Rule Definition Language and a Bimachine Compiler

Ivan Peikov
ivan.peikov@gmail.com

Abstract

This paper presents a new Rule Definition Language (RDL) designed to allow fast and natural linguistic development of rule-based systems. We also describe a bimachine compiler for RDL which implements two novel methods: for bimachine construction and composition respectively. As an example usage of the presented framework we develop Porter's stemming algorithm as a set of context-sensitive rewrite rules and compile them into a single compact bimachine.

Keywords

bimachine construction, bimachine composition, context-sensitive rewrite rule, rule-based system, regular function

1 Introduction

The efficient implementation of rule-based systems has traditionally been associated with finite-state techniques. It all started when Kaplan and Kay's finite-state transducers framework [1] was first developed. It suggested that a regular rewrite rule

$$\phi \rightarrow \psi \ / \ \lambda \ \_ \ \rho$$

can be efficiently encoded by a finite-state transducer (FST). This would allow to simulate the sequential application of several rules by running a single FST, obtained through composition. For practical applications when linear running time is desired FSTs could be further transformed into a deterministic device such as subsequential transducer or a bimachine [2].

A lot of work has been done in this direction showing that these or similar approaches can indeed be applied for various common linguistics tasks such as POS-tagging [2], speech processing [3], speech synthesis [4] and others.

In practice, however, the size of a rule-based system usually grows fast while attempting to cover all the complexity a natural language phenomenon contains. This presents a challenge to the rule compilers as the compilation time required for even simple systems easily grows beyond practical bounds. Several attempts were made to design simpler compilers which restrict the rewrite rules in one way or another and then construct directly the final deterministic device without going through the elaborate process of first building a large FST and then determinizing it. A notable example was presented by Skut et al. [5] who managed to efficiently compile together a limited class of regular rules allowing only single letter context focuses. The restriction context focus restriction actually implied that no context overlapping was ever possible, which, in essence, reduced the compilation complexity a lot.

The current work attempts to go one step further and describe a bimachine compiler that doesn't impose any restrictions on the regular rewrite rules and still avoids constructing the intermediate transducers. The compiler comes with, what we hope would be, an easy to use Rule Definition Language (RDL) that provides the full expressive power of regular rewrite rules. Finally, we illustrate the language and compiler usage by implementing Porter's stemming algorithm as a set of 21 rewrite rules, which are then compiled into a single bimachine.

2 The Language

Instead of starting directly with RDL's formal grammar, let's see a simple example first:

```
01 package example1;
02
03 alpha [abc];
04
05 replace: ("ab"|"bc") -> "" / "a"+ _ "c"+;
```

What we see on line 1 is the package definition. Every RDL source file is compiled into a single bimachine, which we refer to as `package`. As we'll see later on, the compiled packages can be reused by importing them into other packages. That is why, the package definition is mandatory and should come first in the source file to define the package name.

The next statement in the file is an alphabet definition. The `alpha` keyword is followed by a set of characters we'll use in the package. There are two atomic types in RDL — sets (enclosed by square brackets) and strings (enclosed by double quotes). One may use any of them in an `alpha` definition. Additionally, there could be one or more alphabet definitions scattered along the source file. It is important, however, that a character must be defined as part of the alphabet before used. Thus, if we swap lines 3 and 5 in the above example, the RDL compiler will complain of unknown character being used in string.

Finally, on line 5 we see a `replace` rule definition. The `replace` keyword identifies the compilation module, which has to compile the rule to follow. The compiler can be extended with other compilation modules which implement different bimachine compilation algorithms. Each rule contains (in its beginning) the name of the compilation module for which it was written, so no confusion could possibly arise. We refer to this name as *rule's type.*

After the type identifier, every rule contains two parts — *rewriting part* and, optionally, a *context restriction part*. If both are present they should be delimited by `/`.

The context restriction part contains left and right regular expressions separated by underscore (_) — exactly as in the classical notation. The regular expressions can contain not only the standard union (`|`), concatenation (`.`) and Kleene star (`*`) but also positive iteration (`+`), optionality (`?`), bounded iteration (`{m,n}`), intersection (`&`), difference (`-`) and negation (`!`) operators.

