Ovide Documentation

From regular expressions and rewrite rules
to finite-state machines

Version 1.4.2

Richard Beaufort
Developed in the framework of generative phonology, rewrite rules are now widely used in many areas of natural language processing (preprocessing, lemmatization, syntactic analysis, grapheme-to-phoneme conversion...) and speech processing (text-to-speech synthesis by Non-Uniform Units selection, and automatic speech recognition).

In general, the overall description of an application domain requires a wide set of rewrite rules. Therefore, the system efficiency will greatly depend on the method used to represent such rules. Finite-state machines (FSMs) such as automata (FSAs) and transducers (FSTs) were shown by Johnson (1972) to be able to model rewrite rules. These machines form an ideal framework: they allow compact representations (Mohri 1994) and accept operations such as union, concatenation and Kleene star (Roche & Schabes 1997) – necessary for describing regular languages –, but also composition (Pereira & Riley 1997, Mohri et al. 1996) – a special kind of intersection that is part and parcel of the compilation of rewrite rules.

Ovide, which is based on an original FSM Library developed at Multitel ASBL, compiles dictionaries, regular languages and rewrite rules into equivalent finite-state machines. Dictionaries, languages and rules may be weighted. Moreover, rules may be optional, because these variants are used in most real applications.
1 About Regular Expressions

1.1 Some definitions

Définition 1.1 (Alphabet). An alphabet is a set of symbols.

For instance, \{a, b, c, \ldots, y, z\} is the alphabet of lower case characters, while \{0, 1, 2, \ldots, 8, 9\} is the alphabet of digits.

Définition 1.2 (Regular language). A regular language is a formal language that is closed under concatenation, intersection, complementation, union and Kleene star.

Note that a class of languages is said to be closed under a given operation when the application of this operation, on a language belonging to this class, gives another language belonging to this class as well. For instance, the result of concatenation on two regular languages is still a regular language.

Définition 1.3 (Regular expression). A regular expression is a syntactic description of a regular language.

This syntactic description defines a language in comprehension, using predefined operators for combining symbols from a given alphabet \(\Sigma\).

Basically, each operator corresponds to a regular operation under which regular languages are closed : concatenation, intersection, complementation, union and Kleene star. However, some combinations of operations may be expressed using single operators. Hence, the list of operators defined on regular expressions is a little bit longer than the list of regular operations.

Définition 1.4 (Empty string). The empty string is the string made of no symbol, and is noted \(\epsilon\).

Définition 1.5 (Free monoid). Given an alphabet \(\Sigma\), the free monoid \(\Sigma^*\) is the set of all strings built from \(\Sigma\), including the empty string.

Thus, the free monoid is a regular language, and any regular language defined on \(\Sigma\) is equal to or included in \(\Sigma^*\).
1.2 Operators available in Ovide

Note that some operators only work on symbols or strings (symbols combined by concatenation only). In the list of operators hereunder, the following notation is in use:

- **a, b, c:** lower cases represent symbols.
- **X, Y:** upper cases represent strings.
- **REG:** represents a regular expression.
- **OP:** represents a single operator.

\[
\begin{align*}
\text{REG}^* & : 0, 1 \text{ or more REG} \\
\text{REG}^+ & : \text{at least one REG} \\
\text{REG}? & : 0 \text{ or one REG} \\
\text{REG}\{n,m\} & : \text{at least } n, \text{ maximum } m \text{ REG} \\
\text{REG}\{n\} & : n \text{ REG} \\
\text{REG}\{n,\} & : n \text{ REG or more} \\
\text{REG}\{,m\} & : 0 \text{ to } m \text{ REG} \\
(\text{REG}) & : \text{grouping (modify the standard precedence of operators, cf. hereunder)} \\
\text{REG}_1 \cdot \text{REG}_2 & : \text{concatenation of } \text{REG}_1 \text{ and } \text{REG}_2 : \text{REG}_1 \text{REG}_2. \text{ Note that this operator is never written. Expressions are directly besides.} \\
\text{REG}_1 \mid \text{REG}_2 & : \text{union of } \text{REG}_1 \text{ and } \text{REG}_2. \text{ Be care of the precedence of operators: } ab|cd = (ab)|(cd) \text{ and not } a(b|c) d \\
[abc] & : \text{equal to } (a|b|c) \\
[a-z] & : \text{equal to } (a|b|...|y|z) \\
[^abc] & : \text{not } abc \text{ (all symbols but } abc) \\
^\text{REG} & : \text{REG must match the beginning of the string} \\
\text{REG}$ & : \text{REG must match the end of the string} \\
"X" & : \text{raw expression, no interpretation of } X, \text{ which is a string that may contain operators, interpreted as single symbols} \\
\text{\textbackslash OP} & : \text{precedes an operator } \text{OP} \text{ that has to be interpreted as a symbol}
\end{align*}
\]

