VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

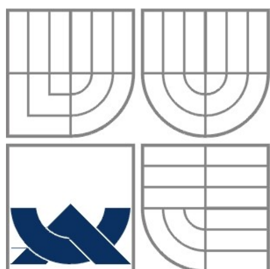
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# SYNTAXÍ ŘÍZENÝ EDITOR POPISNÉHO JAZYKA ISAC A JAZYKA GENERICKÉHO ASSEMBLERU V PLATFORMĚ ECLIPSE

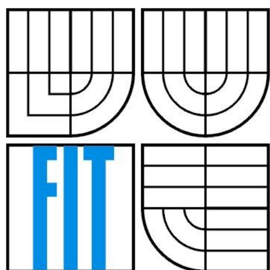
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Boris Šuška
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# SYNTAXÍ ŘÍZENÝ EDITOR POPISNÉHO JAZYKA ISAC A JAZYKA GENERICKÉHO ASSEMBLERU V PLATFORMĚ ECLIPSE

SYNTAX-DIRECTED EDITOR FOR ISAC LANGUAGE AND FOR LANGUAGE OF GENERIC ASSEMBLER IMPLEMENTED AS ECLIPSE PLUGIN

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Boris Šuška
AUTHOR

VEDOUCÍ PRÁCE Ing. Karel Masařík
SUPERVISOR

BRNO 2007

# Abstrakt

Práce se zaoberá vytvořením LR parsra z gramatiky zapsané formálně. Vytvořením lexikálního analyzátoru založeného na deterministickém konečném automatu, který je vytvářen z regulárnich výrazů popisujících lexémy jazyka. Použití vytvoreného syntaktického analyzátoru při konstrukci syntaxí řízeného editoru pod platformou Eclipse.

# Klíčová slova

syntaxí řízený editor, java kolekce, Eclipse IDE, ISAC, asembler, LR parser, deterministický konečný automat, regulární výrazi

# Abstract

This thesis is dealing with creation of LR parser from formally described grammar. I was creating a lexical analyzer based on determinictic finite automaton, which is created from regular expressions. These expressions describe lexemes of language generated by grammar. I used created syntax analyzer to construct syntax directed editor using Eclipse platform.

# Keywords

syntax directed editor, java collections framework, Eclipse IDE, ISAC, assembler, LR parser, deterministic finite automata, regular expressions

# Citace

Boris Šuška: Syntaxí řízený editor popisného jazyka ISAC a jazyka generického assembleru v platformě Eclipse, bakalářská práce, Brno, FIT VUT v Brně, 2007

# Syntaxí řízený editor popisného jazyka ISAC a jazyka generického assembleru v platformě Eclipse

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Karla Masaříka. Další informace mi poskytli prof. RNDr. Alexander Meduna, CSc.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Boris Šuška
15. 5. 2007

© Boris Šuška, 2007.

*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Obsah

Obsah .......................................... 1
1 Úvod .......................................... 3
2 Základné pojmy .......................................... 4
2.1 Jazyky .......................................... 4
2.1.1 Abeceda .......................................... 4
2.1.2 Reťazec .......................................... 4
2.1.3 Dĺžka reťazca .......................................... 4
2.1.4 Konkatenácia reťazcov a mocnina reťazca .......................................... 4
2.1.5 Podreťazec reťazca, predpona a prípona reťazca .......................................... 5
2.1.6 Jazyk, konečné a nekonečné jazyky .......................................... 5
2.1.7 Konkatenácia jazykov a iterácia jazyka .......................................... 5
2.1.8 Regulárne výrazy a regulárny jazyk .......................................... 5
2.2 Gramatiky .......................................... 6
2.2.1 Gramatika .......................................... 6
2.2.2 Derivačný krok a derivácia .......................................... 6
2.2.3 Jazyk generovaný gramatikou G .......................................... 6
2.2.4 Bezkontextová gramatika .......................................... 7
3 Popis riešených častí a použitých technologií .......................................... 8
3.1 Lexikálny analyzátor .......................................... 8
3.2 Syntaktický analyzátor .......................................... 9
3.3 Použité technológie .......................................... 9
3.3.1 Kolekcie .......................................... 9
3.3.2 Spracovanie XML dokumentu metodou SAX .......................................... 15
4 Popis riešenia .......................................... 17
4.1 Lexikálny analyzátor .......................................... 17
4.1.1 Spracovanie regulárnych výrazov .......................................... 17
4.1.2 Popis algoritmu pre pridanie regulárneho výrazu do automatu reprezentujúceho lexikálny analyzátor .......................................... 19
4.2 Syntaktická analýza .......................................... 25
4.2.1 LR Parser .......................................... 25
4.3 Platforma Eclipse .......................................... 28
4.3.1 Eclipse – prehľad .......................................... 28
4.3.2 Vývoj zásuvných modulov .......................................... 28
5 Závěr .......................................... 31
Literatura .......................................... 32

Seznam příloh........................................................................................................................ 33

# 1 Úvod

Vývoj počítačov zaznamenal značný rast, ktorý sa v súčasnej dobe oproti minulosti pomaly ustáluje. Dôležitým faktorom z hľadiska užívateľa je komunikácia, resp. vyjadrovacie prostriedky. Neplatí to, len pre užívateľské aplikácie, ale aj pre programovacie jazyky. Programovaciích jazykov je v súčasnej dobe veľké množstvo a dajú sa rozlišovať do rôznych kategórií napr. podľa princípu na akom sú postavené, podľa účelu, atď. Čím vyššiu formu abstrakcie nám programovací jazyk ponúka, tým rozmanitejšie môžu byť programy v ňom napísané. Jazyk akým sa budem v tejto práci zaoberať je zameraný na popis mikroprocesorov, podľa čoho bola vytvorená jeho syntax a sémantika, a bol pomenovný ISAC. Jazyk ISAC ponúka možnosť popisu mikroporcesoru z pohľadu hardwarového vybavenia, ale aj z pohľadu programovacieho jazyka, ktorým bude mikroprocesor programovaný, z pohľadu asembleru. Vývojové prostredie, založené na platforme Eclipse, pre tento jazyk okrem kompilátoru zahŕňa aj prostriedky, pre simuláciu programov napísaných v popísanom asemblere. Mojou úlohou je vytvoriť syntaxou riadený editor pre tento jazyk a poprípade aj editor pre generovaný asembler. Z toho vyplývajú dielčie časti, ktoré musím splniť, čím splním zadanú úlohu. V prvom rade sa jedná o syntaktickú analýzu, či už jazyka ISAC alebo aj jazyka asembleru a použitie tohto analyzátora pod platformou Eclipse. Vzhľadom na to, že mám vytvárať syntaktický analyzátor pre generovaný asembler, je potreba, aby bolo možné syntaktický analyzátor vhodne konfigurovať, pretože pre každý program v jazyku ISAC sa vygeneruje iný asembler. To by bolo použiteľné aj pre potreby syntaktickej analýzy jazyka ISAC, čím by som v podstate vytváral len jeden analyzátor.

Platforma Eclipse, je open source vývojové prostredie, vyvýjané pod záštitou firmy IBM, ktorého komerčná verzia je známa pod názvom WEB Sphere. Vývojové prostredie Eclispe je zamerané na programovací jazyk Java s podporou zásuvných modulov a ich vývoja pre toto prostredie. Takže v úplnom základe, nám Eclipse ponúka vývojové prostredie pre jazyk Java a vývojové prostredie pre zásuvné moduly napísané v tomto jazyku pre Eclipse IDE.

# 2 Základné pojmy

Nasledujúce základné pojmy, sú nutné k pochopení ďaľšieho textu. Bude sa jednať predovšetkým o teóriu formálnych jazykov. Následujúci text bol čerpaný z [1].

## 2.1 Jazyky

Jedným zo základných pojmov pre vymedzenie jazyka sú pojmy *abeceda* a *reťazec*.

### 2.1.1 Abeceda

**Definícia**: *Abecedou* rozumieme konečnú neprázdnu množinu prvkov, ktoré nazývame *symboly abecedy*.

### 2.1.2 Reťazec

**Definícia**: *Reťazcom* (niekdy sa tiež používá „slovo“ alebo „veta“) nad danou abecedou rozumieme každú konečnú postupnosť symbolov abecedy. Prázdnu postupnosť symbolov, tj. postupnosť, ktorá neobsahuje žiadny symbol, nazývame prázdny reťazec. Prázdny reťazec označuje symbol ε.
Formálne je možnné definovať reťazec nad abecedou Σ takto:

1. prázdný reťazec ε je reťazec nad abecedou Σ,
2. ak je $x$ reťazec nad abecedou Σ a $a \in \Sigma$, potom $xa$ je reťazec nad abecedou Σ.

### 2.1.3 Dĺžka reťazca

**Definícia**: *Dĺžka reťazca* je nezáporné celé číslo udávajúce počet symbolov reťazca. Dĺžku reťazca $x$ značíme $|x|$ a definujeme:

1. pokiaľ $x = \varepsilon$, potom $|x| = 0$,
2. pokiaľ $x = a_1 \dots a_n$, potom $|x| = n$ pre $n \geq 1$ a $a_i \in \Sigma$ pre všetky $i = 1, \dots, n$.

### 2.1.4 Konkatenácia reťazcov a mocnina reťazca

**Definícia**: Nech $x$ a $y$ sú dva reťazce nad abecedou Σ. *Konkatenácia* (zreťazenie) reťazca $x$ s reťazcom $y$ vznikne reťazec $xy$ pripojením reťazca $y$ za reťazec $x$. Operácia konkatenácia je asociatívna, tj. $x(yz) = (xy)z$, ale nie je komutatívna, $xy \neq yx$.

**Definícia**: Nech $x$ je reťazec nad abecedou Σ. Pre $i \geq 0$, $i$-tá mocnina reťazca $x$ ($x^i$), je definovaná:

1. $x^0 = \varepsilon$,
2. pre $i \geq 1$: $x^i = xx^{i-1}$

### 2.1.5 Podreťazec reťazca, predpona a prípona reťazca

**Definícia**: Nech $w$ je reťazec nad abecedou $\Sigma$.

- Reťazec $z$ sa nazýva *podreťazcom* reťazca $w$, ak existujú reťazce $x$ a $y$ také, že $w = xzy$.
- Reťazec $x_1$ se nazývá *prefixom* (predponou) reťazca $w$, ak existuje reťazec $y_1$ taký, že $w = x_1y_1$.
- Reťazec $y_2$ se nazývá *sufixom* (príponou) reťazca $w$, ak existuje reťazec $x_2$, taký, že $w = x_2y_2$.

### 2.1.6 Jazyk, konečné a nekonečné jazyky

**Definícia**: Nech $\Sigma$ je abeceda a $\Sigma^*$ značí množinu všetkých reťazcov nad $\Sigma$ vrátane reťazca prázdneho. Symbol $\Sigma^+$ značí množinu všetkých reťazcov nad $\Sigma$ okrem prázdneho reťazca, tj. $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Množinu $L$, pre ktorú platí $L \subseteq \Sigma^*$ nazývame *jazykom* $L$ nad abecedou $\Sigma$.

**Definícia**: Jazyk $L$ je *konečný*, pokiaľ $L$ obsahuje konečný počet reťazcov, inak je *nekonečný*.

### 2.1.7 Konkatenácia jazykov a iterácia jazyka

**Definícia**: Nech $L_1$ a $L_2$ sú jazyky nad abecedou $\Sigma$. *Konkatenáciu jazykov* $L_1$ a $L_2$ definujeme:

$$L_1L_2 = \{xy: x \in L_1 \wedge y \in L_2\}$$

**Definícia**: Nech $L$ je jazyk nad abecedou $\Sigma$. Iteraci $L^*$ definujeme takto:

1. $L_0 = \{\varepsilon\}$,
2. $L_n = LL_{n\text{-}1}$ pre $n \geq 1$,
3. $L^* = L^0 \cup L^1 \cup \quad \cup L^n$ pre $n \geq 0$

### 2.1.8 Regulárne výrazy a regulárny jazyk

**Definícia**: *Regulárne výrazy* nad abecedou $\Sigma$ a jazyky, ktoré značia, definujeme:

1. $\varnothing$ je regulárny výraz značiaci prázdnu množinu (prázdny jazyk),
2. $\varepsilon$ je regulárny výraz značiaci jazyk $\{\varepsilon\}$,
3. $a$, kde $a \in \Sigma$, je regulární výraz značiaci jazyk $\{a\}$,
4. nech $r$ a $s$ sú regulárne výrazy značiace v poradí jazyky $L_r$ a $L_s$, potom:
   a) $(r.s)$ je regulárny výraz značiaci jazyk $L = L_rL_s = \{xy: x \in L_r \wedge y \in L_s\}$,
   b) $(r+s)$ je regulárny výraz značiaci jazyk $L = L_r \cup L_s = \{x: x \in L_r \vee x \in L_s\}$,
   c) $(r^*)$ je regulárny výraz značiaci jazyk $L = L_r^*$.

