operace nad jednotlivými prvky. Textové prvky mohou měnit svůj text a viditelnost, grafické prvky pak mohou měnit viditelnost a barvu.

## 6.4 Část společného API

Dalším požadavkem bylo vytvoření jednotného prostředí společného pro všechny algoritmy. Hlavním důvodem byla jednoduchá tvorba dalších algoritmů a odstínění problémů společných s načítáním dat a práce s nimi. Toto jednotné prostředí je implementováno v odděleném souboru s názvem ***grammar.as***. Tento soubor musí být vždy přítomen spolu se souborem typu FLA (viz kapitola 6.2), který definuje grafickou základnu této práce. Za přítomnosti těchto 2 souborů jsme schopni snadno implementovat další algoritmus. V další kapitole přejdeme k funkcím, které naše API poskytuje.

### 6.4.1 Datový model

První velice důležitou funkcí společného prostředí je datový model načtené gramatiky. Museli jsme zvolit vhodnou podobu datové struktury, se kterou se bude snadno pracovat. Pro jednoduchost jsme proto zvolili dynamické pole, jehož prvky jsou v podobě řetězce ve speciálním tvaru každé řádky načtené gramatiky. Pravidla pro získání speciálního tvaru nejsou nijak složitá. Jednotlivé prvky zůstanou nezměněny a veškeré přepisovací symboly a symboly „nebo“ se přepíší na znak „|“ (svislítko). Poslední úpravou je odstranění všech bílých znaků, především mezer.

Proto např. pravidlo gramatiky ***S*** => ***aB | bB | C;*** se uloží ve tvaru ***S|aB|bB|C***.

První poznámkou k výše uvedenému modelu může být dotaz na to, jak prostředí rozezná prvky na pravé a na levé straně pravidla (prvky před a za přepisovacím symbolem). Odpověď je velice jednoduchá. Jelikož veškeré algoritmy, probírané v hodinách předmětu KIV/FJP, se týkají pouze bezkontextových gramatik, obsahuje levá strana vždy pouze jeden neterminální symbol. Proto první prvek vždy náleží levé straně (na předchozím příkladu prvky označené modrou barvou), zbytek prvků pak náleží straně pravé (označené červenou). Pro jednoznačnou demonstraci ukážeme princip na jednoduchém příkladu:

| Vstup | | Výsledek |
|---|---|---|
| S => aB \| bC \| B; | | S\|aB\|bC\|B |
| A => a \| Ba \| c; | | A\|a\|Ba\|c |
| B => ac \| bB; | | B\|ac\|bB |
| C => dA \| bC \| cD; | → | C\|dA\|bC\|cD |
| D => ac \| bG; | | D\|ac\|bG |
| E => f \| Bc \| b; | | E\|f\|Bc\|b |
| G => aB; | | G\|aB |

Každý řádek výsledného formátu je uložen jako jeden prvek dynamického pole. V našem případě by mělo pole celkem 7 prvků. S tímto formátem gramatiky poté umí pracovat veškeré funkce společného prostředí, o kterých si povíme později.

### 6.4.2 Načítání gramatiky

Datový model načítané gramatiky jsme si již představili. Nyní se seznámíme se samotným načtením vstupní gramatiky. Toto načtení není samo o sobě nijak zajímavé. Ze vstupní gramatiky se odstraní veškeré bílé symboly a přepisovací symbol se přepíše na znak „|“ (svislítko). Zajímavější částí samotného načítání je ovšem formát vstupní gramatiky a implementace algoritmu ke zjišťování chyb v této gramatice. Vstupní gramatika má předem určený formát, který je pro každý algoritmus shodný. Aby byl vstupní formát dodržen, je nutné se seznámit s určitými pravidly:

- Dodržení správného zápisu přepisovacího symbolu ve tvaru „=>“. Jiný způsob zápisu není akceptovatelný.
- Každé pravidlo gramatiky musí být ukončeno středníkem (1 pravidlo != 1 řádek).

- Levá strana pravidla smí obsahovat pouze neterminální symbol (bezkontextová gramatika)
- V gramatice jsou povolené pouze množiny symbolů [a-z],[A-Z],[0-9] a [>,=,|].

Jelikož mohou uživatelé chybovat, bylo nutné vytvořit algoritmus pro detekci možných chyb ve vstupní gramatice. Tento algoritmus musí být schopen detekovat základní chyby a poté co možno nejpřesněji uživatele informovat. Proto jsme definovali celkem 5 základních uživatelských chyb, které jsme se pokusili ošetřit. Jedná se o chyby typu:

- **Chyba 1** : Absence nebo špatná syntaxe přepisovacího symbolu "=>". Chybu může způsobit také absence středníku na konci některého pravidla a tím způsobit dvojitý výskyt přepisovacího symbolu "=>" v jednom pravidle.
- **Chyba 2** : V gramatice se objevuje více svislítek "|" za sebou.
- **Chyba 3** : Byl vložen některý z nepovolených znaků. Uživatel smí zadávat pouze znaky [a-z],[A-Z],[0-9] a [>,=,|].
- **Chyba 4** : Levá strana některé gramatiky obsahuje nepovolený symbol (povolen je pouze neterminální symbol).
- **Chyba 5** : Nebyla zadána žádná vstupní gramatika.

Výše zmíněný seznam chyb je naše řešení schopné detekovat a do logu vypíše vždy odpovídající varování. Kromě tohoto varování se ve většině případů vypíše i číslo řádku, ve kterém k dané chybě došlo. Příklad takového výstupu lze vidět na následujícím obrázku.

**Vaše zádání**

s => aB | bB | C;
A => a | B% | c;
B => aS | bB;
C => dA | bC | cD;
D => ac || bG;
E > f | Ac | b;
G => a;

**Log**

Chyba 4, řádek 1 : Levá strana některé gramatiky obsahuje terminální symbol.

Chyba 3, řádek 2 : Vložil jste některý z nepovolených znaků. Vkládejte pouze znaky [a-z],[A-Z],[0-9] a [>,=,|].

Chyba 2, řádek 5 : V gramatice se objevuje více svislítek "|" za sebou.

Chyba 1, řádek 6 : Absence nebo špatná syntaxe přepisovacího symbolu "=>". Chybu může způsobit také absence středníku na konci některého pravidla a tím způsobit dvojitý výskyt přepisovacího symbolu "=>" v jednom pravidle.

Obr 9 : Ukázka zadání nekorektní gramatiky

### 6.4.3 Funkce rozhraní

Nyní přecházíme k té nejdůležitější části společného rozhraní, a to k funkcím, které jsou uživateli poskytnuty. Všechny tyto funkce se nacházejí opět v externím zdrojovém souboru s názvem ***grammar.as***. Dříve, než budeme moci používat dané funkce, musíme vytvořit instanci objektu třídy grammar. V ActionScriptu lze tuto instanci vytvořit příkazem:

```
var promenna:grammar = new grammar();
```

po vytvoření této instance můžeme již naplno využívat funkce, které nám jsou nabízeny. Nyní si zde vypíšeme seznam těchto funkcí, vysvětlíme jejich význam a seznámíme se s jejich vstupními parametry a návratovou hodnotou.

- **vratGramatiku()** – funkce, která vrátí dynamické pole s načtenou vstupní gramatikou. Jedná se pouze o návratovou funkci a jejímu použití musí předcházet funkce pro načtení gramatiky
- **zjistiPrvek(radek, poradi)** – z načtené gramatiky zjistí prvek na daných souřadnicích. Pokud tedy potřebuji vrátit symbol na levé straně prvního řádku, použiji parametry souřadnic 1,1.
- **zjistiPravouStranu(radek)** – funkce, která vrátí dynamické pole všech prvků, které se nacházejí na pravé straně daného řádku.
- **vratBezDuplicity(inArr:Array)** – funkce, která odstraní duplicitní prvky z dynamického pole, udaného jako parametr a vrátí dynamické pole, ve kterém se jeden prvek vyskytuje pouze jednou.
- **zmenBarvu(prvek, barva)** – jedná se o funkci pro vizualizaci. Změní barvu prvku s názvem „prvek“. Možných barev je celkem 5, a to ve tvaru „cervena“, „seda“, „zelena“, „cerna“ a „zluta“.
- **viditelnost(prvek, viditelnost)** – funkce, která mění viditelnost prvku s názvem „prvek“. Parametr „viditelnost“ pak může nabývat hodnot true nebo false. Tuto operaci lze také provést pomocí vnitřní funkce ActionScriptu (viz kapitola 6.3.3).

- **otestujANacti(nacistVizualizaci:Boolean)** – hlavní funkce pro načtení vstupní gramatiky. Funkce zkontroluje korektnost zadání uživatele a podle toho buď gramatiku načte, nebo vypíše chyby. Jediný parametr určuje to, jestli se má načtená gramatika rovnou vizualizovat ve velké vizualizaci. Tato funkce je použita u prvních dvou algoritmů. Funkce `otestujANacti()` dále využívá dalších funkcí společného rozhraní, které nejsou již přístupné při vytváření algoritmu.
- **getLoc(pole,promenna)** – funkce, která zajišťuje jednoduchou lokalizaci jmenných prvků. Více informací o lokalizaci viz kapitola 6.4.4.
- **getLocLog(pole,extPromenna,poleIntPromennych:Array,htmltext:Boolean)** - funkce, která zajišťuje složitou lokalizaci textového výpisu. Na rozdíl od předchozí lokalizační funkce podporuje vkládání proměnných hodnot. Více informací o lokalizaci viz kapitola 6.4.4.

### 6.4.4 Lokalizace

Předchozí kapitola nás seznámila se společným rozhraním pro všechny algoritmy. Ovšem vytvoření tohoto společného rozhraní by nebylo možné bez použití externí lokalizace veškerých textů v algoritmu. Jako příklad si vezmeme první algoritmus pro zjištění nedostupných symbolů a druhý algoritmus pro zjištění prázdnosti jazyka. I když se oba algoritmy tváří velice podobně, první veliký rozdíl je v konečném verdiktu, který nám algoritmus nabídne. Zatímco u prvního algoritmu je výsledkem upravená gramatika bez nepotřebných pravidel, druhý algoritmus pouze rozhodne, jestli je jazyk prázdný nebo ne. Nadpis okna pro zobrazení výsledků je tedy u obou případů jiný. U prvního algoritmu je toto okno nadepsáno jako „Výsledná gramatika“, u druhého pak jako „Výsledek“. Pokud by byly hodnoty těchto nadpisů definovány napevno přímo ve zdrojovém kódu, nemohlo by existovat společné rozhraní a tyto rozdíly by bylo potřeba rozlišit. Abychom mohli myšlenku společného rozhraní splnit, museli jsme zavést i možnost externí lokalizace všech textových prvků v algoritmu. Jak můžeme vidět na obrázku 5 v kapitole 6.3, všechny textové prvky jsou nastaveny na název proměnné daného prvku. Veškeré nadpisy a texty se po spuštění algoritmu načítají z externího souboru s názvem **`loc`**, který se musí nacházet ve stejném adresáři jako grafická nástavba ve formátu FLA a musí striktně splňovat předem daný formát. Nejdříve si popíšeme všechny povinné proměnné, které se musí v externím souboru vyskytovat.

```
&empty="prázdný text pro smazání textu"
&title="hlavní nadpis v hlavičce algoritmu"
&subtitle="jednořádkový text pod nadpisem v hlavičce algoritmu"
&input="nadpis prvního okna pro zadání vstupní gramatiky"
&rules="nadpis druhého okna pro vypsání pravidel"
&algorithm="nadpis třetího okna pro vypsání algoritmu"
&result="nadpis čtvrtého okna pro vypsání výsledku"
&menu="nadpis hlavní nabídky"
&log="nadpis okna pro textový výpis (log)"
&help="text tlačítka pro nápovědu"
&function="nadpis dodatečně funkce(např. follow)"
&firstSmallVis="nadpis první malé vizualizace"
&secondSmallVis="nadpis druhé malé vizualizace"
&loadGrammar="text druhého tlačítka v menu pro načtení gramatiky"
&firstButton="text prvního tlačítka pro generování gramatiky"
&thirdButton="text třetího tlačítka pro reset algoritmu"
&visualisationWarningClose="text tlačítka pro odstranění hlášky"
&visualisationWarning="varování v oblasti vizualizace při spuštění"
&grammarFailed="výpis při porušení limitu vizualizace"
&logGrammarSuccessLoad="výpis do logu po úspěšném načtení gramatiky"
&grammarError="chybová hláška v logu při chybné gramatice"
&grammarErrorMessage="chybová hláška v okně při chybné gramatice"
&grammarError1="chybová hláška při výskytu chyby č.1"
&grammarError2="chybová hláška při výskytu chyby č.2"
&grammarError3="chybová hláška při výskytu chyby č.3"
&grammarError4="chybová hláška při výskytu chyby č.4"
&grammarError5="chybová hláška při výskytu chyby č.5"
&helpText=text vyskakovací nápovědy v algoritmu
```

Obr 10 : Ukázka zdrojového souboru

Pokud bude v lokalizačním souboru některá z povinných proměnných chybět nebo bude chybná, výsledek se projeví na funkci algoritmu. Lokalizační soubor může obsahovat i více proměnných, které závisí pouze na potřebě vývojáře nového algoritmu. Tyto proměnné pak může využívat ve zdrojovém kódu samotného algoritmu.