1.3 Operator precedence in Ovide

\[(1) \ "X" \ \text{\textbackslash OP} \]
\[(2) ^\text{REG} \ \text{REG}$ \]
\[(3) (\text{REG}) \]
\[(4) [abc] \ [^abc] \]
\[(5) \text{REG}^* \ \text{REG}^+ \ \text{REG}? \ \text{REG}\{n,m\} \ \text{REG}\{n\} \ \text{REG}\{n,\} \ \text{REG}\{,m\} \]
\[(6) \text{REG}_1 \cdot \text{REG}_2 \ \text{REG}_1 \mid \text{REG}_2 \]

Note. Operators on a same line have the same precedence.
1.4 Predefined strings in Ovide

1. *Any alphabet* :
   - .
   - \textbackslash NUM
   - \textbackslash x[0-F][0-F]

2. *ASCII only* :
   - \textbackslash d
   - \textbackslash w
   - \textbackslash s
   - \textbackslash D
   - \textbackslash W
   - \textbackslash S
   - \textbackslash a
   - \textbackslash b
   - \textbackslash f
   - \textbackslash n
   - \textbackslash r
   - \textbackslash t
   - \textbackslash v

1.5 Examples

Here are some examples of regular expressions, their “meaning” and their FSM representations. All examples work on the same alphabet : \{a-j\}.

- abc|de(f|i)+
  - abc or de followed by at least one f or i.
2. About rewrite rules

2.1 Standard rewrite rules

Basically, a rewrite rule describes a transduction between two regular expressions. However, rewrite rules take generally the following form:

$$\phi \rightarrow \psi :: \lambda \_ \rho$$ (2.1.1)

which expresses that $\phi$ must be rewritten $\psi$ when surrounded by $\lambda$ and $\rho$. In this notation, $\lambda$, $\phi$, $\rho$ and $\psi$ are regular expressions. $\lambda$, $\phi$ and $\rho$ is called the upper part of the rule, while $\psi$ is the lower part of the rule.

Note that two constraints are expressed on sets of rules:

1. Rules are sorted from the most specific to the most general, so that a given rule may be applied only if no rule, more specific and relevant, has been met.
2. Any context of a given rule \((\lambda, \rho)\) that is the target of another rule will be rewritten as well: in short, all relevant rules for a given input will be applied, simultaneously or recursively.

Ovide observes these constraints.

### 2.2 Weighted rules

Many applications require a way of ranking all possible solutions for a given input. It is particularly true in applications related to natural language processing, like spelling correction, syntactic analysis or speech recognition. Mohri & Sproat (1996) were the first to propose an extension to weighted rewrite rules. This extension is also available in Ovide.

Weighted rewrite rules take the form:

\[
\phi \rightarrow \psi :: \lambda \_ \rho/\omega \tag{2.2.1}
\]

which expresses that the replacement \(\phi \rightarrow \psi\), performed when \(\phi\) is surrounded by \(\lambda \_ \rho\), receives the weight \(\omega\).

### 2.3 Optional rules

In some applications, like spelling correction, one wishes to align strings of symbols. String alignment is classically solved using Dynamic Time Warping. However, finite-state machines can provide a more convenient solution. The idea is to create a finite-state transducer \(F\) which allows optional transformations. \(F\) can then be thought of as a filter between two strings \(s_1\) and \(s_2\) to be aligned:

\[
s_1 \circ F \circ s_2 \tag{2.3.1}
\]

For this kind of application, weighted optional rules are required and take the form:

\[
\phi \? \rightarrow \psi :: \lambda \_ \rho/\omega \tag{2.3.2}
\]

which expresses that the replacement \(\phi \? \rightarrow \psi\) is facultative.

### 2.4 The empty string

In Ovide, the index of the first symbol of an alphabet is always 1. Hence, the index of the empty string is 0, and can be represented using two notations:

- decimal notation: \(\backslash 0\)
- hexadecimal notation: \(\backslash x00\)
3. Sections in Ovide

Here are examples of insertion and deletion rules, using the hexadecimal notation:

- Insertion: \x00 \rightarrow a :: b \_ c / 0.1
- Deletion: a \rightarrow \x00 :: b \_ c / 0.1

3 Sections in Ovide

[INFO]
* [CLASSIN]
* [CLASSOUT]
* [LANGIN] [DICIN]
* [RULE] [COND]
* [LANGOUT] [DICOUT]
* [INCLUDE]
* [COMPILE]

- Sections preceded by an asterisk (*) are optional.
- Sections presented on a same line are mutually exclusive: use just one of them.
- The sorting of the sections has to be observed carefully.

