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

Bakalářská práce

Generování magických pravidel z ozdobeného logického programu

Plzeň 2016 Jiří Láska

Místo této strany bude zadání práce.

Prohlášení

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

V Plzni dne 4. května 2016

Jiří Láska

Abstract

Generation of magic rules from adorned logic program

This bachelor's thesis discusses magic rules generating. It is an essential part of the Magic Sets method. The Magic Sets method is a relatively complex optimization for logic programs' evaluation. A part of this thesis is an application written in the Java language. The application implements the very algorithm of a magic rules generator. The application processes a given text input.

Keywords: magic rules, logic programming, evaluation optimization

Abstrakt

Generování magických pravidel z ozdobeného logického programu

Tato bakalářská práce se zaobírá konstrukcí magických pravidel. Je to důležitá součást Metody magických množin. Metoda magických množin je poměrně komplexní optimalizace vyhodnocování logických programů. Součástí práce je aplikace v Java jazyce. Aplikace implementuje samotný algoritmus konstrukce magických pravidel, pracující s daným textovým vstupem.

Klíčová slova: magická pravidla, logické programování, optimalizace vyhodnocování

Obsah

1 Úvod

Logický jazyk zachycuje svět v relacích. Typické uplatnění nacházíme ve znalostních systémech.

V logickém jazyce pokládáme dotazy vzhledem k určitému modelu (databázím faktů a pravidel). Při vyhodnocování takového dotazu se musí prohledat strom řešení, který může býti dosti košatý a přitom některé větve nemusí k řešení vůbec vést. Metoda magických množin [6] se snaží takovéto nadbytečné vyhodnocování uzlů ve stromu omezit.

Tato práce se zabývá jen jednou částí, z kterých se skládá Metoda magických množin.

Rozebereme si použitou terminologii, kterou použijeme při návrhu algoritmů tvorby magických pravidel, tzn. popis toho, co je to logický jazyk, rovněž si definujeme potřebné konstrukce pro formální popis, přičemž budeme stavět nad formálními jazyky.

Také si popíšeme hlavní části realizace, tím myslíme samotnou implementaci algoritmů konstrukce magických pravidel a načítání vstupních dat. Zároveň si přiblížíme zvolený funkcionální styl programování.

2 Použitá terminologie

Popíšeme si z jakých elementů se skládá logický jazyk, kterým se budeme zabývat, a také ostatní pojmy z jiných oblastí, které při návrhu budeme potřebovat. Z těch dále v této kapitole definujeme konstrukce vytvořené pro potřeby popisování algoritmů tvorby magických pravidel.

2.1 Logický jazyk

V tomto oddíle definujeme pojmy užívané v logickém programování.

Definice 2.1 *Symbol* je řetězec znaků Ω^+ s nosnou abecedou Ω obsahující malé i velké alfanumerické znaky a podtržítko.

$$\Omega = \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, \_\}$$

Symbol nezačíná numerickým znakem. Dále pak použijeme jisté třídy symbolů:

- začínající malým písmenem `[a-z][a-zA-Z0-9_]*`
- začínající velkým písmenem `[A-Z][a-zA-Z0-9_]*`
- podtržítko `_`

Příklad 2.1 Symboly s malým počátečním písmenem:
`symbol, predicate, fIRST_SMALL_1`

Příklad 2.2 Symboly s velkým počátečním písmenem:
`Variable, Symbol_2`

Příklad 2.3 Symbolem není:
`123, 0not_symbol, 1_abc, @$#, !alphabeta, +, _underline`

Definice 2.2 *Konstanta* je buď číslo nebo řetězec.

- *číslo* – je buď:

  - 0
  - celé – řetězec numerických znaků nezačínající nulou s možným znaménkem mínus, vyskytujícím se před samotnými numerickými znaky
    `-?[1-9][0-9]*`
  - desetinné – to je celé číslo rozšířené o desetinnou část, část je oddělená desetinnou tečkou
    `-?(0|[1-9][0-9]*)\.[0-9]+`

- *řetězec* – je řetězec libovolných znaků ohraničený apostrofy.

Příklad 2.4 Celá čísla mohou vypadat:
`123, -58, 9`

Příklad 2.5 Desetinná čísla mohou vypadat:
`1.0, 2.3, -0.47`

Příklad 2.6 Číslem není:
`-0, 21.38.6, 123a`

Příklad 2.7 Ukázka řetězců:
`'string', 'with characters: #@\'`

Příklad 2.8 Řetězcem není:
`'not string' next part', 'sepa' 'rated'`

Definice 2.3 *Proměnná* je jakýkoliv symbol začínající velkým písmenem. Výjimečný případ je podtržítko, to je také proměnná a má speciální význam, označuje *anonymní proměnnou*. Každý jednotlivý výskyt tohoto symbolu je brán jako unikátní proměnná.

Příklad 2.9 Ukázka proměnných:
`Var1, X, _`

Definice 2.4 *Term* je buď konstanta nebo proměnná.

Příklad 2.10 Ukázka termů:
`Var, 'const', _, -2.56`

Příklad 2.11 Termem není:
`small(X, Y), one('string')`

Definice 2.5 *Predikátový symbol* je symbol začínající malým písmenem.

Příklad 2.12 `predicate, rule, sMALL, q_1`

Definice 2.6 *Atom* má tvar: predikátový symbol následovaný seznamem termů. *Základní atom* je takový atom, kde všechny termy jsou konstantami; neobsahuje proměnnou.

Příklad 2.13 Ukázka atomů:
`predicate(X, Y), atom(X, 'const')`

Příklad 2.14 Základní atom:
`basic_atom(2.5,  'const')`

Definice 2.7 *Výskyt predikátu* je buď atom hlavičky pravidla nebo pozitivní literál v těle pravidla. Začíná predikátovým symbolem a je následován seznamem argumentů o délce, která odpovídá aritě[1] daného predikátu. Jako argumenty dosazujeme termy. Výskyt predikátu, který neobsahuje žádné proměnné, se označuje *základní výskyt predikátu*.

Příklad 2.15 `rule(X, Y), related('abc','def', Z)`

Definice 2.8 *Literál* – literály jsou dvojího druhu:

- pozitivní – výskyt predikátu v těle pravidla
- negativní – negace pozitivního; to je výskyt predikátu v těle pravidla uvozený klíčovým slovem `not`.

[1] u predikátů se arita nazývá také místnost

Příklad 2.16 `positive(A), not negative(B)`

Definice 2.9 *Konjunkce literálů* představuje seznam literálů, literály jsou oddělené čárkou.

Příklad 2.17 Toto jako celek je konjunkce:
`atom1(A), atom2(B, C), not atom3('const')`

Definice 2.10 *Hlavička pravidla* je tvořena jedním atomem.

Definice 2.11 *Tělo pravidla* je tvořeno buď jedním literálem nebo konjunkcí literálů, tělo je na začátku od hlavičky odděleno sekvencí `:-`.

Příklad 2.18 `:- p(X), q(X, Y)`

Definice 2.12 *Pravidlo* se skládá z hlavičky a těla, celé je ukončeno tečkou.

Příklad 2.19 Ukázka pravidla:
```
rule1(Object) :- relation1(Object, A), property1(A).
```

Definice 2.13 *Bezpečné pravidlo* je takové pravidlo, kde všechny proměnné vyskytující se v hlavičce pravidla a v negativních literálech v těle pravidla se také vyskytují v nějakém pozitivním literálu v těle pravidla.

Příklad 2.20 Ukázka bezpečného pravidla:
```
safe_rule(A, B) :- rel(A, C), fact1(C), fact2(B).
```

Příklad 2.21 Není bezpečným pravidlem:
```
r(X, Y, Z) :- f(X), q(Z, X).
q(U, V) :- not f(U), g(V).
```

Definice 2.14 *Fakt* je podobný pravidlu, je to degenerované pravidlo. Fakt se skládá pouze z hlavičky, která je zakončená tečkou.

Příklad 2.22 `fact(object).`

Definice 2.15 *Dotaz* je degenerované pravidlo, které neobsahuje hlavičku. Začíná sekvencí `?-`, pro naše účely bude dotaz tvořen jedním pozitivním literálem.

Příklad 2.23 `?- relation1(Object1, Object2).`

Pokud původní dotaz toto nesplňuje, vytvoříme pro něj dodatečné pravidlo, které v těle obsahuje původní dotaz a má patřičnou hlavičku, tu pak použijeme v dotazu.

Příklad 2.24

```
rule2(Obj1, Obj2) :-
    rule1(Obj1), relation2(Obj1, Obj2), property2(Obj2).
?- rule2(Objt1, Obj2).
```

Definice 2.16 *Predikát* je určen predikátovým symbolem a místností neboli počtem argumentů, definován je pak množinou pravidel nebo faktů s odpovídající hlavičkou.

Definice 2.17 *Extenzionální predikát* je takový predikát, který je definován množinou obsahující jen fakta. Dále existují také *vestavěné predikáty*, ke kterým se v častých případech chováme jako k extenzionálním.

Definice 2.18 *Intenzionální predikát* je takový predikát, který je definován množinou obsahující jen pravidla.

2.2 Formální jazyk

Formální jazyk [5] použijeme jako podpůrnou část při konstrukci algoritmu tvorby magických pravidel.

Definice 2.19 *Abeceda* je jakákoliv neprázdná konečně spočetná množina.

Příklad 2.25

$$\Sigma = \{a, b, c, \cdots, z\}$$

Definice 2.20 *Slovo* w nad abecedou Σ je posloupnost symbolů z dané abecedy o délce $n = |w|$. Taktéž označováno *řetězec*.

$$w = w_1 w_2 \dots w_n \quad , \quad \forall w_i \in \Sigma$$

Dále se samotným symbolem abecedy $c \in \Sigma$ budeme také zacházet jako se slovem o délce $|c| = 1$.

Příklad 2.26

$$w = slovo$$

Definice 2.21 *Prázdné slovo*, značené ϵ, má délku nula $|\epsilon| = 0$, je to prázdná posloupnost, tj. neobsahuje žádné prvky.

Definice 2.22 *Uzávěr abecedy* značíme Σ^*, uzávěrem abecedy se míní množina všech slov nad danou abecedou.

Příklad 2.27 Mějme abecedu $\Sigma = \{a, b, c\}$, což je nějaká neprázdná konečně spočetná množina. Poté uzávěr abecedy je

$$\Sigma^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ...\}$$

Definice 2.23 *Řetězení slov* $s \in \Sigma^*$ a $p \in \Pi^*$ zapisujeme jako $s \circ p \in (\Sigma \cup \Pi)^*$, přičemž platí:

$$\begin{aligned} s &= s_1 s_2 \dots s_n \quad p = p_1 p_2 \dots p_m \\ s &\circ p = s_1 s_2 \dots s_n p_1 p_2 \dots p_m \end{aligned}$$

Příklad 2.28 Mějme slova $s = ahoj$, $p = světe$, pak jejich zřetězení

$$s \circ p = ahojsvěte$$

Definice 2.24 *Formální jazyk* L nad abecedou Σ je libovolná podmnožina uzávěru abecedy $L \subset \Sigma^*$, je to nějaká množina slov konečné délky nad určitou abecedou.