Pro aplikaci textu na některý textový prvek v algoritmu slouží dvě funkce společného rozhraní. Jedná se o funkce:

- **getLoc(pole,promenna)** – funkce, která zajišťuje jednoduchou lokalizaci jmenných prvků. Jako parametr pole se udává název jmenného prvku algoritmu, kterému je daný text určen. Parametrem promenna je poté název proměnné v externím souboru (např. `&helpText`). Tento parametr však nesmí nikdy obsahovat symbol &, který pouze v lokalizačním souboru uvádí proměnnou, ale není součástí samotného názvu. Oba parametry je nutné psát v uvozovkách. Praktické použití této funkce může tedy vypadat následovně: `getLoc("inputname","input");`

- **getLocLog(pole,extPromenna,poleIntPromennych:Array,htmltext:Boolean)** - funkce, která zajišťuje složitou lokalizaci textového výpisu. Na rozdíl od předchozí lokalizační funkce podporuje vkládání proměnných hodnot. Parametry `pole` a `extPromenna` mají stejnou funkci, jako parametry v předchozím případě funkce `getLoc(pole,promenna)`. Parametr poleIntPromennych je dynamické pole hodnot proměnných, které se v textu mají vyskytovat. V lokalizačním souboru je místo pro takovou proměnnou označené symbolem # (hashmark). Princip budeme demonstrovat na jednoduchém příkladu:

  *&grammarError0=Chyba 0, řádek # : chybí středník ";" na konci celé gramatiky.*

  Lokalizační proměnná `grammarError0` jasně definuje, že v textu bude číslo řádku proměnlivá hodnota (#). Tuto hodnotu je potřeba uložit jako prvek pole v proměnné `poleIntPromennych`**.** Pořadí prvků této proměnné bude odpovídat pořadí symbolů # v textu lokalizační proměnné.

  Posledním parametrem je pravdivostní hodnota `htmlText`. Při nastavení hodnoty `true` bude text, načtený z lokalizační proměnné, interpretován jako ořezaný HTML kód. Lze tedy např. jednoduše zvýraznit některé části textu pomocí primitiv <b></b>.

Kompletní příklady lokalizačních souborů můžeme vidět v příloze této práce. Všechny texty, umístěné v lokalizačních souborech, jsou pouze zkušební a jejich finální znění závisí pouze na budoucím poskytovateli.

Nyní jsme si ukázali, jak dodržet správnou strukturu lokalizačního souboru a jak použít některou z lokalizačních proměnných v našem algoritmu. Díky podpoře externí lokalizace není v budoucnu problém vytvořit pro algoritmy více jazykových schémat bez jakéhokoliv zásahu do samotného programového kódu. V rámci této práce byla vytvořena pouze česká lokalizace.

### 6.4.5 Generování gramatik

Další usnadňující funkcí společného rozhraní je možnost generování předem uložených gramatik. Často se stává, že uživatel nepotřebuje zjistit výpočet vlastní gramatiky, ale potřebuje demonstrovat funkčnost algoritmu na libovolném příkladu. V tomto ohledu je vytváření vlastní vstupní gramatiky nežádoucí faktor zejména z důvodu časové náročnosti. Proto jsme vytvořili možnost jednoduchého generování předem definovaných gramatik. Tyto gramatiky jsou uloženy v externím textovém souboru s názvem template.txt. Formát tohoto souboru musí být striktně dodržen a styl formátování můžeme vidět na následujícím příkladu:

```
&template1=<p>S => aB | bB | C;</p><p>A => a | BA | c;</p><p>B => aS | bB;</p><p>C => dA | bC | cD;</p><p>D => ac | bG;</p><p>E => f | Ac | b;</p><p>G => a;</p>

&template2=<p>S => aB | bC | B;</p><p>A => a | Ba | c;</p><p>B => ac | bB;</p><p>C => dA | bC | cD;</p><p>D => ac | bG;</p><p>E => f | Bc | b;</p><p>G => aB;</p>
```

Pravidla gramatiky jsou ve tvaru HTML kódu. Každé pravidlo je uzavřené v odstavcových elementech <p>. Každá gramatika je definovaná pod určitým názvem proměnné. Název musí být vždy ve tvaru `&templateX`, kde `X` je celočíselná hodnota v intervalu *1*...n. Tyto hodnoty musí být vzestupného charakteru a posloupnost musí být vždy úplná bez vynechaných hodnot. Příklad správné posloupnosti můžeme demonstrovat např. jako *„1,2,3,4,5,6"*. Příklady nesprávné posloupnosti jsou např. *„1,2,3,5,7"* nebo *„3,4,5,6"*. Použití mezer mezi jednotlivými prvky je čistě uživatelská záležitost a program dokáže gramatiku zpracovat jak s mezerami, tak i bez nich. Gramatika v externím souboru musí mít stejný tvar jako gramatika, kterou uživatel vkládá do vstupního textového pole algoritmu. Přepisovací symbol tedy musí mít tvar vždy „=>" a symbol „nebo" je reprezentován jako znak „|" (svislítko). Na konci každého pravidla je vyžadován středník. Pokud nejsou tyto podmínky splněny, není zaručena správnost načtené gramatiky a funkčnosti algoritmu. V samotném algoritmu je generování gramatiky vyvoláno stisknutím prvního funkčního tlačítka v menu s nadpisem „Generuj".

## 6.5 Implementace algoritmu pro zjištění nedostupných symbolů

Poté, co jsme si prošli veškeré informace o grafické stránce a o společném prostředí algoritmů, nezbývá nic jiného, než přejít k samotné implementaci jednotlivých algoritmů. Jako první jsme si zvolili algoritmus pro odstranění nedostupných symbolů. Důvodem byla především jednoduchost daného problému. Teoretický popis funkce tohoto algoritmu je vysvětlen v kapitole 3.4.3, proto se zde budeme zabývat pouze konkrétním řešením naší implementace.

Samotný zdrojový kód algoritmu se nachází ve složce ***algorithms*** v souboru s názvem ***algorithm1.as***. Prvním úkolem, který jsme museli udělat, bylo deklarování důležitých proměnných, které musely být platné pro celý rozsah algoritmu. Jednalo se o proměnné:

- **var radky:Array** – v tomto dynamickém poli budeme uchovávat kompletní strukturu načtené gramatiky (viz kapitola 6.4.1)
- **var terminaly:Array** – dynamické pole pro ukládání již prošlých terminálních symbolů.
- **var neterminaly:Array** – dynamické pole pro ukládání již prošlých neterminálních symbolů.
- **var prvkyVAlgoritmu:Array** – dynamické pole, do kterého ukládáme všechny dostupné symboly. Pokud na konci algoritmu existují symboly, které v tomto poli nejsou, jedná se o symboly nedostupné, které náš algoritmus vyhledává.
- **var gramatika:grammar** – instance objektu `grammar` je důležitou částí algoritmu. Bez této instance nebudeme moci využívat funkcí společného rozhraní (viz kapitola 6.4)

Všechny tyto proměnné je poté nutné inicializovat v konstruktoru třídy.

Následuje vytvoření hlavní cyklické funkce celého algoritmu. Tato funkce musí striktně dodržet název, pomocí kterého je poté vyvolávána stisknutím druhého funkčního tlačítka v menu. Podoba funkce je následující:

```
function dalsi(){
   //kód algoritmu při jednom stisknutí druhého funkčního tlačítka v menu
}
```

Nyní se nabízí otázka, jak rozlišit, kolikrát bylo dané tlačítko stisknuto. Algoritmus nemusí vždy po stisknutí tlačítka vykonávat stejnou cyklickou funkci. Číslo kroku (počet stisknutí tlačítka) je uložen v proměnné `_root.krok`, která je indexována od 0. V případě našeho algoritmu reprezentuje první stisknutí tlačítka načtení gramatiky, dalších *n* stisknutí reprezentuje průchod algoritmem a poslední stisknutí vypisuje konečný verdikt. Je tedy zřejmé, že naše cyklická funkce musí mít rozlišeny celkem 3 podoby. Jelikož známe počet stisknutí tlačítka pro průchod algoritmem, není již obtížné rozlišení použití potřebné podoby funkce. V našem případě cyklická funkce vypadá takto:

```
function dalsi(){
      if(_root.krok == 0){
            //kód pro načtení gramatiky
      }
      if(_root.krok > 0 && _root.krok < n){
            //kód pro průběh algoritmu
      }
      if(_root.krok == n){
            //kód pro vypsání konečného verdiktu
      }
      _root.krok++;
}
```

Je důležité nezapomenout na konci cyklické funkce inkrementovat hodnotu `_root.krok`. Pokud by bylo potřeba použít více rozlišení průběhu cyklické funkce, nabízelo by se místo použití podmínky *if* použití přepínače *switch*.

První podoba cyklické funkce je pro všechny implementované algoritmy společná. Jedná se o načtení a uložení vstupní gramatiky pomocí funkce `otestujANacti()` a o zajištění informačních výpisů pomocí funkcí `getLoc()`. Díky těmto funkcím, které poskytuje společné rozhraní algoritmu, je tato část algoritmu naprosto triviální a není potřeba se jí více věnovat.

Druhá část cyklické funkce je nejdůležitější a obsahuje samotný procházecí mechanismus celého algoritmu. Jako první vezmeme první řádek gramatiky a do množiny algoritmu přidáme startovací neterminál (většinou S) a ostatní neterminální prvky na pravé straně pravidla. Je jisté, že všechny tyto prvky jsou dostupné právě ze startovacího symbolu. Nesmíme zapomenout naplnit těmito prvky i pole `prvkyVAlgoritmu`, které jsme deklarovali na začátku algoritmu. Toto pole je právě pro další průchod velice důležité. První prvek tohoto pole (startovací symbol) jsme již prošli. Pokud pole obsahuje i další neterminální prvek, přejdeme na něj a stejně jako u startovacího symbolu vyhledáme tento prvek na levé straně pravidel a pokud ho nalezneme, do pole `prvkyVAlgoritmu` přidáme opět všechny neterminální symboly, na které se můžeme z daného prvku přepsat. Je tedy vidět, že pole `prvkyVAlgoritmu` procházíme od indexu 0 směrem nahoru a každý procházený prvek tohoto pole nám může na konec pole přidat další množinu dostupných prvků a pole nám tímto způsobem roste. Konec průchodu algoritmu nastává tehdy, když projdeme poslední prvek a ten nám nepřidá žádné další dostupné symboly.