As you can see, only INFO is required. However, the compilation will give a result only if at least a language (LANGIN or DICIN) or a rule set (RULE or COND) is defined.

4 INFO

This section accepts the following keywords:

- ALPHAIN and ALPHAOUT
- SUBALPHAIN and SUBALPHAOUT
- APPLY
- FILTER

4.1 ALPHAIN and ALPHAOUT

ALPHAIN is necessary while ALPHAOUT is optional. By default, ALPHAOUT is equal to ALPHAIN.
1. **ASCII** : ASCII characters (256 symbols, ISO-8859-1)
2. **ALPHA** : alphabetic characters, including [a-z], [A-Z] and their accentuated counterparts (115 symbols) from 0 to 9 (10 symbols)
3. **ALPHANUM** : both **ALPHA** and **NUM** symbols (125 symbols)
4. **yourfile.symbol** : a user-defined alphabet, one symbol a line. First line may define \( \epsilon \) = \( \epsilon \) = \( \epsilon \) = where \( \epsilon \) is a string of maximum 8 characters. Note that standard epsilon is \( \backslash x00 \), not so difficult to use...

**Warning.** Ovide is *case sensitive* and must often compare several alphabets (on names and symbols). Hence, for a same alphabet, always use the same case in your declarations.

### 4.2 SUBALPHAIN and SUBALPHAOUT

Often, several FSMs work on the same alphabet, but don’t use the same part of this alphabet. In this case, using the whole alphabet in compiling all FSMs is time consuming and doesn’t improve the result.

The solution could be to define a special alphabet for each FSM with only those symbols really used for the compilation. However, this solution doesn’t work, because it won’t be possible to compose resulting FSMs together : common symbols of different (sub-)alphabets won’t share the same internal code value. For instance, for the ASCII alphabet where "A" has code 65, a first reduced alphabet could have code 1 for "A" if "A" is the first symbol of the subset, while a second could have code 32 for "A" if "A" is the 32th symbol of the subset...

**SUBALPHAIN** and **SUBALPHAOUT** are useful in this case. They allow the user to define in extension a really useful subset of either **ALPHAIN** or **ALPHAOUT**, but keep the original code value of the whole alphabet.

The use of **SUBALPHAIN** and **SUBALPHAOUT** is straightforward : just give between square brackets the list of useful symbols.

For instance, let’s say that the input alphabet is the ASCII set, with 256 symbols. But you only use symbols between A and Z, a and n and the space. Define your input subset like this:

```
ALPHAIN = ASCII
SUBALPHAIN = [A-Za-n \ ]
```
where "\ " is the space symbol.

The definition of an output subset is only allowed if input and output alphabets are different. For instance, on an output alphabet of phonemes, you could define the following subset:

\[
\text{ALPHOUT} = \text{Phonemes} . \text{symbol} \\
\text{SUBALPHOUT} = [\text{ieEa} \ a\sim \text{e}\sim \text{9}\sim \text{o}\sim \text{pbm}]
\]

**Warning.** Of course, the definition of a subset is allowed as soon as the corresponding alphabet has been declared, but not before... The following example shows you a wrong use of the SUBALPHAIN definition:

\[
\text{SUBALPHAIN} = [\text{A-Za-n}\ ] \\
\text{ALPHAIN} = \text{ASCII}
\]

### 4.3 APPLY

APPLY is an **optional** value.

\[
\text{APPLY} = \text{OUTPUT} \\
\text{INPUT}
\]

1. OUTPUT : is the default value, because it corresponds to the behaviour of a standard composition. On two FSMs \( T_1 \) and \( T_2 \) composed together \( (T_1 \circ T_2) \), the standard composition takes only the common interface \( (i.e. \text{the output alphabet of } T_1) \) into account.

2. INPUT : the idea is that a rule may be applied *if and only if* the input symbol has not been changed yet by another rule. Hence, the composition \( (T_1 \circ T_2) \) takes the *input* alphabet of \( T_1 \) into account as well.