Příklad 2.29 Pro abecedu $\Sigma = \{a, e, l, m, o, s\}$, můžeme mít jazyk

$$L = \{ema, mele, maso\}$$

Definice 2.25 *Zřetězení jazyků* obsahuje všechna řetězení dvojic slov kartézkého součinu původních jazyků. Označujeme jej $L_1 \circ L_2$, pak

$$L_1 \circ L_2 = \{v \circ w \mid v \in L_1 \wedge w \in L_2\}$$

Příklad 2.30 Mějme jazyky $L = \{Tomáš, Martin\}$,
$K = \{Novák, Omáčka\}$, jejich zřetězení

$$L \circ K = \{TomášNovák, TomášOmáčka, MartinNovák, MartinOmáčka\}$$

2.3 Jazyk slov s unikátním výskytem symbolů

Popíšeme si jazyk, který dále budeme potřebovat pro popis operací při tvorbě těl magických pravidel. Jazyk budeme značit $U_\Sigma \subset \Sigma^*$, případně jeho obdobu $U_\Sigma^+ = U_\Sigma \setminus \{\epsilon\}$.

$$U_\Sigma = \{\epsilon\} \cup \left\{c \circ w \mid c \in \Sigma \wedge w \in U_{\Sigma \setminus \{c\}}\right\}$$

Příklad 2.31 Pro abecedu $\Sigma = \{a, b, c\}$, takový jazyk vypadá:

$$U_\Sigma = \{\epsilon, a, b, c, ab, ac, ba, bc, ca, cb, abc, acb, bac, bca, cab, cba\}$$

2.4 Projekce slova do abecedy

Projekci slova w do abecedy Π budeme značit $P_\Pi(w)$ a definujeme ji takto:

$$P_\Pi(w) = p_\Pi(w_1) \circ p_\Pi(w_2) \circ \cdots \circ p_\Pi(w_n)$$

Přičemž $p_\Pi(w_i)$ rozhoduje o tom, zda se slovo w_i ponechá nebo nahradí prázdným.

$$p_\Pi(c) = \begin{cases} c & c \in \Pi \\ \epsilon & c \notin \Pi \end{cases}$$

Příklad 2.32 Pro abecedu $\Pi = \{a, \ldots, z\} \setminus \{a, e, i, o, u\}$ a slovo $w = loremipsum$, projekce vypadá

$$P_{\Pi}(w) = lrmpsm$$

2.5 Řetězení slov při zachování unikátnosti symbolů

Toto speciální řetězení budeme značit $a \circ_U b$, kde $a \in U_{\Sigma}^{+}, b \in U_{\Pi}^{+}$ jsou nějaká slova s unikátním výskytem symbolů. Pro toto zřetězení platí:

$$a \circ_U b \in \left(U_{\Sigma}^{+} \circ_U U_{\Pi}^{+}\right) \quad , \quad \left(U_{\Sigma}^{+} \circ_U U_{\Pi}^{+}\right) = \left(U_{\Sigma}^{+} \circ U_{\Pi \setminus \Sigma}^{+}\right)$$
$$a \circ_U b = a \circ P_{\Pi \setminus \Sigma}(b)$$

Příklad 2.33 Pro slova $a = lorem$, $b = ipsum$, zřetězení vypadá:

$$a \circ_U b = loremipsu$$

Povšimněte si chybějícího symbolu m.

3 Metoda magických množin

Předpokládejme dotaz v logickém jazyce skládající se z jednoho predikátu, některé argumenty jsou navázané na konstanty a jiné obsahují proměnné.

Pro každý argument v dotazu platí, že k jeho jednotlivým hodnotám lze přiřadit třídu rozkladu prostoru řešení. Pro každý argument máme jeden takový rozklad prostoru řešení. Jestliže se v dotazu vyskytují konstanty, tak jejich přítomnost nám pevně určuje třídy v daných rozkladech. Řešení se nachází v daných třídách rozkladu, takže řešení se musí nacházet v jejich průniku, to nám jistým způsobem omezuje prostor řešení.

Metoda magických množin je optimalizace vyhodnocování dotazu položeném v logickém jazyce, která přihlíží na toto omezení prostoru řešení konstantními argumenty. Bez optimalizace se běžně prohledává celý prostor řešení i mimo průnik těchto tříd příslušejících k daným konstantním argumentům.

Metoda magických množin přepisuje program vůči tvaru konkrétního dotazu, tvarem se zde rozumí navázanost argumentů. Nezáleží na konkrétních hodnotách argumentů, nýbrž na tom, zda je argument navázán na konstantu či proměnnou.

Program se přepisuje do magické formy, vznikají další pravidla tzv. magická. Ty jsou při vyhodnocování předřazena původním originálním, díky tomu dochází k brzkému vyhodnocování těch částí pravidel, které odpovídají argumentům v dotazu navázaným na konstanty, tak dochází k ořezávání prostoru řešení.

Metoda se skládá ze dvou hlavních kroků:

- Konstrukce ozdobeného programu
- Konstrukce magického programu

3.1 Konstrukce ozdobeného programu

Ozdobený program se skládá z ozdobených pravidel. Pro každé pravidlo v původním programu vznikne $2^n - 1$ ozdobených pravidel, kde n je místnost predikátu, jež se vyskytuje v hlavičce pravidla. Ozdobené pravidlo má ozdobenou hlavičku a každý výskyt intezionálního predikátu v těle pravidla. Ozdobený výskyt predikátu má rozšířený predikátový symbol o příponu, označující, které argumenty jsou navázané[1]. Přípona se skládá z podtržítka a řetězce znaků nad abecedou $\{b, f\}$, počet znaků (bez podtržítka) odpovídá místnosti predikátu. Přípona má tvar `_[bf]{n}` pro n-místný predikát.

3.2 Graf předávání informace

Zachycuje, jak se předává informace od navázaných proměnných v hlavičce pravidla. Je to ohodnocený orientovaný graf, orientace hran směřuje od hlavičky k výskytům predikátů v těle pravidla, ohodnocení jsou množiny proměnných, skrze které prochází informace od jednoho výskytu predikátu k druhému.

3.3 Konstrukce magického programu

Konstrukce magické programu se skládá z několika částí, jednou z nich se tato práce zabývá. Je to právě generování magických pravidel.

Pro každé ozdobené pravidlo vytváříme množinu magických pravidel o mohutnosti n, kde n je počet intenzionálních predikátů v těle ozdobeného pravidla, které mají některé proměnné a jsou pro dané pravidlo navázané.

3.4 Konstrukce magického pravidla

3.4.1 Použitá notace

Ohledně grafů předávání informace budeme užívat následujícího značení:

- $V(G)$ – množina vrcholů grafu G
- $E(G)$ – množina hran grafu G
- $f \to t$ – hrana z vrcholu f do vrcholu t

[1] to je taková proměnná, která zde představuje argument, co v prostředí výskytu predikátu je navázána na nějakou hodnotu

Pro logická pravidla máme značení:

- $\{Body(R)\}$ – množina výskytů predikátů v těle pravidla R, tzn. abeceda sestavená ze znaků slova $Body(R)$
- $Body(R)$ – tělo pravidla R skládající se z unikátních výskytů predikátů s daným pořadím neboli slovo nad abecedou $\{Body(R)\}$

3.4.2 Vstupní a výstupní data

Tvorba magických pravidel potřebuje ozdobená pravidla a příslušné grafy předávání informace. Pravidlo mající v hlavičce n argumentů lze ozdobit až 2^n způsoby, ovšem případ, kdy jsou všechny argumenty volné, se vynechává, efektivně je případů $2^n - 1$. Pro každou takovou instanci ozdobení existuje odpovídající graf předávání informace.

Ozdobený program a dodatečné informace budou rozděleny na množinu dvojic (ozdobené pravidlo, graf předávání informace), které budou jednotlivě zpracovány algoritmem pro generování magických pravidel.

Z každé dvojice (ozdobené pravidlo, graf předávání informace) vznikne množina magických pravidel, tento soubor množin bude buď přidán k původním datům na vstupu, a nebo uložen zvlášť. Celý výstupní soubor množin rozdělen na množin, které budou připojeny ke korespondujícím vstupním dvojicím.

3.4.3 Algoritmy konstrukce

Pro tvorbu magických pravidel užijeme dvou algoritmů, pro zpracování jedné hrany grafu předávání informace [Alg. 2] a hlavní pro zpracování celého magické pravidla [Alg. 1], který využívá hranového algoritmu.

Algoritmus prochází výskyty predikátů v těle ozdobeného pravidla. Jestliže dochází k předávání informace (tzn. existuje hrana v grafu) směrem k danému intenzionálnímu predikátu, pro který sestrojujeme magické pravidlo, zpracujeme tuto hranu pomocí [Alg. 2]. Získáme různé části těla sestrojovaného magického pravidla, ty jsou oddělené, protože chceme, aby docházelo k vyhodnocování určitých částí dříve než k vyhodnocování jiných. Mezivýsledky spojujeme do těchto oddělených částí nezávisle na sobě a až po průchodu celým tělem ozdobeného pravidla spojíme výsledné části do jednoho celku, tak nám vznikne tělo magického pravidla.

Algoritmus 1: Konstrukce magického pravidla

Vstup: ozdobené pravidlo r a jeho graf předávání informace g_r
Výstup: magické pravidlo m_p pro výskyt predikátu p v těle pravidla r

Postup:

1 $P_{sh} \leftarrow \emptyset$ `/* speciální hlavičkový predikát */`
2 $P_m \leftarrow \emptyset$ `/* magické predikáty */`
3 $P_o \leftarrow \emptyset$ `/* původní predikáty a extenzionální či vestavěné predikáty */`

`/* každý výskyt predikátu v těle pravidla */`
4 **foreach** $q \in \{Body(r)\}$ **do**
  `/* z kterého vede hrana do predikátu` p `*/`
5 $e = (q \rightarrow p)$
6 **if** $e \in E(g_r)$ **then**
    `/* zpracujeme hranu` e `algoritmem zpracování hrany [Algoritmus 2] */`
7 $(\partial P_{sh}, \partial P_m, \partial P_o) \leftarrow$ `Algoritmus2`(r, g_r, e)
8 $P_{sh} \leftarrow P_{sh} \circ_U \partial P_{sh}$
9 $P_m \leftarrow P_m \circ_U \partial P_m$
10 $P_o \leftarrow P_o \circ_U \partial P_o$
  end
end
`/* Sestavení těla magického pravidla m_p */`
11 $Body(m\_p) \leftarrow P_{sh} \circ_U P_m \circ_U P_o$

Pro jednotlivé hrany grafu předávání informace sestrojujeme části těla magického pravidla. Rozlišujeme tři případy na základě predikátu, od kterého se předává informace:

1. speciální hlavičkový – nahradíme jeho magickou formou
2. intenzionální – ponecháme původní formu a přidáme jeho magickou, pokud existuje
3. extenzionální či vestavěný – ponecháme původní formu a [Alg. 2] rekurzivně spustíme s tímto predikátem

---

Algoritmus 2: Zpracování hrany

Vstup: ozdobené pravidlo r, jeho graf předávání informace g_r a hrana $q \to p \in E(g_r)$

Výstup: části těla P_{sh}, P_m, P_o

Postup:

```
1   P_sh ← ∅ /* speciální hlavičkový predikát        */
2   P_m ← ∅ /* magické predikáty                    */
3   P_o ← ∅ /* původní predikáty, extenzionální či vestavěné
     predikáty                                     */
4   switch druh predikátu q do
5       case speciální hlavičkový do
6           P_sh ← P_sh ∘_U (m_r)
        end
7       case intenzionální do
8           P_o ← P_o ∘_U (q)
9           if ∃ vázaný argument obsažený v q then
10              P_m ← P_m ∘_U (m_q)
            end
        end
11      case extenzionální či vestavěný do
12          P_o ← P_o ∘_U (q)
13          foreach (s → q) ∈ E(g_r) do
14              (p_sh, p_m, p_o) ← Algoritmus2(r, g_r, (s → q))
15              P_sh ← P_sh ∘_U p_sh
16              P_m ← P_m ∘_U p_m
17              P_o ← P_o ∘_U p_o
            end
        end
    end
```

Pro lepší názornost si ukažme možný průchod algoritmem.

Příklad 3.1 Nechť máme ozdobené pravidlo:

```
preceding_sibling_bbbb(Line1, Line2, Element1, Element2) :-
  self_bb(Line1, Element1),
  xml(Line1, Element1, Order1, Level),
  self_bb(Line2, Element2),
  xml(Line2, Element2, Order2, Level),
  lt(Order2, Order1),
  parent_bfbf(Line1, Line3, Element1, Element3),
  parent_bbbb(Line2, Line3, Element2, Element3).
```

A k němu příslušný graf předávání informace, viz diagram 3.1.