**Definícia**: Nech $L$ je jazyk. $L$ je *regulárny jazyk*, pokiaľ existuje regulárny výraz $r$, ktorý jazyk $L$ značí.

## 2.2 Gramatiky

Tento pojem súvisí veľmi úzko s problémom reprezentácie jazyka. Gramatika, ako najznámejší prostriedok pre reprezentáciu jazykov, splňuje základné požiadavky kladené na reprezentáciu konečných i nekonečných jazykov, požiadavku konečnosti reprezentácie. Chomského hierarchia jazykov, vymedzuje štyry typy gramatík. V ďalšom texte se zameram predevšetkým na bezkontextovú gramatiku.

### 2.2.1 Gramatika

**Definícia**: *Gramatika G* je štvorica $G = (N, \Sigma, P, S)$, kde:

- $N$ je konečná množina nonterminálnych symbolov,
- $\Sigma$ je konečná množina terminálnych symbolov, $N \cap \Sigma = \emptyset$,
- $P$ je konečná podmnožina kartézského součinu $(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$,

prvok $(\alpha, \beta)$ množiny $P$ nazývame *prepisovacím pravidlom* a zapisujeme $\alpha \rightarrow \beta$,

- $S \in N$ je východzí (počiatočný) symbol gramatiky.

### 2.2.2 Derivačný krok a derivácia

**Definícia**: Nech $G = (N, \Sigma, P, S)$ je gramatika a nech $u$, $v$ sú reťazce z $(N \cup \Sigma)^*$. *Príama derivácia* je binárna relácia $\Rightarrow$ vyjadrujúca: $u\alpha v \Rightarrow u\beta v$, kde $\alpha \rightarrow \beta$ je nejaké prepisovacie pravidlo z $P$.

**Definícia**: Nech $G = (N, \Sigma, P, S)$ je gramatika a $\lambda$, $\mu$ sú reťazce z $(N \cup \Sigma)^*$. Medzi reťazcami $\lambda$, $\mu$ platí relácia $\Rightarrow^+$ nazývaná *derivácia*, pokiaľ existuje postupnosť priamych derivácií $v_{i-1} \Rightarrow v_i$, kde $i = 1, \dots, n$, $n \geq 1$, taká, že platí:

$$\lambda = v_0 \Rightarrow v_1 \Rightarrow \dots \Rightarrow v_{n-1} \Rightarrow v_n = \mu$$

**Definícia**: Ak v gramatike G platí pre reťazce $\lambda$, $\mu$ relácia $\lambda \Rightarrow^+ \mu$ alebo identita $\lambda = \mu$, potom píšeme $\lambda \Rightarrow^* \mu$. Relácia $\Rightarrow^*$ je tranzitívnym a reflexívnym uzáverom relácie priamej derivácie $\Rightarrow$.

**Definícia**: Nech $G = (N, \Sigma, P, S)$ je gramatika a nech $u \in \Sigma^*$, $v \in (N \cup \Sigma)^*$, $\alpha \rightarrow \beta \in P$ je pravidlo. Potom *najľavejšiu deriváciu* nazveme reláciou $u\alpha\, v \Rightarrow_{lm} u\beta v$ a *najpravejšiu deriváciu* reláciu $v\alpha\, u \Rightarrow_{rm} v\beta\, u$.

### 2.2.3 Jazyk generovaný gramatikou *G*

**Definícia**: Nech $G = (N, \Sigma, P, S)$ je gramatika. Reťazec $\alpha \in (N \cup \Sigma)^*$ nazývame *vetnou formou*, ak platí $S \Rightarrow^* \alpha$, tj. reťazec $\alpha$ je generovaný z východzieho symbolu $S$. Vetná forma, ktorá obsahuje len terminálne symboly, se nazývá *veta*. *Jazyk* $L(G)$, generovaný gramatikou $G$, definujeme množinou všetkých viet: $L(G) = \{w \colon w \in \Sigma^* \wedge S \Rightarrow^* w\}$.

### 2.2.4 Bezkontextová gramatika

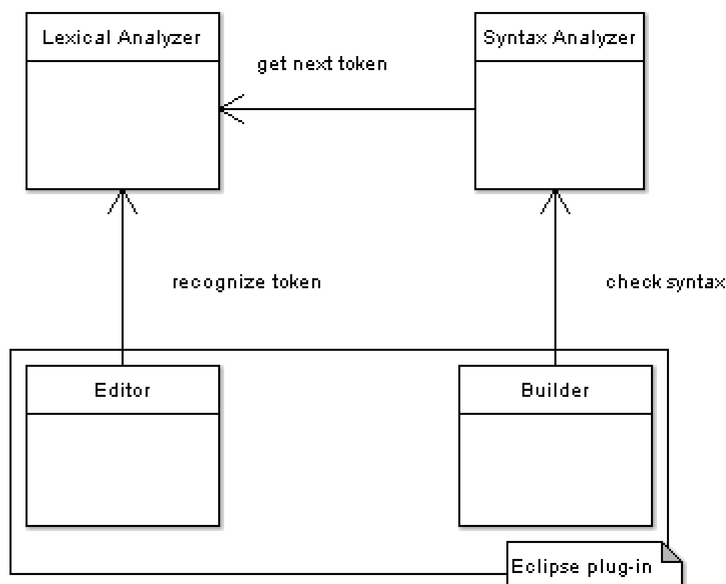
**Definícia**: *Bezkontextová gramatika* je podľa Chomského klasifikácie typu 2 a obsahuje pravidlá tvaru:

$$A \rightarrow \gamma, \quad A \in N, \quad \gamma \in (N \cup \Sigma)^*.$$

#### 2.2.4.1 LR gramatiky

LR gramatiky reprezentují největšiu triedu bezkontextových gramatík, ku ktorým je možné vždy zostrojiť deterministický analyzátor, ktorý robí analýzu zdola-hore. Vstupný reťazec sa číta zľava doprava a vytvára se pravý rozklad.

# 3 Popis riešených častí a použitých technologií.




*Obrázok 1: Koncept aplikácie*

## 3.1 Lexikálny analyzátor

Lexikálny analyzátor je založený na princípe deterministického konečného automatu, kde postupne prijíma vstupný reťazec znak po znaku, na základe čoho mení svoj aktuálny stav. Podľa definície je konečný automat pätica – konečná množina stavov, vstupná abeceda, konečná množina pravidiel (pri prijatí konkrétneho znaku sa má prejsť z aktuálneho stavu do pravidlom určeného), štartovací stav a množina konečných stavov. Deterministický konečný automat je taký automat, ktorý môže zmeniť stav len pri prijatí znaku a pre každý prijatý znak pre aktuálny stav existuje najviac jedna možnosť zmeny stavu určeného pravidlom.

Vzhľadom na to, že je požadovaná konfigurovateľnosť, je potreba zvoliť si vhodnú formu. Ja som si zvolil takú formu, ktorá je dostatočne obecná, aby nebola závislá na konkrétnom formáte vstupu, ale aby bolo možné, tak ako to je v prípade generovaného asembleru použiť aj iný ako presne definovaný vstup. Toto je dosiahnuté tým, že je možné zvoliť si vlastný prostriedok na nakonfigurovanie lexikálneho aj syntaktického analyzátoru – kde sú možnosti odkiaľ vziať definíciu gramatiky neobmedzené. Okrem toho existuje základný prostriedok, ktorý konfiguruje analyzátory podľa popisu vo formáte so syntaxou XML, popisujúceho gramatiku formálnym zápisom s rozšireným popisom konečných symbolov popísaných regulárnym výrazom.

## 3.2 Syntaktický analyzátor

Jedná sa o časť prekladača, ktorý k svojej práci využíva lexikálny analyzátor. Vytvára syntaktický strom pomocou tokenov získaných od lexikálneho analyzátoru. Na základe toho, či je alebo nie je možné vytvortiť syntaktický strom je alebo nie je analyzovaný zdrojový kód syntakticky správny. Vytvorenie syntaktického stromu sa riadi gramatiku, presnejšie pravidlami tejto gramatiky. Gramatika je založená na konečnej množine gramatických pravidiel, podľa ktorých sa generuje reťazec jazyka, generovaného gramatikou. Podľa definície je gramatika štvorica – množina terminálov (tokenov), množina nonterminálov, konečná množina pravidiel a počiatočný symbol z množiný nonterminálov.

## 3.3 Použité technológie

Keďže je program písaný pre platformu Eclipse, ktorá je postavená na technológií Jave, všetky časti som vytvoril v programovacom jazyku Java. Jazyk Java je objektovo orientovaný programovací jazyk, ktorý je prekladaný do bytekódu, ktorý je spustiteľný na ľuboveľnej platforme, na ktorej je k dispozícií virtuálny stroj, ktorý bytekód interpretuje. Popis celého jazyka je rozsiahly, preto sa zamerám na časti, ktoré som prevažne používal. Pre vývoj aplikácie som použil distribúciu JDK (Java Devolepement Kit), ktorá oproti distrubúcii JRE (Java Runtime Enviroment) hlavne obsahuje kompilátor, debuger a dokumentáciu k Java API (samozrejme aj ďalšie nástroje, ktoré sa používajú pri vývoji pod touto platformou, ktoré som však nemal dôvod použiť). Z verzií, ktoré sú k dispozícii som použil poslednú dostupnú – verziu 6. Pre spustenie programu je požadovaná minimálna verzia 5, pretože som pri implementácii nepoužil žiadne zvláštnosti verzie 6.

Z Java API som použil nasledovné časti:

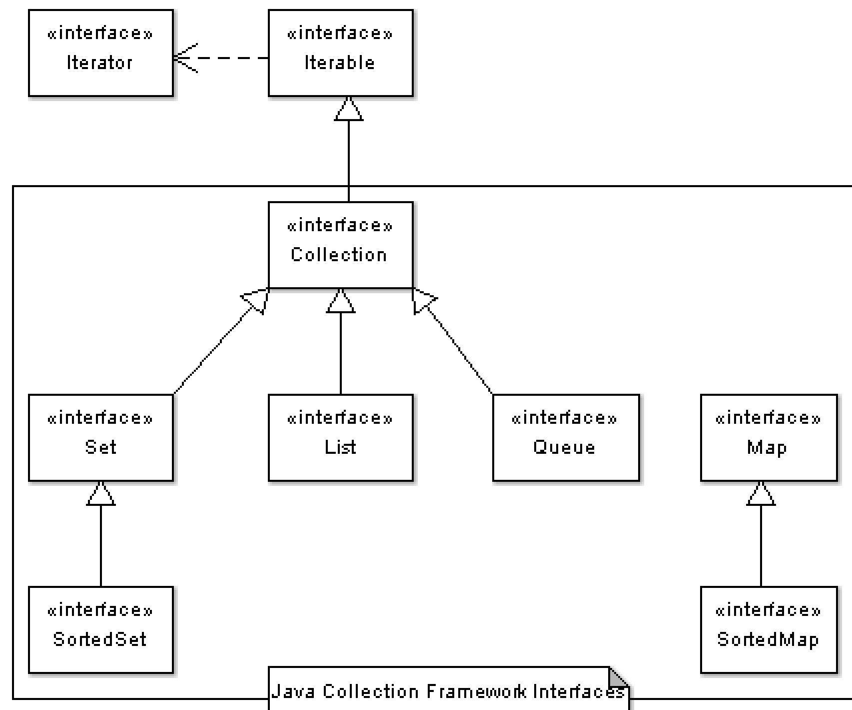
- Collection framework – kolekcie.
- XML SAX parsing – Spracovanie XML dokumentu metódou SAX.