**Example.** We have 2 rules:

\[
\text{rule 1} = a \rightarrow b \\
\text{rule 2} = b \rightarrow c
\]

We have a state with 3 transitions:

\[
\text{a :a} \\
\text{b :b} \\
\text{a :b}
\]

Behaviour with APPLY=OUTPUT:


\[
\begin{align*}
a : a & \rightarrow a : c \\
b : b & \rightarrow b : c \\
a : b & \rightarrow a : c \\
\end{align*}
\]

Behaviour with **APPLY=INPUT** :

\[
\begin{align*}
a : a & \rightarrow a : b \\
b : b & \rightarrow b : c \\
a : b & = a : b \\
\end{align*}
\]

4.4 **FILTER**

**FILTER** is an **optional** value.

\[
\text{FILTER} = \text{yourfile.fsm}
\]

During the rule compilation process, input and output symbols are automatically aligned as a **standard cross product**. For instance, `couvent -> k u^- v a^-` will be aligned:

\[
\begin{align*}
cou\text{v}\text{e}nt \\
k u^- v a^- \backslash x00 \backslash x00 \backslash x00 \\
\end{align*}
\]

This optional filter allows one to give the compiler a list of **preferential alignments**.

For instance, `v` is better with `v` than with `u`...

Thus, the filter is an FSM, which can be compiled from an Ovide file containing a simple list of **optional** weighted rewrite rules. For a complete example of an Ovide file using a filter, cf. section 17.3. On our example, the best possible alignment is:

\[
\begin{align*}
cou\text{v}\text{e}nt \\
k u^- \backslash x00 v a^- \backslash x00 \backslash x00 \\
\end{align*}
\]

**Warning.** The resulting FSM is **not always** smaller because you have used a filter. In contrast, it is at least more logical...

5 **CLASSIN**

This section allows the declaration of input classes and markers that will be used further in the Ovide file. A class is a regular expression, while a marker is interpreted as a special kind of symbols.
5.1 Standard classes

A class is declared on a line. The first string of the line is the class name, separated by blanks (spaces or tabulations) from the class definition. Here are some examples:

```
VOWEL a|e|i|o|u|y
CONS b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|z
```

For calling a class, its name is put between brackets. For example,

```
<VOWEL>
```

A class may be used as soon as it has been defined. Thus, in the declaration of another class, for instance:

```
VOWEL a|e|i|o|u|y
CONS b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|z
CHAR [<CONS><VOWEL>]
```

The class name is a string combining only alphabetic and/or numeric ASCII symbols: [a-z], [A-Z], [0-9] and ".". Note that "." and accentuated characters are forbidden.

```
EPS
NAME
VERB01
TASK-1
```

The class definition is a standard regular expression, including input symbols and operators:

```
abc
ab|c
(a ?b)|c
[a-z]+`
```

Of course, this section only accepts symbols from the input alphabet...

5.2 Special case: the markers

A marker is called like a class, between brackets. That is why we allow the declaration of markers in the class section. For example,

```
<MARKER>
```

**Definition.** A marker is a special symbol that does not belong to the alphabet. Hence, a marker cannot be confused with real symbols. It is inserted into a rewrite rule, in order to identify a phenomenon and to check its evolution. A marker may be seen as a *trigger*, a *mask* or a *freezer*. 
1. The trigger tells a required condition has been met, and allows the application of a given rule.

2. The mask hides a regular expression that could be rewritten by a further rule.

3. The freezer is temporarily inserted between two regular expressions, in order to prevent the creation of a new rule that could be rewritten by a further rule.

The way a marker is declared does not depend on its kind. The kind is never specified, but can be discovered in the way a marker is used.

Declaration. The first string of the line is the marker name, separated by blanks (spaces or tabulations) from the marker definition. The definition of a marker is not regular expression; it is only an identifier started by "":

- ATRIG &1
- AMASK &2
- AFREZ &3

6 CLASSOUT

This section allows the definition of output classes, defined on output symbols, that will be used later in the Ovide file.

7 LANGIN

On each line, a regular expression made of input symbols and operators. The result is the UNION of all expressions in a single input language.

8 DICIN

On each line a string of input SYMBOLS (no operators). The result is the UNION of all strings in a single input dictionary.