The rewriting part may contain union, concatenation, all kinds of iterations and optionality of rewrite expressions. A rewrite expression maps regular expression to a string (or something that evaluates as a string). The mapping is expressed through the arrow (`->`) operator. If no arrow operator is specified in a rewriting expression the expression is treated as identity rewriting (e.g. it doesn't change the context focus but merely constrains it). For instance, the following rewriting part

```
("a" -> "b") . "c"{3,} . ("a" -> "b")
```

would rewrite two a's with b's if they are separated by three or more c's. The middle rewriting expression acts as restriction and leaves the c's unmodified.

For cases when several rules need to be grouped together before passed to the respective compilation module a rule block might be used. In the case of `replace` module, a block of rules is treated as if they need to be applied simultaneously (with some conflict resolution strategy) and not sequentially one after the other. Rule blocks are surrounded by curly braces as in the following example:

```
replace: {
   "a" -> "A";
   "b" -> "B";
   "c" -> "C";
}
```

RDL also supports variables and functions which we won't cover in detail here as they are only important for notation simplification.

The language has two special types of statements that were added with reusability in mind — `include` and `import` statements. The first one merely includes another source file and the second allows for previously compiled packages to be reused without recompilation. As every package and every rule is actually compiled into a separate bimachine, there is no trouble mixing them as needed.

3 The Compiler

The RDL compiler goes through several phases while compiling a package source file.

Tokenization

At this stage the input file is read and split into a list of tokens. Except for its string representation each token contains the exact position (file name, line number and column) where it was located in the input. This information aids for better error reporting in the further stages. This is also the time when `include`'s are expanded and `alpha` statements are interpreted in order to build the alphabet set for the input file. Further on, each string and set token is checked for unknown characters, and failure is reported if any. Cyclic inclusion is also detected at this stage of the compilation process.

Syntax parsing

As usual, the list of tokens obtained during the tokenization stage is fed into the syntax parser. Due to RDL's syntax simplicity its parsing is pretty straight forward. The parser only needs one pass through the token list with no more than two look-aheads at a time. This makes it possible to run the tokenizer and parser simultaneously the latter reading what previous outputs and throwing it away after it is not needed any more. The result from the parsing process is a list of parse trees, each one representing a rule block or import statement.

Expansion

Further on, all variables and function calls are expanded. This is the time when all undefined variables and functions are reported resulting in compilation failure. As a result from this stage the parse trees from the previous stage are modified and contain only strings and sets in their leaves. No variable or function call tokens are left in the trees.

Optimization

At this phase, we attempt to merge strings and sets when possible. For example after this stage the parse tree for the following expression:

```
"a" . ("b" . "c")
```

will contain a single string node with value of `"abc"`. The same optimization is done for sets connected with the union (`|`) operator.

Bimachines compilation

This is the phase when the compilation modules come along. Each compilation module provides two routines - for syntax tree checking and for bimachine compilation. The compiler is not aware how exactly bimachines are compiled out of parse trees - this is taken care of by the respective compilation module. All the compiler has to do is find the compilation module, required by the currently compiled rule, and call its routines to first verify that the syntax tree satisfies the module's requirements, and then compile the syntax tree into a bimachine. The compilation details and algorithms are hidden in the modules which makes the compiler easily extensible.

`import` trees are handled by the compiler - it simply loads the requested package from the file specified by the respective `import` statement.

The result from this stage is a list of bimachines all sharing a single alphabet.

Package compilation

Finally, all bimachines constructed or imported during the previous stage are composed into a single bimachine, which is written into the package output file. We'll cover the bimachine composition algorithm separately in the next section.

4 Bimachine composition

Before we set out the bimachine composition algorithm, we need several definitions.

Definition 1 (Composition) *Let $\mathcal{B}'$ and $\mathcal{B}''$ are bimachines, defined as follows*

$$\begin{array}{lll} \mathcal{B}' = \langle \mathcal{A}'_L, \mathcal{A}'_R, \psi' \rangle & \text{, where} & \mathcal{A}'_L = \langle \Sigma, Q'_L, q'_L, \delta'_L \rangle \\ & & \mathcal{A}'_R = \langle \Sigma, Q'_R, q'_R, \delta'_R \rangle \\ \mathcal{B}'' = \langle \mathcal{A}''_L, \mathcal{A}''_R, \psi'' \rangle & \text{, where} & \mathcal{A}''_L = \langle \Sigma, Q''_L, q''_L, \delta''_L \rangle \\ & & \mathcal{A}''_R = \langle \Sigma, Q''_R, q''_R, \delta''_R \rangle \end{array}$$

We define the result from bimachine composition (or simply composition) of $\mathcal{B}'$ and $\mathcal{B}''$ as a bimachine $\mathcal{B}$ such that for any $\alpha \in \Sigma^*$, $\mathcal{B}(\alpha) = \mathcal{B}''(\mathcal{B}'(\alpha))$.