### 3.3.1 Kolekcie

Kolekciu si môžeme predstaviť ako sklad objektov. Do kolekcie môžeme objekty pridávať, môžeme ich uberať, vyhľadávať a cez celú kolekciu môžeme prechádzať – pričom nemusí byť dodržané poradie v akom sme objekty do kolekcie vložili. Kolekcie si môžeme rozdeliť na také, ktoré môžu obsahovať rovnaké objekty a na také, ktoré rovnaké objekty obsahovať nemôžu. Pri pohľade na Java API, čo je základná skupina tried, nájdeme kolekcie v balíčku java.util. Kolekcie sú tu zastúpené rozhraniami, ktoré reprezentujú dostupnú funkčnosť a triedamy, ktoré predstavujú konkrétnu implementáciu funkčnosti. Základným rozhraním pre kolekcie je Collection, od ktorého dedia nasledujúce tri základné:

- Set – predstavuje kolekciu, ktorá nemôže obsahovať duplicitné hodnoty
- List – predstavuje kolekciu, ktorá duplicitné hodnoty obsahovať môže

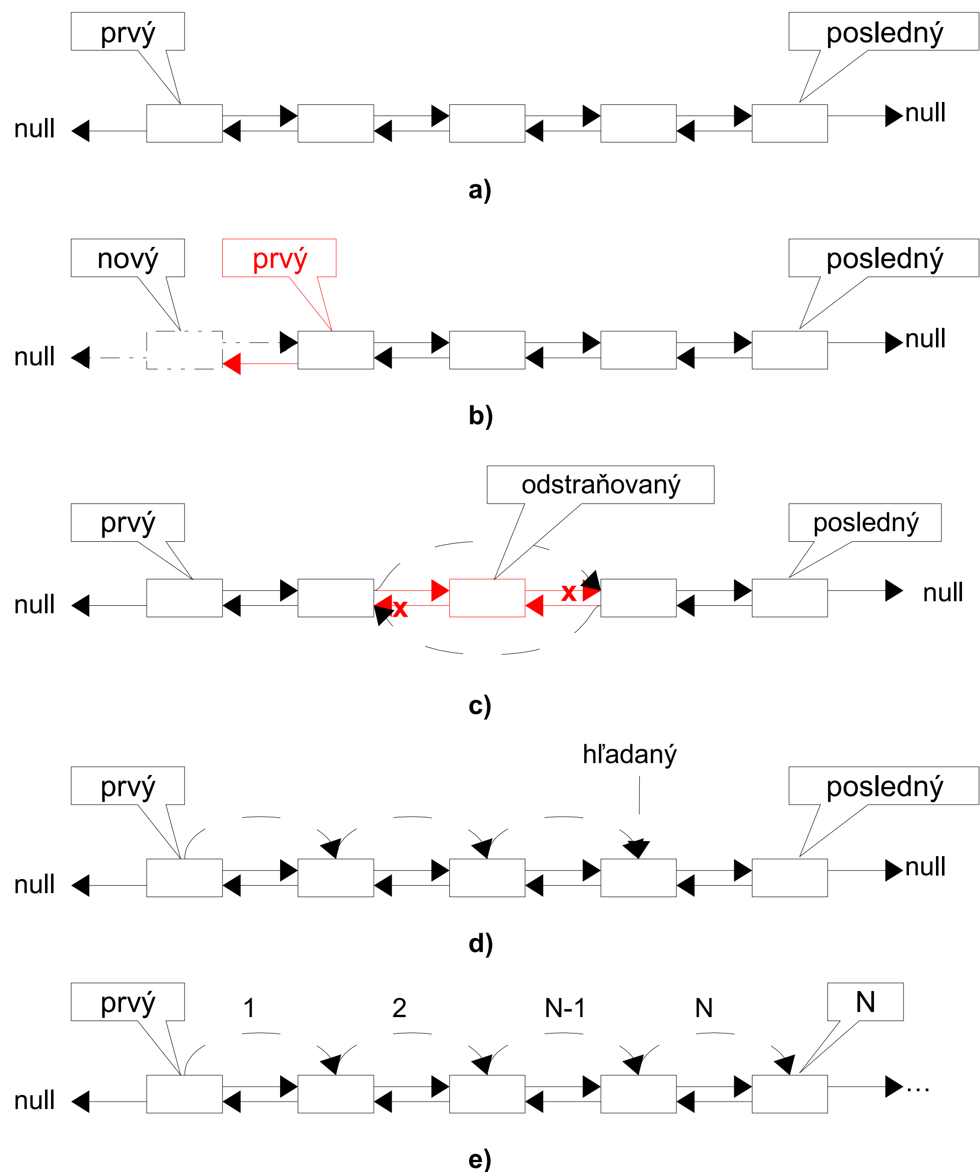
- Queue – môže obsahovať duplicitné hodnoty, pri pridávaní sa môže riadiť prioritou pridávaných prvkov, od čoho závisí poradie v akom ich vyberáme




*Obrázok 2: Kolekcie*

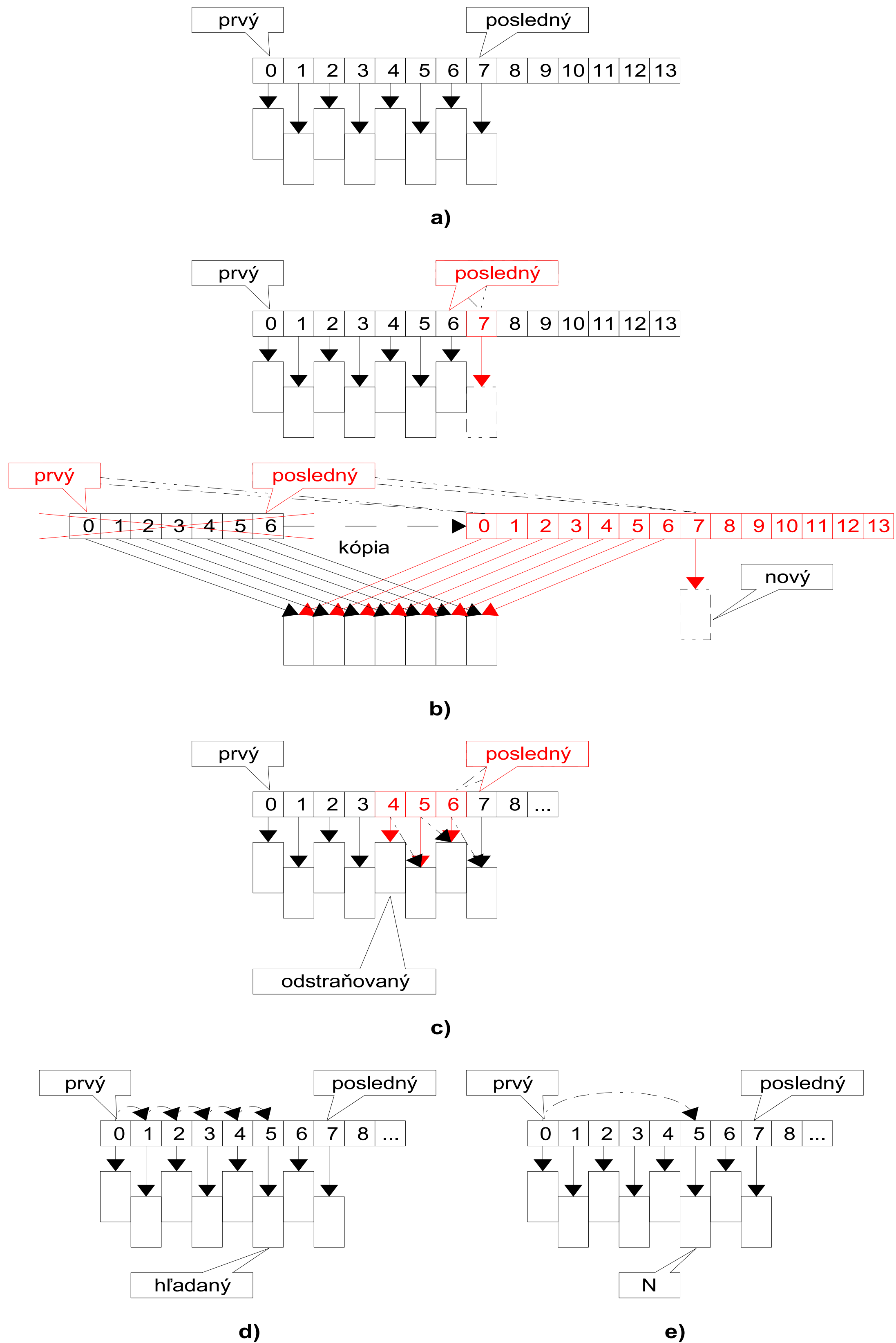
Preto, aby sme mohli používať kolekcie, rozhrania nestačia, je potreba použivať, aj ich implementáciu. Implelmentáciu volíme podľa toho, ako s danou kolekciou chceme ďalej pracovať. Implementácie pre rozhranie List:

**LinkedList** – Implementácia je založená na princípe dvojsmerne viazaného zoznamu. Pri vkladaní objektu do zoznamu, sa na prvú (alebo poslednú – záleží od implementácie) položku záznamu naviaže vkladaný objekt. Pri vyhľadávaní objektu sa musí prechádzať celý zoznam postupne. Prístup k náhodnému prvku podľa poradia je rovnako náročný pretože sa musia prejsť všetky položky od začiatku až k vyhľadávanému prvku. Odobranie prvku zo zoznamu, je len o tom, že sa previažiu referencie medzi susednými prvkami, čo nie je až tak náročné – najzložitejšie je požadovaný prvok nájsť.




*Obrázok 3: a) Štruktúra dvojsmerne viazaného zoznamu; b) Pridanie prvku; c) Odstránenie prvku; d) Vyhľadanie prvku; e) Prístup k náhodnému prvku*

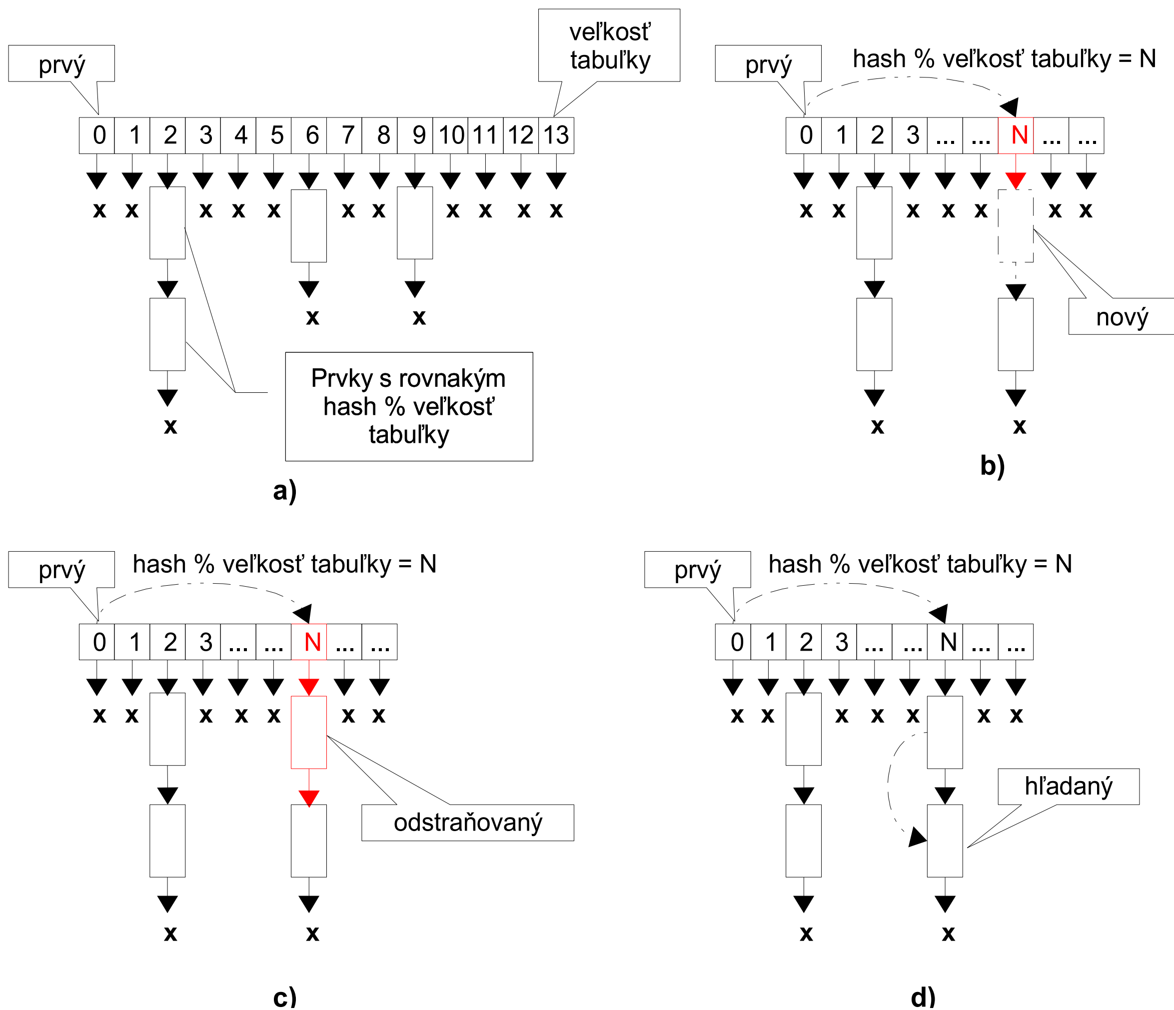
**ArrayList** – Implementácia je založená na poli objektov. Vkladaný prvok sa vloží na poslednú voľnú pozíciu v poli, pokiaľ je pole zaplnené, automaticky sa zväčší. Pri vyhľadávaní objektu v kolekcií sa musí prechádzať celým poľom postupne cez všetky prvky, až k poslednému . Na druhej strane, pri prístupe k náhodnemu prvku v kolekcií sa pristupuje priamo k prvku bez prieťahov. Pri odstraňovaní prvku z kolekcie, sa skopíruje tá časť poľa, ktorá je za odstraňovaným prvkom až do konca od pozície odstraňovaného prvku a posledný prvok sa označí ako voľný. Táto operácia je dosť náročná a keď si uvedomíme, že sa toto robí pre každý odstraňovaný prvok, je potreba sa zamyslieť, kedy túto implementáciu použiť.