Hint. Interesting because quickly compiled.
9 RULE

(1) \( A \rightarrow B : : C \_ D / W \)
(2) \( A \rightarrow B : : C \_ D \)
(3) \( A : : C \_ D / W \)
(4) \( A \rightarrow B / W \)
(5) \( A \rightarrow B \)
(6) \( A?\rightarrow B : : C \_ D / W \)
(7) \( A?\rightarrow B : : C \_ D \)
(8) \( A?\rightarrow B / W \)
(9) \( A?\rightarrow B \)

(1) \( A \) becomes \( B \) and gets the weight \( W \) when surrounded by \( C \) and \( D \)
\( A, C \) and \( D \) : regular expression defined on input symbols
\( B \) : regular expression defined on output symbols
\( W \) : a weight, so far that weights are declared as standard (STD) or user-defined (DEF)

(2) \( A \) becomes \( B \) when surrounded by \( C \) and \( D \). No weight.

(3) \( A \) gets the weight \( W \) when surrounded by \( C \) and \( D \), but is not rewritten.

(4) \( A \) becomes \( B \) wherever it occurs, and gets the weight \( W \).

(5) \( A \) becomes \( B \) wherever it occurs, but this conversion is not weighted.

(6-9) \( ?\rightarrow \) means optional rule.
Hence, the result is that 2 paths are created for \( A \) in the machine.
One does not change \( A \), the other applies the rule.

Notes about start (\( ^ \)) and end (\( \$ \)) operators :

- Only \( C \) may use ^ at its beginning : it expresses that left context must start the string.
- Only \( D \) may use $ at its end : it expresses that right context must end the string.
- No other parts of the rules may use these operators.

Note about weights :

Weights in our rewrite rules are distances or probabilities. Be care, however, that weights \( w \)
are represented as inverse of logarithms:

\[-\log(P(w))\]  

(9.0.1)

Hence,

1) a smaller weight is **better** than a bigger one, and 0 is the best weight.
2) a negative weight is **forbidden.** The result of a best path algorithm will **be wrong** if negative weights are defined.
3) weights are **summed** along a given path.

10 COND

Same section as RULE, but indicates that **all** rules are optional. The rewrite operator is not important here, it may be ?-> or -> as well.

11 LANGOUT

On each line, a regular expression made of output symbols and operators. The result is the **union** of all expressions in a single output language.

12 DICOUT

On each line a string of output symbols (no operators). The result is the **union** of all strings in a single output dictionary.

**Hint.** Interesting because quickly compiled.

13 INCLUDE

13.1 Principle

This section allows the user to include other Ovide or FSM files. These files will be composed with the set of rules contained in the current file. Of course, the composition implies some constraints on the alphabets used by the files: all files must have a compatible interface for the composition.

So what? How does it work? A file \( F_1 \) working on the pair of alphabets \( \{\text{A} : \text{B}\} \) includes another file \( F_2 \). The file \( F_2 \) works on another pair of alphabets. In this case, the input alphabet of \( F_2 \) must be \( \text{B} \), while the output alphabet of \( F_2 \) may be whatever alphabet (\( \text{B} \), \( \text{A} \) or a new alphabet \( \text{C} \)).

What is the result? \( F_1 \circ F_2 \) will work on the pair of alphabet \( \{\text{A} : F_2[\text{output}]\} : \)}
13. INCLUDE

\{A :B\} \circ \{B :A\} \rightarrow \{A :A\}
\{A :B\} \circ \{B :B\} \rightarrow \{A :B\}
\{A :B\} \circ \{B :C\} \rightarrow \{A :C\}

Notes:

1. Of course, **a file may include more than one file**. In this case, remember that the pair of alphabets of the machine in building changes with every new inclusion.

2. **An included file may include files itself.** This inclusion system has no end and allows one to design machines thought of as multi-layers languages. Of course, at the end, the built machine only works on the input alphabet of the first machine and the output alphabet of the last included file.

**Warning.** What about the alphabets? Well, an including file works like standard files. This means that you have to give the ALPHAIN only if your file works on a single alphabet, both ALPHAIN and ALPHAOOUT if your file works on 2 alphabets. Note that,

- if your “including file” only... includes files, you have to give **only** the ALPHAIN of your first included file.
- if you include several files, don’t never give the ALPHAIN of the first included file and the ALPHAOOUT of the last included file : it won’t work... The reason is straightforward : you have to give a **compatible** interface between the main file and the first included file.

**A correct example.** \(F_1\), working on \{A :B\}, includes :
- \(F_2\{B :C\}\)
- \(F_3\{C :D\}\)
- \(F_4\{D :A\}\)

As you can see,
- \(F_2\) works on \{B :C\}. The pair of alphabets becomes \{A :C\}.
- \(F_3\) works on \{C :D\}. The pair of alphabets becomes \{A :D\}.
- \(F_4\) works on \{D :A\}. The pair of alphabets becomes \{A :A\}.

