Rational Expression Parser

Vivien Delmon

Technical Report n^o0810, March 2008
revision 1887

The VAUCANSON library is designed to manipulate automata and transducers. Therefore we need a rational expression parser which deals with transducers.

The current rational expression parser only takes as input simply weighted rational expression (weithed with Integers). The new parser allows us to specify any kind of weight and any kind of monoid like free monoid product. Both of these features are mandatory if we want to deal with transducers. The new parser is also less restrictive and provides more freedom to the user who can easily change the form of the grammar used to write its expression.

La bibliothèque VAUCANSON permet de manipuler des automates et des transducteurs. Le parser d'expression rationnelles doit donc lui aussi traiter ces différentes structures.

Malheureusement l'ancien parser ne permettait pas de lire des expressions rationnelles décrivant des transducteurs ou même des automates à poids autres que des nombres. Le nouveau parser permet de lire des expressions rationnelles contenant des poids de toutes sortes et des alphabets définis sur des produits de monoïdes. Ces différentes améliorations permettent d'interpréter des expressions rationnelles complexes représentant entre autres des transducteurs.

Keywords
parser, rationnal expression, transducer, bison

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
vivien.delmon@lrde.epita.fr – VAUCANSON

Copying this document

Contents

Introduction

The VAUCANSON library is a finite state machine platform designed to manipulate automata and transducers. The input system can read either a XML file representing an automaton or a rational expression that can be transformed into an automaton. The XML parser uses the rational expression parser to parse the labels of the transitions that can be rational expressions. If the rational expression parser does not allow enough expressiveness all the library is penalized.

The current rational expression parser of VAUCANSON cannot parse all kinds of rational expressions that represent automata or transducers. It can only parse an expression that represents a Boolean automaton or a simply weighted automaton (weighted with Integers).

Firstly we will see what kind of expression we want to parse with VAUCANSON and what kinds of problems were implied by the old parser and its grammar. Then we will present a new grammar and discuss its advantages and inconveniences. Finally, we will compare different parser generator and present our implementation of the new parser based on Bison and some problems we had to solve during this implementation.

Chapter 1

Preliminaries

1.1 Weighted rational expression

1.1.1 Definition

The algebraic structure of a monoid is required to define the alphabet of a rational expression.

Definition 1 *A system* $(\mathbb{K}, \otimes, \overline{1})$ *is a monoid if:*

- $\otimes$ *is associative:* $\forall a, b, c \in \mathbb{K}, (a \otimes b) \otimes c = a \otimes (b \otimes c)$*,*
- $\overline{1}$ *is an identity element for* $\otimes$*:* $\forall a \in \mathbb{K}, a \otimes \overline{1} = \overline{1} \otimes a = a$*.*

The algebraic structure of a semiring is also required to define a weighted rational expression.

Definition 2 *A system* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ *is a semiring if:*

- $(\mathbb{K}, \oplus, \overline{0})$ *is a commutative monoid with* $\overline{0}$ *as identity element for* $\oplus$*,*
- $(\mathbb{K}, \otimes, \overline{1})$ *is a monoid with* $\overline{1}$ *as identity element for* $\otimes$*,*
- $\otimes$ *distributes over* $\oplus$*:* $\forall a, b, c \in \mathbb{K}, (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$ *and* $c \otimes (a \oplus b) = (c \otimes a) \oplus (c \otimes b)$*,*
- $\overline{0}$ *is an annihilator for* $\otimes$*:* $\forall a \in \mathbb{K}, a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$*.*

The classical rational expressions are defined over the Boolean semiring $(\mathbb{B}, \vee, \wedge, \top, \bot)$. These expressions answer $\top$ if they recognize the word and $\bot$ if they do not.

Definition 3 *A rational expression is a series parameterized by a semiring and a monoid.*

- *The monoid represents the alphabet of the expression often called* Σ*,*
- *The semiring describes how weights are evaluated.*

1.1.2 Examples

The rational expression "a + b" defined over the Boolean semiring and the alphabet $\Sigma = \{a, b\}$ recognizes the word "a" and the word "b".