*Obrázok 4: a) Štruktúra ArrayList-t; b) Pridanie prvku (aj v prípade zmeny veľkosti alokovaného poľa); c) Odstránenie prvku; d) Vyhľadanie prvku; e) Prístup k náhodnému prvku*

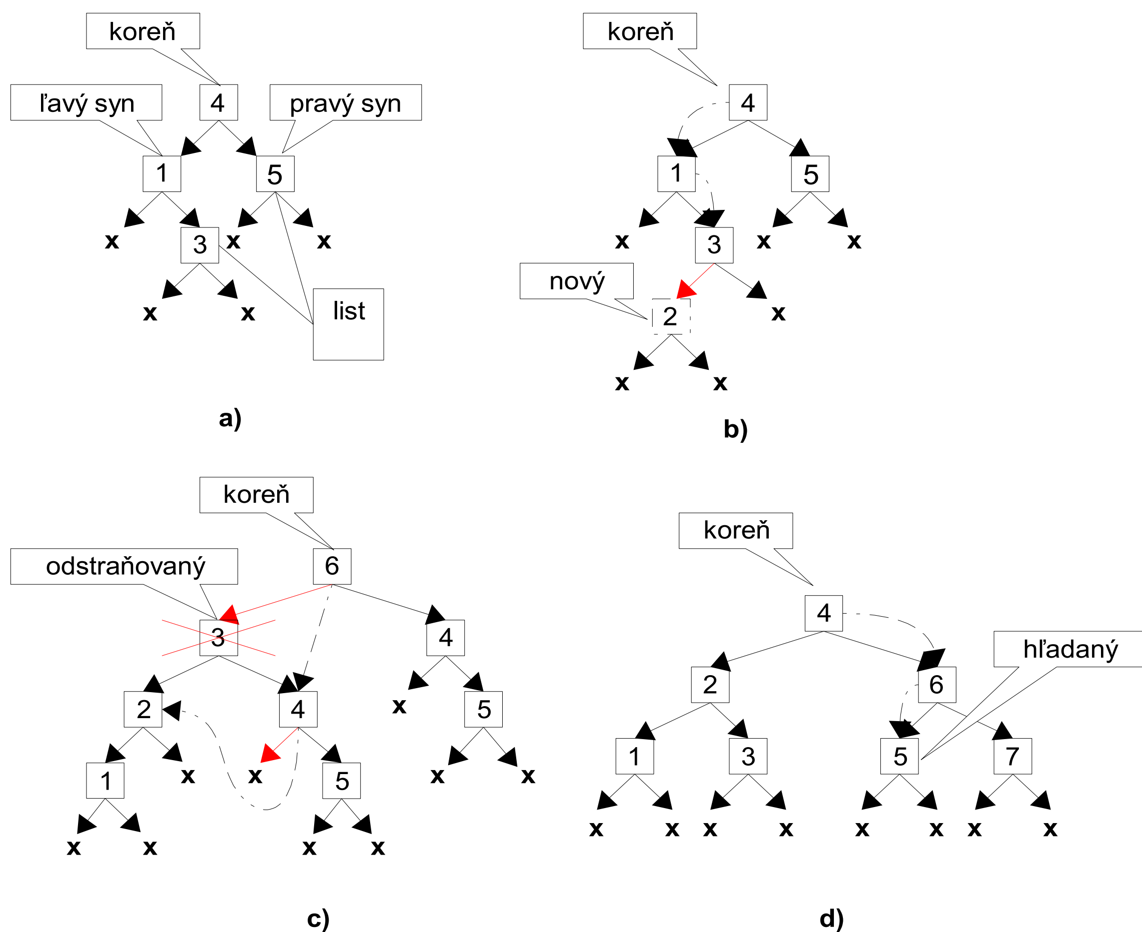
Implementácie pre rozhranie Set:

**HashSet** – Implemetácia je založená na princípe tabulky s rozptýlenými položkami (hash table). Pri vkladaní sa zisťuje číselná reprezentácia (hash kód) instancie objektu (konkrétneho objektu), čo si môže objekt riadiť sám na základe svojho vnútorného stavu. Po nájdení takejto číselnej reprezentácie sa v tabuľke nájde pozícia kam bude instancia vložená. Keďže objektov máme nekonečne veľa a veľkosť tabuľky je obmedzená, využívame zvyšok po delení, aby sme redukovali hash kód na veľkosť tabuľky. Preto môže byť pozícia už obsadená a tak na uloženie ďalšieho prvku použije zoznam s viazanými položkami, tak ako tomu bolo u triedy LinkedList s tým rozdielom, že ak sa objekt v zozname nachádza nepridá sa, pretože rozhranie Set definuje, že kolekcia nemôže obsahovať duplicitné prvky. Vyhľadávanie v takejto dátovej štruktúre je pomerne rýchle, pretože sa na začiatku nájde pozícia podľa hash kódu a potom sa prechádza príslušný zoznam. Pokiaľ sa prejde celý zoznam prislúchajúci pozícií podľa hash kódu a objekt sa vňom nenachádza, znamená to, že sa objekt v kolekcii nenachádza vôbec.




*Obrázok 5: [**x** = null] a) Štruktúra tabuľky s rozptýlenými položkami; b) Pridanie prvku; c) Odstránenie prvku; d) Vyhľadanie prvku;*

**TreeSet** – Implementácia založená na princípe binárneho stromu. Ponúka možnosť vkladané objekty radiť, podľa zvolených kritérií, pokiaľ sa táto možnosť nevyužije, použije sa tzv. prirodzené radenie. Porovnanie objektov je dôležité preto, aby sme vedeli, kam objekt pri vkladaní umiestniť. Pri vkladaní objektu začíname vždy od koreňového elementu, pokiaľ neexistuje, vkladaný objekt je použitý ako koreňový element. Ak sme objekt nevložili pokračujeme rekurzívne ľavou vetvou (za koreňový prvok budeme považovať ľavý element akuálneho koreňového elementu) ak je vkladaný objekt menší než koreňový, inak rekurzívne pokračujeme pravou stranou. Táto implementácia ako jediná ponúka možnosť radenia objektov, čo znamená, že pri prechode cez kolekciu môžeme očakávať objekty v poradí podľa určitého kľúča (ak sme sa nespolahli na prirodzené radenie). Vyhľadanie objektu v kolekcii je pomerne rýchle, pretože sa vždy od koreňového elementu postupuje vetvou podľa toho , či je objekt väčší alebo menší ako koreňový. Tým pádom sa pri každom kroku, keď objekt nebol nájdený, počet objektov, ktoré ešte neboli prehľadávané, zníži o polovicu. Oproti implementácií HashSet je táto implementácia menej výkonná, z dôvodu vkladania prvkov (vždy sa musí prechádzať stromom ako pri vyhľadávaní). Doporučené je používať túto kolekciu len keď chceme radiť objekty.




*Obrázok 6: [**x** = null] a) Štruktúra binárneho stromu; b) Pridanie prvku; c) Odstránenie prvku; d) Vyhľadanie prvku;*

| | Vkladanie | Prechádzanie | Vyhľadávanie | Náhodný pristup | Odstraňovanie | Radenie |
|---|---|---|---|---|---|---|
| *LinkedList* | ☺ | ☺ | ☹ | ☹ | ☺ | - |
| *ArrayList* | 😐 | ☺ | ☹ | ☺ | ☹ | - |
| *HashSet* | 😐 | 😐 | ☺ | - | ☺ | - |
| *TreeSet* | 😐 | ☺ | 😐 | - | ☺ | ☺ |

*Tabuľka 1: Porovnanie implementácií podľa efektivity v jednotlivých prípadoch*

Medzi kolekcie sa častokrát radí aj tzv. mapa, rozhranie Map. Toto rozhranie od rozhrania Collection nededí, ale dá sa prezentovať ako množina (Set), ktorá nesie užitočné dáta uložené pod kľúčom. Jedná sa o dvojicu kľúč-hodnota, kde každá hodnota je ukrytá pod kľúčom. Každý kľúč musí byť unikátny, preto si množinu kľúčov môžeme predstaviť ako množinu (rozhranie Set). Tu je vidieť aj dôvod toho, prečo rozhranie Map nenedí od rozhrania Collection. Je to práve tým, že kolekcie slúžia na uchovávanie objektov (častokrát rovnakého typu), kdežto mapa slúži na uchovávanie dvojice kľúč-hodnota. Najdôležitejšie u mapy je to, že si môžeme ukladať hodnoty podľa klúča a potom k nim rovnako pristupovať. Implementácie pre rozhranie Map:

**HashMap** – Množinu kľúčov má založenú na princípe HashSet-u, s tým rozdielom, že pre každú položku pridáva možnosť uloženia hodnoty, čím sa z HashSet-u stáva HashMap.

**TreeMap** – Podobne ako u HashMap-y, aj tu je množina kľúčov založená na princípe implementácie rozhranie Set, tentokrát implementácia TreeSet. Tým pádom preberá aj jeho možnosti a to usporiadať mapu podľa ľubovoľného kľúča.

### 3.3.2 Spracovanie XML dokumentu metodou SAX

Spracovanie XML dokumentu metodou SAX je založená na odchytávaní udalostí, ktoré vznikajú pri prechode XML dokumentom. Tým pádom táto metóda nie je nijak náročná. Oproti metóde DOM, kde sa vytvára objektový model dokumentu, čo je pamäťovo dosť náročné. Rozdieľ v týchto metodách spočíva hlavne v tom, že pri použití metódy SAX nemôžeme dokument upravovať, kdežto metóda DOM túto možnosť ponúka. V Java API sú potrebné triedy organizované do balíčkov java.xml.parsers a org.xml.sax. Triedu pre spracovanie XML dokumentu nájdeme v balíčku java.xml.parser s menom SAXParser. Táto trieda je abstraktná, takže nie je možné vytvoriť jej instanciu. V tomto prípade je použitý návrhový vzor factory, čiže nová instancia sa vyvára pomocou tzv. factory triedy, v tomto prípade SAXParserFactory, ktorej instancia sa vytvorí statickou metódou newInstance(), keď sa vyhľadáva vhodná implementácia SAXParserFactory, viac je možné nájsť v JAXP špecifikácií. Až potom je možné vytvoriť SAXParser, metódou newSAXParser(). Po získaní parseru ešte potrebujeme XMLReader – je to nový spôsob práce so SAX parserom, pred tým postačoval len parser. XMLReader získame metódou newXMLReader(). Na to, aby sme ho mohli použiť, potrebujeme mu predať instanciu triedy, ktorá implementuje rozhranie, podľa toho čo chceme

odchytávať. Instancii tejto triedy budú predávané všetky udalosti, ku ktorým počas prechodu dokumentom dôjde. Rozhrania, ktoré môžeme implementovať sú:

- ContentHandler – rozhranie pre spracovanie udalostí ohľadom obsahu. Dôležité sú hlavne metódy startElement a endElement.
- ErrorHandler – rozhranie pre spracovanie chýb v dokumente. Chyby sa rozlišujú na varovania, chyby s ktorých sa dá ešte zotaviť a chyby, z ktorých sa už zotaviť nedá. Viac o chybách je popísané v W3C XML 1.0 Recommendation.