**A wrong example.** \(F_1\), working on \{A :B\}, includes :
- \(F_2\{B :C\}\)
- \(F_3\{B :C\}\)
- \(F_4\{C :D\}\)

In this case,
• $F_2$ works on $\{B : C\}$. The pair of alphabets becomes $\{A : C\}$.

• $F_3$ works on $\{B : C\}$, like $F_3$. This will interrupt the process because $\{B : C\}$ is not compatible with the current interface, $\{A : C\}$.

How to solve this problem? Actually, $F_2$ and $F_3$ must be a single file...

### 13.2 Syntax

The main principle is that files are given **without extension**. For instance,

```
categories
syntax
```

The system will first try to open a precompiled FSM (here, "categories.fsm" and "syntax.fsm"). If the FSM does not exist yet, the system will then try to open an Ovide file (thus, "categories.regexp" and "syntax.regexp"). Of course, the process will be aborted if neither files exist.

**Note.** The Ovide file is automatically recompiled if the "-r" option (for rebuild) is given on the command line (cf. section 16.2).

If you want to load an FSM that has no corresponding Ovide file, specify the ".fsm" extension, like this:

```
categories.fsm
syntax.fsm
```

**Warning.** In this case, the FSM is binary and must know its own alphabets. For the way of adding alphabets in an FSM, please see the FSM Library Documentation.

### 14 COMPILATION

The main compilation process is by default a composition ($\circ$) of 3 machines (if they exist, of course):

```
(LANGIN|DICIN) \cdot (RULE|COND) \cdot (LANGOUT|DICOUT)
```

This section allows the user to define another main compilation.

The principle is to use **precompiled files**, defined in the INCLUDE section. For instance,
15. Some special operators

Some rules are difficult to describe synthetically. Here is an example:

```
categories_A  # works on A :B
categories_B  # works on A :B
syntax       # works on B :C
```

To use these files in this section, just prefix their names with `@`:

```
@categories_A
```

The syntax of this section is equivalent to the syntax of regular expressions. However, one more operator has been defined for this section, which allows composition: `˚`.

Notes:

1. In our example, `categories_A` cannot be composed with `categories_B`, because they work on the same pair of alphabets. But they can be combined by union or concatenation, for instance.

2. In contrast, `categories_A` must be composed with `syntax`. They cannot be combined by union or concatenation, because they do not work on the same pair of alphabets.

Here is an example of correct compilation rule:

```
(categories_A|categories_B) ˚ syntax
```

You may declare several compilation lines. In this case, all lines are combined together by union. For instance,

```
@categories_A ˚ syntax
@categories_B ˚ syntax
```

are combined by union, which makes them equivalent to the preceding example.

Warning. With a COMPILE section, you must only use an INFO section. All other sections are forbidden.

15 Some special operators

15.1 The strict operator

Some rules are difficult to describe synthetically. Here is an example:
\[ i \ a \rightarrow \ a \ i \n\]
\[ i \ b \rightarrow \ b \ i \n\]
\[ i \ c \rightarrow \ c \ i \n\]
\[ \ldots \n\]

In these case, whatever the symbol which follows \( i \), \( i \) and the symbol are switched. However, the following line does not model this rule:

\[ i \ . \rightarrow \ . \ i \n\]

because this line means “\( i \) followed by any symbol” becomes “any symbol followed by \( i \)”:

\[ i \ a \rightarrow \ . \ i \n\]
\[ i \ b \rightarrow \ . \ i \n\]
\[ i \ c \rightarrow \ . \ i \n\]
\[ \ldots \n\]

For solving this problem, we defined the **strict operator** : \%. This operator, preceding a set of symbols, like \( . \) or a class (cf. sections 5 and 6), means that each symbol of the set must be processed separately. Thanks to this operator, the correct rule can be expressed like this:

\[ i \ . \% \rightarrow \% \ . \ i \n\]

**Recommendations.**

1. This operator must be written on both sides of the rule.
2. Authorized sections : only RULE (cf. Section 9).
3. Operator precedence : between (4) and (5) of the table of operator precedence (cf. Section 1.3).

### 15.2 The complement operator

The identifier \( \text{REG} \) shows the place of symbols.

\[ \text{REG}! \quad : \quad \text{complement of the language } \text{REG} \n\]

**Difference between Not and Complement.** The **Not** operator works on symbols. The **Complement** operator works on languages: it computes the difference between the language represented by \( \Sigma^* \) ("any finite string of symbols, including the empty string") and the language defined by the user.
15. Some special operators

\[ \hat{a} = (\Sigma - a) \]

Not

\[ a! = (\Sigma^* - \{a\}) \]

Complement

Recommendations.