Toto je veškerý princip průchodu algoritmem. Kromě toho zapisujeme do polí `terminaly` a `neterminaly` odpovídající prvky, na které jsme při průchodu narazili a staráme se o kontrolní výpisy do okna s algoritmem a do logu. V neposlední řadě pak vytváříme vizualizaci. Ta probíhá pouze zobrazením načtené gramatiky a barevným značením prvků, které prohledáváme, a pravidel, která jsou dostupná.

Poslední třetí částí je v případě tohoto algoritmu vypsání nové gramatiky, která je složena pouze z dostupných symbolů. Gramatika se tvoří tak, že se prochází pole `prvkyVAlgoritmu` a pro každý neterminální symbol v tomto poli je vyhledáno pravidlo gramatiky, ve kterém se daný neterminál nachází na levé straně. Pokud některý z prvků nebude v množině dostupných symbolů, nebude pravidlo s jeho přepisem do finální gramatiky zahrnuto. Výsledná gramatika se poté vypíše do okna pro zobrazení výsledku a patřičný komentář je vypsán i do okna s logem.

## 6.6 Implementace algoritmu pro zjištění prázdnosti jazyka

Druhým algoritmem naší praktické části se stal algoritmus pro zjištění prázdnosti jazyka. Teoretický základ k tomuto algoritmu můžeme nalézt v kapitole 3.4.4. My se zde budeme opět zabývat způsobem implementace našeho řešení.

Samotný zdrojový kód algoritmu se nachází stejně jako v předchozím příkladě ve složce ***algorithms*** v souboru s názvem ***algorithm1.as***. První částí implementace bylo vytvoření proměnných, které budeme v průběhu celého algoritmu potřebovat. Jednalo se o proměnné:

- **var radky:Array** – v tomto dynamickém poli budeme uchovávat kompletní strukturu načtené gramatiky (viz kapitola 6.4.1)
- **var neterminaly:Array** – dynamické pole pro ukládání již prošlých neterminálních symbolů.
- **var prvkyVAlgoritmu:Array** – dynamické pole, do kterého ukládáme všechny dostupné symboly. Pokud na konci algoritmu existují symboly, které v tomto poli nejsou, jedná se o symboly nedostupné, které náš algoritmus vyhledává.
- **var gramatika:grammar** – instance objektu grammar je důležitou částí algoritmu. Bez této instance nebudeme moci využívat funkcí společného rozhraní (viz kapitola 6.4)
- **var dobrePravidlo:Boolean** – pravdivostní hodnota, která určuje, jestli je procházené pravidlo „dobré“.
- **var povoleni:Boolean** – pravdivostní hodnota, které určuje, jestli se v právě procházeném pravidle nachází prvek tvořený pouze z terminálů a „dobrých“ neterminálů. Pokud tomu tak je (hodnota je true), je i neterminální prvek na levé straně pravidla označen jako „dobrý“.
- **var dobreNeterminaly:Boolean** – pravdivostní hodnota, která nám rozlišuje dvojí průchod algoritmem. První průchod hledá pouze pravidla přepisující na terminální prvky (hodnota false), každý další průchod už hledá pravidla, která se dají přepsat nejen na terminální symboly, ale i na „dobré“ neterminální symboly (hodnota true).

- **var pocetOpakovani** – číselná hodnota, která nám počítá počet opakování průchodu gramatikou, způsobené nalezením některých „dobrých“ neterminálů a tím pádem novým průchodem s platností i těchto nových „dobrých“ neterminálů.
- **var vstupOK:Boolean** – pravdivostní hodnota, která indikuje správné načtení vstupní gramatiky. V našem případě je nutné tuto proměnnou vytvořit pro pozdější použití.

Všechny tyto proměnné je poté nutné inicializovat v konstruktoru třídy.

Následuje vytvoření hlavní cyklické funkce celého algoritmu. Tato funkce má stejný tvar jako v případě předchozího algoritmu, ve kterém byla důkladně popsána. Není tedy nutné se zde zabývat popisováním jejího tvaru a rovnou přejdeme k rozdělení průchodu této funkce, tentokrát na 4 části.
První částí je stejně jako u předchozího algoritmu načtení vstupní gramatiky pomocí funkce `otestujANacti()` a výpis informací pomocí funkce `getLoc()`.
Druhá část je samotný průběh algoritmu. Ten postupně prochází každé pravidlo vstupní gramatiky a hledá v něm přepis na prvek složený pouze z terminálních symbolů. Pokud v celém pravidle existuje na pravé straně alespoň jeden takový prvek, označíme neterminální symbol na levé straně jako tzv. „dobrý“ a vložíme ho do pole `prvkyVAlgortimu`. Poté, co projdeme tímto stylem všechna pravidla, pokusíme se zjistit, zda jsme při průchodu nalezli některé „dobré“ neterminální symboly. Toto je řešeno vytvořením kopie pole `prvkyVAlgortimu` před samotným průchodem gramatiky. Po průchodu je porovnána neaktualizovaná kopie a pole `prvkyVAlgortimu`. Pokud jsou proměnné rozdílné, je zřejmé, že byly v aktuálním průchodu nalezeny nějaké „dobré“ neterminální symboly. Pokud tomu tak je, je nutné projít celou gramatiku stejným způsobem znova s rozdílem, že akceptujeme prvky obsahující nejen terminální symboly jako v předchozím případě, ale právě i „dobré“ neterminální symboly, které jsme v předchozím průchodu nalezli a uložili do pole `prvkyVAlgortimu`. Stejným způsobem projdeme opět celou gramatiku a pokud i v tomto průchodu přibudou některé nové „dobré“ neterminály, budeme muset opět projít znova celou gramatiku s platností pro nově nalezené „dobré“ neterminály. Průchod může být ukončen dvojím způsobem. První způsob nastane tehdy, když projdeme pravidla gramatiky a v tomto průchodu nenalezneme žádné nové „dobré“ neterminální symboly. Tento způsob ukončení velice často končí verdiktem,

že jazyk je prázdný. Druhý způsob ukončení nastává v momentě, kdy zjistíme, že všechna pravidla jsou označena jako „dobrá“. Tehdy můžeme s jistotou konstatovat, že jazyk není prázdný. Toto je veškerý princip průchodu algoritmem. Kromě toho zapisujeme do pole `neterminaly` odpovídající prvky, na které jsme při průchodu narazili a staráme se o kontrolní výpisy do okna s algoritmem a do logu. V neposlední řadě pak vytváříme vizualizaci. Ta probíhá pouze zobrazením načtené gramatiky a barevným značením prvků, které prohledáváme a detekcí „dobrých“ symbolů.

Třetí částí cyklické funkce je stav, kdy dochází k testování, jestli byly při průchodu nalezeny nějaké nové „dobré“ neterminální symboly. Pokud ano, resetuje určité proměnné a opakuje průchod znovu.

Poslední čtvrtá část zajišťuje vypsání konečného verdiktu. V tomto případě se jedná pouze o odpověď typu „jazyk je prázdný“ nebo „jazyk není prázdný“. Výsledný verdikt se poté vypíše do okna pro zobrazení výsledku a patřičný komentář je vypsán i do okna s logem.

Samotný algoritmus pro zjištění prázdnosti jazyka je velmi podobný předchozímu jazyku pro zjištění nedostupných symbolů. Proto i principiálně jsou oba algoritmy velice podobné a např. i vizualizace je tvořena stejným způsobem. V následujících kapitolách si představíme implementaci dalších algoritmů, které jsou principiálně zcela rozdílné a i samotná vizualizace je demonstrována jiným stylem.

## 6.7 Implementace algoritmu first

Třetím algoritmem byl pro praktickou část této práce určen algoritmus funkce first. Jak už zde jednou zaznělo, algoritmus funkce first se poměrně hodně liší oproti dvěma předchozím algoritmům. Implementace tedy byla brána jako jakýsi test našeho společného rozhraní, jestli budeme pomocí něj schopni daný problém úspěšně vyřešit. Funkčnost algoritmu funkce first je vysvětlena v kapitole 3.4.9, proto můžeme přejít k samotné implementaci našeho řešení.

Samotný zdrojový kód algoritmu se nachází ve složce ***algorithms*** v souboru s názvem ***algorithm1.as***. První částí implementace bylo vytvoření proměnných, které budeme v průběhu celého algoritmu potřebovat. Jednalo se o proměnné:

- **var radky:Array** – v tomto dynamickém poli budeme uchovávat kompletní strukturu načtené gramatiky (viz kapitola 6.4.1)
- **var prvkyVAlgoritmu:Array** – dynamické pole, které reprezentuje množinu F. Postupně jsou do něj vkládána pravidla, která odpovídají krokům 1b a 1c. Více informací o těchto pravidlech nalezneme v kapitole 3.4.9.
- **var prubezneVysledky:Array** – dynamické pole, do kterého při průchodu algoritmu postupně vkládáme terminální symboly, o kterých v danou chvíli jistě víme, že patří do množiny F funkce first.
- **var pouzitaPravidla:Array** – dynamické pole, do kterého ukládáme posloupnost pravidel, které jsme při průchodu algoritmem použili. Více informací o těchto pravidlech nalezneme v kapitole 3.4.9.
- **var gramatika:grammar** – instance objektu grammar je důležitou částí algoritmu. Bez této instance nebudeme moci využívat funkcí spol. rozhraní (viz kapitola 6.4)
- **var pocetOpakovani** – číselná hodnota, která nám počítá počet opakování průchodu gramatikou, způsobené nalezením některých „dobrých“ neterminálů a tím pádem novým průchodem s platností i těchto nových „dobrých“ neterminálů.
- **var vstupOK:Boolean** – pravdivostní hodnota, která indikuje načtení vstupní gramatiky. V našem případě je nutné tuto proměnnou vytvořit pro pozdější použití.

Všechny tyto proměnné je opět nutné inicializovat v konstruktoru třídy.

Stejně jako u předchozích algoritmů, je i zde základem funkčnosti cyklická funkce dalsi(). Tu musíme opět rozdělit do několika částí a tím určit průběh algoritmu. V tomto případě si vystačíme se třemi částmi funkce.

První částí je již tradičně samotné načtení vstupní gramatiky. To se opět provádí pomocí funkce `otestujANacti()`, ovšem tentokrát s parametrem false, který zakazuje vykreslení načtené gramatiky do vizualizace. Jak jsme již zmínili, vizualizace tohoto algoritmu proběhne zcela jiným systémem. Kromě načtení vstupní gramatiky se zde musíme postarat i o načtení dodatečné funkce, tedy symbolu uvnitř závorek first(), pro který hledáme prvky v množině F. Následují již tradičně výpisy do logu a ostatních informačních prvků pomocí funkcí `getLoc()` a `getLocLog()`.