The weighted rational expression "6(2 a + 3 b)" defined over the semiring $(\mathbb{Z}, +, \times, 0, 1)$ and an alphabet $\Sigma = \{a, b\}$ associates 12 with the word "a" and 18 with the word "b".

1.1.3 Rational expression weighted by a rational expression

If the set $\mathbb{K}$ of our semiring is a set of weighted rational expression, we can define weighted rational expression weighted by weighted rational expressions. This kind of construction is used to represent some kinds of transducers.
The rational expression "({a} b + {b} a)*" transforms every "a" in "b" and every "b" in "a" in a word. For example, the word "abbaab" becomes "baabba". We can see in this expression that we need to specify what is a weight and what is word to avoid ambiguity.
We can imagine far more complicated expressions with an arbitrary number of recursions in the weights.

- "{3 a} b + {4 b} a"
- "{{3 a} a} b + {{4 a} b} a"

Chapter 2

The old parser and its limits

2.1 The grammar

```
exp ::= '(' exp ')'
    | exp '+' exp
    | exp '.' exp
    | exp exp
    | exp '*'
    | weight ' ' exp
    | exp ' ' weight
    | '0'
    | '1'
    | word
```

This grammar is directly based on characters without any token abstraction. We had some complains about VAUCANSON's users who wanted to use alphabets composed of 0 and 1. With this grammar it is clearly impossible to parse such expression without ambiguity.
For example, the rational expression "1 1" defined over the semiring $(\mathbb{Z}, +, \times, 0, 1)$ and an alphabet $\Sigma = \{0, 1\}$ can represent four different expressions: "weight(1) exp(1)", "exp(1) weight(1)" and each time we have "exp(1)", the "1" can be either the "1" of the alphabet or the identity of the series ('1' in this grammar).
Moreover this grammar uses the space character as an active character. It impedes the user who cannot use it to clarify its rational expression. What is even worse, the two expressions "1 1" and "11" do not have the same meaning.
Noticing these problems, we decided to adapt our grammar and parser to deal with any kind of finite state machine and any kind of alphabet.

The first idea was to adapt the previous parser by changing the grammar, but this parser is a handwritten recursive parser and it is very difficult to insert new rules in this kind of parser without changing a lot of code. Since we had some ideas to extend the grammar we wanted a clear grammar and an easy to adapt implementation. The choice of a generated parser seems to be the best one. In this kind of parser we clearly write the grammar and if we want to change a rule, it is just one rule to adapt and sometimes some priorities to change. This solution is the one we decides to implement with the new grammar that is introduced in the next section.

Chapter 3

The new parser

3.1 The grammar

```
%start exp;

exp :
  rexp
  ;

rexp :
    OPAR   rexp   CPAR
  | rexp   PLUS   rexp
  | rexp   TIMES  rexp
  | rexp   rexp
  | rexp   STAR
  | WEIGHT rexp
  | rexp   WEIGHT
  | ONE
  | ZERO
  | WORD
  ;
```

A similar grammar have already been proposed by Raphaël Poss and Thomas Claveirole in a discussion that can be read in annexe A of this report. Their grammar had the aim to be POSIX compliant to help any user who already manipulates rational expressions to easily find his way. In our grammar we can also imagine a POSIX mode with `PLUS` as "`|`", `STAR` as "*" and without activating the third rules about `TIMES`.
This thing is possible because in our grammar the representation of each symbol is abstracted by a token which can be changed by the user. This allows more freedom for the user who can use any string of characters to represent any token of the grammar.
For instance, if someone wants an automaton with `0` and `1` in its alphabet he can redefine the representation of `ONE` and `ZERO` and can choose something like `eps` and `null`.
The character space is not used anymore and can be used directly or even replaced by any other character if we need it in our alphabet.

3.2 Spirit or Bison

The first step before writing a generated parser is to choose between all the existing libraries that generate C++ parser. We already knew how to use Bison and we wanted to test Spirit. Some other parser generators are available like ANTLR, APG (not available on some distribution), BtYacc, QLALR (part of Qt) but they are closed to Bison so we decided to chose Bison to represent them.

