

YRX tagged regular expression

Remo Dentato

rdentato@users.sourceforge.net

20 January, 2008

1. Introduction

This document describes the regular expressions algorithm used in YRX, (<http://yrx.googlecode.com>), a tool to create lexical scanners.

The way YRX operates is very similar to `re2c` (<http://re2c.sourceforge.net>) where the scanners are embedded into regular C code. The main difference is that YRX uses *tagged automata* to capture parts of the matching text.

The most complete work on tagged automata I'm aware of is from Ville Laurikari whose regular expressions library `tre` (<http://laurikari.net/tre>) offers a POSIX compliant regexp extended with submatch capability.

The YRX algorithm described here, converts a set of regular expressions in a NFA and then in a DFA. All expressions are intended to be anchored at the first character.

No claim is made on compatibility with any other regex package (Posix, PCRE, ...)

2. From RE to NFA

To convert a regular expression on an alphabet Σ into a NFA, YRX proceeds as described in figure 1 where:

- a and b are characters of Σ ;
- X and Y are generic regular expressions;
- ϵ denotes empty transitions that can be followed without consuming any input;
- λ denotes transitions toward the final state 0 and must be followed only when no other match is possible.

Every state has a λ -transition but for simplicity only those that carry a set of tags are shown in the diagrams.

Note that the expression $(X)^*$ captures 0 or more repetitions of X not just the last one. In other words, the expression $(ab)^*$ captures `ababab` when matching the input text `abababcde`.