Druhou částí cyklické funkce je samotné chování algoritmu a jeho průchod. Tato část je již zcela odlišná, než předchozí implementace. V teoretické části jsme si o algoritmu funkce first řekli, že ho lze počítat třemi způsoby. Pro naši implementaci jsme použili druhý způsob, tedy výpočet s tečkou. Vytvoříme tedy množinu F, kterou zpočátku naplníme základním tvarem pravidla přepisovacího symbolu (viz krok 1a v kapitole 3.4.9) a následně budeme postupně vkládat pravidla, které lze na prvky v množině F aplikovat (kroky 1b, 1c, 1d a 2 v kapitole 3.4.9). Jelikož krok 2 znamená pouze zjištění výsledků z množiny F a krok 1d je pouze pravidlo pro opakování předchozích kroků, je zřejmé, že hlavními pravidly budou pouze kroky 1b a 1c, které se budou starat o naplňování množiny F. Pro tyto pravidla jsou v algoritmu vytvořeny samostatné funkce, které nám budou daný problém řešit. Jedná se o funkce:

- **function pravidlo1b(prvkyVAlgoritmu, begin, gramatika:grammar):Array** – funkce, reprezentující použití pravidla pro krok 1b. Nejdříve si projdeme vstupní parametry této funkce. Parametr `prvkyVAlgoritmu` je dynamické pole, které obsahuje pravidla uvnitř množiny F. Hodnota `begin` označuje číslo řádku množiny F, od kterého máme začít prohledávat pravidla platná pro krok 1b. Parametr gramatika je poté pouze poskytnutí instance na společné prostředí algoritmů. Funkce má návratovou hodnotu v podobě dynamického pole. Funkce začne prohledávat jednotlivá pravidla v množině F a snaží se najít prvek ve tvaru `B --> beta . A gama` . Pokud nějaký platný prvek nalezne, zjistí neterminální symbol umístěný za tečkou a uloží do výsledného pole návratové hodnoty. Toto se bude opakovat pro každý řádek v intervalu *begin…n*. Po prohledání celé množiny F

se funkce ukončí a vrátí pole s neterminálními symboly, které se nacházejí za tečkou. Samotná část cyklické funkce pak toto pole vezme a podle symbolů v něm nalezne odpovídající pravidla ve vstupní gramatice a vloží je do množiny F.

- **function pravidlo1c(prvkyVAlgoritmu, begin, gramatika:grammar):Array -** funkce, reprezentující použití pravidla pro krok 1c. Vstupní parametry i návratová hodnota jsou totožné s předchozí funkcí `pravidlo1b()`. Funkce začne prohledávat jednotlivá pravidla v množině F a snaží se najít prvek ve tvaru `B --> delta .` . Pokud je prvek v tomto tvaru nalezen, uloží se do návratového pole prvek z levé strany aktuálního pravidla (v našem případě prvek B). Tento postup se opět opakuje pro všechna pravidla v intervalu *begin…n*. Samotná část cyklické funkce poté vezme vrácené pole a v množině F hledá prvek ve tvaru `. prvek-vraceneho-pole`. Pokud pravidlo s tímto prvkem nalezne, vloží toto pravidlo do množiny F ještě jednou s tvarem prvku `prvek-vraceneho-pole .` .

- **function pravidlo2(prvkyVAlgoritmu):Array -** funkce, reprezentující použití pravidla pro krok 1c. Vstupním parametrem je pouze naplněná množina F. Funkce poté prochází tuto množinu a hledá terminální symboly, které se nacházejí bezprostředně za tečkou. Tyto symboly vloží do pole, které na konci funkce vrátí. Toto pole už obsahuje konečné výsledky funkce first.

Kromě naplňování množiny F se druhá část cyklické funkce stará i o vizualizaci. Ta zobrazuje postupné plnění množiny F. Pokud je v této množině nalezen prvek, na který lze aplikovat krok 1b nebo 1c, bude tento prvek obarven patřičnou barvou, jejíž význam je vysvětlen v logu. Malá vizualizace poté zobrazuje momentální množinu výsledných prvků a posloupnost použitých pravidel.

Poslední část cyklické funkce zajistí vytvoření konečného verdiktu. Známe všechna pravidla v množině F a potřebujeme zjistit prvky odpovídající funkci first. Pro zjištění výsledků použijeme výše zmíněnou funkci `pravidlo2()`, která nám vrátí pole výsledných hodnot. Tyto hodnoty jsou nakonec vypsány do informačního okna pro výsledek a algoritmus je ukončen.

## 6.8 Implementace algoritmu follow

Poslední algoritmus funkce follow je velice příbuzný předchozímu algoritmu funkce first. Průběhy algoritmů jsou totožné a liší se pouze zněním jednotlivých kroků. Funkčnost algoritmu funkce first je vysvětlena v kapitole 3.4.10, proto můžeme přejít k samotné implementaci našeho řešení.

Samotný zdrojový kód algoritmu se nachází ve složce ***algorithms*** v souboru s názvem ***algorithm1.as***. První částí implementace bylo vytvoření proměnných, které budeme v průběhu celého algoritmu potřebovat. Jednalo se o proměnné:

- **var radky:Array** – v tomto dynamickém poli budeme uchovávat kompletní strukturu načtené gramatiky (viz kapitola 6.4.1)
- **var prvkyVAlgoritmu:Array** – dynamické pole, které reprezentuje množinu F. Postupně jsou do něj vkládána pravidla, které odpovídají krokům 1b a 1c. Více informací o těchto pravidlech nalezneme v kapitole 3.4.9.
- **var prubezneVysledky:Array** – dynamické pole, do kterého při průchodu algoritmu postupně vkládáme terminální symboly, o kterých v danou chvíli jistě víme, že patří do množiny F funkce first.
- **var pouzitaPravidla:Array** – dynamické pole, do kterého ukládáme posloupnost pravidel, které jsme při průchodu algoritmem použili. Více informací o těchto pravidlech nalezneme v kapitole 3.4.9.
- **var gramatika:grammar** – instance objektu grammar je důležitou částí algoritmu. Bez této instance nebudeme moci využívat funkcí spol. rozhraní (viz kapitola 6.4)
- **var mnozinaNe:Array** – dynamické pole, které je naplněno neterminálními symboly, které se mohou přepsat na prázdné slovo e. Funkce pro naplnění této množiny probíhá na počátku průběhu algoritmu.
- **var pocetOpakovani** – číselná hodnota, která nám počítá počet opakování průchodu gramatikou, způsobené nalezením některých „dobrých“ neterminálů a tím pádem novým průchodem s platností i těchto nových „dobrých“ neterminálů.
- **var vstupOK:Boolean** – pravdivostní hodnota, která indikuje načtení vstupní gramatiky. V našem případě je nutné tuto proměnnou vytvořit pro pozdější použití.

Všechny tyto proměnné je opět nutné inicializovat v konstruktoru třídy.

Základem funkčnosti algoritmu je cyklická funkce `dalsi()`. Ta se opět dělí do tří částí. První část se kromě načtení vstupní gramatiky pomocí funkce `otestujANacti()` s parametrem false zabývá i zjištěním množiny Ne, která osahuje neterminální symboly, které se mohou přepsat na prázdné slovo e (krok 1 v kapitole 3.4.10). Vizualizace tohoto algoritmu pak probíhá stejně jako u algoritmu funkce first. Kromě načtení vstupní gramatiky se zde musíme postarat i o načtení dodatečné funkce, tedy symbolu uvnitř závorek follow(), pro který hledáme prvky v množině F. Poté následují již tradičně výpisy do logu a ostatních informačních prvků pomocí funkcí `getLoc()` a `getLocLog()`.

Druhou částí cyklické funkce je samotné chování algoritmu a jeho průchod. Tato část funkce follow je totožná s funkcí first. Opět je použit způsob výpočtu s tečkou a opět množina F závisí na množině kroků, pomocí kterých je naplňována. Vytvoříme tedy množinu F, kterou zpočátku naplníme vstupním fiktivním pravidlem (viz krok 2a v kapitole 3.4.10) a následně budeme postupně vkládat pravidla, která lze na prvky v množině F aplikovat (kroky 2b, 2c a 2d, 2e a 3 v kapitole 3.4.10). Jelikož krok 3 znamená pouze zjištění výsledků z množiny F a krok 2e je pouze pravidlo pro opakování předchozích kroků, je zřejmé, že hlavními pravidly budou pouze kroky 2b, 2c a 2d, které se budou starat o naplňování množiny F. Pro tyto pravidla jsou v algoritmu vytvořeny samostatné funkce, které nám budou daný problém řešit. Jedná se o funkce:

- **function pravidlo2b(prvkyVAlgoritmu, begin, gramatika:grammar):Array** - funkce, reprezentující použití pravidla pro krok 2b. Nejdříve si projdeme vstupní parametry této funkce. Parametr `prvkyVAlgoritmu` je dynamické pole, které obsahuje pravidla uvnitř množiny F. Hodnota begin označuje číslo řádku množiny F, od kterého máme začít prohledávat pravidla platná pro krok 2b. Parametr gramatika je poté pouze poskytnutí instance na společné prostředí algoritmů. Návratová hodnota je v podobě dynamického pole. Funkce začne prohledávat jednotlivá pravidla v množině F a snaží se najít prvek ve tvaru `B --> gama .` . Pokud je tento prvek nalezen, do návratového pole je vložen symbol z levé strany daného pravidla (v našem případě symbol B). Tento princip opakujeme pro každý řádek množiny F v intervalu *begin…n*. Pověřená část cyklické funkce vezme vrácené pole a jeho prvky vyhledává na pravé straně pravidel vstupní gramatiky. Pokud je v pravidle tento prvek nalezen, bude pravidlo přidáno do množiny F a hledaný prvek bude přepsán do tvaru `prvek-vraceneho-pole .` .

- **function pravidlo2c(prvkyVAlgoritmu, begin, gramatika:grammar):Array** - funkce, reprezentující použití pravidla pro krok 2c. Vstupní parametry i návratová hodnota jsou totožné s předchozí funkcí `pravidlo2b()`. Funkce začne prohledávat jednotlivá pravidla v množině F a snaží se najít prvek ve tvaru `C --> alfa . B beta` . Pokud je tento prvek nalezen, do návratového pole je vložen neterminální symbol nacházející se bezprostředně za tečkou (v našem případě symbol B). Tento postup je opakován pro všechna pravidla množiny F v intervalu *begin…n*. Příslušná část cyklické funkce prochází prvky vráceného pole a hledá je na levé straně pravidel vstupní gramatiky. Pokud je prvek na některé levé straně nalezen, je pravidlo přidáno do množiny F a všechny prvky na pravé straně tohoto pravidla budou začínat tečkou.

- **function pravidlo2d(prvkyVAlgoritmu:Array, begin, mnozinaNe:Array, gramatika:grammar):Array** – funkce, reprezentující použití pravidla pro krok 2d. Vstupní parametry i návratová hodnota jsou totožné s předchozí funkcí `pravidlo2c()`. Funkce navíc obsahuje další vstupní parametr `mnozinaNe`, který obsahuje pole prvků, patřící do této množiny. Funkce začne prohledávat jednotlivá pravidla v množině F a snaží se najít prvek ve tvaru `C --> alfa . B* beta` , kde `B*` je neterminální symbol patřící do množiny Ne. Pokud je tento prvek nalezen, uloží se do návratového pole symbol nacházející se bezprostředně za tečkou. Příslušná část cyklické funkce vezme prvek vráceného pole, nalezne jeho pozici v pravidle množiny F a toto pravidlo vloží do množiny F znovu s tečkou posunutou o jeden prvek doprava.

- **function pravidlo1(radky, gramatika:grammar):Array** – funkce, která naplní množinu Ne platnými symboly. Vstupním parametrem je načtená gramatika a instance třídy grammar. Funkce prochází vstupní gramatiku a hledá všechny neterminální symboly, které se mohou přepsat na prázdné slovo e. Pokud jsou v jednom průběhu nalezeny nové platné neterminální symboly, je průchod opakován a jsou hledány neterminální symboly, přepisující se také na platné symboly, nalezené v předchozím průběhu. Cyklus se opakuje do doby, než se během jednoho cyklu nenaleznou žádné nové platné neterminální symboly. Funkce nakonec vrátí dynamické pole reprezentující množinu Ne.