3.2.1 Spirit

Spirit (de Guzman, 2003) is a C++ parameterized parser generator part of the Boost Library. VAUCANSON already depends on this library that is why we wanted to try an implementation with Spirit. Unfortunately the Spirit library needs to know the token representation at compilation time which is incompatible with our approach. Moreover the Spirit library is a scannerless parser. Due to this structure we cannot separate the scanner and the parser, it is why we cannot write our own scanner separately.
Since the Spirit library is totally static and we want a dynamic aspect for our parser it seemed really difficult to adapt our parser to use Spirit and we finally droped this idea.

3.2.2 Bison

Bison (Donnelly and Stallman, 2006) is a C/C++ parser generator and people usually use Flex to generate their scanner. But the problem is still the same. Flex, as Spirit, needs to know the token representation before generating the scanner.
But this time the scanner and the parser are two different entities and we can use Bison to generate our parser without using Flex. It is what we chose so we implemented a scanner which only abstracts the token representation to give a uniform queue of tokens to the scanner. This approach has another advantage due to the dynamic structure of our parser and the static structure of the VAUCANSON library, we will precise this advantage in a further section 3.3.2.

3.3 Implementation

First of all we will see how VAUCANSON stores a rational expression. Then we will see why is it a problem when we want to use Bison and our solution to this problem. Finally we will present our implementation of the scanner.

3.3.1 A rational expression in VAUCANSON

```
typedef typename rat::exp<typename automaton_t::monoid_elt_value_t,
                          typename automaton_t::semiring_elt_value_t>
        krat_exp_impl_t;

typedef Element<typename automaton_t::series_set_t,
                krat_exp_impl_t>
        krat_exp_t;
```

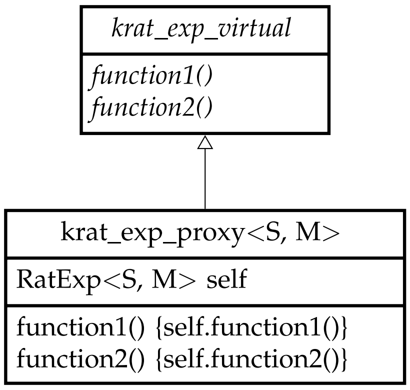
The design pattern ELEMENT (Régis-Gianas and Poss, 2003) is used in VAUCANSON to associate the concept of series with our implementation. Our implementation is defined in the class `rat::exp` which is an abstract syntax tree in which the following nodes are available:

- product
- sum
- star (Kleene's star)
- right weight and left weight (semiring element)
- word (monoid element)
- one (identity of the monoid)
- zero (zero of the monoid)

The `krat_exp_t` structure provides all necessary operators to construct and manipulate series. We just have to parse the expression and delegate the construction of the expression to this structure.

3.3.2 A problem of template

The parser generated by Bison cannot be parameterized but the class `krat_exp_t` of VAUCANSON is parameterized. We have to use a bridge to manipulate a static object in a dynamic function. Some work about it has already been done in the LRDE ((Pouillard, 2006), (Duret-Lutz, 2000)) and we chose to implement the external polymorphism pattern described in (Duret-Lutz, 2000).
This pattern consists of a `DataProxy` that contains the static data structure and an abstract superclass `AbstractData` that declares the different functions we have to use in the dynamic function. This gives us the following UML schema:



In the parser, we are sure that we manipulate only one type of `krat_exp_t` and we can downcast it. The next example is the implementation of an operator in the proxy class, other functions of the proxy are similar.

```
template <class S, class T>
krat_exp_virtual*
krat_exp_proxy<S, T>::operator+(krat_exp_virtual* exp)
{
  return new krat_exp_proxy<S, T>(self + (dynamic_cast<krat_exp_proxy
      <S, T>*> (exp))->self);
}
```

3.3.3 Scanner

The scanner constructs a queue of token by reading a string of characters.

```
typedef std::queue<std::pair<krat_exp_parser::token_type,
                             krat_exp_parser::semantic_type> >
        tok_queue_t;
```

The `token_type` is an enumeration given by the generated parser and `semantic_type` is an union of `krat_exp_virtual`, `semiring_virtual` and `std::string` also given by the generated parser.