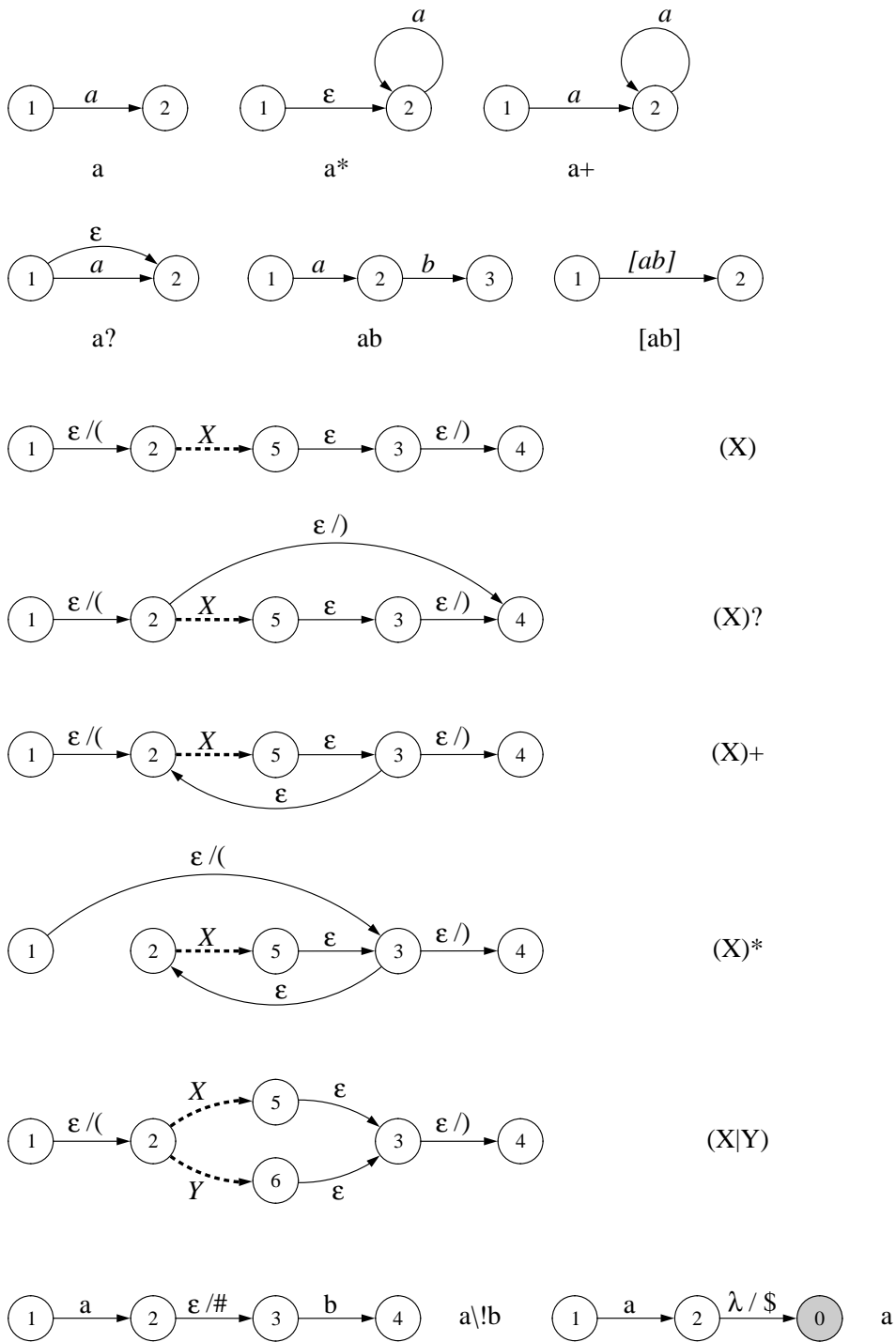


Figure 1. Construction of NFA

3. Tagged transition

The common algorithms that deal with the RE->NFA->DFA translation require to be modified to work properly when tagged transitions are introduced. One common assumption in those algorithms is that the two expressions $a(b)$ and $(a)b$ are equivalent. This is true when we are only interested in the language recognized by the RE but becomes false when we also want

the captured text enclosed in the parenthesis.

We'll consider tags in the form: $\tau_{x,k}^d$ where x is the *type* of the tag, d its *displacement* and k the index of the expression the tag is related to. (Remember that YRX transform a set of RE into a DFA).

When an arc containing the tag $\tau_{x,k}^d$ is chosen, a variable $v_{x,k}$ is given the value $(pos - d)$ where pos is the current position within the input text.

The tag *increment* operation $()^+$ is defined such that: $(\tau_{x,k}^d)^+ = \tau_{x,k}^{(d+1)}$

The increment operation over a set of tags $S = \{s_1, s_2, \dots, s_n\}$ is defined such that:

$$(S)^+ = \{(s_1)^+, (s_2)^+, \dots, (s_n)^+\}$$

Given two sets of tags S and R the following operations are defined:

$$\begin{aligned} S \cup_{\tau} R = \{ & \tau_{x,k}^d \mid (\tau_{x,k}^d \in S \wedge \nexists d': \tau_{x,k}^{d'} \in R) & \text{tag-union} \\ & \vee (\tau_{x,k}^d \in R \wedge \nexists d': \tau_{x,k}^{d'} \in S) \\ & \vee (\exists d', d'': \tau_{x,k}^{d'} \in S \wedge \tau_{x,k}^{d''} \in R \wedge d = sel_x(d', d'')) \} \end{aligned}$$

$$\begin{aligned} S \cap_{\tau} R = \{ & \tau_{x,k}^d \mid (\tau_{x,k}^d \in S \wedge \exists w, d': \tau_{w,k}^{d'} \in R) & \text{tag-intersection} \\ & \vee (\tau_{x,k}^d \in R \wedge \exists w, d': \tau_{w,k}^{d'} \in S) \\ & \vee (\tau_{x,k}^d \in S \wedge \tau_{x,k}^d \in R) \} \end{aligned}$$

$$S \setminus_{\tau} R = \{ \tau_{x,k}^d \mid \tau_{x,k}^d \in S \wedge \tau_{x,k}^d \notin R \} \quad \text{tag-difference}$$

The *tag-union* is defined as the regular union on sets except that if there are two tags of the same type and they also refer to the same RE, $\tau_{x,k}^{d'}$ and $\tau_{x,k}^{d''}$ only one of them is included. Which one is chosen depends on the type of the tag.

The *tag-intersection* is defined as the regular intersection on sets except that if one of the sets contains tags that refer to the RE k and the other doesn't they are all included in the result.