- **function pravidlo3(prvkyVAlgoritmu,radky):Array** - funkce, reprezentující použití pravidla pro krok 3. Vstupními parametry jsou pouze naplněná množina F a vstupní gramatika. Funkce dále prochází množinu F a hledá terminální symboly, které se nacházejí bezprostředně za tečkou. Tyto symboly vloží do pole, které na konci funkce vrátí. Pokud je zjištěno, že množina F obsahuje pravidlo, kde se startovací symbol vstupní gramatiky přepisuje na prvek ve tvaru `S --> alfa .`, je do návratového pole vloženo i prázdné slovo e. Toto pole už obsahuje konečné výsledky funkce follow.

Kromě naplňování množiny F se druhá část cyklické funkce stejně jako u algoritmu funkce first stará i o vizualizaci. Ta zobrazuje postupné plnění množiny F. Pokud je v této množině nalezen prvek, na který lze aplikovat kroky 2b, 2c nebo 2d, bude tento prvek obarven patřičnou barvou, jejíž význam je vysvětlen v logu. Malá vizualizace poté zobrazuje momentální množinu výsledných prvků a posloupnost použitých pravidel.
Poslední část cyklické funkce zajistí vytvoření konečného verdiktu. Známe všechna pravidla v množině F a potřebujeme zjistit prvky odpovídající funkci follow. Pro zjištění výsledků použijeme výše zmíněnou funkci `pravidlo3()`, která nám vrátí pole výsledných hodnot. Tyto hodnoty jsou nakonec vypsány do informačního okna pro výsledek a algoritmus je ukončen.

# 7 Testování

Abychom ověřili funkčnost a uživatelskou přívětivost vytvořených algoritmů, nechali jsme je otestovat na pěti nezávislých uživatelech. Test probíhal celkem v šesti bodech:

- Funkčnost algoritmu pro nalezení nedostupných symbolů
- Funkčnost algoritmu pro zjištění prázdnosti jazyka
- Funkčnost algoritmu funkce first
- Funkčnost algoritmu funkce follow
- Uživatelská přívětivost a přehlednost grafického rozraní
- Srozumitelnost daného problému

První čtyři body se týkají funkčnosti samotných algoritmů. Uživatelé zde testují algoritmy a hledají případné chyby. Předposlední test se týká celkové přehlednosti a přívětivosti grafického rozhraní. Hodnocen je design, funkční prvky a intuitivnost prostředí. Jelikož vzhled je u všech algoritmů stejný, proběhl pouze jeden test pro všechny algoritmy obecně. Poslední test se týká srozumitelnosti algoritmů. Jedná se především o správné pochopení problému, který algoritmus popisuje.

Test proběhl formou poskytnutí internetového odkazu se všemi algoritmy a následnému vyplnění výsledného formuláře. Testování je zcela anonymní.

Pro testování bylo vybráno celkem 5 uživatelů, z toho 3 uživatelé jsou studenty informatiky. Zbylé 2 uživatelé nikdy nebyli do problému formálních gramatik zainteresováni a jejich výsledky budou velice důležité především pro poslední test.

**Výsledky testování** (min. hodnocení 0/3, max. hodnocení 3/3)

| otázka/uživatel | 1.uživatel | 2.uživatel | 3.uživatel | 4.uživatel | 5.uživatel |
|---|---|---|---|---|---|
| **Funkčnost 1. algoritmu** | 3/3 | 3/3 | 3/3 | 3/3 | 3/3 |
| **Funkčnost 2. algoritmu** | 3/3 | 3/3 | 3/3 | 3/3 | 3/3 |
| **Funkčnost 3. algoritmu** | 3/3 | 3/3 | 3/3 | 3/3 | 3/3 |
| **Funkčnost 4. algoritmu** | 3/3 | 3/3 | 3/3 | 3/3 | 3/3 |
| **Přehlednost a design** | 2/3 | 2/3 | 3/3 | 2/3 | 3/3 |
| **Pochopení problému** | 3/3 | 3/3 | 2/3 | 1/3 | 2/3 |
| **Celkový dojem** | 3/3 | 3/3 | 2/3 | 3/3 | 3/3 |

Vyplněné formuláře jsou zveřejněny v přílohách této práce

# 8 Závěr

Již v prvním ročníku navazujícího studia fakulty aplikovaných věd jsem poznal hlubší problematiku formálních jazyků a překladačů díky stejnojmennému předmětu. Tato problematika mě poměrně zaujala a předmět KIV/FJP patřil mezi mé oblíbené. Proto ihned, co jsem v seznamu témat diplomových prací viděl tuto možnost, neváhal jsem a ihned si téma rezervoval. Po následné schůzce s Richardem Lipkou, který mě seznámil s podrobnostmi práce, jsem si byl svou volbou naprosto jistý.

Na samotném tématu diplomové práce se mi líbila především volnost výběru algoritmů a technologie pro zpracování. Nic nebylo striktně zadáno a já měl možnost si rozšířit programátorské obzory o další programovací technologii. Adobe Flash byl pro mě od počátku studia neznámou technologií, do které jsem neviděl. Bylo tedy potřeba prostudovat několik publikací a manuálů ([Z13]), abych získal povědomí o možnostech tohoto nástroje. Jistě by bylo snazší použít např. Javascript, jehož základy mi byly již známé, ale touha po poznání něčeho nového byla o něco silnější. Dalším důvodem pro volbu této práce byla vidina toho, že výsledné aplikace budou pomáhat budoucím studentům, které čeká stejný boj, jako mě do této doby.

V průběhu vytváření implementace jsem nenarazil na žádný větší problém. Hlavní snahou celé implementace bylo vytvořit jednotné prostředí, které by bylo stejné pro všechny algoritmy. Jedná se o prostředí, které se postará o načtení vstupních dat a potenciálnímu programátorovi nových algoritmů poskytne pohodlné funkce pro snazší vývoj. Dále bylo snahou vytvořit externí lokalizaci všech textů uvnitř algoritmů. Veškeré tyto snahy se nám podařilo bez problémů a v celém rozsahu realizovat. Díky externí lokalizaci není problém vytvořit více jazykových verzí pro použití algoritmu, aniž by někdo musel zasahovat do programového kódu. Společné API se nám také osvědčilo při návrhu nových algoritmů. Samotný vzhled i funkčnost řešení jsme se pokusili přizpůsobit na míru přímo studentům a naše řešení pečlivě otestovali na samotných studentech i nezainteresovaných lidech, jejichž hodnocení bylo více než kladné. Původní záměr bylo vytvořit řešení pro 3 algoritmy. Vzhledem k časové rezervě byl zcela dobrovolně vytvořen ještě jeden algoritmus mimo rozsah práce. Výsledkem této práce je tedy implementace 4 algoritmů v podobě webové aplikace s hlavním využitím na hodinách předmětu KIV/FJP pro samotné studenty. Všechny body zadání byly splněny včetně dobrovolných funkcionalit nad rámec zadání.

Komunikace se zadavatelem probíhala bez problému především díky oboustranné ochotě. Každá část zpracování byla se zadavatelem vždy důkladně probrána a každá realizovaná funkce podrobně otestována. Programový text jsem se snažil dostatečně okomentovat pro snazší orientaci potenciálních programátorů, kteří budou mít za úkol pokračovat v implementaci dalších algoritmů. Zdrojové kódy společného rozhraní a všech algoritmů jsou zveřejněny v příloze této práce, stejně jako ukázky externích souborů pro lokalizaci a pro generování gramatik. Nad rámec zadání byla vytvořena internetová stránka, obsahující zmíněnou čtveřici algoritmů. V příštím semestru (ZS 2012) je plánováno zveřejnění na oficiálních stránkách předmětu KIV/FJP na Coursewaru a tím pádem je uděleno i povolení pro používání implementovaných algoritmů po dobu neurčitou.

# 9 Literatura a zdroje


- [Z1] - BI-PJP – Programovací jazyky a překladače [online]. Dostupné z <http://fit.cvut.cz/predmety/bi-pjp>.
- [Z2] - Kocur, Pavel. Úvod do teorie konečných automatů [online]. 2000. Dostupné z <http://www.kvd.zcu.cz/cz/materialy/kafj/_kafj1.html>.
- [Z3] - Mička, Pavel. Algoritmus – Algoritmy.net [online]. Dostupné z <http://algoritmy.net>.
- [Z4] - Lipka, Richard. Výuka FJP [online]. 2010. Dostupné z <http://home.zcu.cz/~lipka/fjpcvic.php?cviceni=9>
- [Z5] – ČVUT. Algoritmus funkce First [online]. Dostupné z <http://moon.felk.cvut.cz/~pjv/Jak/_info/i531/FirstApplet.html>
- [Z6] – ČVUT. Algoritmus funkce Follow [online]. Dostupné z <http://moon.felk.cvut.cz/~pjv/Jak/_info/i531/FollowApplet.html>
- [Z7] – SeeYK – A Visualisation of the CYK Algorithm [online]. Dostupné z <http://www2.tcs.ifi.lmu.de/SeeYK/>.
- [Z8] – JFLAP [online]. Dostupné z <http://www.jflap.org/>.
- [Z9] – Markus, Holzer. Volume 1547/1998. Lecture Notes in Computer Science [online]. 450-451, DOI: 10.1007/3-540-37623-2_41. Dostupné z <http://www.springerlink.com/content/7k5p2lwlfvp6e94e/>.
- [Z10] – QFSM [online]. Dostupné z <http://qfsm.sourceforge.net/about.html>.
- [Z11] – ANTLR Parser Generator [online]. Dostupné z <http://www.antlr.org/>.
- [Z12] – AutomataTools [online]. Dostupné z <http://members.fortunecity.com/boroday/Automatatools.html>.
- [Z13] – František, Sabovčik. Seriál ActionScript [online]. Dostupné z <http://programujte.com/clanky/36-serial-actionscript>

- [L1] - Chomsky, Noam. Three Models for the Description of Language. *IRE Transactions on Information Theory.* s. 113–124.
- [L2] - Wróblewski, Piotr. Algoritmy : Datové struktury a programovací techniky. Brno: Computer press, 2004. 351 s. ISBN 80-251-0343-9.
- [L3] - Doc. RNDr. Češka Milan, Doc. Ing. Rábová Zdena. Gramatiky a jazyky. Brno: 1992. 141 s.
- [L4] - Kerman, Phillip. ActionScript ve Flashi Podrobná příručka. Brno: Computer press. 528 s.
- [L5] - Yank, Kevin. Začínáme s JavaScriptem. Praha: Zoner press, 2008. 333 s. ISBN 978-80-86815-94-7
- [L6] - Levine, Robert – Fraser, Simon. Formal Grammar: Theory and Implementation. Oxford University Press, 1992. 439 s.

# 10 Přílohy

## I. Hodnocení uživatelů (formuláře)

**Formulář pro testování algoritmů**

**Datum**...11. 5. 2012......  **Jméno**(nepovinné)...........................

Správnou odpověď zřetelně zakřížkujte. Pro každou otázku může být zvolena vždy maximálně jedna odpověď.

---

**1) Jak hodnotíte správnost a funkčnost algoritmu pro nalezení nedostupných symbolů?**

- [x] a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**2) Jak hodnotíte správnost a funkčnost algoritmu pro zjištění prázdnosti jazyka?**

- [x] a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**3) Jak hodnotíte správnost a funkčnost algoritmu funkce first?**

- [x] a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**4) Jak hodnotíte správnost a funkčnost algoritmu funkce follow?**

- [x] a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**5) Jak hodnotíte grafické rozhraní algoritmů?**

- a) Design algoritmů i rozložení grafických prvků hodnotím výborně. Vše bylo zcela přehledné a intuitivní.
- [x] b) Design algoritmů je povedený, ale chvilku mi trvalo zvyknout si na uspořádání jednotlivých prvků.
- c) Vzhled se mi příliš nezamlouvá, ale rozložení prvků je intuitivní.
- d) Algoritmy jsou velice nepřehledné a uspořádání prvků je nelogické.