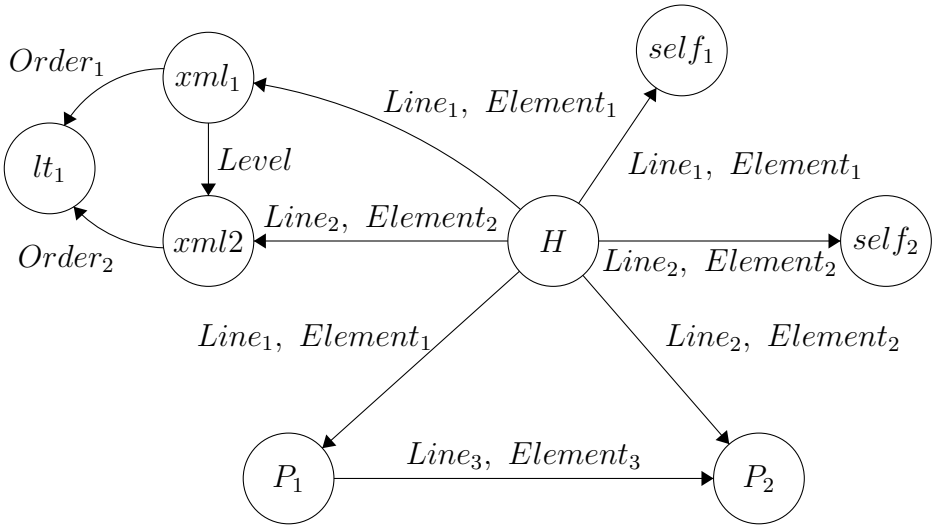

Diagram 3.1: Graf předávání informace (příklad 3.1)

Dolní indexy označují pořadí výskytu daného predikátu v těle pravidla, to platí, když si odmyslíme přípony ozdobení. H značí speciální hlavičkový predikát nebo-li `preceding_sibling`, P je pak zkratka pro `parent`.

V těle máme čtyři výskyty intenzionálních predikátů:

1. `self_bb(Line1, Element1)`
2. `self_bb(Line2, Element2)`
3. `parent_bfbf(Line1, Line3, Element1, Element3)`
4. `parent_bbbb(Line2, Line3, Element2, Element3)`

Pro každý výskyt provedeme konstrukci magického pravidla (algoritmus 1).

Pro výskyt predikátu `self_bb(Line1, Element1)` podle řádek **4** až **6** v algoritmu 1 budeme zpracovávat jen jednu hranu $H \rightarrow self_1$, tomu odpovídají řádky **7** až **10** v algoritmu zpracování hrany (algoritmus 2), dle řádky **4** rozlišíme další zpracování podle typu predikátu, z kterého hrana vychází. V tomto případě se jedná o speciální hlavičkový predikát, ten zpracujeme na řádkách **5** až **6**. Nakonec sestrojíme tělo magického pravidla na řádce **11** v algoritmu 1.

Výsledné pravidlo bude vypadat následovně:

```
m_self_bb(Line1, Element1) :-
  m_preceding_sibling_bbbb(Line1, Line2, Element1, Element2).
```

Pro další dva výskyty predikátů `self_bb(Line2, Element2)` a `parent_bfbf(Line1, Line3, Element1, Element3)` nastává stejná situace. Zpracování posledního výskytu intenzionálního predikátu `parent_bbbb(Line2, Line3, Element2, Element3)` bude zajímavější, proto si to rozebereme podrobněji.

Podle řádek **4** až **6** v algoritmu 1 máme pro zpracování dvě hrany $P_1 \rightarrow P_2$ a $H \rightarrow P_2$. Vezměme hranu $P_1 \rightarrow P_2$, na řádce **4** v algoritmu 2 zpracování hrany dojde k určení druhu predikátu, z kterého vychází hrana. Nyní je to intenzionální predikát P_1, aplikujeme případ počínající řádkou **7**, řádka **8** zapisuje původní nemagický predikát do samostatné části P_o vyhrazené na tento typ predikátů. Dále podle řádek **9** a **10** je ověřeno, zda existuje magická forma predikátu, který jsme si zaznamenali v předešlém kroku. Jestliže magická forma existuje, zapíšeme ji do vyhrazené části P_m. Poté zpracuje hranu $H \rightarrow P_2$ stejným způsobem jako v předchozích případech. Sestrojíme tělo magického pravidla spojením částí P_{sh}, P_m a P_o (dle řádky **11** v algoritmu 1).

Pravidlo vypadá následovně:

```
m_parent_bbbb(Line2, Line3, Element2, Element3) :-
  m_preceding_sibling_bbbb(Line1, Line2, Element1, Element2),
  m_parent_bfbf(Line1, Element1),
  parent_bfbf(Line1, Line3, Element1, Element3).
```

V následujícím příkladě 3.2 si ukážeme zbývající větve algoritmu, které zahrnují rekurzivní zpracování hran.

Příklad 3.2 Mějme ozdobené pravidlo:

```
intersection_ffbb(Line1, Line2, Element, Level1) :-
  xml(Line1, Element, Order, Level1),
```

```
xml(Line2, Element, Order, Level1),
dec(Level2, Level1),
intersection_bbfb(Line1, Line2, _, Level2).
```

a k němu příslušný graf předávání informace, viz diagram 3.2:

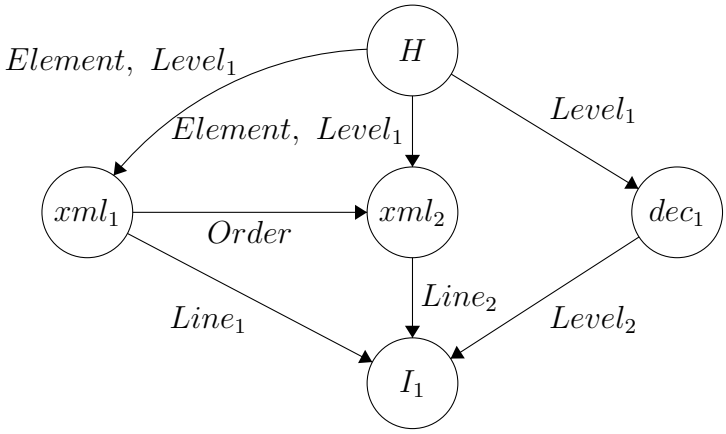

Diagram 3.2: Graf předávání informace (příklad 3.2)

Opět H značí speciální hlavičkový predikát `intersection_ffbb(Line1, Line2, Element, Level1)` a I_1 je jediný ozdobený predikát v těle pravidla. Zde vznikne jen jedno magické pravidlo pro výskyt predikátu `intersection_bbfb(Line1, Line2, _, Level2)`. Ze začátku máme ke zpracování tři hrany $xml_1 \to I_1$, $xml_2 \to I_1$ a $dec_1 \to I_1$.

Zpracujme hranu $xml_1 \to I_1$, xml_1 je extenzionální predikát (příp. vestavěný), podle řádky **4** algoritmu 2 dojde k vyhodnocení větve na řádcích **11** až **17**, zde dochází k rekurzivnímu zpracování hran v grafu předávání informace. Nejdříve na řádku **12** přidáme původní tvar extenzionálního predikátu do vyhrazené části P_o pro sestavování těla magického pravidla a pak vezmeme predikát z něhož vychází hrana (v tomto případě xml_1) a spustíme algoritmus 2 zpracování hran nad hranami vcházející do tohoto predikátu, výsledky rekurzivních volání pochopitelně přidáme do našich částečných mezivýsledků. Dojde ke zpracování hrany $H \to xml_1$ a do části těla P_{sh} nám přibude magická forma speciálního hlavičkového predikátu.

Další hrana ke zpracování určená algoritmem 1 je $xml_2 \to I_1$, i v tomto případě dojde podobně také k rekurzi. Nejdříve do části P_o přidáme predikát xml_2 a pak v rekurzivním zpracování dojde na hrany $xml_1 \to xml_2$ a $H \to xml_2$, kdy při zpracovávání hrany $xml_1 \to xml_2$ rekurze pokračuje s hranou $H \to xml_1$. Přidávané predikáty se v daných částí již vyskytují, takže speciální operace na slučování, definovanou v oddíle 2.5, je vlastně zahodí.

Obdobně u poslední zbývající hrany $dec_1 \rightarrow I_1$ přidáme do části P_o predikát dec_1 podle řádky **12** v algoritmu 2 a rekurzivně zpracováváme hranu $H \rightarrow dec_1$, přidáme magickou formu speciálního hlavičkového predikátu do části P_{sh}, jelikož přidávání se děje speciální operací, nedochází k duplikaci predikátů, část P_{sh} zůstane stejná.

Nakonec podle řádku **11** algoritmu 1 vše sloučíme do jednoho těla a výsledek je následovný:

```
m_intersection_bbfb(Line1, Line2, Level2) :-
  m_intersection_ffbb(Element, Level1),
  xml(Line1, Element, Order, Level1),
  xml(Line2, Element, Order, Level1),
  dec(Level2, Level1).
```

4 Funkcionální paradigma v Javě

Java je převážně jazyk objektově orientovaný, má velice dlouhou historii. Jeho vývoj je tak více konzervativnější a jazyk Java nebyl navrhnut proto, aby v něm šlo programovat ve všech možných paradigmatech. Samozřejmě jako každý programovací jazyk se inspiruje jinými jazyky a jednou za čas dojde k velkému rozhodnutí a takto rozšířené jazyky přiberou do svého standardu nové způsoby, nová paradigmata. Jeden takový velký krok udělala *Java 8*, umožňuje nám psát programy funkcionálním způsobem.

Jedna z věcí, které takovéto jazyky s historií musí brát v potaz, je zpětná kompatibilita. Proto také jazyky s dlouho historií velké kroky nedělají často, velice snadno se může stát, že přerostou v „monstra“. Avšak v případě funkcionálního stylu programování je to tak odlišný přístup k řešení problémů, kdy je opravdu výhodné ho použít, že i přes různé možné potíže se toto paradigma dostalo do Javy.

V této kapitole se podíváme na implementační detaily toho, jak je funkcionální paradigma zakomponováno do Javy.

4.1 Lambda funkce

Lambda funkce jsou základním prvkem ve funkcionálním programování, jsou to anonymní funkce. Ve verzi *Java 7* a starších se vyskytují oblasti, kde by taková lambda funkce bylo vhodná. Staré verze Javy to řeší anonymními třídami, velice známý případ se nám naskýtá při programovaní GUI ve *Swing*u, když chceme přidat odezvu na stisknutí tlačítka. V tomto případě anonymní třída implementuje rozhraní `ActionListener`, vidět je to v ukázce 4.1.

Z pohledu zpětné kompatibility *Java 8* zavádí mechanismus pro lepší zakomponování lambda funkcí. Víceméně jde o to, aby lambda funkce byly *first-class* objektem jazyka, tedy, že lambda funkce jde použít na místech, kde to čekáme.

V našem případě jde anonymní třídu implementující `ActionListener` nahradit lambda funkcí, to je vidět v ukázce 4.2. Když to zobecníme, tak lambda funkce může použít všude tam, kde se po anonymní třídě požaduje jen jedna metoda, pak kompilátor Javy ztotožní lambda funkci s touto metodou.

```
JButton button = new JButton("Button");

button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Clicked");
    }
});
```

Ukázka 4.1: Anonymní třída pro obsluhu tlačítka

```
JButton button = new JButton("Button");

button.addActionListener(e -> System.out.println("Clicked"));
```

Ukázka 4.2: Obsluha tlačítka lambda funkcí

Povšimněte si, že bylo použito zkráceného zápisu, který je omezen na jeden výraz. Když použijeme blok se složenými závorkami, tak nejsme nijak omezeni. Zkrácený zápis má ještě jednu vlastnost, kdybychom psali funkci s jedním `return` příkazem, tak stačí použít zkrácený zápis a za šipku lambda funkce napsat jen výraz nacházející se v `return` příkazu.