In the following we'll refer to four types of tags:

Tag	Symbol	Var	$sel_x()$
Begin Capture	$(_n^d$	bc_n	max()
End Capture	$)_n^d$	ec_n	min()
Mark	$\#^d$	mrk	min()
Match	$\d	end	min()

The selection functions are chosen so to capture *longest-leftmost* subexpression.

4. From NFA to DFA

The determinization algorithm processes each state once removing ϵ -transitions first (possibly introducing new ambiguous transitions) and then resolving any ambiguities:

```

pushonce( stack, 1 )
while not isempty( stack ) do
    state = pop( stack )
    removeeps( state )
    mergearcs( state )
end
    
```

The algorithm uses a stack to ensure that only reachable states are visited. The function `pushonce()` only pushes a value in the the stack if that value has not already been pushed previously. The function `mergearcs()` will push into the stack the destination states so that the computation continues.

4.1. Elimination of ϵ -transitions

The key idea is that an ϵ -transition from a state x to a state y implies that all states reachable from y are also reachable from x . In other words the ϵ -transition from x to y can be removed and replaced with the set of arcs x to the states to the states reachable from y .

During this step we'll need to check if the NFA has ϵ -loops to avoid entering an infinite loop.

Figure 2 shows how labels and tags are propagated during ϵ -transitions removal. A is a set of characters, T_1 and T_2 are set of tags.

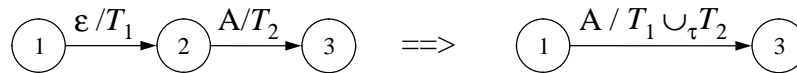


Figure 2. Removing ϵ -transitions

4.2. Resolving ambiguous transitions

The intuition behind figure 3 is that in state 1, if we get the character a , we can't decide between going to state 2 or to state 3. So we postpone the decision for when we'll have access to another character (state 4). The increment on the tags displacement will account for the fact that the tags should have been applied one character earlier. The general case is shown in figure 4

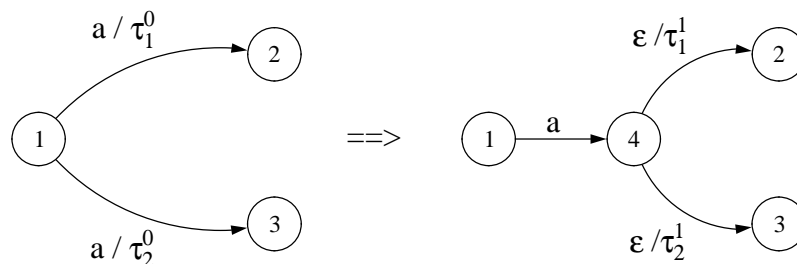


Figure 3. Merging transitions

where A and B are set of characters, T_1 and T_2 set of tags.

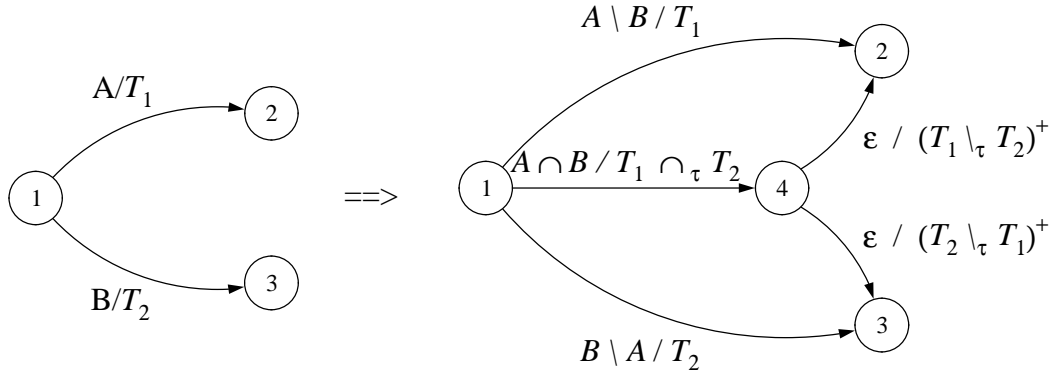


Figure 4. Merging transitions

Unfortunately an algorithm based on this simple intuition does not always converge. There are times where ambiguities cannot be resolved simply looking ahead and we would enter in an endless loop, merging the same states over and over. Figure 5 shows the case for the expression $(a^*)a$. We merged the states 2 and 3 in step i), creating state 4. In step iii), starting from 4, we