**6) Porozuměli jste pouze s pomocí tohoto nástroje problému, který algoritmus řeší?**

a) Pochopení dané problematiky mi nedělalo problém ani u jednoho z výše zmíněných algoritmů.
b) Pochopil jsem problém tří ze čtyřech algoritmů.
c) Polovinu algoritmů jsem pochopil, zbytek mi nedával smysl.
d) Pochopil jsem pouze jeden nebo žádný algoritmus.

**7) Jak hodnotíte práci jako celek?**

a) Práce je velice vydařená. Za celou dobu testování jsem nenarazil k ničemu, co bych vytknul.
b) Práci hodnotím velice kladně. Během testování jsem narazil na pár drobných nedostatků, které ovšem nejsou nijak zásadní.
c) Práce je podle mého názoru průměrná. Některé věci se mi zamlouvali, jiné mi nedávali žádný smysl. Práce by potřebovala rozhodně ještě doladit.
d) Práci hodnotím jako nepřínosnou. Téměř se vším v této práci jsem značně nespokojen.

**8) Jaké je Vaše povolání?**

a) jsem student technické fakulty
b) jsem student humanitní fakulty
c) pracuji v IT sféře
d) pracuji v jiném odvětví
e) jiné ..........................................................................................

## Formulář pro testování algoritmů

Datum..11.5.................... Jméno(nepovinné)..........................

Správnou odpověď zřetelně zakřížkujte. Pro každou otázku může být zvolena vždy maximálně jedna odpověď.

---

**1) Jak hodnotíte správnost a funkčnost algoritmu pro nalezení nedostupných symbolů?**

a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
b) Algoritmus se choval správně až na pár drobných výjimek.
c) Algoritmus vykazoval větší množství chyb a nepřesností.
d) Nepodařilo se mi algoritmus vůbec spustit.

**2) Jak hodnotíte správnost a funkčnost algoritmu pro zjištění prázdnosti jazyka?**

a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
b) Algoritmus se choval správně až na pár drobných výjimek.
c) Algoritmus vykazoval větší množství chyb a nepřesností.
d) Nepodařilo se mi algoritmus vůbec spustit.

**3) Jak hodnotíte správnost a funkčnost algoritmu funkce first?**

a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
b) Algoritmus se choval správně až na pár drobných výjimek.
c) Algoritmus vykazoval větší množství chyb a nepřesností.
d) Nepodařilo se mi algoritmus vůbec spustit.

**4) Jak hodnotíte správnost a funkčnost algoritmu funkce follow?**

a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
b) Algoritmus se choval správně až na pár drobných výjimek.
c) Algoritmus vykazoval větší množství chyb a nepřesností.
d) Nepodařilo se mi algoritmus vůbec spustit.

**5) Jak hodnotíte grafické rozhraní algoritmů?**

a) Design algoritmů i rozložení grafických prvků hodnotím výborně. Vše bylo zcela přehledné a intuitivní.
b) Design algoritmů je povedený, ale chvilku mi trvalo zvyknout si na uspořádání jednotlivých prvků.
c) Vzhled se mi příliš nezamlouvá, ale rozložení prvků je intuitivní.
d) Algoritmy jsou velice nepřehledné a uspořádání prvků je nelogické.

**6) Porozuměli jste pouze s pomocí tohoto nástroje problému, který algoritmus řeší?**

a) Pochopení dané problematiky mi nedělalo problém ani u jednoho z výše zmíněných algoritmů.
b) Pochopil jsem problém tří ze čtyřech algoritmů.
c) Polovinu algoritmů jsem pochopil, zbytek mi nedával smysl.
d) Pochopil jsem pouze jeden nebo žádný algoritmus.

**7) Jak hodnotíte práci jako celek?**

a) Práce je velice vydařená. Za celou dobu testování jsem nenarazil k ničemu, co bych vytknul.
b) Práci hodnotím velice kladně. Během testování jsem narazil na pár drobných nedostatků, které ovšem nejsou nijak zásadní.
c) Práce je podle mého názoru průměrná. Některé věci se mi zamlouvali, jiné mi nedávali žádný smysl. Práce by potřebovala rozhodně ještě doladit.
d) Práci hodnotím jako nepřínosnou. Téměř se vším v této práci jsem značně nespokojen.

**8) Jaké je Vaše povolání?**

a) jsem student technické fakulty
b) jsem student humanitní fakulty
c) pracuji v IT sféře
d) pracuji v jiném odvětví
e) jiné ..........................................................................................

## Formulář pro testování algoritmů

**Datum**..11.5.................... **Jméno**(nepovinné)...—......................

Správnou odpověď zřetelně zakřížkujte. Pro každou otázku může být zvolena vždy maximálně jedna odpověď.

---

**1) Jak hodnotíte správnost a funkčnost algoritmu pro nalezení nedostupných symbolů?**

- a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**2) Jak hodnotíte správnost a funkčnost algoritmu pro zjištění prázdnosti jazyka?**

- a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**3) Jak hodnotíte správnost a funkčnost algoritmu funkce first?**

- a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**4) Jak hodnotíte správnost a funkčnost algoritmu funkce follow?**

- a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**5) Jak hodnotíte grafické rozhraní algoritmů?**

- a) Design algoritmů i rozložení grafických prvků hodnotím výborně. Vše bylo zcela přehledné a intuitivní.
- b) Design algoritmů je povedený, ale chvilku mi trvalo zvyknout si na uspořádání jednotlivých prvků.
- c) Vzhled se mi příliš nezamlouvá, ale rozložení prvků je intuitivní.
- d) Algoritmy jsou velice nepřehledné a uspořádání prvků je nelogické.

**6) Porozuměli jste pouze s pomocí tohoto nástroje problému, který algoritmus řeší?**

a) Pochopení dané problematiky mi nedělalo problém ani u jednoho z výše zmíněných algoritmů.
b) Pochopil jsem problém tří ze čtyřech algoritmů.
c) Polovinu algoritmů jsem pochopil, zbytek mi nedával smysl.
d) Pochopil jsem pouze jeden nebo žádný algoritmus.

**7) Jak hodnotíte práci jako celek?**

a) Práce je velice vydařená. Za celou dobu testování jsem nenarazil k ničemu, co bych vytknul.
b) Práci hodnotím velice kladně. Během testování jsem narazil na pár drobných nedostatků, které ovšem nejsou nijak zásadní.
c) Práce je podle mého názoru průměrná. Některé věci se mi zamlouvali, jiné mi nedávali žádný smysl. Práce by potřebovala rozhodně ještě doladit.
d) Práci hodnotím jako nepřínosnou. Téměř se vším v této práci jsem značně nespokojen.

**8) Jaké je Vaše povolání?**

a) jsem student technické fakulty
b) jsem student humanitní fakulty
c) pracuji v IT sféře
d) pracuji v jiném odvětví
e) jiné ...............................................................................................

## Formulář pro testování algoritmů

**Datum**..13.5.2012..................                **Jméno**(nepovinné)........—.................

Správnou odpověď zřetelně zakřížkujte. Pro každou otázku může být zvolena vždy maximálně jedna odpověď.

---

**1) Jak hodnotíte správnost a funkčnost algoritmu pro nalezení nedostupných symbolů?**

- [x] a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**2) Jak hodnotíte správnost a funkčnost algoritmu pro zjištění prázdnosti jazyka?**

- [x] a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**3) Jak hodnotíte správnost a funkčnost algoritmu funkce first?**

- [x] a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**4) Jak hodnotíte správnost a funkčnost algoritmu funkce follow?**

- [x] a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**5) Jak hodnotíte grafické rozhraní algoritmů?**

- [x] a) Design algoritmů i rozložení grafických prvků hodnotím výborně. Vše bylo zcela přehledné a intuitivní.
- b) Design algoritmů je povedený, ale chvilku mi trvalo zvyknout si na uspořádání jednotlivých prvků.
- c) Vzhled se mi příliš nezamlouvá, ale rozložení prvků je intuitivní.
- d) Algoritmy jsou velice nepřehledné a uspořádání prvků je nelogické.

**6) Porozuměli jste pouze s pomocí tohoto nástroje problému, který algoritmus řeší?**

a) Pochopení dané problematiky mi nedělalo problém ani u jednoho z výše zmíněných algoritmů.
b) Pochopil jsem problém tří ze čtyřech algoritmů.
c) Polovinu algoritmů jsem pochopil, zbytek mi nedával smysl.
d) Pochopil jsem pouze jeden nebo žádný algoritmus.

**7) Jak hodnotíte práci jako celek?**

a) Práce je velice vydařená. Za celou dobu testování jsem nenarazil k ničemu, co bych vytknul.
b) Práci hodnotím velice kladně. Během testování jsem narazil na pár drobných nedostatků, které ovšem nejsou nijak zásadní.
c) Práce je podle mého názoru průměrná. Některé věci se mi zamlouvali, jiné mi nedávali žádný smysl. Práce by potřebovala rozhodně ještě doladit.
d) Práci hodnotím jako nepřínosnou. Téměř se vším v této práci jsem značně nespokojen.

**8) Jaké je Vaše povolání?**

a) jsem student technické fakulty
b) jsem student humanitní fakulty
c) pracuji v IT sféře
d) pracuji v jiném odvětví
e) jiné ......................................................................................

## Formulář pro testování algoritmů

Datum..13. 5. 2012..................  Jméno(nepovinné)..............................

Správnou odpověď zřetelně zakřížkujte. Pro každou otázku může být zvolena vždy maximálně jedna odpověď.

---

**1) Jak hodnotíte správnost a funkčnost algoritmu pro nalezení nedostupných symbolů?**

- a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**2) Jak hodnotíte správnost a funkčnost algoritmu pro zjištění prázdnosti jazyka?**

- a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**3) Jak hodnotíte správnost a funkčnost algoritmu funkce first?**

- a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**4) Jak hodnotíte správnost a funkčnost algoritmu funkce follow?**

- a) Během testování jsem neobjevil žádnou chybu. Algoritmus byl zcela funkční.
- b) Algoritmus se choval správně až na pár drobných výjimek.
- c) Algoritmus vykazoval větší množství chyb a nepřesností.
- d) Nepodařilo se mi algoritmus vůbec spustit.

**5) Jak hodnotíte grafické rozhraní algoritmů?**

- a) Design algoritmů i rozložení grafických prvků hodnotím výborně. Vše bylo zcela přehledné a intuitivní.
- b) Design algoritmů je povedený, ale chvilku mi trvalo zvyknout si na uspořádání jednotlivých prvků.
- c) Vzhled se mi příliš nezamlouvá, ale rozložení prvků je intuitivní.
- d) Algoritmy jsou velice nepřehledné a uspořádání prvků je nelogické.

**6) Porozuměli jste pouze s pomocí tohoto nástroje problému, který algoritmus řeší?**

a) Pochopení dané problematiky mi nedělalo problém ani u jednoho z výše zmíněných algoritmů.
b) Pochopil jsem problém tří ze čtyřech algoritmů.
c) Polovinu algoritmů jsem pochopil, zbytek mi nedával smysl.
d) Pochopil jsem pouze jeden nebo žádný algoritmus.

**7) Jak hodnotíte práci jako celek?**

a) Práce je velice vydařená. Za celou dobu testování jsem nenarazil k ničemu, co bych vytknul.
b) Práci hodnotím velice kladně. Během testování jsem narazil na pár drobných nedostatků, které ovšem nejsou nijak zásadní.
c) Práce je podle mého názoru průměrná. Některé věci se mi zamlouvali, jiné mi nedávali žádný smysl. Práce by potřebovala rozhodně ještě doladit.
d) Práci hodnotím jako nepřínosnou. Téměř se vším v této práci jsem značně nespokojen.