1. Authorized sections: classes, languages, rules and conditions.
2. Operator precedence: line (5) of the table of operator precedence (cf. Section 1.3).

15.3 The composition operator

The identifier \texttt{REG} shows the place of symbols.

\[ \texttt{REG} \ast \texttt{REG} \quad (\text{the degree symbol}) \]

This operator allows the user to specify that 2 transducers musts be composed together.

Recommendations.

1. If both machines are automata instead of transducers, this operator performs an intersection. Cf. next point (15.4) for more information about this.

2. Authorized sections: only \texttt{COMPILE} (cf. Section 14).

3. Operator precedence: line (6) of the table of operator precedence (cf. Section 1.3).

15.4 The projection operator

The identifier \texttt{REG} shows the place of symbols.

\[ \texttt{REG}>i \quad : \quad \text{first projection (input) of the FST REG} \]

\[ \texttt{REG}>o \quad : \quad \text{second projection (output) of the FST REG} \]

Imagine you compute a (W)FST, and you would like to get the first (resp. the second) projection of this FST, namely the FSA working only on the input (resp. the output) symbols of the FST. In this case, you need to project the FST. For getting the first projection (input), use \texttt{>i}. For getting the second projection (output), use \texttt{>o}.
Recommendations.

1. if you want to build the intersection of 2 FSTs,
   (a) first, project both of them in order to get the FSAs you need,
   (b) second, compose them together.

For instance, $T_1$ works on alphabets $\{A:B\}$, $T_2$ works on alphabets $\{B:C\}$. You may build the intersection of these transducers, in so far as you take the second projection of $T_1$ and the first projection of $T_2$:

$$T_1 \circ T_2$$

2. Authorized sections : only COMPILE (cf. Section 14).

3. Operator precedence : line (5) of the table of operator precedence (cf. Section 1.3).

16 Command line

By typing ./ovide on the command line, the following help will be printed on the standard output:

```
ovide  mainfile.regexp  [-t]  [-g]  [-r]  [-c]  [-q]  [-mr|-bf]  [-s:]  [-d...]
       [-x...]  [-min=N]

-t     create fsmtxt
-g     create graph
-r     rebuild includes
-c     classes in output FSM
-q     quiet mode (no print)
-mr|-bf algorithms for compiling rules : mr=mohri (default),
     bf=beaufort
     Default is mohri
-s:    sort on 2 features from [iows] (Input, Output, Weight, State)
     Default is "-siw" (input, weight)
-d...  output directory is ... (relative or absolute)
-x...  add suffix ... to output files
-min=N only for DICIN. The N value is the number of lines
        compiled before minimizing
     Default is 10,000
```

16.1 Standard behaviour

If the only argument on the command line is a file name (e.g. mainfile.regexp), this file is compiled using the default options and a single output is created : the corresponding FSM
(mainfile.fsm). Default options are:

- The output file is created in the current directory.
- No rebuild: included files are not rebuilt if the corresponding FSM is found.
- No classes: transitions are simple symbols.
- Standard rewrite rule algorithm (Mohri’s) is used.
- Verbose mode. The current step is printed on the standard output.
- State transitions are sorted on: (1) Input, (2) Weight. Corresponds to the option "-siw" on the command line.

16.2 Options

-t create fsmtxt

Create a human-readable FSM (extension is .fsmtxt). For more information about the syntax of this kind of file, please see the FSM Library Documentation.

-g create graph

Create a graphical FSM (extension is .graph) that can be viewed using the AT&T tool Graphviz. For more explanation, see http://www.graphviz.org/.

-r rebuild includes

If the mainfile.regexp contains included files, this option forces Ovide to rebuild these includes as well. Otherwise, only the main file is rebuilt.

-c classes in output FSM

In order to minimize the machine as much as possible, transitions (belonging to a same state) that reach the same next state with the same weight can be combined in a single transition represented as a class of symbols.

If the machine is dumped, the hashmap containing these classes is stored in the binary format of the machine. Note that, in this case, alphabets (input and output) must be given to the machine when you load it for a composition. For the way of adding alphabets in an FSM, please see the FSM Library Documentation.
-q quiet mode (no print)

In the standard mode, the current step of the algorithm is printed on the standard output: current section, current regular expression or rule, current operation (composition, minimization), etc. If you are no interested in this kind of information, this last flag is for you...