Here, for simplicity, we agree that the bimachines to compose share a single alphabet — both in their input and output.

Definition 2 *Let $\mathcal{B} = \langle \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$ is a bimachine. We extend transitively its output function with the following inductive definition*

- $\psi^*(q_1, \epsilon, q_2) = \epsilon$
- $\psi^*(q_1, a\alpha, q_2) = \psi(q_1, a, \delta^*_R(q_2, \widetilde{\alpha})) \circ \psi^*(\delta_L(q_1, a), \alpha, q_2)$

Let $\mathcal{B}'$ and $\mathcal{B}''$ are bimachines as defined in Definition 1. We'll build in several steps a bimachine $\mathcal{B} = \langle \mathcal{A}_L, \mathcal{A}_R, \psi \rangle$, realizing their composition.

First, we construct a nondeterministic finite-state automaton (NFA)

$$\mathcal{A}^N_L = \langle \Sigma, Q^N_L, S^N_L, \Delta_L \rangle$$

where $Q^N_L = Q'_L \times Q''_L \times Q'_R$, the set of starting states is $S^N_L = \{q'_L\} \times \{q''_L\} \times Q'_R$ and transition relation is $\Delta_L \subseteq Q^N_L \times \Sigma \times Q^N_L$, for which $\langle \langle p_1, q_1, r_1 \rangle, a, \langle p_2, q_2, r_2 \rangle \rangle \in \Delta_L$ iff the following hold:

1. $\delta'_L(p_1, a) = p_2$
2. $\delta'_R(r_2, a) = r_1$
3. $\psi'(p_1, a, r_2) = \beta \in \Sigma^*$
4. $\delta''^*_L(q_1, \beta) = q_2$

We construct $\mathcal{A}_L$ – the left automaton of the composition $\mathcal{B}$, by determinizing $\mathcal{A}^N_L$.

Absolutely symmetrically, we build the right automaton of $\mathcal{B}$, starting with

$$\mathcal{A}^N_R = \langle \Sigma, Q^N_R, S^N_R, \Delta_R \rangle$$

where $Q^N_R = Q'_R \times Q''_R \times Q'_L$, the set of starting states is $S^N_R = \{q'_R\} \times \{q''_R\} \times Q'_L$ and transition relation is $\Delta_R \subseteq Q^N_R \times \Sigma \times Q^N_R$, for which $\langle \langle s_1, t_1, u_1 \rangle, a, \langle s_2, t_2, u_2 \rangle \rangle \in \Delta_R$ iff the following hold:

1. $\delta'_R(s_1, a) = s_2$
2. $\delta'_L(u_2, a) = u_1$
3. $\psi'(u_2, a, s_1) = \beta \in \Sigma^*$
4. $\delta''^*_R(t_1, \widetilde{\beta}) = t_2$

By analogy with the left automaton, we construct $\mathcal{A}_R$ by determinizing $\mathcal{A}^N_R$.

We define the output function $\psi : Q_L \times \Sigma \times Q_R \to \Sigma^*$ to produce $\psi(L, a, R) = \omega$ iff there exist $\beta \in \Sigma^*$ and states $\langle p, q, r \rangle \in L$, $\langle s, t, u \rangle \in R$, such that

1. $\delta'_L(p, a) = u$
2. $\delta'_R(s, a) = r$
3. $\psi'(p, a, s) = \beta$
4. $\psi''^*(q, \beta, t) = \omega$

The proof of correctness is trivial and too tedious for the scope of the current paper.

5 Bimachine construction

In this section we describe a method for bimachine construction from a general regular rewrite rule. The proposed construction contains three stages each one of which building an intermediate bimachine. Those three bimachines are finally composed into the desired result bimachine, realizing the regular rewrite rule. We use the *leftmost-longest* disambiguation strategy which is probably the most natural and widely used in practice. Other strategies are also implementable within the proposed construction method (although we won't show how explicitly).