**8) Jaké je Vaše povolání?**

a) jsem student technické fakulty
b) jsem student humanitní fakulty
c) pracuji v IT sféře
d) pracuji v jiném odvětví
e) jiné ..........................................................................................

# II. Uživatelský manuál pro rozhraní algoritmů

Tato část slouží jako průvodce pro uživatele vytvořených algoritmů. Jelikož je grafické a funkční rozhraní společné pro všechny algoritmy, bude stačit obecný popis použití.

**Systémové požadavky**

Ke spuštění algoritmu je potřeba mít na PC nainstalován Adobe Flash Player (nejlépe poslední verzi). Algoritmus lze pak přímo spustit ve Flash Player okně nebo pomocí webového prohlížeče s podporou Adobe Flash. Pokud tento doplněk není nainstalován nebo nepracuje správně, algoritmus se nespustí nebo se místo něj objeví chybové hlášení.

Pro spuštění aplikace na smartphonu nebo tabletu je potřeba mít zařízení s novějšími verzemi OS Android, Windows Mobile, Windows Phone a RIM (Blackberry). Vzhledem k ignorování technologie Adobe Flash společností Apple nejsou tyto algoritmy dostupné pro zařízení Iphone, Ipad a Ipod.

**Externí zdroje**

Pokud splňujete požadavky, klazené v předchozím odstavci, objeví se Vám po spuštění samotný layout aplikace. Aplikace načítá veškeré texty z externího datového souboru. Pokud se Vám texty v algoritmu nezobrazují nebo nejsou korektní, znamená to, že datový soubor je poškozen nebo že aplikace nemá k tomuto souboru přístup (soubor není ve složce s SWF souborem nebo neexistuje).

Nepokoušejte se stahovat samotný SWF soubor do počítače pro offline použití (pro funkčnost je potřeba i několik dalších datových souborů), aplikace bude nefunkční.

**Vstupní gramatika**

Pro vyzkoušení aplikace máte celkem 2 možnosti.

**První možnost** je, že zadáte do pole „Vaše zadání“ vlastní vstupní gramatiku, na kterou chcete aplikovat algoritmus. Vstupní gramatika musí splňovat předem daný formát:

- Použití pouze znaků [a..z], [A...Z], [0..9], =, >, |
- Levá strana pravidla musí obsahovat pouze neterminální symbol [A..Z]
- Každé pravidlo musí být ukončeno středníkem (;)
- Tvar přepisovacího symbolu je (=>)
- Tvar symbolu „nebo“ je (|)

Výskyt bílých znaků je ve vstupní gramatice povolen. Parser tyto znaky po načtení odstraní. Pokud udělá uživatel při zadání gramatiky chybu, aplikace se pokusí tuto chybu najít a informuje o ní do logu.

**Druhou možností** je vygenerování vstupní gramatiky pomocí tlačítka „Generuj“. Generované gramatiky jsou čerpány z externího souboru a jejich tvar je pouze na poskytovateli aplikace. Předpokládá se, že tyto předlohy budou mít správný tvar a ukázkový příklad pro použití na daný algoritmus. Pokud generování gramatik není korektní, chyba je na straně poskytovatele.

**Načtení gramatiky**

Pokud má uživatel gramatiku správně vepsanou do pole „Vaše zadání“, stiskne tlačítko „Načti gramatiku“ pro její načtení do aplikace. Jak již bylo zmíněno výše, pokud obsahuje gramatika syntaktické chyby, bude o tom uživatel informován v logu.

**Průběh algoritmu**

Po načtení korektní gramatiky začíná průběh algoritmu. Každý algoritmus zpravidla provádí jiné operace. Každý krok algoritmu je důkladně popsán v textovém výpisu, který pomocí barevných označení spolupracuje s velkou vizualizací. Tlačítko „Další krok“ bude provádět kroky algoritmu do doby, kdy to bude možné. Na konci průběhu se toto tlačítko změní na „Vygeneruj gramatiku“. Po stisknutí tohoto tlačítka bude do okna „Výsledek“ vypsán konečný verdikt algoritmu.

Pokud po vygenerování výsledku budu chtít aplikaci znova použít, je nutné stisknout tlačítko „Vynuluj“. Toto tlačítko lze stisknout i kdykoli v průběhu algoritmu pro jeho přerušení a opětovnou přípravu pro zadání vstupní gramatiky.

**Nápověda**

Každý algoritmus má v pravém horním rohu tlačítko s nápovědou. Po jeho stisknutí se objeví okno, ve kterém si můžete přečíst nápovědu k funkčnosti aplikace. Text této nápovědy opět závisí na poskytovateli aplikace, jelikož je čerpán z externího souboru.

# III. Struktura algoritmů

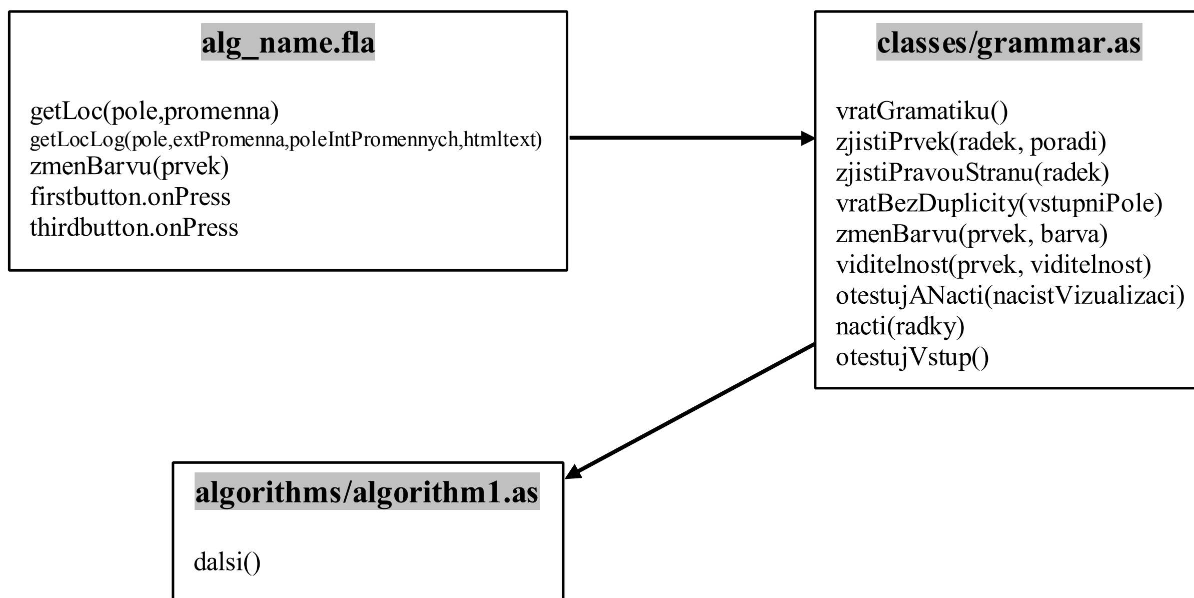
Každý z algoritmů má stejnou strukturu:

```
algorithms        folder
classes           folder
loc               text file
template.txt      text file
alg_name.fla      fla file
alg_name.swf      swf file
```

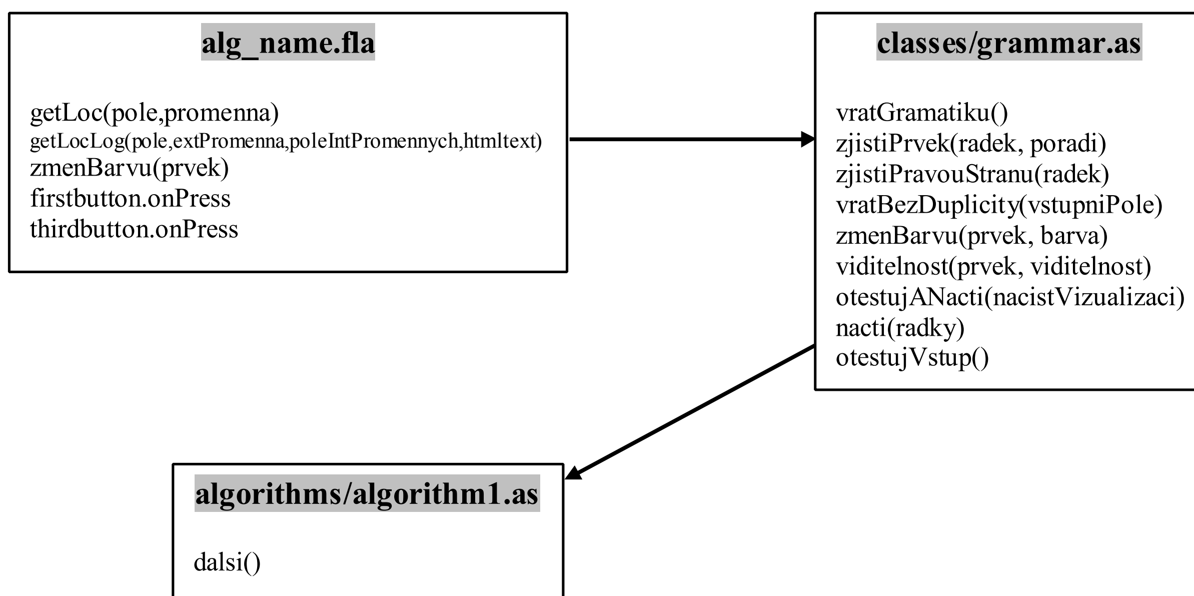
- **algorithms** – složka s třídou příslušného algoritmu. Tento soubor musí splňovat název algorithm1.as. Pokud bude název odlišný, aplikace s tímto souborem nebude pracovat. Více informací o souboru algorithm1.as v praktické části této práce.
- **classes** – složka s třídou popisující práci s gramatikou . Tento soubor se musí nazývat grammar.as. Pokud bude název odlišný, aplikace s tímto souborem nebude pracovat. Jedná se o hlavní část společného uživatelského rozhraní, které poskytuje důležité funkce algoritmům.
- **loc** – textový soubor s veškerými texty, které se objevují v algoritmu.
- **template.txt** – textový soubor s definovanými gramatikami. Tyto gramatiky je možné načítat do vstupního pole algoritmu pomocí tlačítka „Generuj".
- **alg_name.fla** – v tomto souboru se nachází celá grafická část aplikace včetně úprav všech barev algoritmu. Tento soubor lze otevřít např. pomocí nástroje Adobe Flash Professional. Kromě grafické části poskytuje i některé funkce třídě grammar.as.
- **alg_name.swf** – soubor s algoritmem, spustitelný ve Flash playeru.