Po nastavení vhodného rozhrania, použijeme metódu parse() triedy XMLReader. Ako parameter jej môžeme predať instanciu triedy InputStream. Tým pádom sa môže jednať o ľubovoľný zdroj odkiaľ parser dáta prijíma (súbor, miesto v sieti,...). Viac o vstupno-výstupnom systéme v Jave je popísané v dokumentácií k balíčku java.io. Za zmienku tiež stojí balíček java.nio, ktorý od verzie 1.4 predstavuje nový a rýchlejší spôsob práce so súbormi.

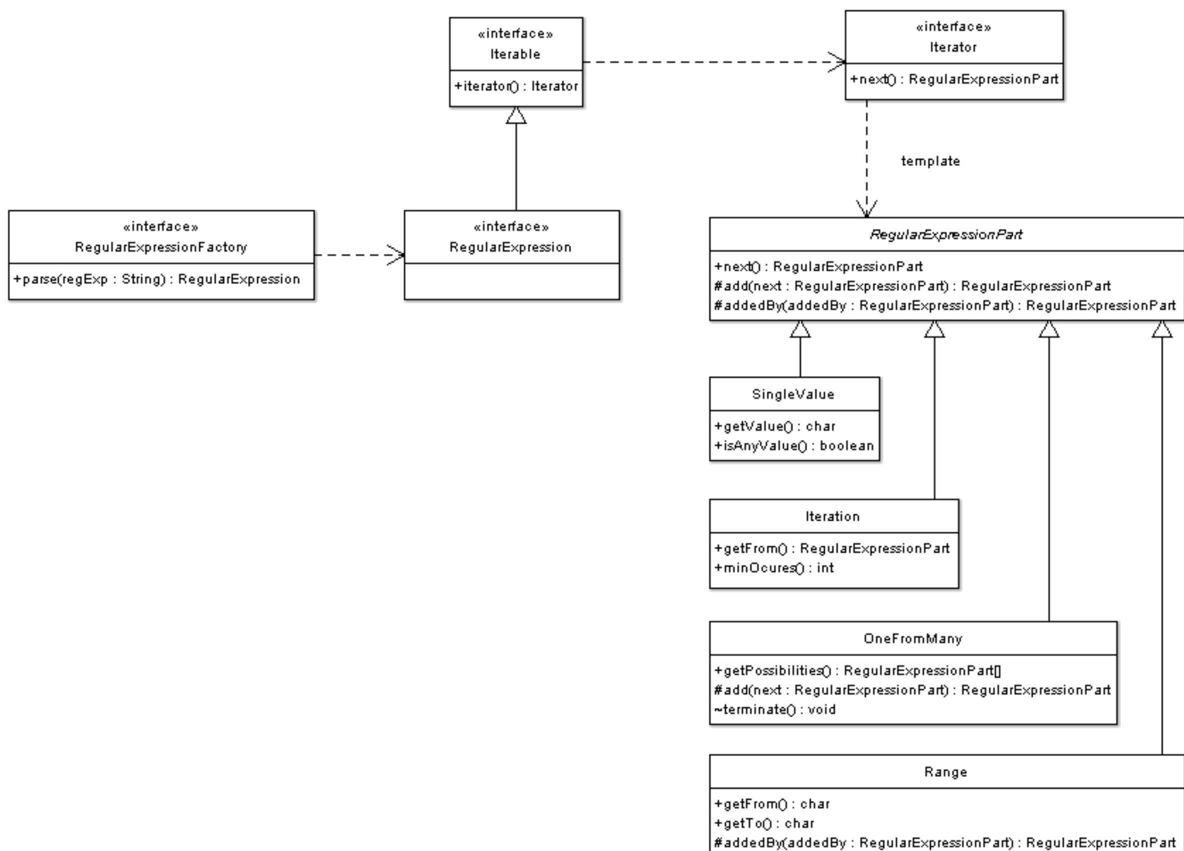
# 4 Popis riešenia

## 4.1 Lexikálny analyzátor

Lexikálny analyzátor bol vyvynutý ako samostatná časť, nezávislá od ostatných častí aplikácie. Pri vytváraní išlo o to vytvoriť deterministický konečný automat, ktorý je definovaný skupinou regulárnych výrazov, kde každý regulárny výraz predstavuje jednu lexému. Žiaľ regulárne výrazy ponúkané v Java API (balíček java.util.regex) nepostačovali mojím potrebám, pretože mňa zaujíma samotný regulárny výraz, nie to či daný reťazec mu zodpovedá alebo nie. Kvôli tomu som musel vytvoriť vlastný nástroj na spracovanie regulárnych výrazov. Potom som mohol o regulárnom výraze získať potrebné informácie.

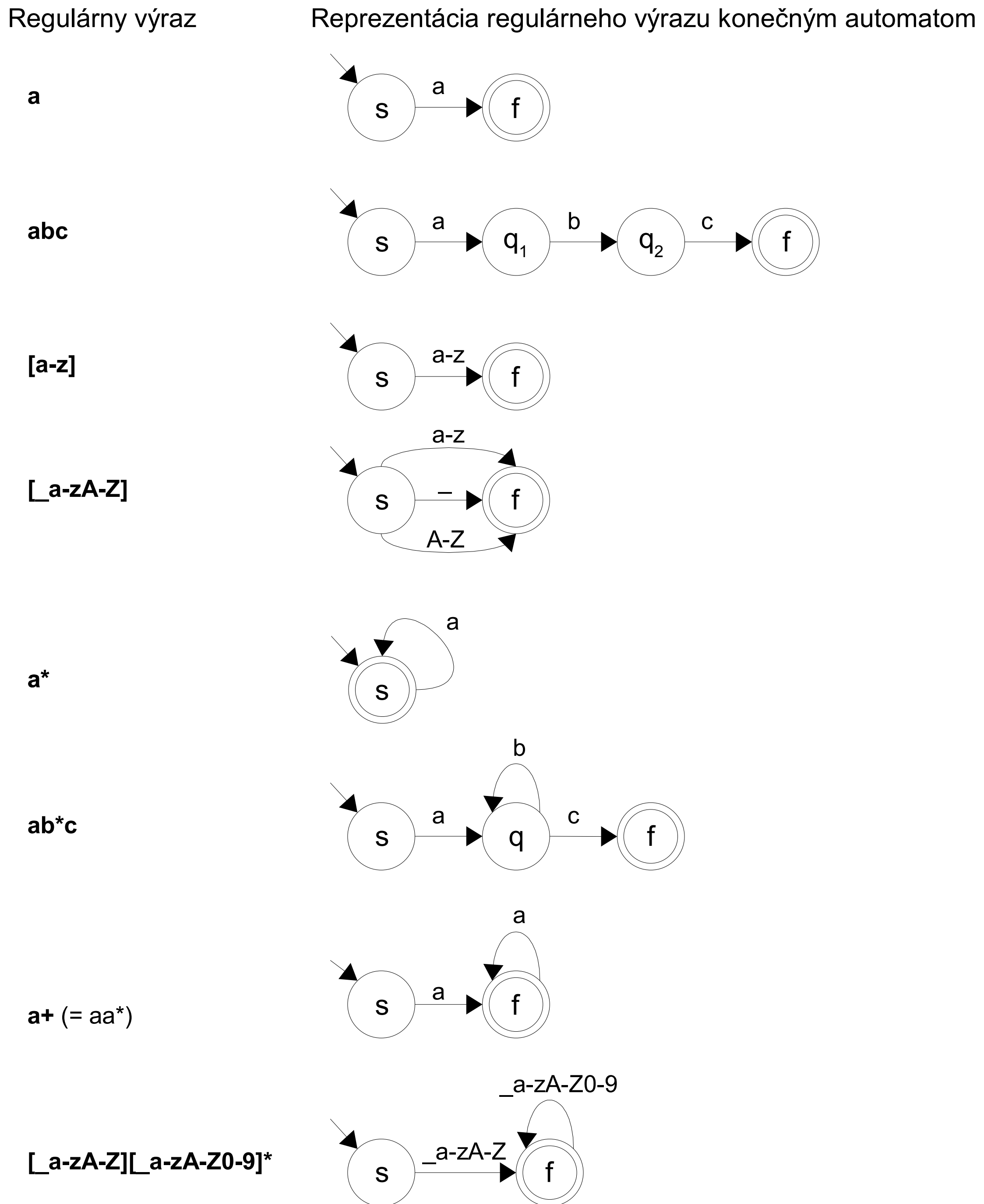
### 4.1.1 Spracovanie regulárnych výrazov

Na obrázku je objektový model nástroja pre spracovanie regulárnych výrazov. Jedná sa o objektovú reprezentáciu regulárneho výrazu, tak aby sa mi ďalej dobre spracovával. Použitie rozhraní je z dôvodu tzv. low coupling, čo vyjadruje malé zviazanie s okolím. Tým pádom je možné vytvoriť ľubovoľnú reprezentáciu regulárneho výrazu a použiť ju so zvyškom aplikácie, pokiaľ táto implementácia implementuje dané rozhrania.




*Obrázok 7: Diagram tried pre spracovenie regulárnych výrazov*

Po úspešnom kroku vytvorenia regulárneho výrazu nasleduje jeho pridanie do vytváraného deterministického konečného automatu. V prvom rade pre každý regulárny výraz vytvorím, ich reprezentácie konečným automatom, ktoré potom spojím do jedného finálneho automatu. Podľa nasledujúcich príkladov reprezentujem regulárny výraz na konečný automat.




*Obrázok 8: Príklady reprezenácie regulárneho výrazu konečným automatom*

Dôležitým poznatkom je, že každý konečný automat, ktorý vznikne je deterministický, má práve jeden koncový stav a pri prechode do tohoto stavu prešiel vždy všetkými stavmi.

### 4.1.2 Popis algoritmu pre pridanie regulárneho výrazu do automatu reprezentujúceho lexikálny analyzátor

Regulárny výraz reprezentovaný konečným automatom potrebujem pridať k existujúcemu deterministickému konečnému automatu, tak, aby stále ostal deterministiký.

Pre nasledujúci algoritmus vyhradím pojmy, ktoré budem v tomto význame ďalej používať.

- *konečný automat* – alebo len automat, mám na mysli existujúci automat, ktorý bude výsledkom spojenia všetkých regulárnych výrazov.
- *pridávaná vetva* – alebo len vetva, je pridávaný reguláerny výraz reprezentovaný ako konečný automat
- *hrana* – je spojenie aktuálneho stavu zo stavom, do ktorého sa automat dostane, keď prijme symbol, ktorým je hrana označená. Hrana je označená vždy práve jedným symbolom.
- *prechod* – je to skupina hrán spojujúca rovnaké stavy
- *stavom reprezentovaná hodnota* – každý regulárny výraz reprezentuje lexému, do automatu sa to prenesie tak, že každý stav automatu si uchováva hodnotu, ktorá reprezentuje lexému, preto v každom stave automatu vieme akú lexému automat momentálne prijíma.

1. Pokiaľ konečný automat nemá žiadne stavy, štartujúci stav pridávanej vetvy označím ako štartujúci stav automatu.
2. Porovnávam každú hranu vedúcu zo štartovacieho stavu automatu s každou hranou vedúcou zo štartovacieho stavu vetvy.
3. Pokiaľ nájdem totožné prechody, tj. prechod v automate a prechod vo vetve, kde ku každej hrane v prechode automatu existuje hrana označená rovnakým symbolom ako v prechode vetvy a naopak, potom:
   3.1. Ak odkazovaný stav vetvy je koncovým stavom a odkazovaný stav automatu nie je koncovým stavom, potom zmen odkazovanému stavu automatu reprezentovanú hodnotu na hodnotu odkazovaného stavu vetvy a odkazovaný stav automatu označ ako koncový stav.
   3.2. Ak odkazovaný stav vetvy nie je koncovým stavom a odkazovaný stav automatu nie je tiež koncovým stavom, potom zmen odkazovanému stavu automatu reprezentovanú hodnotu na nedefinovanú len v prípade, že sa ich reprezentované hodnoty líšia.
   3.3. Opakuj od bodu 2, s tým, že za štartovací stav automatu sa bude považovať odkazovaný stav automatu a za štartovací stav vetvy sa bude považovať odkazovaný stav vetvy.
4. Pokiaľ nájdem kolízne prechody, tj. prechod v automate a prechod vo vetve, kde najmenej k jednej hrane v prechode automatu existuje hrana označená rovnakým symbolom

ako v prechode vetvy a zároveň k najmenej jednej hrane v prechode automatu neexistuje hrana označená rovnakým symbolom ako v prechode vetvy, potom:

4.1.Nájdem prienik prechodov, tj. hrany v prechode automatu a hrany v prechode vetvy, ktoré sú označené rovnakým symbolom.