Word

The scanner reads the string until it finds a token. When it finds a token if there was some characters before, it translates them into a word using the parameterized function `parse_word`.

```
template<class S, class T>
bool parse_word(Element<S,T>& dest,
                const std::string& s,
                typename std::string::const_iterator& i)
```

The Element created is used to construct a `krat_exp_virtual`.

Weight

A weight is delimited by two strings, one for the beginning and one for the end. By default it is "{" and "}". When the first one is found the scanner tries to find the second one. All characters in between are given to the `parse_weight` function.

```
template<class S, class T>
bool parse_weight(Element<S,T>& dest,
                  const std::string& s,
                  typename std::string::const_iterator& i)
```

The Element created is used to to construct a `semiring_virtual`.

Other token

Any other token is associated with the string representing it. Except the token `ONE` and the token `ZERO`, which are created using a function that returns the identity and the annihilator of the series. These functions take as input the structure of the rational expression that contains the defined alphabet.

```
Element<S, T> w = identity_as<T>::of(e_.structure())
Element<S, T> w = zero_as<T>::of(e_.structure())
```

Chapter 4

Future work and possible improvements

For the moment, we used the default representation to recognize the token but the structure of our scanner is already prepared to accept different representation. We need to find a way to store this representation in the rational expression object. In addition, we have to find a place for this representation in the automaton created with the rational expression. In the scanner class the token representation is stored in a vector of strings and we can change it when the scanner class is created.

Some automata family are defined using exponent like "$(a+\varepsilon)^n$" which is often use to test ε-removal. Some other extension of the grammar could be interesting.

Some automata family like the ones using Integers in the alphabet are implemented in VAUCANSON but have never been used due to the lack of the previous parser. We should revive them by creating the interface to show some examples that can be parsed with this new parser.

Conclusion

We saw why the old parser impedes the user who cannot write complicated rational expressions easily. The new parser with its new grammar allows more freedom to the user. In addition, the user will be able to change the token of the grammar to adapt its automata family to any kind of situation without changing any line of code.
The static/dynamic bridge between the parser and VAUCANSON allows us to compile the parser in a library and to link it only at the end, which provides less compilation time. This example shows us that dynamic function can be used in a fully static library.
Finally, the new parser provides more freedom to the user who can write much more complicated expressions. In addition the new parser is generated and the grammar is written using a BNF formalism which is much more understandable and easier to extend.

Bibliography

de Guzman, J. (2003). Spirit user's guide. http://www.boost.org/doc/libs/1_35_0/libs/spirit/index.html.

Donnelly, C. and Stallman, R. (2006). Bison, the yacc-compatible parser generator. http://www.gnu.org/software/bison/manual/index.html.

Duret-Lutz, A. (2000). Olena: a component-based platform for image processing, mixing generic, generative and oo programming. http://www.lrde.epita.fr/dload/papers/gcse00-yrw/olena.html.

Pouillard, N. (2006). Dynamization of c++ static libraries. Technical report, EPITA Research and Development Laboratory (LRDE).

Régis-Gianas, Y. and Poss, R. (2003). On orthogonal specialization in C++. Technical report, EPITA Research and Development Laboratory (LRDE). http://www.lrde.epita.fr/cgi-bin/twiki/view/Publications/200307-Poosc.