## IV. UML diagram

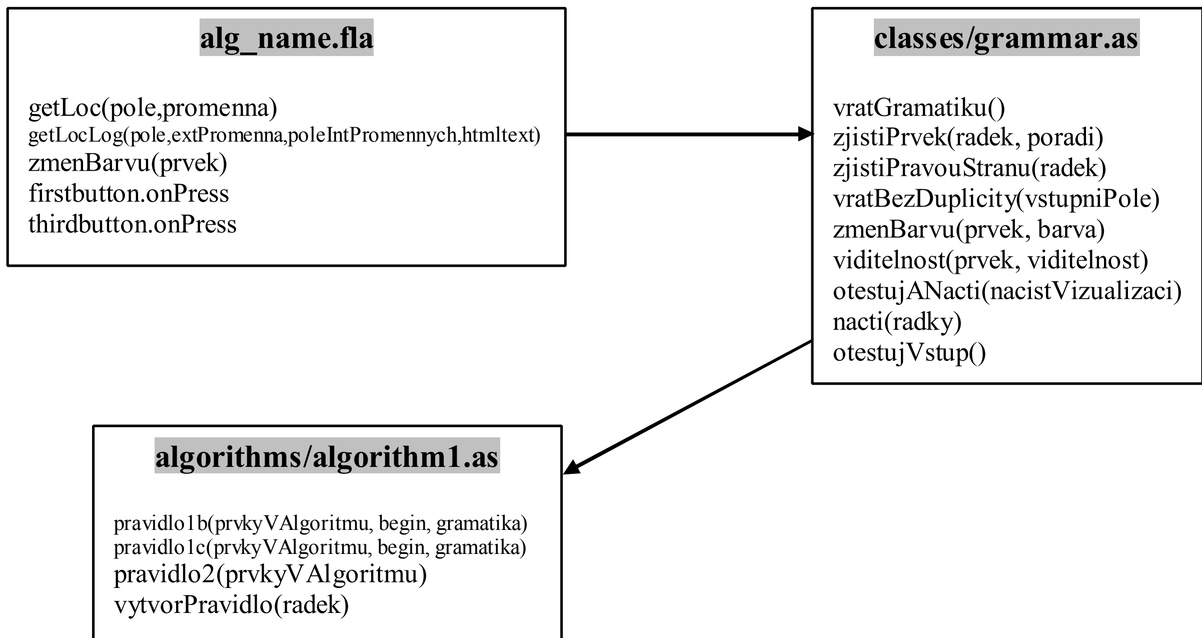
### Algoritmus pro zjištění nedostupných symbolů



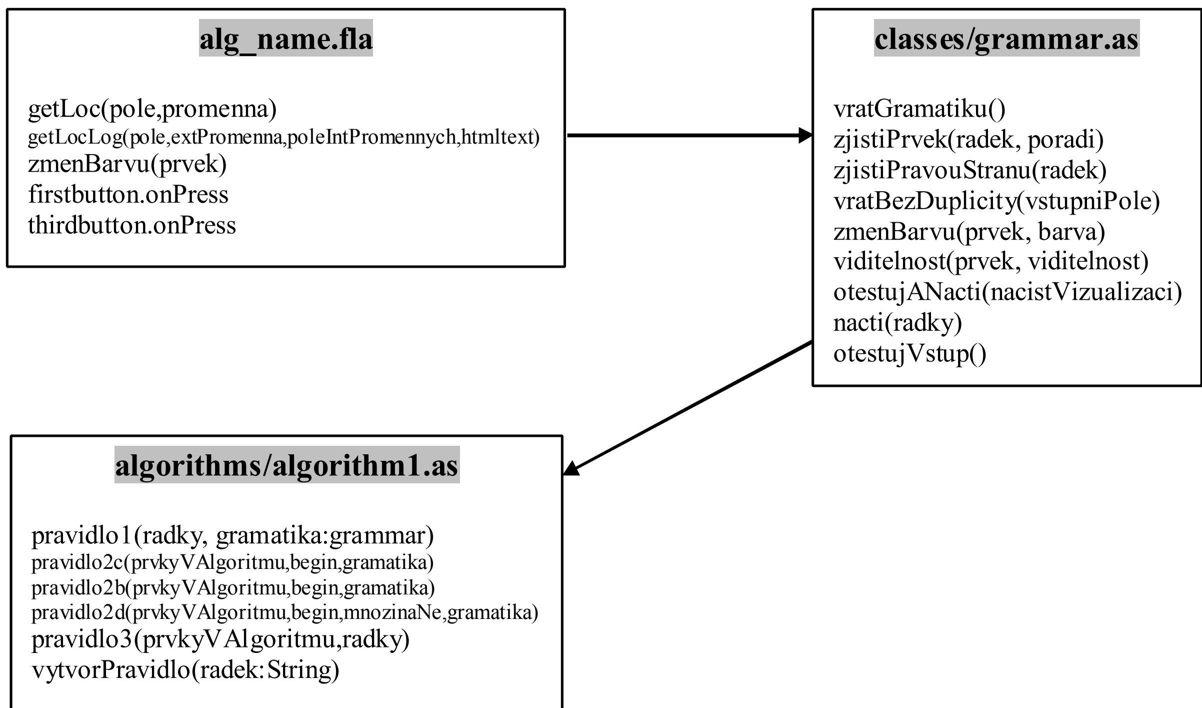

### Algoritmus pro zjištění prázdnosti jazyka



### Algoritmus funkce first




### Algoritmus funkce follow



# V. Ukázka lokalizačního souboru

```
&empty=
&title=Algoritmus pro funkci follow
&subtitle=Vytvořeno v rámci diplomové práce výhradně pro výuku předmětu KIV/FJP na ZČU
&input=Vaše zadání
&rules=Výpis pravidel
&algorithm=Výpis algoritmu
&result=Výsledek
&menu=Menu
&log=Log
&help=Nápověda
&helpTitle=Nápověda
&function=follow
&firstSmallVis=Průběžný výsledek
&secondSmallVis=Použitá pravidla
&nextStep=Další krok
&loadGrammar=Načti gramatiku
&getGrammar=Vygenerovat gramatiku
&firstButton=Generuj
&thirdButton=Vynuluj
&visualisationWarningClose=Odstranit tuto hlášku
&noFollow=Nezadali jste do políčka follow žádnou hodnotu. Stiskněte reset a napravte to.
&visualisationWarning=V této vizualizaci budete moci postupně sledovat algoritmus
naplňování množiny F. Pokud by množina F obsahovala více než 10 pravidel, bude vizualizace
deaktivována.
&grammarFailed=Počet řádků množiny F přesáhl limit 10. Vizualizace je nyní deaktivována.
Pro kontrolu použijte textovou podobu algoritmu v poli "Výpis algoritmu".
&logGrammarSuccessLoad=-Gramatika úspěšně načtena.
&logRule1a=-Aplikujeme pravidlo 1a. Do množiny F přidáme vstupní pravidlo # => #.
&logStep=-Procházíme #. řádek množiny F.
&logNoRule=-Pro tento řádek nelze aplikovat žádné pravidlo.
&logRule1Ok=-Jako první operaci jsme provedli zjištění množiny prvků, které mohou končit
prázdným slovem e (i rekurzivně). Množina "Ne" tedy obsahuje prvky #.
&logRule2cOk=-V tomto řádku byla zjištěna platnost pravidla 2c. Toto pravidlo lze aplikovat
na všechny neterminály, před kterými se nachází tečka (konkrétně prvky # , ve vizualizaci
označené žlutou barvou. Pokud tento prvek ve vizualizaci není vidět, je pravděpodobně
překreslen barvou jiného pravidla, aplikovaného ihned po tomto). Vezmeme tyto prvky a do
množiny F přidáme ze vstupní gramatiky všechna pravidla, jejíž levá strana obsahuje některý
z těchto prvků.
&logRule2cKo=-V tomto řádku sice byla zjištěna platnost pravidla 2c (ve vizualizaci prvky
označené žlutou barvou), ale přidaná pravidla se v množině F již vyskytují. Proto
neprovádíme žádnou akci.
&logRule2bOk=-V tomto řádku se vyskytuje tečka na konci některého prvku za neprázdným
řetězcem (ve vizualizaci označeného zelenou barvou. Pokud tento prvek ve vizualizaci není
vidět, je pravděpodobně překreslen barvou jiného pravidla, aplikovaného ihned po tomto).
Můžeme tedy uplatnit pravidlo 2b. Vezmeme levou stranu tohoto řádku (prvek #) a vyhledáváme
ji na pravé straně v množině pravidel vstupní gramatiky. Tyto pravidla přidáme do množiny F
a tečku vložíme za prvek #.
```

&logRule2bKo=-V tomto řádku sice byla zjištěna platnost pravidla 2b (ve vizualizaci prvky označené zelenou barvou), ale přidaná pravidla se v množině F již vyskytují. Proto neprovádíme žádnou akci.
&logRule2bNotFound=-V tomto řádku sice byla zjištěna platnost pravidla 2b (ve vizualizaci prvky označené zelenou barvou), ale levá strana tohoto pravidla (prvek #) nebyla na pravé straně vstupní gramatiky nikde nalezena. Proto neprovádíme žádnou akci.
&logRule2dOk=-V tomto řádku byla zjištěna platnost pravidla 2d. Toto pravidlo lze aplikovat na všechny neterminály z množiny "Ne", před kterými se nachází tečka (konkrétně prvky # , ve vizualizaci označené červenou barvou). Vezmeme celý tento řádek a do množiny F ho přidáme ještě jednou, tentokrát s tečkou za prvkem #.
&logRule2dKo=-V tomto řádku sice byla zjištěna platnost pravidla 2d (ve vizualizaci prvky označené červenou barvou), ale přidaná pravidla se v množině F již vyskytují. Proto neprovádíme žádnou akci.
&logFinalResults=-Tímto krokem jsme prošli všechny řádky množiny F. Výsledkem funkce first() jsou všechny terminály bezprostředně za tečkou. Výsledek můžete vidět v okně "Výsledek".
&finalResultOk=Výsledky funkce follow(#) jsou: #.
&finalResultKo=Vámi zadaná gramatika nebo prvek, pro který je hledána funkce follow(), není korektní. Výsledkem není žádný symbol.
&grammarError=<p>Zadaná gramatika není korektní. Vložte Vaší gramatiku do pole "Vaše zadání" a dbejte na správnou syntaxi. Dodržujte symboly pro: </p><br><p>přechod : "=>"</p><p>nebo : "|"</p><p>konec pravidla : ";"</p><br>
&grammarErrorMessage=Špatně zadaná gramatika. Více informací v logu.
&grammarError0=Chyba 0, řádek # : chybí středník ";" na konci celé gramatiky.
&grammarError1=Chyba 1, řádek # : Absence nebo špatná syntaxe přepisovacího symbolu "=>". Chybu může způsobit také absence středníku na konci některého pravidla a tím způsobit dvojitý výskyt přepisovacího symbolu "=>" v jednom pravidle.
&grammarError2=Chyba 2, řádek # : V gramatice se objevuje více svislítek "|" za sebou.
&grammarError3=Chyba 3, řádek # : Vložil jste některý z nepovolených znaků. Vkládejte pouze znaky [a-z],[A-Z],[0-9] a [>,=,|].
&grammarError4=Chyba 4, řádek # : Levá strana některé gramatiky obsahuje terminální symbol.
&grammarError5=Chyba 5, řádek # : Nezadali jste žádnou gramatiku do pole "Vaše zadání".

# VI. Ukázka souboru s gramatikami ke generování

```
&template1=<p>S => aSAb;</p><p>S => AB;</p><p>A => Bb;</p><p>A => aA;</p><p>B =>
bB;</p><p>B => e;</p>
&template2=<p>S => aSAb | AB;</p><p>A => Bb | aA;</p><p>B => bB | e;</p>
&template3=<p>S => aB | bB | C;</p><p>A => a | BA | c;</p><p>B => aS | bB;</p><p>C => dA |
bC | cD;</p><p>D => ac | bG;</p><p>E => f | Ac | b;</p><p>G => a;</p>
&template4=<p>S => aB | bC | B;</p><p>A => a | Ba | c;</p><p>B => ac | bB;</p><p>C => dA |
bC | cD;</p><p>D => ac | bG;</p><p>E => f | Bc | b;</p><p>G => aB;</p>
&template5=<p>S => aB | bB;</p><p>A => a | BA;</p><p>B => aS | bB;</p>
&template6=<p>S => aB | bB | C;</p><p>A => a | BA | c;</p><p>B => aS | bB;</p><p>C => dA |
bC | cD;</p><p>D => ac | b;</p>
```