4.2.Nájdem prioritnejší prechod, tj prechod, ktorý má menší počet všetkých hrán.

4.3.Ak stav odkazovaný prioritnejším prechodom nie je koncový a stav odkazovaný menej prioritným prechodom je koncový, potom stav odkazovaný viac prioritným prechodom označ ako koncový a reprezentovanú hodnotu nastav podľa stavu, ktorý je odkazovaný menej prioritným prechodom.

4.4.Ak menej prioritný prechod vychádza zo skutočného štartovacieho stavu automatu, potom ho nahradím prechodom, ktorý obsahuje všetky hrany menej prioritného prechodu okrem hrán, ktoré má spoločné s viac prioritným a pridám nový prechod medzi skutočný štartovací stav automatu a stav odkazovaný viac prioritným prechodom, ktorý bude obsahovať všetky hrany viac prioritného prechodu.

4.5.Ak nebol vykonaný predchádzajúci bod, potom pridám nový prechod medzi stav z ktorého vychádza viac prioritný prechod a stav odkazovaný menej prioritným prechodom, ktorý bude obsahovať všetky hrany menej prioritného prechodu okrem hrán, ktoré má spoločné s viac prioritným.

4.6.Pokiaľ stav odkazovaný viac prioritným stavom odkazuje na iný stav, než sám na seba, potom opakuj od bodu 2, s tým, že za štartovací stav automatu sa bude považovať odkazovaný stav automatu a za štartovací stav vetvy sa bude považovať odkazovaný stav vetvy.

5. Pokiaľ neboli nájdené totožné ani kolízne prechody pridám nový prechod medzi štartovací stav automatu a stav odkazovaný prechodom vetvy, ktorý obsahuje všetky hrany prechodu vetvy.

6. Pre všetky novo pridané a zmenené prechody, ktoré spojujú rovnaké stavy urobím zjednotenie hrán, tj. ponechám hrany ktoré sú vo všetkých prechodoch totožné, čoho výsledkom je skutočne pridaný prechod.

7. Skutočný štartovací stav vetvy označím ako stav na vymazanie

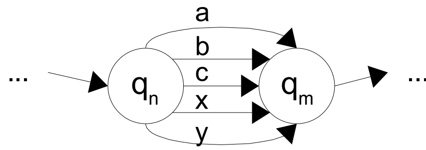
8. Pre každý stav označený na vymazanie zistím, či je odkazovaný aspoň jednou novo pridanou hranou

8.1.ak áno, potom tomuto stavu zruším označenie stav na vymazanie

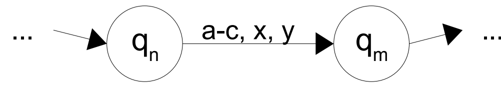
8.2.ak nie, potom všetky stavy, ktoré sú spojené najmenej jednou hranou s týmto stavom označím stavmi na vymazanie a aktuálny stav vymažem.

9. Opakujem krok 8, kým existuje nejaký stav na vymazanie.

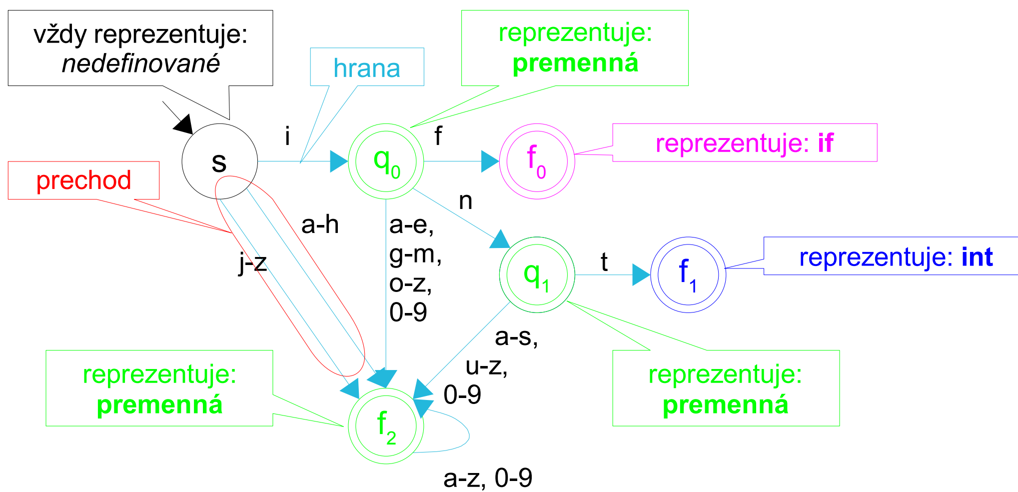
Skutočnosť




Skrátený zápis

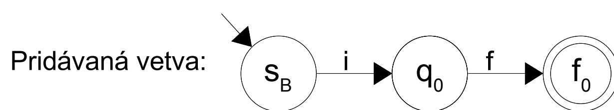


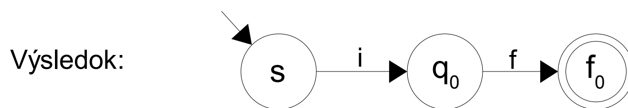

*Obrázok 9: Ukážka skráteného zápisu označenia hrán, čo bude použité v ďalších príkladoch. Ďalšia možnosť skráteného zápisu označenia hrany je a-c, x-y*

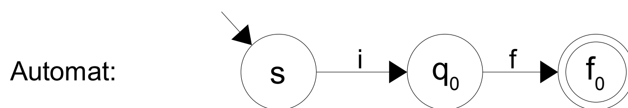


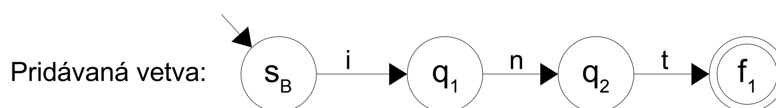

*Obrázok 10: Vyznačenie vyhradených pojmov*

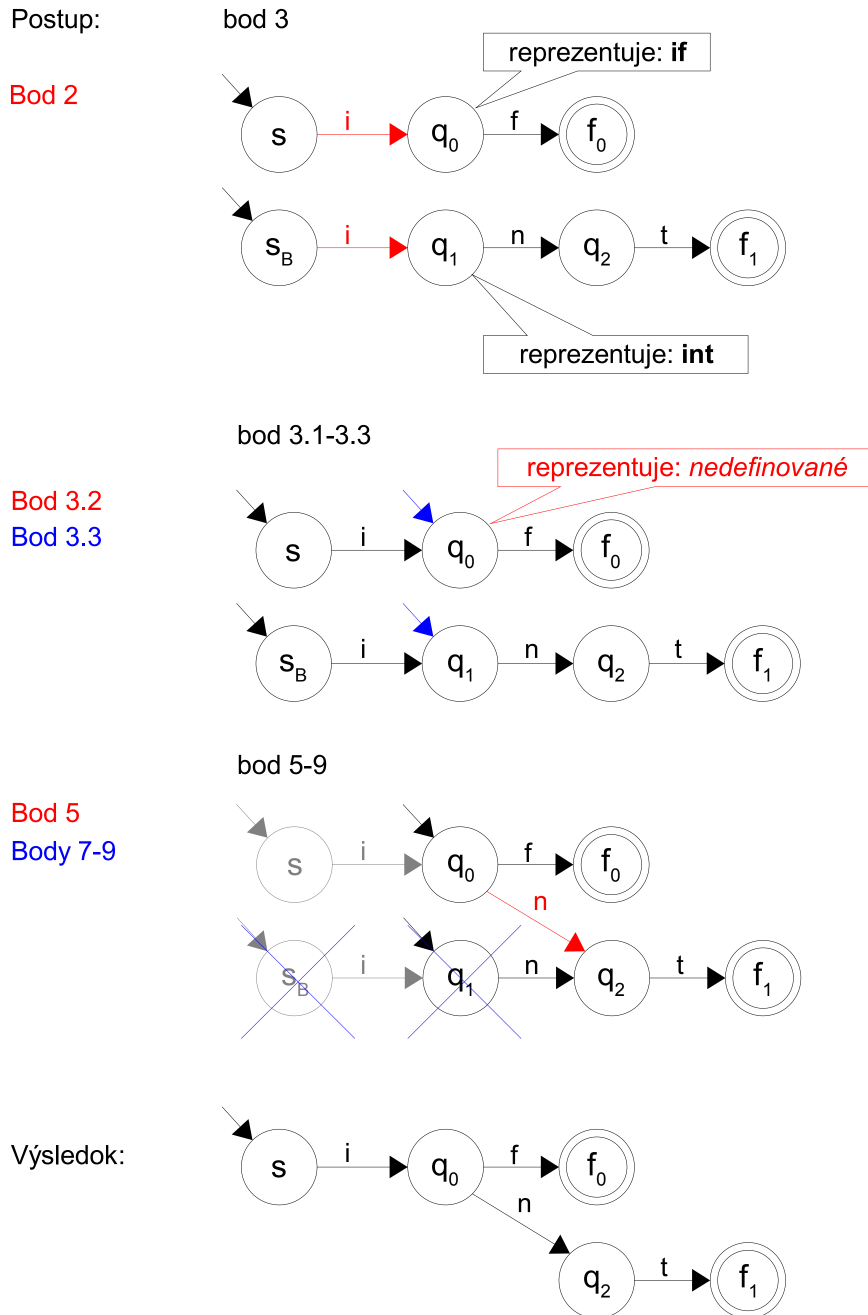
Automat: <prázdny>



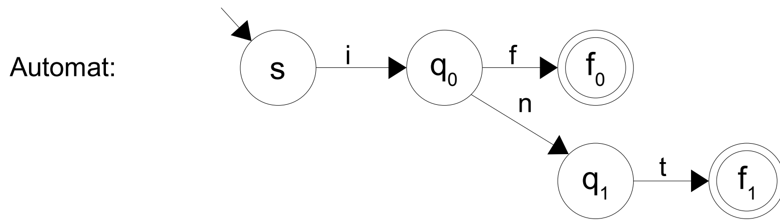



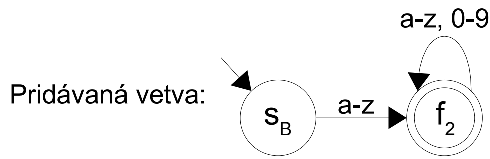



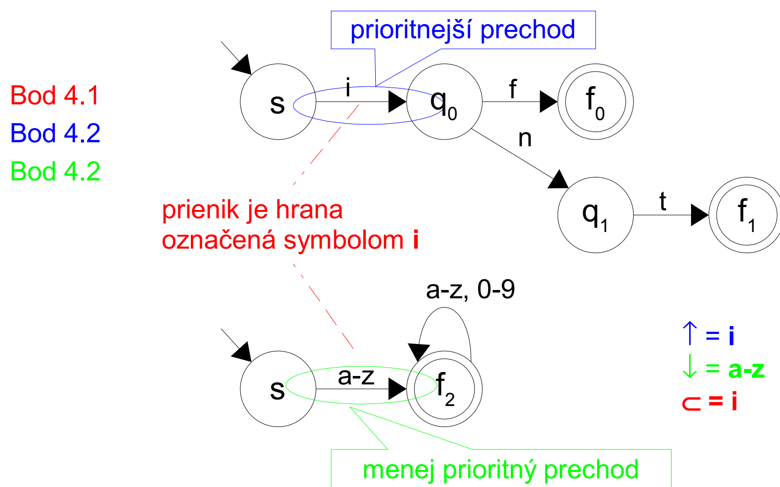



*Obrázok 11: Príklad algoritmu pre  pridanie regulárneho výrazu reprezentovaného konečným automatom do deterministického konečného automatu – kroky 1-3.*