We assume that a τ / λ _ ρ rewrite rule is given, where λ and ρ are regular expressions and τ is a regular relation.

5.1 Marking the contexts

In this initial stage we construct a bimachine that would enrich its output with marker characters wherever the left and right constraints (prefix and suffix) of the rewrite rule are satisfied. For this purpose we extend the alphabet with two unused characters, `<` and `>` and call them left and right marker respectively.

Let Σ be the rule alphabet (both input and output) and $\omega \in \Sigma^*$ is an input word of length n. Left marker should be inserted at position i of ω iff $\overline{\omega_1 \omega_2 \ldots \omega_i} \in \mathcal{L}(\Sigma^* \lambda)$. Symmetrically, a right marker should be inserted at position j of ω iff $\overline{\omega_j \omega_{j+1} \ldots \omega_n} \in \mathcal{L}(\rho \Sigma^*)$.

Additionally, we would like to treat specially the empty context and insert another `<>` marker sequence at positions where both the left and right constraints hold.

For example, if we take the rule:

```
replace: "a"* -> "b";
```

where the context constraints are empty (e.g. always satisfied) the word `aaa` should be marked as

```
<><a><><a><><a><>
```

As already explained the left/right marker is inserted exactly before the position where the left/right constraint happens to be true (in our case this is anywhere) and an additional pair of markers is inserted to denote the empty context (this is done for purely technical reasons).

This behavior is implemented by a bimachine constructed as follows. First we construct two deterministic automata (DFA) $\mathcal{A}_i = \langle \Sigma, Q_i, q_i, F_i, \delta_i \rangle$ (for $i = 1, 2$) which accept exactly the languages of $\Sigma^*\lambda$ and $\Sigma^*\widetilde{\rho}$ respectively. The only requirement for these automata is that no transition ever enters back into initial state. That is $\delta_i(q, a) \neq q_i$ for any $q \in Q_i$ and $a \in \Sigma$ $(i = 1, 2)$.

Then for any $s_1 \in Q_1$, $s_2 \in Q_2$ and $a \in \Sigma$ we define the output function $\psi(s_1, a, s_2) = e_1 r_1 l_1 a r_2 l_2 e_2$ where

$$e_1 = \begin{cases} < & s_1 = q_1 \wedge s_1 \in F_1 \\ \epsilon & otherwise \end{cases}$$

$$r_1 = \begin{cases} > & \delta_2(s_1, >) \in F_1 \\ \epsilon & otherwise \end{cases}$$

$$l_1 = \begin{cases} < & s_1 \in F_1 \\ \epsilon & otherwise \end{cases}$$

and symmetrically for r_2, l_2 and e_2.

5.2 Filtering out the wrong markers

The purpose of this second phase is to construct a bimachine which will accept as input a word with left and right markers (inserted by the previous phase) and only leave those of them that surround the proper leftmost longest contexts. In the previously given example, the desired bimachine should output

```
<aaa>
```

Informally speaking, the *leftmost longest* replacement strategy requires that one traverses the input from left to right and deletes the markers (left or right) which appear to be in the middle of a word from the language of $< \tau >$.

In order to implement this behavior, we'll construct a *left output-driven bimachine*. The output-driven bimachines differ from the classical ones in their running semantics — their left (or right) automaton traverses not the input word but the output. The advantage of this semantics is that the output function can influence the behavior of bimachine's left automaton. In the current case, this means that by deleting certain markers from the input one can trick the left automaton into running as if they never existed. Having this power we are able to ignore markers that don't surround the proper contexts (e.g. those selected by the leftmost longest strategy).

It is shown [6] that any output-driven bimachine can be transformed into an equivalent classical bimachine.

First, we construct a nondeterministic finite state acceptor for the context focus τ. Then we add two new states - s and f and make them the only initial and accepting states respectively. Transitions are added from s to all states that were previously initial over the left marker and from all states that we previously accepting to f over the right marker. Finally, loop transitions over both markers are added for all other states in the acceptor. The result is an automaton $\mathcal{A}$ that accepts all instances of τ surrounded by left and right maker with possibly other markers inserted in the middle.

The right automaton $\mathcal{A}_R$ of the output-driven bimachine is obtained by structurally reversing $\mathcal{A}$, extending it to accept Σ^* as prefix and finally determinizing it. Similarly, we construct the left automaton $\mathcal{A}_L$ of the bimachine by extending $\mathcal{A}$ to accept any prefix from Σ^* then determinizing it.