Appendix A

Exchange between Raphaël Poss, Thomas Claveirole and Jacques Sakarovitch in vaucanson.private

```
Raphaël Poss 29/03/2007

Coucou tous,

suite à discussion avec Thomas, voici une nouvelle proposition de grammaire pour
les expressions dans Vaucanson.

Nous avons cherché les propriétés suivantes pour les expressions:

 - compatibilité POSIX
 - syntaxe générique
 - implémentation triviale
 - lecture facile non ambigue (impliqué par la compatibilité POSIX)

Le résultat est ... la grammaire POSIX elle-même (celle de base avec juste les
opérateurs "()|*"), étendue avec les multiplicités de la forme {=N}, N pouvant
être n'importe quoi (y compris... une expression). En bonus: la liberté
d'implémenter les syntaxes POSIX supplémentaires sans souci : [], {n}, etc.

J'ai écrit une implémentation "proof of concept", jointe à ce mail. Ça s'utilise
comme suit:

 python parse2.py "ab*"
 python parse2.py "a|b"
 python parse2.py "a{=2}|b{=3}"
 (etc)

Python 2.4 ou supérieur requis.
```

Vos avis nous intéressent.

Thomas Claveirole 30/03/2007

Je me permet d'entrer dans les détails et d'expliciter les cheminements intellectuels qui nous ont amené au choix d'un {=N} suffix pour les multiplicités :

1. {=N} est une séquence de caractère invalide dans les expressions POSIX, ce qui nous garanti qu'aucune expression POSIX bien formée n'aura pas une signification différente dans cette syntaxe et dans la syntaxe POSIX.

2. La présence d'un caractère ouvrant *et* d'un caractère fermant pour délimiter les poids permet d'embarquer sans ambiguïté des poids d'un type non trivial, comme par exemple d'autre expressions rationnelles (utile dans le cas de transducteurs, par exemple). Et ce, avec plusieurs niveau de récursion.

Par exemple :
```
a{=b*{=c{=4}}}|aa{=ba{=cc{=1}}}
```

(Si je ne m'abuse, c'est un transducteur multi-bandes avec multiplicités dans Z)

3. Mettre les poids en notation suffixe nous permet d'éviter une ambiguïté avec les opérateurs *, + et {}. Cette ambiguïté ce poserait si {=N} préfixait un bloc. De plus, tous les opérateurs unaire POSIX sont suffixes, ça semble plus cohérent de rester dans cette lignée.

4. On avait pensé à des choses genre {+N}, {-N}, {0} ou encore {N..} et {N.N}, mais ça n'avait pas vraiment de sens dans les contextes ou les poids ne sont pas numériques. On a également songé à employer \N, mais on perd l'avantage 2, et ça pose des problèmes avec l'introduction de caractères ASCII en octal. Et puis nous trouvions cette notation encore pire que {=N}. Sans compter que je crois que Raphaël avait peut-être d'autre idées en têtes avec le \.

Jacques Sakarovitch 30/03/2007

Merci à vous deux pour cette proposition et contribution à Vaucanson. Merci à Thomas en particulier d'avoir ajouté des explications explicatives à la grammaire envoyée par Raphaël. Je n'ai pas le temps juste maintenant de répondre au fond, et je ne suis d'ailleurs pas le plus compétent pour le faire. Deux remarques néanmoins:

1) La proposition qui semblait avoir été adoptée de délimiter les 'coefficients' par des crochets est refusée parce que cela permet d'implémenter les "syntaxes supplémentaires" de POSIX. Il faut évaluer le coût d'une notation plus lourde

contre celui de renoncer à une partie de ces syntaxes supplémentaires.

2) La grammaire des expressions rationnelles avec multiplicité est une donnée.
Vous la trouvez dans ETA p.427, ou dans le papier "Derivatives of rational
expressions with multiplicity" de Sylvain et moi, Theoret. Comput. Sci. 332
(2005) 141-177, chargeable depuis ma page web:

Rational expressions are then given by the following grammar:

```
E := 0|1| a\in A | (E+E) | (E.E) | (E^*) | (kE) k\in K | (Ek) k\in K
```

Je vois mal comment les poids en position suffixe permettent de 'simuler' cette
grammaire. Par exemple, comment écrit-on:

```
(k(((ha)^*)g))
```

où a est une lettre, k, h, g sont des poids, et la multiplication n'est pas
commutative.