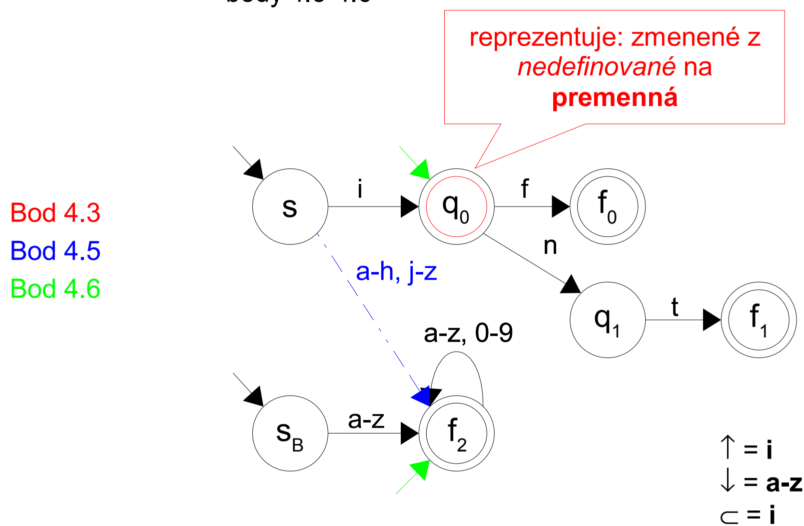


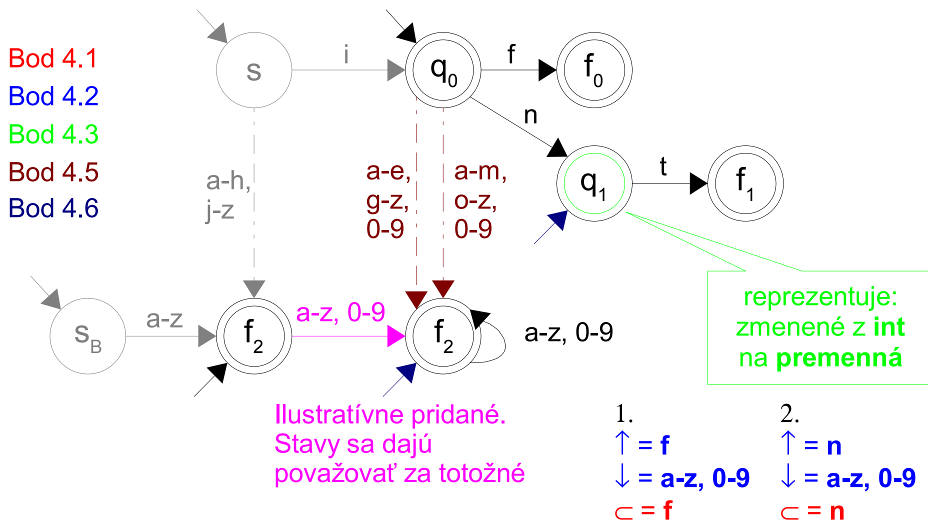

Postup: body 4.1 a 4.2



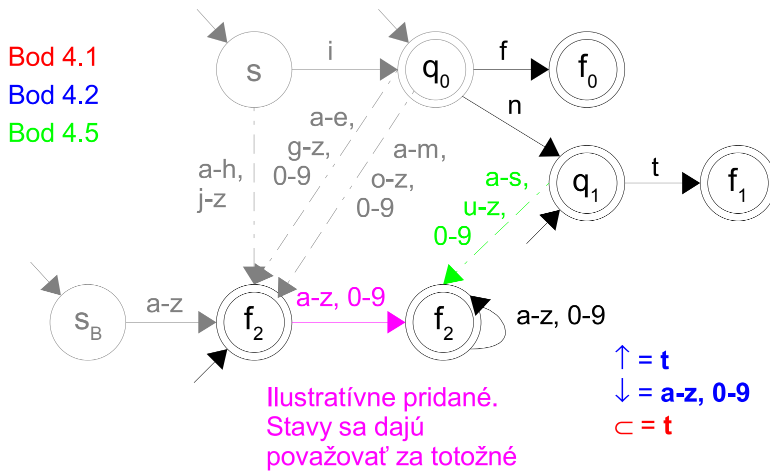

body 4.3-4.6



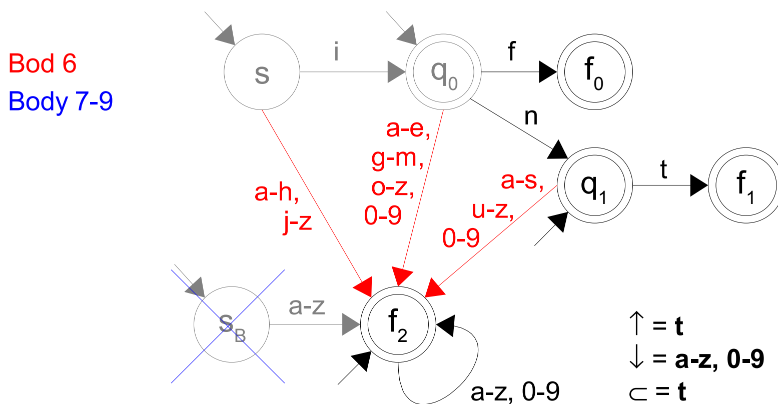
Bod 4, pre stavy určené predchádzajúcim krokom



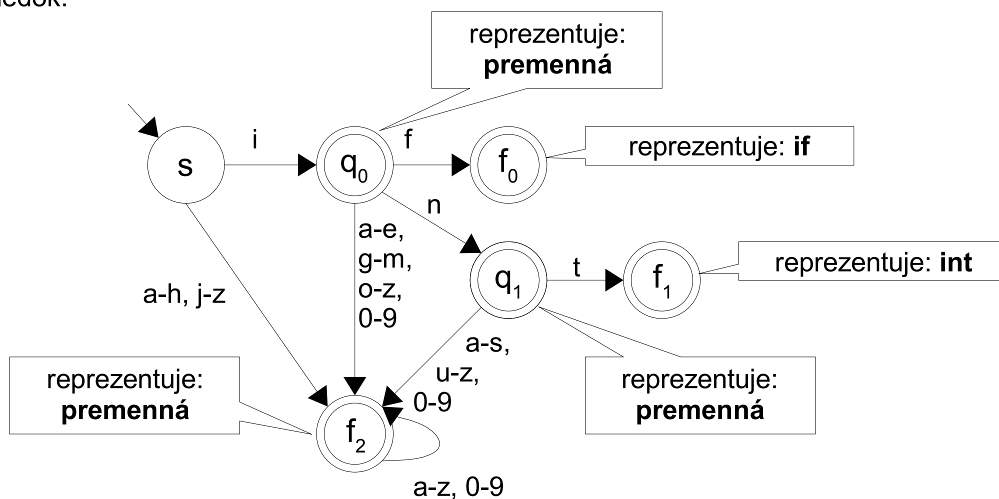

Bod 4, pre stavy určené predchádzajúcim krokom




Body 6-9 – nebola splnená podmienka v bode 4.6 a 5



Výsledok:




*Obrázok 12: Príklad algoritmu pre pridanie regulárneho výrazu reprezentovaného konečným automatom do deterministického konečného automatu – krok 4.*

# 4.2 Syntaktická analýza

Keďže zdrojový súbor, ktorý chcem analyzovať mohol by vytvorený podľa pravidiel najsilnejšej známej gramatiky – LR gramatiky, preto som si pre syntaktickú analýzu zvolil najkomplexnejšíi nástroj a to LR parser.

## 4.2.1 LR Parser

LR parser je založený na princípe rozšíreného zásobníkového automatu $M$ zo stavmi $Q = \{q_0, q_1, ..., q_k\}$, kde $q_0$ je štartovacím stavom a pracuje s dvomi tabuľkami, tzv. „Action part“ a „Go-to part“.

Go-to part tabuľka sa riadi podľa stavu zásobníkového automatu, ktorý ostane na vrchole zásobníku po použití redukčného pravidla a nonterminálu, ktorý má použité pravidlo na ľavej strane. Viac je to objasnené v nasledujúcom odstavci. Obsahom v tejto tabuľke je nový stav automatu, v ktorom sa má zásobníkový automat nachádzať po aplikovaní zmieneného pravidla. V tabuľke sa nachádzajú aj prázdne miesto, ktoré však nemajú význam, pretože sa nemôže stať, že po aplikovaní pravidla by sme dostali takú konfiguráciu pre túto tabuľku, kde by bolo prázdne miesto. Taká chyba, by bola odhalená dávno predtým.

Action part tabuľka sa riadi podľa aktuálneho stavu rozšíreného zásobníkového automatu a symbolu získaného od lexikálneho analyzátora na vstupe. Obsah v tejto tabuľke môže mať štyry významy a to:

1. ulož vstupný symbol na zásobník a zmeň aktuálny stav zásobníkového automatu na $q_n$.

2. redukuj vrchol zásobníka aplikovaním pravidla gramatiky a zmeň aktuálny stav zásobníkového automatu podľa Go-to part tabuľky, pričom použi stav, ktorý je na vrchole zásobníku po aplikovaní pravidla.
3. syntaktická analýza bola úspešná
4. chyba – neočakávaný symbol na vstupe.

#### 4.2.1.1 Algoritmus vytvorenia pravého rozkladu vstupného reťazca symbolov LR parserom

Tabuľky podľa ktorých analýza prebieha sú vytvorené podľa bezkontextovej LR gramatiky $G = (N, \Sigma, P, S)$ (bude vysvetlené ďalej) a pre vstupný reťazec symbolov platí $x = \Sigma^*$. Do rozšíreného zásobníkového automatu je vždy ukladaná dvojica $(u, v)$, kde $u \in (N \cup \Sigma)$ a $v \in Q$.

1. Uložím na zásobník dvojicu $(\$, q_0)$, kde $\$ \in \Sigma$ a vyjadruje ukončovací symbol.
2. Aktuálny stav automatu nastavím na $q_0$.
3. Prečítam aktuálny symbol na vstupe.
4. Nasledujem podľa pokynov v „Action part" tabuľke na priesečníku aktuálneho stavu automatu a prečítaného vstupného symbolu.
5. Opakujem od bodu 3, kým analýza neskončila chybou alebo úspechom.

#### 4.2.1.2 Konštrukcia LR tabuliek

Na skonštruovanie tauliek, podľa ktorých sa riadi syntakická analýza je možné použiť niekoľko rôznych algoritmov. Tu sú príklady niektorých základných:

- Simple LR (SLR) – výpočetne najnáročnejší, ale jednoduchý, pričom vzniká málo stavov rozšíreného zásobníkového automatu.
- Canonical LR – výpočetne menej náročný, ale vzniká veľký počet stavov rozšíreného zásobníkového automatu, čo spôsobuje spomalenie behu analyzátora.
- Lookahead (LALR) – výpočetne najmenej náročný, pričom vzniká rovnaký počet stavov ako použitím SLR algoritmu.

Zdalo by sa, že použitie LALR algoritmu by mohlo byť najvýhodnejšie, ale oproti jeho výhodám je jeho nevýhodou zložitosť. Po zvážení toho ako bude analyzátor primárne použitý, tj. ako nástroj syntaxou riadeného editora pod platformou Eclipse náročnosť na výpočet je menším problémom, čo sa dá redukovať ďalšími prostriedkami diskutovanými neskôr. Z tohoto dôvodu som zvolil algoritmus Simple LR.

##### Algoritmus Simple LR

Pracujem s bezkontextovou LR gramatikou $G = (N, \Sigma, P, S)$. Pred samotným algoritmom zavediem pojmy, ktoré budem v algoritme používať:

- *Item* – keď pre bezkontextovú gramatiku existuje pravidlo $A \rightarrow x$, kde x = yz, potom A → y•z je *Item*.