4.2 Stream API

Ve funkcionálních programovacích jazycích máme běžně k dispozici funkce vyššího řádu, to jsou funkce, které mají jako argumenty další funkce. Časté je zpracování různých kolekcí [2] jejich průchodem a při něm se zpracovávají jednotlivé položky. Standardní funkce vyššího řádu jsou např.: `map` a `filter`. Další z věcí je, že z pravidla potřebujeme provést několik operací za sebou nad těmito kolekcemi či výsledky předešlých operací, potřebujeme operace vlastně řetězit. V jiných programovacích jazycích než je Java bývá syntaxe na to uzpůsobená, jelikož je to velice častá záležitost a pro funkcionální styl programování přímo stěžejní.

V *Java 8* je v JDK balík `java.util.function`, který zastřešuje práci s funkcemi. Řetězení operací se dá provést skrze metodu `Function.compose` a její obdoby. To není nijak přívětivé pro zápis delších funkcionálních kusů kódu.

Java to řeší poměrně elegantním způsobem a sice pro velké množství standardních kolekcí zavádí *Stream API*, to je zpřístupněné přes metody instance `stream()` a `parallelStream()`. Vlastně je to metoda rozhraní `Collection`, takže jediný standardní druh kolekce, pro který není *Stream API* tak přímočaré, je asociativní mapa.

Stream API nám umožňuje použít tečkovou notaci a tak jednoduše řetězit jednotlivé operace s kolekcí, znázorněno je to v ukázce 4.3.

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9);

OptionalDouble avg =
    numbers.stream()
           .filter(x -> x % 2 == 0)
           .mapToInt(x -> x*x)
           .average();

System.out.println(avg);
```

Ukázka 4.3: Řetězení operací

4.3 Reference na metodu

Další zjednodušení zápisu a souvisí to také s tím, že funkce jsou nyní v Javě *first-class* objektem, je umožněno vytvářením referencí na metody. Reference na metody můžeme jednoduše použít na místech, kde bychom jinak použili lambda funkce. Používá se operátoru `::`, na který jsme zvyklý z jiných programovacích jazyků a leckdy znamená něco podobného. V ukázce 4.4 lze vidět použití.

Existují čtyři varianty referencí na metodu. Popišme si je:

1. `Class::staticMethod` reference na statickou metodu – tato varianta je nejpřímočařejší, reference se chová jako lambda funkce, která přijímá stejný seznam argumentů:
   `(arg1, arg2) -> Class.staticMethod(arg1, arg2)`

```
List<String> input = Arrays.asList("13","24","129");

List<String> output =
    input.stream()
           .map(Integer::parseInt)
           .filter(x -> x % 2 != 0)
           .map(Object::toString)
           .collect(Collectors.toList());

System.out.println(output);
```

Ukázka 4.4: Reference na metody

2. `instance::method` reference na normální metodu (metodu instance) – toto je nestatický protějšek předešlé varianty. Tentokrát reference přijímá o jeden argument navíc a tím je samotná instance třídy (nebo chcete-li reference na instanci), nad kterým má být metoda vykonána. Je to první argument v seznamu. Ve smyslu lambda funkce to vypadá následovně:
   `(obj, arg1, arg2) -> obj.method(arg1, arg2)`

3. `Class::method` reference na normální metodu přes třídní identifikátor – to je alternativní způsob zápisu reference na normální metodu. Jednou z výhod je, že v kombinovaném výrazu je jasně vidět, jaká metoda se má na mysli. Kód je tak přehlednější. Má to i své úskalí, pokud změníme okolní kód a ve výsledku pracujeme s prvky jiné (nekompatibilní) třídy, budeme muset takové reference přepisovat na novou třídu. Odpovídající lambda funkce je stejná jako předchozí:
   `(obj, arg1, arg2) -> obj.method(arg1, arg2)`

4. `Class::new` reference na konstruktor – poslední variantou je speciální reference. Na konstruktor lze pohlížet jako na statickou funkci třídy, jen její volání má zvláštní syntaxi. Reference na konstruktor nás zbavuje nutnosti použít tento syntaktický cukr. Lambda funkce, která odpovídá takové referenci, vypadá následovně:
   `() -> { return new Class();}`

5 Programová realizace

Tato práce zpracovává jednu část celé metody magických množin, předešlé části již byly nějakým způsobem zpracovány. Z důvodů návaznosti byl zvolen stejný programovací jazyk *Java*.

Vstup pro generování magických pravidel odpovídá výstupu práce zabývající se grafy předávání informace [3]. Aplikace se skládá z několika částí, jmenovitě to jsou:

- práce se vstupem a výstupem
- vnitřní reprezentace dat
- převod textového vstupu do vnitřní reprezentace
- hlavní část

Jejich vzájemný vztah je vidět na UML diagramu 5.1.

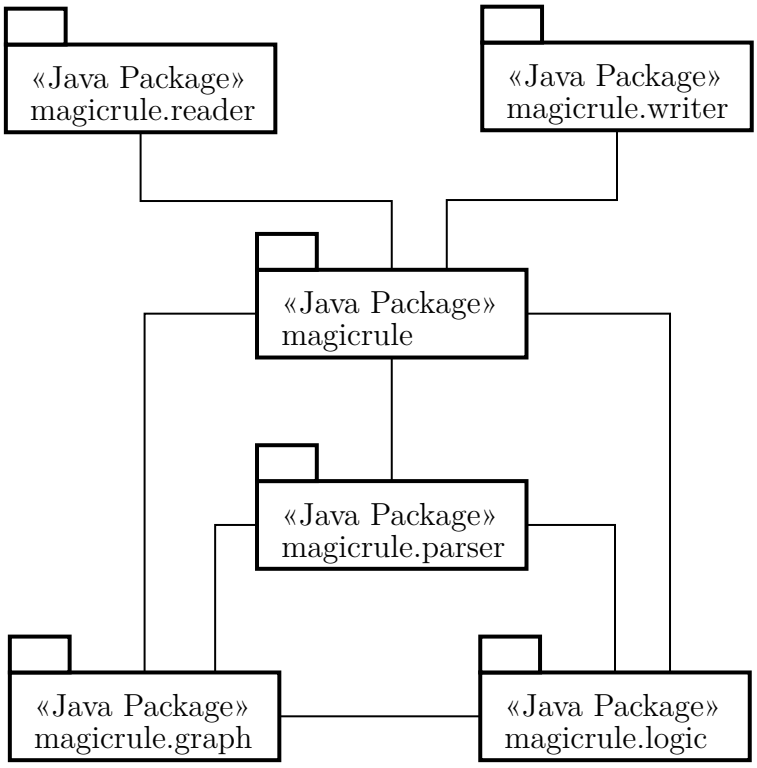

Diagram 5.1: Celkový pohled

5.1 Vstupní data

Vstupem je adresář, kde se nachází veškerá pravidla logického programu. Struktura je koncipována tak, že v hlavním adresáři se nachází podadresáře. Jeden podadresář je pro jedno pravidlo. V každém podadresáři jsou textové soubory. Pro jednu variantu ozdobení existuje v podadresáři příslušný textový soubor.

Textové soubory mají obsah rozdělen do několika sekcí. Každá sekce má hlavičku a obsahu, kde hlavička se skládá z jedné řádky. Tyto dvě části jsou odděleny jednou prázdnou řádkou, jednotlivé sekce jsou pak odděleny dvěma řádkami (příp. větším počtem). Soubory na vstupu obsahují následující sekce:

- `Original rule:` – původní pravidlo
- `Modified rule: (bfbf)` – upravené pravidlo
- `SIP:` – úplný graf předávání informace
- `FSIP:` – konečný graf předávání informace
- `Adorned rule:` – ozdobené pravidlo

Pro naše účely jsou stěžejní sekce ozdobené pravidlo a úplný graf předávání informace.

V ukázce 5.1 je vidět, jak je graf předávání informace reprezentován v textové podobě. Každá řádka obsahu sekce představuje jednu hranu grafu, ta se skládá ze začátku a konce a je navíc ohodnocená, začátek obsahuje ve složených závorkách cestu vždy vedoucí od speciálního hlavičkového predikátu do predikátu odkud vychází hrana definována na daném řádku. Cesta je zapsána ve tvaru seznamu odděleného čárkami, až na poslední prvek, který je oddělen středníkem. Následuje samotná hrana, která je tvořena pomlčkami a na konci většítkem, znaky dohromady tak představují šipku. Do hrany je vepsáno ohodnocení, což je seznam proměnných předávající informaci, prvky jsou zase oddělovány čárkami. Šipka je ohodnocením rozdělena na dvě části, obě mají minimální délku takovou, že se v nich vyskytují alespoň dvě pomlčky. Konec hrany je potom jen jeden predikát, v kterém hrana končí.

Sekce ozdobeného pravidla, jak je vidět v ukázce 5.2, obsahuje jen jedno pravidlo, které má v těle ozdobené jen intenzionální predikáty. Ozdobení predikátů v těle využijeme právě pro rozpoznání jejich druhů při tvorbě magických pravidel.

5.2 Výstupní data

Výstupem je adresářová struktura shodná s adresářovou strukturou vstupu. Jednotlivé soubory pro různá ozdobení pravidel jsou rozšířeny o sekci obsahující magická pravidla, jednotlivá pravidla jsou oddělena jednou prázdnou řádkou, jak je vidět v ukázce 5.3.

5.3 Struktura aplikace

V této části uvedeme, jak korespondují jednotlivé *java* balíky aplikace s dříve zmíněným rozvrhnutím aplikace.

Hlavní balík `magicrule` řeší zpracování celého vstupu a také samotnou tvorbu magických pravidel (to se děje v souboru `Record.java`), což bude dále podrobněji rozebráno. Mimo to v souboru `Util.java` spolu se souborem `Pair.java` jsou naimplementovány dodatečné metody a struktury z oblasti funkcionální programování, které jsou standardní v jiných programovacích jazycích, avšak v *Java 8* nejsou v základu dostupné. Jednou takovou běžnou funkcionalitou je funkce *zip*, u které jsem kód převzal z velmi známého zdroje [4], další je běžná datová struktura páru.

Balíky `magicrule.reader` a `magicrule.writer` se starají o čtení a zápis sekcí, pracují s instancemi třídy `Record`.

Další důležitou součástí je samotné reprezentování dat vnitřními strukturami, kde data jsou ve formě pravidel a grafů předávání informace, to je zajištěno pomocí balíků dvou `magicrule.logic` a `magicrule.graph`, přičemž druhý využívá třídy `Symbol` definované v tom prvním.

Vstup je poměrně složitý (tím jsou myšleny syntaxe obsahů sekcí) a tudíž jeho parsování je řešeno v samostatném balíku `magicrule.parser`. Samotné parsování vstupu je poměrně komplikovaná záležitost, proto ji probereme v následujícím oddíle podrobněji.

5.4 Parsování vstupu

Samotný proces je rozložen do dvou fází [1]: lexikální analýzy a syntaktické analýzy. Parser je navržen dostatečně obecným způsobem, aby mohl zpracovávat jak pravidla logického programu, tak i grafy předávání informace, za pomoci příslušných definic lexerů a konečných automatů pro syntaktickou analýzu. Ve zjednodušeném UML diagramu 5.2 jsou vztahy nastíněny.

Ve třídě `Definitions` jsou veškeré potřebné instance pro lexikální analýzu (množiny `Lexer`ů) a syntaktickou analýzu (asociativní mapa přechodů konečných automatů). V našem případě je lexikální analýza rozdělena na dvě fáze, takže ve výsledku vstup prochází třemi fázemi:

1. `lexicalize()` – lexikální analýza (rozkouskování vstupu na lexémy)
2. `tokenize()` – lexikální analýza (příprava tokenů pro další fázi)
3. `parse()` – syntaktická analýza a načítání do vnitřní struktury

Popišme si jednotlivé instance a jejich funkce. Dvě množiny `Lexer`ů pro lexikální analýzu logických pravidel `logicProgramLexers` a pro lexikální analýzu grafů předávání informace `graphLexers`. Dvě asociativní mapy přechodů pro syntaktickou analýzou prováděnou konečnými automaty, jedna z map `ruleParserTransitions` je pro syntaktickou analýzu logického pravidla a druhá z map `graphParserTransitions` pro grafy předávání informace. Výčtové typy `Logic` a `Graph` určují jednotlivé druhy tokenů na vstupu. Výčtové typy `RuleParserState` a `GraphParserState` představují stavy konečných automatů.