The output function of the output-driven bimachine is defined to delete left markers when the left and right automata have reached states q_1 and q_2 respectively such that $q_1 \cap \delta_R(q_2, <)$ doesn't contain s or contains anything other than s or f. The first condition means that the focus of the rule is not matched even though the left constraint is satisfied. The second reason for marker deletion means that the marker is met in the middle of a context focus that hasn't yet finished, e.g. in case of context overlapping. Symmetrically, the right marker is deleted whenever $\delta_L(q_1, >) \cap q_2$ doesn't contain f or contains anything other than s or f. The first condition again means that the context focus is not satisfied, while the second one means that either some overlapping context has finished within the current one or there exist a longer context focus which should be preferred by the leftmost longest replacement strategy.

Finally, we transform the output-driven bimachine into an equivalent standard bimachine that reads only its input.

5.3 Finally replacing

Having marked exactly the leftmost longest context focuses that satisfy prefix, focus and suffix constraints of the rule it is trivial to do the actual replacement. Ambiguity might only occur if τ is ambiguous regular relation. In this case disambiguation is possible on the basis of an arbitrary transition ordering (*first-come* replacement preference is a good candidate).

After constructing the bimachines from 5.1, 5.2 and 5.3 we compose them and remove the markers from the result. This produces a bimachine realizing the given regular rewrite rule.

It is important to note that other replacement strategies (such as *leftmost shortest*, *rightmost longest* and *rightmost shortest*) can be easily implemented by slightly changing only the bimachine from 5.2. The RDL compiler allows all four strategies (leftmost/rightmost longest/shortest) to be applied when constructing bimachine for rewrite rule.

6 Going cheaper

While experimenting with the so constructed bimachines it became evident that they are not at all minimal. A certain degree of redundancy was observed even at the lowest level of the constructions. Due to bimachine composition's product nature even the smallest amount of superfluous states in the composed bimachines multiplied in the result which tended to produce large bimachines even for a small set of not so complex rules.

It is true indeed that the problem of bimachine minimization is not yet solved. However, there are certain techniques that can help significantly reduce at least the obvious redundancies. We'll discuss one of them now.

Definition 3 (Bimachine state equivallence) *Let $\mathcal{B} = \langle \mathcal{A}_1, \mathcal{A}_2, \psi \rangle$ be a bimachine where $\mathcal{A}_i = \langle \Sigma, Q_i, q_i, F_i, \delta_i \rangle$ is a DFA and $\psi : Q_1 \times \Sigma \times Q_2 \to \Sigma^*$ is the output function. We say that $s_{1,2} \in Q_1$ are equivalent iff the following are true:*

- $\delta_1(s_1, a) = \delta_1(s_2, a)$ *for any* $a \in \Sigma$
- $\psi(s_1, a, q) = \psi(s_2, a, q)$ *for any* $a \in \Sigma$ *and* $q \in Q_2$

Symmetrically, we say that $s_{1,2} \in Q_2$ are equivalent iff the following are true:

- $\delta_2(s_1, a) = \delta_2(s_2, a)$ *for any* $a \in \Sigma$
- $\psi(q, a, s_1) = \psi(q, a, s_2)$ *for any* $a \in \Sigma$ *and* $q \in Q_1$

It becomes evident from the definition of when two bimachine states are equivalent that merging equivalent states doesn't change the regular function represented by the bimachine or in other words results in smaller yet equivalent bimachine. We use that fact in the following iterative algorithm:

Algorithm 1 Remove equivalent states

```
Ensure: there are no equivalent states
1: repeat
2:     merge all equivalent states in $\mathcal{A}_L$
3:     merge all equivalent states in $\mathcal{A}_R$
4: until no states were merged
```

Note that the merging to the left and to the right might not succeed removing all equivalent states at once because the merge operation can possibly generate new equivalences.

Although simple, this technique helped reduce the size of the bimachine in the example to follow in the next section approximately 40 times.

7 Example: Porter's stemmer

Porter's stemming algorithm [7] is a set of 62 context rules which are applied sequentially in batches over a list of words. Although it is neither too complex nor too large, Porter's stemmer is a good example of a rule-based system implementable in RDL, because it utilizes a lot of the extended regular expressions' capabilities and relies heavily on sophisticated context restrictions.