Raphaël Poss 30/03/2007

Facile: `{=k}(a{=h}*{=g})`
Ou plus explicitement: `({=k})(((a{=h})*){=g})`

Si je reprends la grammaire ETA, voici les équivalences:

E := 0

On écrit: {=0} (chaine vide affectée du poids 0)
C'est équivalent dans la grammaire actuelle de Vaucanson à "0 1", qui
est bien équivalent au 0 des séries

E := 1

On écrit: (rien) (la chaine vide)
C'est équivalent dans la grammaire actuelle de Vaucanson à "1".

E := a\in A

Pas de changement: on écrit a

E := (E+E)

On écrit: E|E

E := (E.E)

```
On écrit: EE
Si on veut vraiment expliciter l'opération on peut écrire aussi (E)(E)

E := (E^*)

Pas de changement: on écrit E*

E := (kE) k\in K | (Ek) k\in K

On écrit: {=k}E ou E{=k}

Cette dernière syntaxe unique est effectivement éloignée de l'idée
initiale de distinguer "la multiplication à gauche" de "la
multiplication à droite". Cependant on remarque que cela permet bien de
noter tout ce qu'on sait déjà noter dans Vaucanson:

x E y => {=x}E{=y}

Ou dans certains cas aussi E{=x}{=y} (quand il n'y a pas encore de
    coefficients dans E)

(2a+3b)4 => (a{=2}|b{=3}){=4}

4(2a+3b) => {=4}(a{=2}|b{=3})

Il est vrai que pour {=k}E, on triche. La nouvelle notation proposée est
en réalité équivalente à la notation Vaucanson "(k 1).E" et non "k E".
Cependant pour citer Thomas l'effet sémantique est le même et la
distinction structurelle n'intéresse pas l'utilisateur dans la plupart
des cas. Si la distinction structurelle est importante, dans ce cas on a
envie d'une syntaxe qui lève toute ambiguité, par exemple le format XML
explicite pour les expressions.

Voilà, est-ce que c'est plus clair comme cela?

-------------------------------------------------------------------------------

Thomas Claveirole 30/03/2007

Décidément, je n'arriverai jamais à comprendre la différence entre un
poids à gauche et un poids à droite. Il me semblait que ((ka)h) et
(k(ha)) étaient équivalents. Du coup, j'imaginais écrire des choses
de la forme a{=k}{=h}.

Sinon, il est toujours possible d'avoir deux opérateurs suffixes, par
exemple {left k} et {right k} (bien sûr on peut choisir un lexique
moins verbeux [1]). Globalement, je n'aime pas le mélange
préfixe/suffixe. Ça introduit des d'ambiguïtés lorsque l'on veut
retirer les parenthèses qui ne sont pas agréable à manipuler. Par
```

exemple:

```
{=k}a{=h} => a{left k}{right h} ou a{right h}{left k} ?
{=k}a* => a{left k}* ou a*{left k} ?
```

Bien sûr, on peut toujours s'en sortir avec un choix de désambiguïsation plus ou moins arbitraire, et requérir l'usage de parenthèse lorsque l'on veut quelque chose contraire à ce choix (là où avec une notation tout suffixe les parenthèses sont inutiles). Mais je n'aime pas l'introduction de comportement implicite du type « {=k}a* signifie a*{left k} ». Autant ça ne poserait pas de problèmes à l'utilisateur averti, autant ce serait une source d'embrouille pour les autres utilisateurs. Je parle en connaissance de cause, puisque la syntaxe actuelle possède le même genre de comportement, et que cela a déjà été pour moi et plusieurs fois une source de surprises par le passé.

Reste aussi l'argument, de moindre importance, que tout faire en notation suffixe permettrai de conserver à la fois une certaine cohérence avec la notation POSIX, et une certaine cohérence de manière générale.

[1] Par exemple, voici quelques idées :

```
{l=N} et {r=N}
{k.=N} et {.k=N}
{<N} et {>N}
{.N} et {N.} (J'aime beaucoup, mais ça a pas l'air LL(1). :-( )
{[N} et {]N}
```