```
SIP:
    {following_sibling_h} ------------------ Line1, Element1 --> self.1
    {following_sibling_h} ------------------ Line1, Element1 --> xml.1
    {following_sibling_h} ------------------ Line2, Element2 --> self.2
    {following_sibling_h} ------------------ Line2, Element2 --> xml.2
    {following_sibling_h} ------------------ Line1, Element1 --> parent.1
    {following_sibling_h} ------------------ Line2, Element2 --> parent.2
    {following_sibling_h; xml.1} ---------- Level ------------> xml.2
    {following_sibling_h; xml.1} ---------- Order1 ----------> gt.1
    {following_sibling_h, xml.1; xml.2} -- Order2 ----------> gt.1
    {following_sibling_h; parent.1} ------ Line3, Element3 --> parent.2
```

Ukázka 5.1: Úplný graf předávání informace

```
Adorned rule:

following_sibling_bbbb(Line1, Line2, Element1, Element2) :-
  self_bb(Line1, Element1),
  xml(Line1, Element1, Order1, Level),
  self_bb(Line2, Element2),
  xml(Line2, Element2, Order2, Level),
  gt(Order2, Order1),
  parent_bfbf(Line1, Line3, Element1, Element3),
  parent_bbbb(Line2, Line3, Element2, Element3).
```

Ukázka 5.2: Ozdobené pravidlo

```
Magic rules:

m_self_bb(Line1, Element1) :-
  m_following_sibling_bbbb(Line1, Line2, Element1, Element2).

m_self_bb(Line2, Element2) :-
  m_following_sibling_bbbb(Line1, Line2, Element1, Element2).

m_parent_bfbf(Line1, Element1) :-
  m_following_sibling_bbbb(Line1, Line2, Element1, Element2).