1. Rozšírim gramatiku o „štartovacie pravidlo“, tj. pravidlo, ktoré má na ľavej strane úplne nový nonterminál $S' \notin N$, a na pravej štartovací nonterminál gramatiky $S$, potom $G = (N \cup S', \Sigma, P \cup S' \rightarrow S , S')$.
2. Zistím *Closure*, ktorý sa zisťuje pre konkrétny *Item I* a to nasledovne
   2.1. Inicializujem $Closure(I) = \{I\}$
   2.2. Pre každý *Item* v $Closure(I)$ v tvare $A \rightarrow y \bullet Bz$ zistím, či existuje pravidlo B → x, ak áno pridám $B \rightarrow \bullet x$ do $Closure(I)$
   2.3. Ak bolo v predchádzajúcom kroku niečo pridané, opakujem od bodu 2.2
3. Zistím množinu $\Theta_G$ pre gramatiku *G,* čo je množina všetkých $X_n$ takých, ktoré pre všetky pravidla gramatiky v zápise $A \rightarrow Y_0 Y_1 \ldots Y_n$, sa dajú napísať ako $X_{n+1} = X_n Y_{n+1}$ pre $n \geq 0$, $X_n \in N \cup \Sigma$ a $X_0 = \varepsilon$
4. Zistím množiny *Contents*, ktoré sa určujú pre každý prvok množiny $\Theta_G$ nasledovným postupom:
   4.1. Inicializuj $Contents(\varepsilon) = Closure(S' \rightarrow \bullet S)$ a ostatné $Contents(x) = \emptyset$, kde $x \in \Theta_G - \{\varepsilon\}$.
   4.2. Pre každý *Item* v $Contents(b)$, kde $b \in \Theta_G$ v tvare $A \rightarrow y \bullet Bz$, kde $B \in N \cup \Sigma$ zistím, či existuje $bB \in \Theta_G$, ak áno, pridám $Closure( A \rightarrow y \bullet Bz)$ do $Contents(bB)$
   4.3.Ak bolo v predchádzajúcom kroku niečo pridané, opakujem od bodu 4.2
5. Riadky oboch tabuliek označím prvkami z množiny $\Theta_G$
6. Stĺpce „Action part“ tabuľky označím terminálmi gramatiky a stĺpce „Go-to part“ tabuľky označím nonterminálmi
7. Pre každé $I \in Contents(x)$, kde $x \in \Theta_G$
   7.1.ak $I = A \rightarrow n \bullet Bm$, kde $B \in N$, potom ak existuje $Contents(y)$, ktorý obsahuje *Item* $A \rightarrow nB \bullet m$ pridaj do „Go-to part“ tabuľky na priesečník *x* a *B* hodnotu *y.*
   7.2.ak $I = A \rightarrow n \bullet Bm$, kde $B \in \Sigma$, potom ak existuje $Contents(y)$, ktorý obsahuje *Item* $A \rightarrow nB \bullet m$ pridaj do „Action part“ tabuľky na priesečník *x* a *B* príkaz ulož vstuný symbol na zásobník a zmeň aktuálny stav zásobníkového automatu na *y.*
   7.3.ak $I = S' \rightarrow S \bullet$, potom pridaj do „Action part“ tabuľky na priesečník *x* a $\$ \in \Sigma$ (ukončovací symbol), že analýza úspešne skončila.
   7.4.ak $I = A \rightarrow y \bullet$ ($A \neq S'$), potom pre $a \in Fallow(A),$ čo je množina všetkých terminálov, ktoré môžu nasledovať priamo za *A* vo vetnej forme gramatiky, pridaj do „Action part“ tabuľky na priesečník *x* a *a* príkaz „redukuj vrchol zásobníka aplikovaním pravidla $A \rightarrow y$ a zmeň aktuálny stav zásobníkového automatu podľa Go-to part tabuľky, pričom použi stav, ktorý je na vrchole zásobníku po aplikovaní pravidla“.

## 4.3 Platforma Eclipse

Platforma Eclipse je vývojovým prostredím pre programovací jazyk Java s podporou zásuvných modulov. Tieto moduly umožňujú rozšíriť aplikáciu o nové funkcie, nastavenia a vzhľad. Dokonca je možné použiť základ Eclipse pre svoju aplikáciu, ako je tomu napríklad u aplikácie Blueprint Software Modeler a mnohých iných.

### 4.3.1 Eclipse – prehľad

Keďže sa v základe jedná o vývojové prostredie pre jazyk Java, dá sa odneho čakať, že obsahuje nejaký vyspelejší editor tohoto jazyka. Čo je v skutku pravda, ale rozhodne sa nejedná len o editor. Keď pôjdem postupne, musím najprv spomenúť perspektívy. Perspektíva sa v Eclipse rozumie definícia zobrazených panelov a ich rozvrhnutie. To znamená aj to, že sa na to isté môžem pozerať iným pohľadom, čo sa aj využíva napríklad u takej Java perspektívy - mám zobrazený hlavne zdrojový kód, hierarchické usporiadanie súborov s ktorými pracujem a poprípade konzolu alebo súhrn varovaní a chýb... Po prepnutí do perspektívy Debuggeru sa rozloženie zmení práve tak, aby som mal k dispozícii výpisy obsahu premenných, aktuálne zanorenie programu... Skrátka perspektívy len definujú spôsob usporiadania jednotlivých častí.

Tieto časti sa dajú primárne rozlíšiť na pracovný priestor (workspace), pohľady, aktívne ovládacie prvky a editory. Workspace zobrazuje súbory (fyzické alebo logické) s ktorými môžeme pracovať, editory ponúkajú možnosť editovať zmienené súbory, aktívnymi ovládacími prvkami sa rozumejú tlačidlá na panely nástrojov, položky menu a formuláre. Pohľady plnia viac-menej informatívny charakter, častokrát sú závislé od aktuálne označeného súboru v pracovnom priestore alebo aktuálnej polohy kurzoru v editovanom súbore. Toto je základný prehľad z pohľadu uživateľa.

### 4.3.2 Vývoj zásuvných modulov

Pre vývoj zásuvných modulov je doporučené použiť nástroje poskytované priamo vývojármi. Vytvorenie nového modulu začneme tým, keď necháme vytvoriť nový projekt a z ponuky vyberieme Plug-in Project. Wizard nás prevedie nastavením všetkých potrebných častí, dokonca si môžeme nechať vytvoriť vzorový projekt. Skôr by som ale upozornil na dôležité časti. Ďaľej sa obmedzím na to ako postupovať pri vytváraní editora, nebudú tu zahrnuté ostatné časti platformy.

#### 4.3.2.1 plugin.xml

Dôležitým a častokrát prehliadaným alebo zabúdaným je informácia o verzií Eclipse, pre ktorú bol modul naprogramovaný. Udáva sa ako informácia pre preprocesor, direktíva s označením eclipse a s parametrom version. Ďaľšou dôležitou časťou je zaradenie modulu do kategórie akú rozširuje, čo je v prípade editoru atribut point elementu extension nastavený na hodnotu org.eclipse.ui.editors. Potom

už nasleduje element pre konkrétnu skupinu, čiže v tomto prípade element editor, u ktorého meno znamená meno, ktoré bude viditeľné hlavne v kontextovom menu pri voľbe Open with.... Ďaľej tu je možnosť nastavenia prípony súborov, ktoré budeme môcť otvárať, ale čo je najdoležitejšie, tak nastavenie triedy, ktorá bude editor reprezentovať a ktorá povinne dedí od org.eclipse.ui.editors.text.TextEditor. V spomínanej triede je dôležité nastaviť source viewer configuration, čo je trieda ktorá sa stará o to, čo uživateľ vidí a ako editor komunikuje na jeho akcie (napr. pri dvojkliku) a document provider, ktorý je akýmsi oddeľovačom od práce s dokumentom na najnižšej úrovni (čítanie a zápis na disk) a ponúka možnosť práce s dokumentom na objektovej úrovni.

#### 4.3.2.2 META-INF\MANIFEST.MF

Ďalším dôležitým súborom je MANIFEST.MF, v ktorom je treba správne definovať class path, je tu možné zadať súbor s lokalizáciou, ale hlavne je tu definovaná trieda aktivátora pre zdroje používané v module.

#### 4.3.2.3 SourceViewerConfiguration a FileDocumentProvider

Už som spomínal, že trieda, ktorá reprezentuje náš editor potrebuje mať nastavený správny source viewer configuration, aby dokázal komunikovať s uživateľom plne v našej réžií. Ďalšou užitočnou triedou je trieda *FileDocumentProvider*, resp. jej potomok, kde môžeme dokument rozdeliť na určité časti a potom každá z týchto častí je samostatná. Musia sa voliť také časti, ktoré sa nebudú nikdy prekrývať ani do seba nijak zasahovať. Keď už dokument takto rozdelíme, môžeme to v potomkovy triedy *SourceViewerConfiguration* využiť, dielčie časti ďaľej spracovávať. Pokiaľ dokument nijak nerozdelíme na menšie časti môžme spracovávať celý dokument, rozdiel je v tom, že pri rozdelení je každá časť definovaná offsetom (od začiatku dokumentu) a dĺžkou, bez rozdelenia by sme pracovali s jednou veľkou časťou, čo môže byť mnohokrát zbytočné.

Pre rozdelenie dokumentu na časti sa používa trieda *FastPartitioner*, ktorej instancia sa predáva dokumentu (práve v potomkovy triedy *FileDocumentProvider* v metóde *createDocument()*) metódou *setDocumentPartitioner()*. Na druhej strane aj instancii triedy *FastPartitioner* musíme pripojiť dokument metódou *connect()*. Hovoril som však o instancii triedy *FastPartitioner*, ale jej konštruktor očakáva dva parametre. Ako prvý očakáva instanciu triedy, ktorá implementuje rozhranie *IPartitionTokenScanner*, čoho existuje len jediný implementátor a to trieda *RuleBasedPartitionScanner*, ktorej treba predať pole pravidiel (instancie triedy implementujúcej rozhranie *IRule*). Druhý atribút očakáva pole reťazcov, ktoré môže vrátiť skener predaný ako prvý parameter. To ako rozdeliť dokument zatiaľ vieme, ale máme ešte možnosť spracovávať jednotlivé časti ďalej. Pokiaľ to pre nás má význam, čo väčšinou má, môžme v potomkovy triedy *SourceViewerConfiguration* preimplementovať metódu *getPresentationReconciler()*. Po vytvorení instancie triedy *PresentationReconciler*, tejto instancií nastavím damager (pre ošetrovanie zmien) a repairer (pre spracovanie pri načítaní). Pri nastavovaní sú očakávané opäť dva parametre, prvý je

instancia triedy implementujúcej rozhranie *IPresentationDamager*/*IPresentationRepairer* na čo môžem použiť základnú implementáciu triedou *DefaultDamagerRepairer* a pri jej vytváraní použijem instanciu triedy implementujúcej rozhranie *ITokenScanner*, čo najčastejšie býva RuleBasedScanner, ktorému stačí predať pole pravidiel (triedy implementujúce rozhranie *IRule*).

Teraz už len ostáva postarať sa o to, aby sa jednotlivé lexémy zvýrazňovali. K tomuto je potreba predávať každému pravidlu *IToken*, ktorý ako data uchováva instanciu triedy *TextAttribute*.

# 5 Závěr

Touto prácou som prispel k projektu LISSOM, ktorý sa zaoberá vývojom jazyka pre popis mikroprocesorov. Vytvoril som konfigurovateľný syntaktický analyzátor, ktorý dokáže prijať LR gramatiku, čo je najsilnejšia bezkontextová gramatika, podľa tejto gramatiky vytvoriť LR parser a realizovať simuláciu vytvorenia syntaktického stromu vstupného textu. Analyzátor som zakomponoval do vývojového prostredia Eclipse, tak, že som vytvoril syntaxou riadený editor pre jazyk ISAC a jazyk generovaného assembleru.

Pre ďaľší vývoj projektu je v pláne realizovať blokovú syntaktickú analýzu, ktorá by urýchlila spracovanie vstupného textu. Mám v pláne pridať výkonnejší parser, ktorý bude síce spracovávať jednoduchšiu gramatiku, ale rýchlejšie a výber parsra bude na základe gramatiky. Analyzátor, je od zásuvného modulu absolútne nezávislý, preto by som len samotnému analyzátoru chcel pridať podporu pre sémantickú analýzu. Tým by bolo možné vytvoriť konfigurovateľný kompilátor.

# Literatura

[1] Češka, M.: *Teoretická informatika, Učební texty*. Brno, Skriptum FIT VUT Brno 2002.

[2] Lukáš, R., Meduna, A.: *Prednášky k predmetu VYP*, FIT VUT Brno 2006

[3] Spell, B., *Java Programujeme profesionálně*

[4] *Oficiálne stránky programovacieho jazyka Java.* http://java.sun.com

[5] *Java Almanac*.http://www.javaalmanac.com

[6] *Oficiálne stránky Eclipse*. http://www.eclipse.org

# Seznam příloh

Příloha 1. CD so zdrojovými kódmi