The RDL implementation of the stemming algorithm consists of 21 context-sensitive rewrite rules. Follows an example of the first step the stemmer has to make. It combines the following rules:

```
SSES -> SS
IES  -> I
SS   -> SS
S    ->
```

These rules have to be run simultaneously and applied at the end of words. Longer replacement should take precedence when multiple possibilities exist. For instance, the word *pass* should be left unchanged because the third rule matches a longer suffix than the fourth.

Taking advantage of the *leftmost longest* replacement strategy these rules can be implemented as a single RDL rule:

```
replace: ("sses" -> "ss") |
         ("ies"  -> "i" ) |
         ("ss"   -> "ss") |
         ("s"    -> ""  ) / "" _ EOW;
```

Here `EOW` is a variable set to the newline character:

```
EOW = "\n";
```

It is clear that the longest suffix will start first which will make it leftmost of all the possibilities and the replacement strategy will prefer it.

The whole set of 21 RDL rules was compiled into a single bimachine with 4524 states in the left automaton and 433 in the right. The compilation took 28 seconds on a 3.40GHz Intel Pentium 4 CPU and needed 134MB of memory.

8 Conclusion

We presented a bimachine compiler for RDL, based on two new methods for constructing and composing bimachines. The language doesn't impose any severe restrictions on the regular rewrite rules and provides the rule developer with a rich set of finite-state tools. Even though unlimited rewrite rules are allowed, the compiler doesn't use finite-state transducers as an underlying framework but works directly on bimachine level. Our hope is that this will lead to better performance and smaller compiled bimachines. Currently, there is no similar compiler publicly available for comparison, but the result from our experiment with Porter's stemming algorithm implementation in RDL promises that the RDL compiler might indeed become applicable for small-to-middle-sized rule-based systems. Further experiments will prove that right or wrong.

References

[1] Ronald M. Kaplan, Martin Kay. Regular Models of Phonological Rule Systems. *Computational Linguistics*, 20(3): 331-378, 1994.

[2] Emmanuel Roche and Yves Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2): 227-253, 1995.

[3] Mehryar Mohri. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2): 269-312, 1997.

[4] Wojciech Skut, Stefan Ulrich, Kathrine Hammervold. A Flexible Rule Compiler for Speech Synthesis. *Intelligent Information Systems 2004*, 257-266

[5] Wojciech Skut, Stefan Ulrich, Kathrine Hammervold. A Bimachine Compiler for Ranked Tagging Rules. *CoRR cs.CL/0407046*, 2004.

[6] Ivan Peikov. Direct Construction of a Bimachine for Context-Sensitive Rewrite Rule. *http://www.fmi.uni-sofia.bg/fmi/logic/theses/peikov-en.ps*, 2006.

[7] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3): 130-137, 1980.

9 Appendix: RDL implementation of Porter's stemmer

As already mentioned the RDL implementation of Porter's stemming algorithm consists of 21 rewrite rules. They all make use of several variables (defined in the beginning of the RDL source file) and can be divided into 5 groups as suggested by Porter in his original paper [7].

Common definitions and preprocessing

```
package porter;

alpha [abcdefghijklmnopqrstuvwxyz];
alpha "\n";
alpha "\u00005000";
alpha "\u00005001";
alpha "\u00005002";

# Word boundary
BOW = "\n";
EOW = "\n";

# Marker definitions
M = "\u00005000";
S = "\u00005001";
E = "\u00005002";

# Letter definition
A = [abcdefghijklmnopqrstuvwxyz];

# Consonant and vowel conditions
CVp = [aeiou]+;
CVm = [bcdfghjklmnpqrstvwxyz].
      [bcdfghjklmnpqrstvwxz]*.
      [aeiouy].
      [aeiou]*;
CVs = [bcdfghjklmnpqrstvwxyz].
      [bcdfghjklmnpqrstvwxz]*;

# Conditions used in Porter's rewrite rules
MGT0 = BOW . (CVp | CVm) . CVm* . (CVs | CVm);
MGT1 = BOW . (CVp | CVm) . CVm+ . (CVs | CVm);
VGT0 = BOW . (CVp | CVm) . CVm* . (CVs | CVm)?;
MEQ1 = BOW . (CVp | CVm) .        (CVs | CVm);

O = BOW? . A* . [bcdfghjklmnpqrstvwxyz].
                [aeiouy].
                [bcdfghjklmnpqrstvz];

# Prevent rewriting of short words
replace: ("" -> E) . EOW / BOW . A{1,2} _ "";
```

The final replace rule inserts marker character between words of length smaller than 3 and the end-of-word boundary. This is done in order to prevent the processing of these words by the rules to follow.