m_parent_bbbb(Line2, Line3, Element2, Element3) :-
  m_following_sibling_bbbb(Line1, Line2, Element1, Element2),
  m_parent_bfbf(Line1, Element1),
  parent_bfbf(Line1, Line3, Element1, Element3).
```

Ukázka 5.3: Magická pravidla

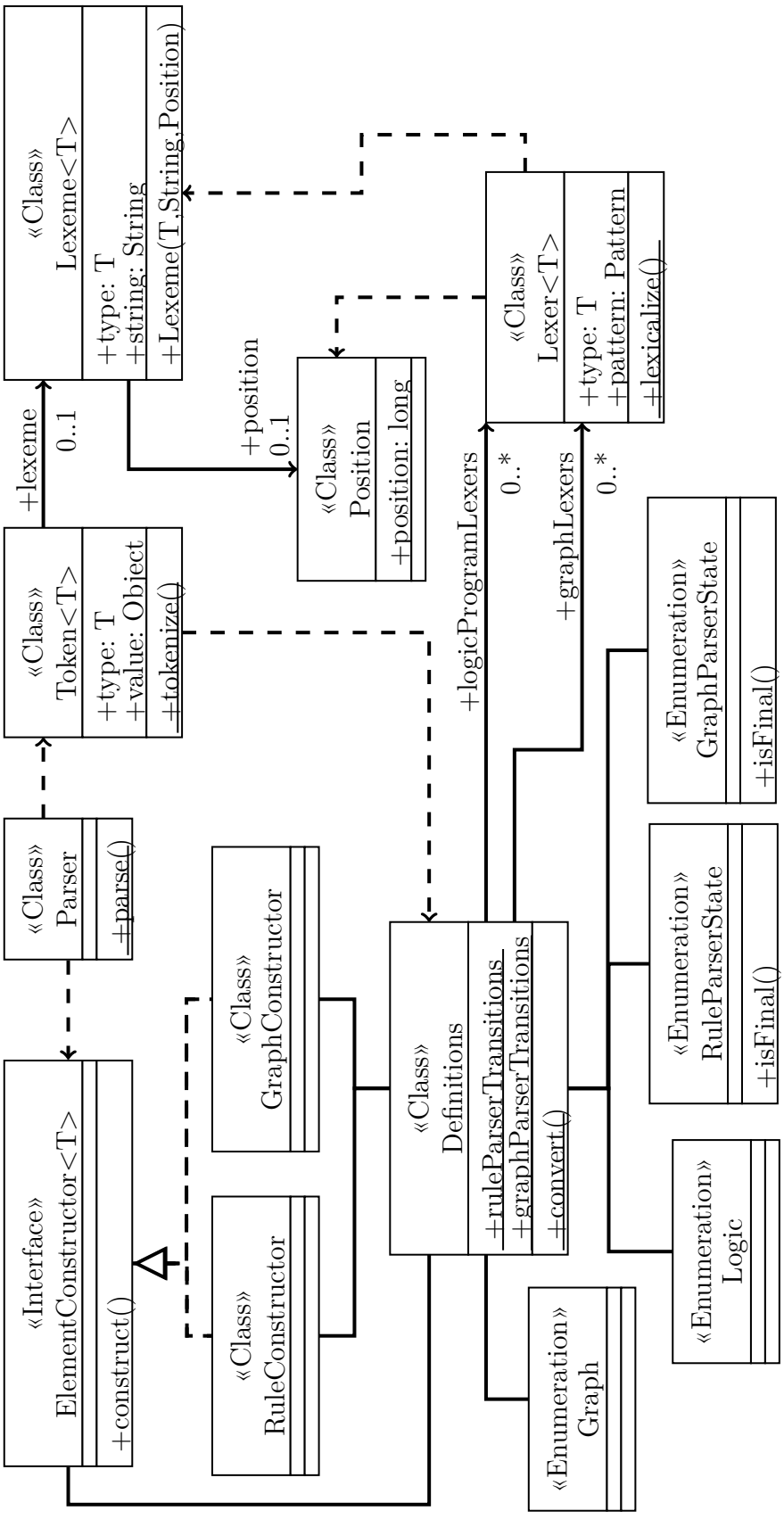

Diagram 5.2: Balík `magicrule.parser`

Konečné automaty při své činnosti sestavují vnitřní reprezentaci dat na vstupu s pomocí rozhraní `ElementConstructor`, jehož metoda `construct()` nám vrátí výsledek konstruování reprezentace. Pro oba dva druhy vstupních dat existují implementující třídy a to jsou `RuleConstructor` spolu s `GraphConstructor`.

Za zmínku také stojí metoda `convert()`, která definuje jakým způsobem se mají lexémy (textový řetězec) převádět na tokeny.

5.4.1 Lexikální analýza

Textový vstup se převádí do lexémů množinou instancí třídy `Lexer`, které efektivně jsou regulárními výrazy pro jednotlivé druhy lexémů. Tyto `Lexery` mají dva charakteristické atributy: již zmíněný regulární výraz a hodnota z výčtového typu označující druh lexému (stejný výčtový typ je použit i u tokenů odpovídajících těmto lexémům). `Lexer` pak dokáže určit, zda se na vstupu nachází daný druh lexému či ne a příp. následně ho načíst.

Lexémy (instance třídy `Lexeme`) jsou jednoduše řetězce označené druhem lexému. Pro rozumné hlášení chyb na vstupu si lexémy drží další informaci, kterou je pozici ve vstupním proudu, při samotném vyhození výjimky je pak na základě pozice a délky řetězce vrácen výňatek vstupu pro rozpoznání problému na vstupu.

Parser je navržen tak, že instance `Lexer`ů musí být po dvojicích vzájemně výlučné, tím jsou myšleny jejich regulární výrazy. Pro části vstupu, které mají být ignorovány, je vyhrazena speciální instance `Lexeru`, která má druh lexému nastaven na hodnotu `null`.

5.4.2 Syntaktická analýza

Ještě před samotnou syntaktickou analýzou dochází k tokenizaci lexémů. `Token` se od `Lexeme` liší tím, že tokeny obsahují dekódované objekty, které jsou v lexémech reprezentované pouze řetězci.

Při syntaktické analýze rovnou vzniká vnitřní reprezentace dat, proces provádí konečný automat. Každý přechod automatu má přiřazenou činnost pro budování vnitřní reprezentace, ty jsou uvedeny v podobě lambda funkcí, takže při přechodech, kdy nedostáváme žádná užitečná data uvádíme prázdné lambda funkce.

Množina přechodů je implementována jako asociativní mapa, jejíž klíče jsou uspořádané dvojice: aktuální stav konečného automatu a druh tokenu na vstupu. A hodnoty mapy jsou uspořádané dvojice: stav konečného automatu, do kterého se má přejít a akce, která má být vykonána. Akce je lambda funkce, která dostává dva argumenty, prvním argumentem je instance implementující `ElementConstructor`, kterým sestavujeme vnitřní reprezentaci a druhým argumentem je aktuální token, který aktivoval daný přechod.

Druh tokenu i stav konečného automatu jsou výčtové typy. Jejich jednotlivé hodnoty jsou pojmenovány systematicky, a to z důvodu jednoduchého hlášení chyb syntaxe. Název stavu odpovídá tomu, jaké tokeny jsou v danou chvíli očekávány, přechody odpovídají různým druhům tokenů.

V následujících stavových diagramech se budeme držet stejné konvence.

5.4.3 Logické pravidlo

Pravidlo se skládá z hlavičky a těla, což je seznam predikátů, případně může být na predikát aplikován operátor *not*, v konečném automatu provádějící syntaktickou analýzu máme pro to dvě hlavní větve (viz diagram 5.3).

Větev od stavu H_{sym} do stavu B analyzuje hlavičku pravidla včetně levé implikace oddělující hlavičku od těla, malý cyklus u stavu H_{arg} načítá argumenty v hlavičce pravidla. Následně ze stavu B vychází ony dvě větve, lépe řečeno cykly, na zpracování prvků v těle pravidla. V těchto větvích jsou malé cykly u stavů P_{arg} resp. $\overline{P_{arg}}$, které načítají argumenty u výskytu predikátů v těle pravidla pro pozitivní literál resp. negativní literál.

Predikátové symboly Sym se liší od obecné definice tím, že neobsahují speciální symboly v našem případě klíčové slovo *not*.

Přechody označené Arg sdružují reálně víc přechodů pro různé typy tokenů a to jsou:

- běžné proměnné
- konstanty
  - čísla
  - řetězce

Přechod $\_Arg$ oproti Arg navíc přijímá anonymní proměnné.

Po přijmutí tečky na konci pravidla přejdeme do jediného koncového stavu θ, z něho nevychází žádné přechody, není očekáván žádný další vstup, je očekáváno pouze jedno pravidlo. Čárka na místo tečky nám značí, že pravidlo nekončí a přecházíme tak do stavu B na místo θ, tzn. že očekáváme další prvek v těle pravidla.

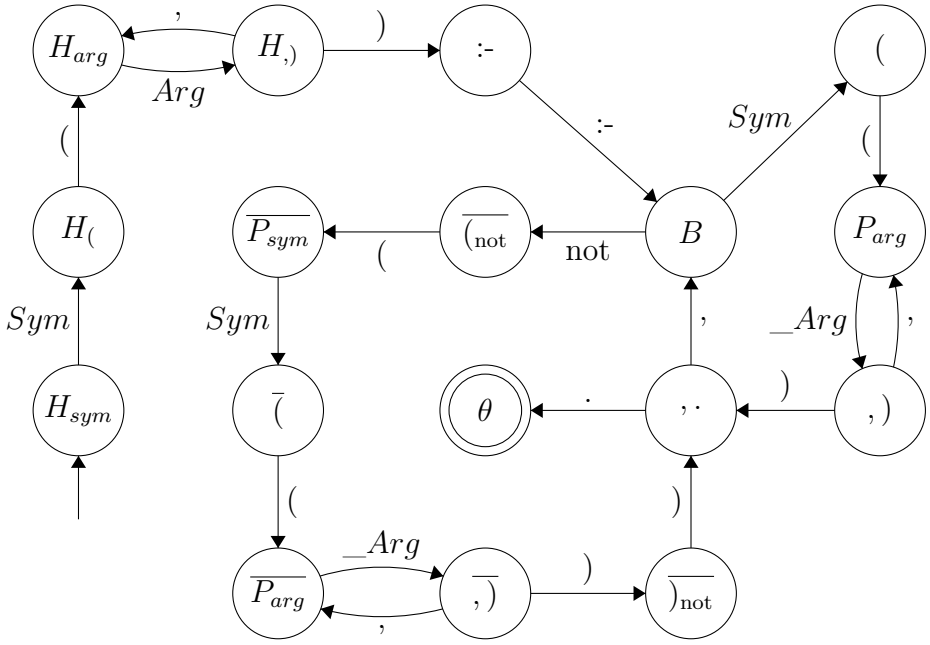

Diagram 5.3: Stavový diagram pro syntaktickou analýzu pravidla

5.4.4 Graf předávání informace

Konečný automat na diagramu 5.4 umožňuje načítat libovolný počet hran grafu, dokonce přijímá i nulový počet hran. Případ, kdy je graf předávání informace prázdný, nastává výjimečně, protože zpracováváme pouze pravidla ohodnocená takovým způsobem, že alespoň jeden argument je navázaný. Může se však stát, že na pozicích, které jsou navázané, se vyskytují místo proměnných konstanty.

Symboly jsou pro načítání grafu definované trochu jinak. Stále platí, že symboly označující proměnné začínají velkým písmenem (ve stavovém diagramu označeno Sym_{var}) a symboly označující predikáty začínají malým písmenem. Rozdíl se nachází v symbolech reprezentujících výskyty predikátů v těle pravidla, v grafu musí být mírně odlišné od těch, co jsou v logických pravidlech. V grafu předávání informace jsou totiž uvedené pouze samotné predikátové symboly bez seznamu argumentů, aby šlo odlišit o jaký výskyt predikátu se v těle jedná, symboly nesou navíc informaci o pořadí výskytu v těle, proto jsou ve stavovém diagramu označeny Sym_{ord}.

Symboly mají navíc příponu začínající tečkou a za ní následuje pořadí výskytu v číselné podobě, symboly tak odpovídají tomu, co nalezneme v sekci vstupního souboru `Modified rule: (***)` (hvězdičky značí libovolný řetězec ozdobení). Rozdíl od normálních predikátových symbolů je ten, že v `Lexer`u pro predikátové symboly v grafu musí být navíc povolena tečka.

Ve stavech P_i a P_n jsou očekávány predikáty, které tvoří cestu od speciálního hlavičkového predikátu k predikátu, který představuje začátek hrany. V malém cyklu u stavu Var se načítají proměnné předávající informaci a stav $Tail$ načítá predikát na konci hrany.

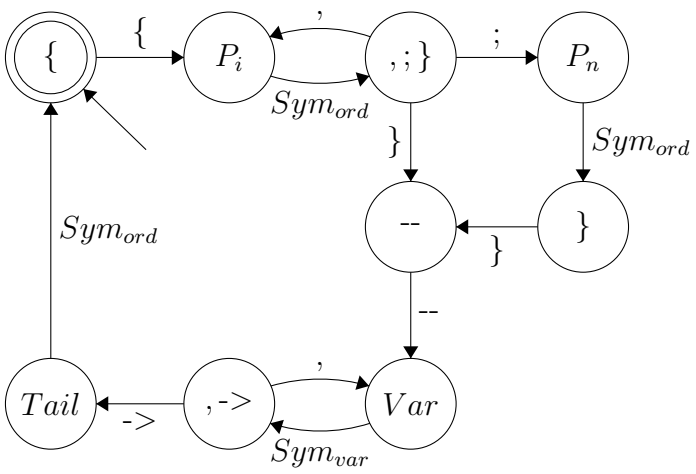

Diagram 5.4: Stavový diagram pro syntaktickou analýzu grafu

5.5 Vnitřní reprezentace dat

Pro konstrukci magických pravidel bylo potřeba vytvořit dvě struktury šité na míru, nepočítaje běžné kombinování struktur ve funkcionálním programování, které bylo zajištěno třídou `Pair` a standardními strukturami v Javě. Ty dvě struktury byly pro logická pravidla a grafy předávání informace, instance těchto struktur jsou vytvářené parserem při čtení vstupu.

5.5.1 Graf předávání informace

Struktura není navržená obecně ve smyslu operací nad grafem. Víceméně do grafu lze jen přidávat hrany, což rozhodně plní účel, neboť grafem nemáme potřebu jakkoliv manipulovat, potřebujeme ho jen načíst ze vstupu a získat z něj relevantní informace. Informace ve struktuře jsou uloženy redundantně a to proto, abychom mohli různé vztahy mezi prvky v grafu získávat efektivním rychlým způsobem. V UML diagramu 5.5 vidíme z jakých tříd se struktura skládá:

- `SIPGraph` – samotný graf předávání informace, UML diagram je zjednodušen a neukazuje jaké operace pro získávání patřičných informacích máme k dispozici, je jich poměrně dost.

- `Predicate` – třída představující vrcholy grafu
- `PassingInformation` – třída pro hrany grafu, které jsou ohodnocené seznamem proměnných skrze, které se předává informace

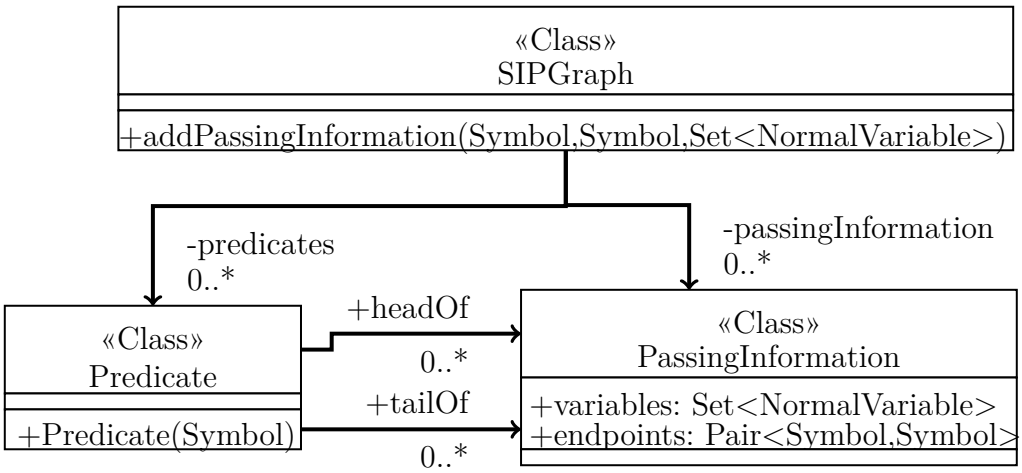

Diagram 5.5: Třídy pro graf předávání informace

5.5.2 Logické pravidlo

Prvky, s kterými se zapisují logické programy, jsou v poměrně hodně hiearchizovaných vztazích, to se odráží v UML diagramu 5.6. Třída `PositionalAtom` nám slouží k převodu mezi symboly v logických pravidlech na symboly v grafech předávání informace, instance této třídy si oproti normální instanci `Atom`u drží informaci o pozici výskytu v těle pravidla. Všechny třídy v hierarchii od `Rule` až po `Constant` a `NormalVariable` mají překryté metody pro porovnání (tzn. `equals(Object)` a `hashCode()`), toho se využívá při určování duplikátů v těle pravidla. `AnonymousVariable` nám umožňuje nastavením atributu `distinct` měnit chování porovnání. Při hodnotě `true` je každá instance odlišné od ostatních, při opačné hodnotě jsou si pak anonymní proměnné rovny.

Třída `Symbol` je hojně použita i v jiných částech aplikace, zejména pak ve vnitřní reprezentaci grafu předávání informace.

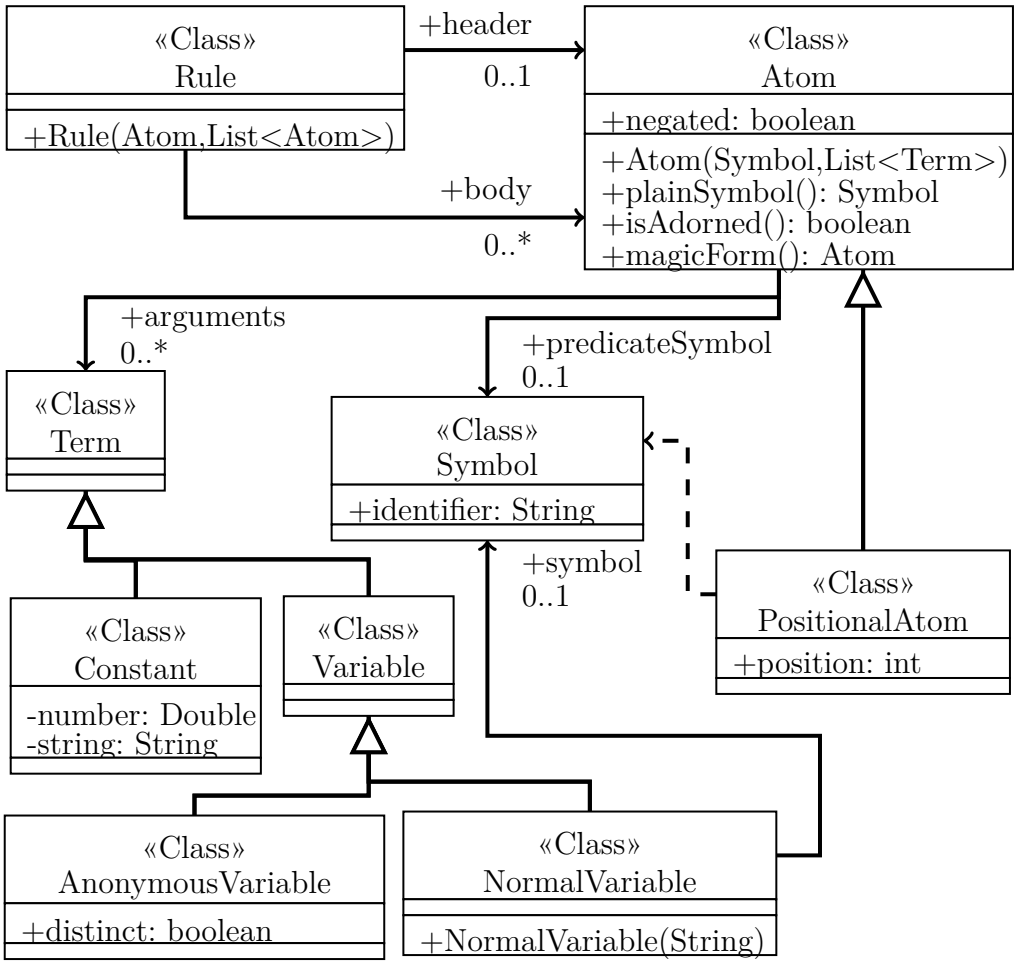

Diagram 5.6: Třídy pro reprezentaci logických pravidel

5.6 Implementace algoritmu tvorby magických pravidel

Již samotné procházení adresářové struktury ve třídě `Main` je psáno funkcionálním způsobem. Není nijak ošetřováno, zda-li na vstupu je stromová struktura nadmnožinou toho, co je vyžadováno, tzn. adresář s podadresáři pro jednotlivé pravidla a v nich soubory obsahující jednotlivá ozdobení pravidel ke zpracování. V podadresářích se nesmí vyskytovat jiné soubory, všechny soubory projdou ke zpracování. Krátce před zpracováním souboru je vypsána jeho cesta a poté případná výjimka ve zpracování.

Hlavní algoritmus pro konstrukci magických pravidel se nachází v třídě `Record`. Nejdříve se v konstruktoru (viz příloha B.1) načítají vstupní data v podobě textových řetězců, tzn. že v konstruktoru probíhá parsování, tj. všechny tři fáze. Zpracovává se řetězec pro logické pravidlo, ten je v argumentu `adornedRule` a také řetězec pro graf předávání informace (argument `SIPGraph`).

Poté následuje metoda `generateMagicForm()` (viz příloha B.2) spouštějící konstrukci magických pravidel nad správnými výskyty predikátů, tj. ozdobené intenzionální predikáty. Ještě před samotným zpracováním jednotlivých intenzionálních predikátů metoda provádí předzpracování těla ozdobeného pravidla. Graf předávání informace pracuje s jednotlivými výskyty predikátů označené pouhými predikátovými symboly, které jsou v identifikátoru rozlišené pořadím v těle, kdežto přirozeně samotné výskyty predikátů v těle pravidla nemají nijak zakódováno pořadí v těle.

Předzpracování se skládá z několika průchodů tělem, tím odteď obecněji myslíme seznam výskytů predikátů. Pro očíslování nejprve potřebujeme vytvořit instance `PositionalAtom` s odpovídajícími původními instancemi třídy `Atom`, abychom mohli vypočítané pořadí zaznamenat. Výskyty predikátů nejsou číslovány v těle globálně, nýbrž vždy vzhledem predikátu, který označují. Jednoduše řečeno každý predikátový symbol má své vlastní číslování v daném těle, když nebereme v potaz ozdobení predikátů. To pro nás znamená, že je potřeba výskyty predikátů před očíslováním rozdělit do skupin. Další průchod seznamem naplní asociativní mapu `positioned`, která je atributem instance třídy `Record` a je použita při dalším zpracování.

Asociativní mapa má jako klíče rozlišitelné predikátové symboly, které mají případné ozdobení odříznuté. Hodnotami jsou pak seznamy obsahující jen ty výskyty predikátů obsažené v původním těle pravidla, které mají daný predikátový symbol.

Pak přichází na řadu poslední průchod předzpracování a to je samotné očíslování výskytů predikátů, je přitom použito funkcionality ze třídy `Util`, tou je funkce `zip` a počítadlo `Counter` implementující rozhraní z *Javy 8* `Supplier` umožňující vytvoření čítače, kterého lze využít jako funkcionálního `Streamu`. Prochází se jednotlivé seznamy v asociativní mapě, pro každý průchod je vytvořeno nové počítadlo začínající na hodnotě 1 a funkcí `zip` jsou jeho hodnoty postupně kombinovány s výskyty predikátů, jsou do nich zapisovány čítané hodnoty.

Dále v metodě `generateMagicForm()` projdeme tělo pravidla a vyfiltrujeme ozdobená pravidla (na to máme k dispozici metodu `isAdorned()` třídy `Atom`). Pro ozdobené výskyty predikátu vytvoříme metodou `magicRule()` (tu si popíšeme dále) příslušná magická pravidla. Pravidla zformujeme do seznamu. V neposlední řadě vytvoříme sekci výstupního souboru `Magic rules:` a do jejího obsahu vypíšeme magická pravidla oddělená prázdnou řádkou. Následně označíme druh sekce a přidáme ji do seznamu k ostatním, s tímto seznamem pak pracuje `Writer`, když zapisuje data do souboru.