-mr|-bf algorithms for compiling rules

In Ovide, the default algorithm which interprets the set of rewrite rules and builds the corresponding FSM is Mohri's. Note that you may still specify it on the command line (-mr).

You may use Beaufort's algorithm (-bf) as well. It is faster, but does not handle yet all kinds of rules: there is still a problem with rules like

\[
a \rightarrow b :: c\Sigma^* \subseteq \Sigma^*d
\]

(16.2.1)

where \(\Sigma^*\) includes \(a\) itself... In the position of \(\Sigma^*\), \(a\) is not rewritten with Beaufort's algorithm.

-s sort on 2 features

By default, transitions of a given state are ranked as follows: (1) Input, (2) Output, (3) Weight, (4) State. You may change this by specifying a new sorting on 2 values among Input, Output, Weight and State. For instance, -siw (Input, Weight), -sow (Output, Weight), -sws (Weight, State), etc. Note that for a composition, the sorting should be -sow for the first machine, -siw for the second one.

Warning. There is no space between the option and its argument.

-d... output directory

This option allows one to give an absolute or relative directory where the result of the compilation will be saved.

Warning. There is no space between the option and its argument. For instance, if the directory is example/graph/, the argument will be -d example/graph/. Note the slash at the end...
-x... suffix

The standard output name is the name of the main file with an .fsm extension instead of .regexp. For instance, if the file to be compiled is syntax.regexp, the compilation result will be syntax.fsm.

The -x option allows one to add a suffix to the output name. For instance, if you give -x_ext, the output name will be syntax_ext.fsm.

Warning. There is no space between the option and its argument, as you can seen in the preceding example.

17 Examples

Here are some examples of Ovide files, with their alphabets.

17.1 An easy example

17.1.a The Ovide file

[INFO]
ALPHAIN = alpha.symbol
ALPHAOUT = NUM
WEIGHT = DEF

[RULE]
a?-> 06 : : ^b _ d / .5 # the rule is facultative...

17.1.b alpha.symbol

\EPSILON=_
a   b   c   d   e   f   g   h   i   j
17.1.c The resulting machine

17.2 With an included file

- The main file works on alphabets `cat.symbol` and `group.symbol (A : B)`.
- The included file works on alphabets `group.symbol` and `phrase.symbol (B : C)`.
- Hence, the interface of both files is `group.symbol (B)`, and the result of the global compilation will be an FSM working on alphabets `cat.symbol` and `phrase.symbol (A : C)`.

17.2.a syntax.regexp

[INFO]

ALPHAIN  = cat.symbol
ALPHAOUT = group.symbol
WEIGHT   = NO

[RULE]

Noun  ->  N
Verb  ->  N : : Det _
Verb  ->  V

[INCLUDE]

syntax_inc

17.2.b syntax_inc.regexp

[INFO]

ALPHAIN = group.symbol
17. Examples

\[ \text{ALPHAOUT} = \text{phrase.symbol} \]
\[ \text{WEIGHT} = \text{NO} \]

\[ \text{[RULE]} \]

\[ V \text{ N} \to \text{ VP} \]
\[ V \to \text{ VP} \]
\[ N \to \text{ NP} \]

17.2.c cat.symbol

\[ \text{\EPSILON=} \_ \]
\[ \text{Det} \]
\[ \text{Noun} \]
\[ \text{Verb} \]

17.2.d group.symbol

\[ \text{\EPSILON=} \_ \]
\[ \text{N} \]
\[ \text{V} \]

17.2.e phrase.symbol

\[ \text{\EPSILON=} \_ \]
\[ \text{NP} \]
\[ \text{VP} \]

17.2.f The resulting machine

![Diagram of the resulting machine]
17.3 With a filter

17.3.a The Ovide file

[INFO]
ALPHAIN = ASCII
ALPHAOUT = Phonemes.symbol
FILTER = filter.fsm

[DICIN]
couvent # the language is thus restricted to "couvent"

[RULE]
couvent -> k u v a~ # to be aligned using the filter

17.3.b filter.regexp

[INFO]
ALPHAIN = ASCII
ALPHAOUT = Phonemes.symbol
WEIGHT = DEF

[RULE]
c ?-> [ksS]
c ?-> \x00 / 1  # c becomes ϵ with a weight of 1
c -> . / 2  # c becomes whatever else with a weight of 2
o ?-> [0o]
o ?-> u / .5
o ?-> \x00 / 1  # o becomes ϵ with a weight of 1
o -> . / 2  # o becomes whatever else with a weight of 2
...

17.3.c The resulting machine
Bibliographie