Step 1a

```
replace: ("sses" -> "ss") |
         ("ies"  -> "i" ) |
         ("ss"   -> "ss") |
         ("s"    -> ""  ) / "" _ EOW;
```

Step 1b

```
replace: ("" -> S) . ("eed"|"ed"|"ing") / "" _ EOW;
replace: (S."eed") -> "ee" / MGT0 _ EOW;
replace: (S.("ed"|"ing")) -> M / VGT0 _ EOW;
replace:  S -> "";

replace:  ( ("at"->"ate") |
            ("bl"->"ble") |
            ("iz"->"ize") |
            ("bb"->"b"  ) |
            ("cc"->"c"  ) |
            ("dd"->"d"  ) |
            ("ff"->"f"  ) |
            ("gg"->"g"  ) |
            ("hh"->"h"  ) |
            ("jj"->"j"  ) |
            ("kk"->"k"  ) |
            ("mm"->"m"  ) |
            ("nn"->"n"  ) |
            ("pp"->"p"  ) |
            ("qq"->"q"  ) |
            ("rr"->"r"  ) |
            ("tt"->"t"  ) |
            ("vv"->"v"  ) |
            ("ww"->"w"  ) |
            ("xx"->"x"  ) |
            (  ""-> S   ) ) . (M -> "") / "" _ EOW;
replace: ( S -> "e"  ) / MEQ1 & O _ "";
replace:   S -> "";
```

Step 1b

```
replace:  "y" -> "i" / VGT0 _ EOW;
```

Step 2

```
replace: ("ational" ->  "ate" ) |
         ("tional"  ->  "tion") |
         ("enci"    ->  "ence") |
         ("anci"    ->  "ance") |
         ("izer"    ->  "ize" ) |
         ("bli"     ->  "ble" ) |
         ("alli"    ->  "al"  ) |
         ("entli"   ->  "ent" ) |
         ("eli"     ->  "e"   ) |
         ("ousli"   ->  "ous" ) |
         ("ization" ->  "ize" ) |
         ("ation"   ->  "ate" ) |
         ("ator"    ->  "ate" ) |
         ("alism"   ->  "al"  ) |
         ("iveness" ->  "ive" ) |
         ("fulness" ->  "ful" ) |
         ("ousness" ->  "ous" ) |
         ("aliti"   ->  "al"  ) |
         ("iviti"   ->  "ive" ) |
         ("biliti"  ->  "ble" ) |
         ("logi"    ->  "log" ) / MGT0 _ EOW;
```

Step 3

```
replace: ("icate" -> "ic") |
         ("ative" -> ""  ) |
         ("alize" -> "al") |
         ("iciti" -> "ic") |
         ("ical"  -> "ic") |
         ("ful"   -> ""  ) |
         ("ness"  -> ""  ) / MGT0 _ EOW;
```

Step 4

```
replace: ("" -> S) . ("al"    |
                      "ance"  |
                      "ence"  |
                      "er"    |
                      "ic"    |
                      "able"  |
                      "ible"  |
                      "ant"   |
                      "ement" |
                      "ment"  |
                      "ent"   |
                      "ion"   |
                      "ou"    |
                      "ism"   |
                      "ate"   |
                      "iti"   |
                      "ous"   |
                      "ive"   |
                      "ize"   ) / "" _ EOW;

replace: (S . "ion") -> "" /
            MGT1 & (BOW . A* . [st]) _ EOW;
replace: (S -> "") / "" _ "ion" . EOW;
replace: (S . A*) -> "" / MGT1 _ EOW;
replace:  S -> "";
```

Step 5a

```
replace: ("e" -> "") / MGT1 _ EOW;
replace: ("e" -> "") / MEQ1 - O _ EOW;
```

Step 5b

```
replace: "ll" -> "l" / BOW . (CVp | CVm) .
                              CVm* .
                             (CVm | CVs) _ EOW;
```

Postprocessing

After all the work is done it remains to remove the marker characters inserted in the beginning, which prevented the rewriting of short words:

```
replace: E -> "" / "" _ EOW;
```