Metoda `magicRule(PositionalAtom)` (viz příloha B.3) odpovídá algoritmu 1 konstrukce magického pravidla. Deklarujeme a inicializujeme si části, z kterých se sestavuje výsledné tělo magického pravidla. V Javě budeme k těmto částem přistupovat jako k množinám, takže tyto části jsou deklarované jako množiny, ovšem chceme strukturu, která si zvládne pamatovat i pořadí vkládaných prvků. Potřebujeme strukturu splňující funkcionalitu požadovanou při definování algoritmu konstrukce magického pravidla, což je řetězení slov při zachování unikátnosti symbolů. To pro nás znamená, že nemůžeme použít běžně používanou implementaci množiny, kterou je `HashSet`. Avšak Java má i další implementace množiny a jedna z nich přesně splňuje naše požadavky, je to `LinkedHashSet`, tímto nám odpadá potřeba vlastní implementace takové struktury.

Argument metody je instance třídy `PositionalAtom`, je to výskyt predikátu v těle pravidla uzpůsobený k práci s grafem předávání informace. V grafu hledáme všechny hrany, které končí v tomto výskytu predikátu, před samotným vyhledáním je ještě instance třídy `PositionalAtom` převedena na symbol ve tvaru, jakým jsou zapsány výskyty predikátu v grafu předávání informace. K převodním metodám se dostaneme dále.

Všechny takové hrany postupně zpracujeme, spustíme nad nimi další metodu `resolveEdge(PassingInformation)` (viz příloha B.4). Ta vrací částečné výsledky jako uspořádanou trojici částí, Java nemá k dispozici generickou třídu pro n-tice jako je tomu v jiných programovacích jazycích, proto jsme také implementovali uspořádanou dvojici třídou `Pair`. Protože jsme potřebovali pracovat s uspořádanou trojicí jen v ojedinělém případě, nebyla vytvořena žádná speciální třída a uspořádaná trojice je reprezentována uspořádanou dvojicí s jedním zanořením. Druhý prvek uspořádané dvojice je rovněž uspořádanou dvojicí.

Po zpracování hrany a získání částečného výsledku je výsledek přidán do připravených částí, z kterých se skládá tělo magického pravidla. Nakonec po zpracování všech hran se všechny tři části spojí do výsledného těla magického pravidla a ještě se vytvoří magická forma hlavičky pravidla, poté vytvoříme instanci třídy `Rule` a vrátíme jako výsledek metody `magicRule(PositionalAtom)`.

Algoritmus 2 zpracování hrany je implementován v metodě `resolveEdge(PassingInformation)`, která má tedy na vstupu hranu `PassingInformation` a vrací uspořádanou trojici částečných výsledků, která je zkonstruována způsobem popsaným výše.

Připravíme části těla pro částečné výsledky, znovu to jsou struktury splňující vlastnosti, že řetězení struktur a vkládání nových prvků vynechává duplikáty a že je zachováváno pořadí zřetězených či vložených prvků, opět byla použita speciální java implementace množiny `LinkedHashSet`.

Dalším krokem je získání výskytu predikátu, z kterého hrana vychází. Ze zpracovávané hrany zjistíme predikátový symbol, z kterého vychází a pomocí převodní funkce predikátový symbol převedeme na výskyt predikátu s pozicí (instance `PositionalAtom`). Převod probíhá tentokrát v opačném směru, máme tedy dvě převodní funkce.

V algoritmu 2 zpracování hrany následuje kód s řídicí strukturou *switch*, kde se rozhoduje podle druhu predikátu, který jsme si v předešlém kroku zjistili. Samotné větve (bloky *case*) rozlišují druhy predikátu poměrně programátorsky přívětivým způsobem. V implementaci algoritmu je řídicí struktura *switch* nahrazená řadou řídicích struktur *if* (příp. *else if* či *else*). Použité podmínky mohou na první pohled vypadat značně komplikovaně.

První případ nastává, jestliže testovaný predikát je speciální hlavičkový. To lze otestovat snadno tím, že porovnáme predikátový symbol testovaného predikátu s predikátovým symbolem v hlavičce ozdobeného pravidla. To je možné díky tomu, že jsme si převedli z grafu predikát na instanci `PositionalAtom`. Pokud nastane první případ jednoduše přidáme speciální hlavičkový predikát v magické formě do vyhrazené části těla.

Druhý případ nastává, jestliže testovaný predikát je intenzionální. Intenzionální predikáty se vyznačují tím, že jsou ozdobené, podle toho je také rozlišujeme od extenzionálních. Jak v případě intenzionálního, tak i v případě extenzionálního predikátu přidáváme do těla původní tvar predikátu. Řídicí struktury *if* jsou zanořené, abychom se neopakovali. Rozlišování druhého a třetího případu se děje ve vnořené řídicí struktuře *if*. V případě intenzionálního predikátu dochází k ověření, zda se předává informace alespoň jednou proměnnou, která je na pozici navázaného argumentu ve zdrojovém predikátu. Po ověření je do části pro magické predikáty přidána magická forma zdrojového predikátu.

Třetí případ nastává, jestliže testovaný predikát je extenzionální nebo také vestavěný. V tomto případě zpracujeme rekurzivně zdrojový predikát stejným způsobem jako predikát, který je momentálně cílem (koncem) ve hraně. Převedeme zdrojový predikát do formy, jak jsou zapsány v grafu. Poté z grafu určíme seznam hran, které vstupují do původně zdrojového predikátu a projdeme postupně všechny hrany. Každou hranu rekurzivně zpracujeme, spustíme nad ní algoritmus 2 zpracování hrany, mezivýsledky přidáváme do postupně budovaných částí těla.

Po zpracování všech hran vrátíme uspořádanou trojici částí těla.

Výše popsané metody využívají dvě převodní metody (viz příloha B.5) mezi výskyty predikátu v těle pravidla a predikátovými symboly v grafu předávání informace. Metody pracují s daty, která byla vytvořena v předzpracování.

1. `getAdornedBySIP(Symbol)` – vstupem je predikátový symbol v grafu předávání informace, ten je převeden na instanci `PositionalAtom`, která již odpovídá výskytu predikátu v těle pravidla.

2. `getSIPByAdorned(PositionalAtom)` – na vstup dosazujeme výskyt predikátu v těle pravidla (instance třídy `PositionalAtom`) a vytvoříme z ní predikátový symbol v grafu předávání informace.

6 Testování

Pro aplikaci byly napsány jednotkové testy a rovněž byla aplikace testována na reálných datech, která byla předem poskytnuta.

6.1 Jednotkové testy

Aplikace je poměrně rozsáhlá a tak je vhodné pro ni napsat jednotkové testy. Knihovna použitá na jednotkové testy je *JUnit 4*. Jelikož je aplikace programována funkcionálním stylem a většina struktur má veřejně přístupné atributy, tak tvorba testovacích objektů byla snadná. Nedošlo tedy na použití *mock* objektů, které jsou vhodné pro mimikování chování instancí se složitým vnitřním stavem. Na druhou stranu tak nebyla potřeba vybírat jednu z některých knihoven implementující *mock* objekty.

Na některých místech testy touto volbou trochu utrpěly, jiným pohledem se dá říct, že odhalily, mírné nedostatky v modulárnosti návrhu implementace. Konkrétně třídy `Reader` a `Writer` pracují přímo se souborem a hlavní třídou `Record`, tudíž při načítání ze souboru dochází k předzpracování dat v konstruktoru a pro zápis je zase potřeba mít připravené sekce k zápisu v instanci třídy `Record`.

6.2 Test s reálnými daty

Pro tento test byla k dispozici vstupní data spolu s daty výstupními, která byla ručně vytvořená a použitá při porovnávání s výstupem generovaným aplikací.

Tento test sloužil především jako zátěžový. Avšak také se ukázalo, že díky testu s reálnými daty se přišlo na výjimečné situace při zpracování vstupu. Nejprve test odhalil potřebu zpracovávat pravidla s negativními literály v těle. Konečný automat pro syntaktickou analýzu musel být rozšířen o další větev pro negativní literály. Třídu `Atom` bylo zase třeba rozšířit o atribut držící tuto informaci.

Další abnormalitou ve vstupních datech byl případ, kdy instanci ozdobení logického pravidla nastala situace, že na pozicích vázaných argumentů se vyskytovaly konstanty a nikoliv proměnné. To mělo za následek to, že sekce obsahující plný graf předávání informace byla prázdná, což původně konečný automat pro syntaktickou analýzu grafu nepřijímal. Konečný automat byl pozměněn, aby přijímal i prázdné grafy.

Byla měřena doba běhu aplikace pro vstup o 19 pravidlech, když vezmeme v úvahu všechna ozdobení, je na vstupu 273 samostatných souborů. První spuštění po překladu trvalo 7,579 s, další spuštění běžela v průměru 6,7 s, to lze připisovat *JIT* technice, kterou Java používá.

7 Závěr

Cílem práce bylo naimplementovat algoritmus konstruující magická pravidla a také navázat na předcházející práci, která se zabývá konstrukcí grafů předávání informace, protože grafy předávání informace spolu s ozdobenými pravidly jsou vstupními daty pro algoritmus konstrukce magických pravidel.

Navázání na předešlou práci znamená využívání jejího výstupu, takže další z úkolů bylo načítání dat z textové souboru ve formátu určeném zmíněnou prací. Těchto cílů se povedlo dosáhnout.

Byla stvořena aplikace v programovacím jazyce Java, ovládání je zprostředkováno příkazovou řádkou. Umožňuje nám určit přes parametry vstupní a výstupní adresář. Výstupní textové soubory obsahují vstupní data rozšířená o jednu sekci, v které jsou zkonstruovaná magická pravidla.

Rychlost běhu aplikace nebyla prioritou, v této práci jde o referenční implementaci algoritmu konstrukce magických pravidel. Důležité je v takovém případě nezabývat se optimalizujícími detaily. Naopak chceme minimalizovat počet chyb při implementaci, z těchto důvodů byl zvolen funkcionální styl programování, který je dostupný od verze *Java 8*. Funkcionální styl programování umožňuje zapisovat algoritmy deklarativním způsobem, který přirozeně napomáhá menší tvorbě chyb během programování. Stinná stránka tohoto stylu programování je mnohdy právě samotný běh aplikace, který je pomalejší, než kdybychom programovali imperativním způsobem.

Pro data takového rozsahu při ruční manipulaci a testování metody Magických množin je necelých 7 s běhu dostatečných.

S jednotkovými testy se podařilo dosáhnout 83,7 % úrovně pokrytí kódu na úrovni jednotlivých instrukcí.

Zdrojové kódy je možné upravit do imperativního stylu programování a tím tak algoritmus optimalizovat v případě, že je nutné v reálném nasazení použít rychlou implementaci. Také je možné některé části kódu paralelizovat a získat tak nějaké urychlení.

Přehled zkratek

- *UML* (Unified Modeling Language) – unifikovaný modelující jazyk pro vizualizaci informace
- *API* (Application Programming Interface) – aplikační programové rozhraní nějakého celku
- *JDK* (Java Development Kit) – vývojové nástroje Javy
- *JIT* (Just in Time) – adaptivní způsob kompilace, stojící na základě analýzy běhu programu

Literatura

[1] *Parsing* [online]. Wikipedia. [cit. 2016/04/16]. Dostupné z: `https://en.wikipedia.org/wiki/Parsing#Overview_of_process`.

[2] Arhipov, A. *Java 8 Explained* [online]. ZeroTurnaround. [cit. 2016/04/22]. Dostupné z: `http://zeroturnaround.com/rebellabs/java-8-explained-applying-lambdas-to-java-collections/`.

[3] Morávka, J. *Generování grafů předávání informace.* Bakalářská práce, Západočeská univerzita v Plzni, 2015.

[4] siki. *Zipping streams using JDK8 with lambda* [online]. stackoverflow, 2014. [cit. 2016/04/16]. Dostupné z: `http://stackoverflow.com/a/23529010/1721219`.

[5] Vais, V. *Úvod do teorie jazyků* [online]. [cit. 2016/05/04]. Dostupné z: `http://home.zcu.cz/~vais/Pred02nov.htm`.

[6] Zíma, M. *Metoda magických množin.* Disertační práce, Západočeská univerzita v Plzni, 2002.

A Uživatelská příručka

A.1 Překlad

Pro uskutečnění překladu potřebujeme JDK alespoň ve verzi *Java 8*, od této verze v Javě přibyla možnost programovat funkcionálně. V adresáři `program/` je projektový soubor vývojové prostředí Eclipse (pro vývoj byl použit *Eclipse Mars*), alternativou je skript `build.xml` (v tomtéž adresáři), to je konfigurační soubor nástroje pro překlad *Apache Ant*. Vytvoření spustitelného JAR souboru provedeme následovně: provedením příkazu `ant CreateJar` v adresáři `program/`. Vytvoří se spustitelný soubor `magicrule.jar`.

A.2 Testy

V souboru `build.xml` jsou cíle pro spuštění jednotkových testů: `WholeSuite` a pak `junitreport`, tím se vytvoří v adresáři `program/junit/` zpráva ve formátu *HTML*, ta nám však neposkytuje všechny informace. Mnohem lepší je spustit jednotkové testy v Eclipsu spolu s nainstalovaným doplňkem *EclEmma Java Code Coverage*, v nabídce druhů spuštění přibude položka pokrytí kódu. Touto položkou spustíme všechny testy třídou `WholeSuite`, kterou najdeme v projektové složce `test`, v balíku `magicrule`.

A.3 Spuštění

Samotná aplikace přijímá na příkazové řádce tři argumenty:

1. volba výstupu (všechny sekce `all` nebo jen magická pravidla `magic`)
2. adresář vstupních dat
3. adresář pro výstupní data

Program s testovacími daty spustíme takto:

```
java -jar magicrule.jar all ../input ../output
```

B Zdrojové kódy

B.1 Konstruktor `Record`

```
/**
 * Instantiates a new record.
 * @param parts the sections
 * @param SIP the SIP graph
 * @param adornedRule the adorned rule
 */
public Record(List<Part> parts, String SIP, String
    adornedRule) {
  this.parts = parts;

  // Parsing of an adorned rule
  List<Lexeme<Definitions.Logic>> lexemes =
    Lexer.lexicalize(Definitions.logicProgramLexers,
          new StringBuilder(adornedRule));
  List<Token<Definitions.Logic>> tokens =
    Token.tokenize(lexemes);
  this.adornedRule = Parser.parse(
    new Definitions.RuleConstructor(),
    Definitions.RuleParserState.HeaderSymbol,
    Definitions.ruleParserTransitions,
    tokens);
  // Parsing of a sideways information passing graph
  List<Lexeme<Definitions.Graph>> SIPLexemes =
    Lexer.lexicalize(Definitions.graphLexers,
          new StringBuilder(SIP));
  List<Token<Definitions.Graph>> SIPTokens =
    Token.tokenize(SIPLexemes);
  this.SIP = Parser.parse(
    new Definitions.GraphConstructor(),
    Definitions.GraphParserState.PathOpeningBrace,
    Definitions.graphParserTransitions,
    SIPTokens);
}
```

B.2 Magický tvar `generateMagicForm`

```
/**
 * Generates magic rules for the given rule.
 */
public void generateMagicForm() {
  // Enumerating of the occurrences
  List<PositionalAtom> bodyPositioned =
    adornedRule.body.stream()
    .map(predicate -> new PositionalAtom(predicate))
    .collect(Collectors.toList());
  // Creating of a map for the conversion from a SIP
     graph symbol to an occurrence of predicate in the
     rule
  positioned = bodyPositioned.stream()
    .collect(Collectors.groupingBy(
      PositionalAtom::plainSymbol,
      Collectors.toList()));

  // Setting up the ordinal positions for the
     conversion from an occurrence of predicate to a
     SIP graph symbol
  positioned.forEach((symbol, positionedAtoms) ->
    Util.zip(
      Stream.generate(Util.counter(START_COUNT)),
      positionedAtoms.stream(),
      Pair<Integer, PositionalAtom>::new)
    .forEach(pair ->
      pair.second.position = pair.first));

  // Collecting of magic rules for every adorned
     occurrence of predicate
  List<Rule> magicRules = bodyPositioned.stream()
    .filter(predicate -> predicate.isAdorned())
    .map(adorned -> magicRule(adorned))
    .collect(Collectors.toList());
  // Writing out the magic rules for the output
  Part magicPart = new Part("Magic rules:",
    magicRules.stream()
    .map(Rule::toString)
```

```
        .collect(Collectors.joining(Writer.newLine +
            Writer.newLine)));
    // Used by Writer for a potential empty result
    magicPart.type = Optional.of(PartType.MAGIC_RULES);
    parts.add(magicPart);
}
```

B.3 Magické pravidlo `magicRule`

```
/**
 * Generates a magic rule for the intensional
    predicate.
 *
 * @param adorned the intensional predicate.
 * @return the magic rule
 */
private Rule magicRule(PositionalAtom adorned) {
    // Prepare parts of rule's body; with unique
       occurrences and preserving order
    Set<Atom> specialHeader = new LinkedHashSet<>();
    Set<Atom> magic = new LinkedHashSet<>();
    Set<Atom> original = new LinkedHashSet<>();

    // For every edge which ends in this occurrence of
       predicate
    SIP.withTail(getSIPByAdorned(adorned)).forEach(
      edge -> {
        // Process the edge
        Pair<Set<Atom>, Pair<Set<Atom>, Set<Atom>>>
            partialResult = resolveEdge(edge);
        // Append the partial result to the parts of
            rule's body
        specialHeader.addAll(partialResult.first);
        magic.addAll(partialResult.second.first);
        original.addAll(partialResult.second.second);
    });

    // Return a new rule with concatenated parts of body
    return new Rule(
      adorned.magicForm(),
```

```
    Stream.concat(
      specialHeader.stream(),
      Stream.concat(
        magic.stream(),
        original.stream()))
    .collect(Collectors.toList()));
}
```

B.4 Zpracování hrany `resolveEdge`

```
/**
 * Resolves only one edge in the SIP graph.
 *
 * @param edge the edge
 * @return the triplet of parts of constructed body of
     a magic rule
 */
private Pair<Set<Atom>, Pair<Set<Atom>, Set<Atom>>>
   resolveEdge(PassingInformation edge) {
  // Prepare parts of rule's body; with unique
     occurrences and preserving order
  Set<Atom> specialHeader = new LinkedHashSet<>();
  Set<Atom> magic = new LinkedHashSet<>();
  Set<Atom> original = new LinkedHashSet<>();

  // An occurrence of predicate from which comes the
     information
  PositionalAtom informationSource =
    getAdornedBySIP(SIP.head(edge));

  // Case of the special header predicate
  if (informationSource.predicateSymbol
      .equals(adornedRule.header.predicateSymbol)) {
    specialHeader.add(adornedRule.header.magicForm());
  } else {
    // Add original form
    original.add(informationSource);

    // The occurrence of predicate is an intensional
       predicate
```

```
    if (informationSource.isAdorned()) {
      // Check whether exists a bound argument in the
         source predicate
      if (edge.variables.stream()
          .anyMatch(argument ->
            informationSource.arguments.stream()
            .filter(term -> term instanceof
               NormalVariable)
            .anyMatch(term -> argument.equals((
               NormalVariable)term)))) {
        // Add magic form
        magic.add(informationSource.magicForm());
      }

    // The occurrence of predicate is an extensional
       predicate
    } else {
      // Recursively process the preceding vertex
      SIP.withTail(getSIPByAdorned(informationSource))
          .forEach(edgeNext -> {
        // Process the edge
        Pair<Set<Atom>, Pair<Set<Atom>, Set<Atom>>>
            partialResult = resolveEdge(edgeNext);
        // Append the partial result
        specialHeader.addAll(partialResult.first);
        magic.addAll(partialResult.second.first);
        original.addAll(partialResult.second.second);
      });
    }
  }

  // Return partial result in parts
  return new Pair<>(specialHeader, new Pair<>(magic,
     original));
}
```

B.5 Převod symbolů

```
/**
 * Converts from the SIP graph to the occurrence in
    the rule.
 *
 * @param symbol the vertex symbol
 * @return the occurrence of an atom
 */
private PositionalAtom getAdornedBySIP(Symbol symbol)
   {
  // Testing for a special header predicate
  if (symbol.identifier.matches(HEADER_REGEX))
    return new PositionalAtom(adornedRule.header);

  // Replacing the symbol with the occurrence of
     predicate
  String[] parts =
    symbol.identifier.split(POSITION_DELIM_REGEX);
  return positioned.get(new Symbol(parts[0]))
    .get(Integer.parseInt(parts[1]) - START_COUNT);
}

/**
 * Converts from the occurrence in the rule to the SIP
     graph.
 *
 * @param adorned the occurrence of an atom
 * @return the vertex symbol
 */
private Symbol getSIPByAdorned(PositionalAtom adorned)
    {
  return new Symbol(adorned.plainSymbol().identifier +
      POSITION_DELIM + adorned.position);
}
```