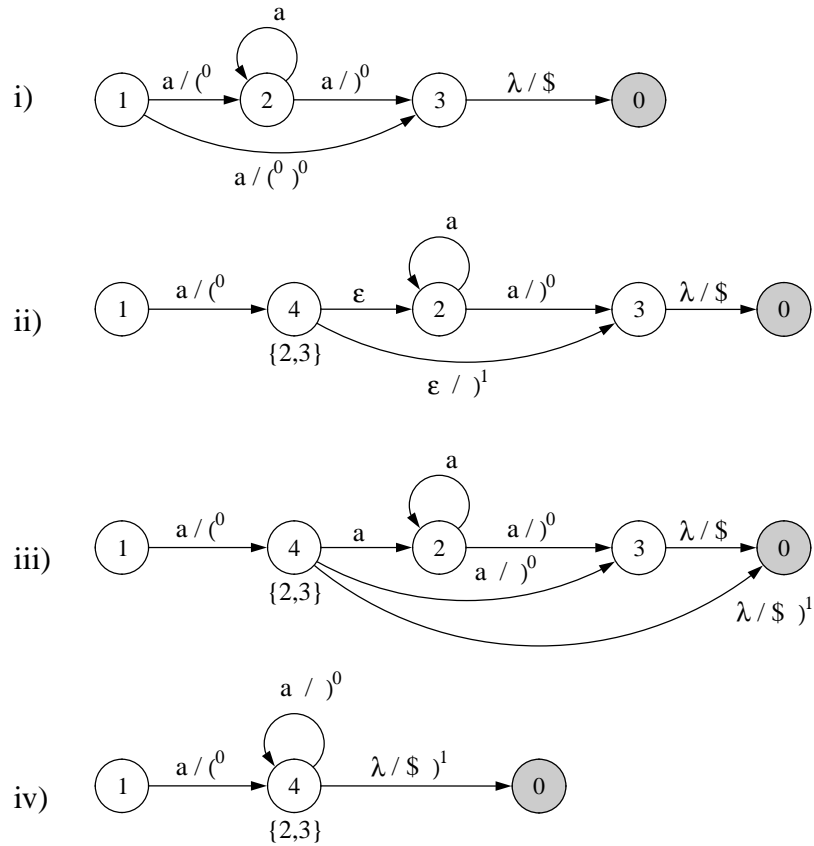


Figure 5. Avoiding infinite loop

should merge 2 and 3 again, and we would never stop. In these cases, to avoid the infinite loop, instead of merging the states again we'll create an arc from 4 into itself

5. Examples

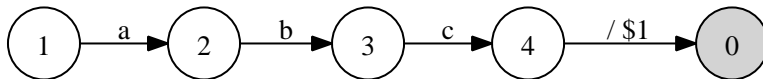
This section contains graphs of the DFAs generated by the current version of YRX and rendered with AT&T GraphViz (<http://www.graphviz.org>).

Capture parenthesis are represented with letters so that: a1_2 is $(_{1,1}^2$, A1_2 is $)_{1,1}^2$, b3 is $(_{2,3}^0$, B3_1 is $)_{2,3}^1$, etc.

5.1. Example 1

Just a simple string.

abc

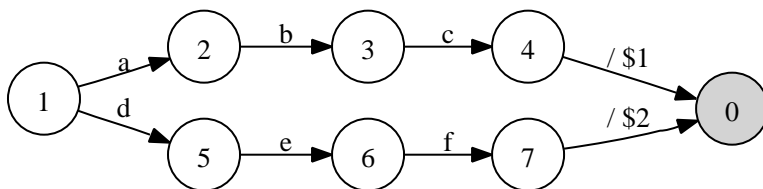


5.2. Example 1

Two strings.

abc

def

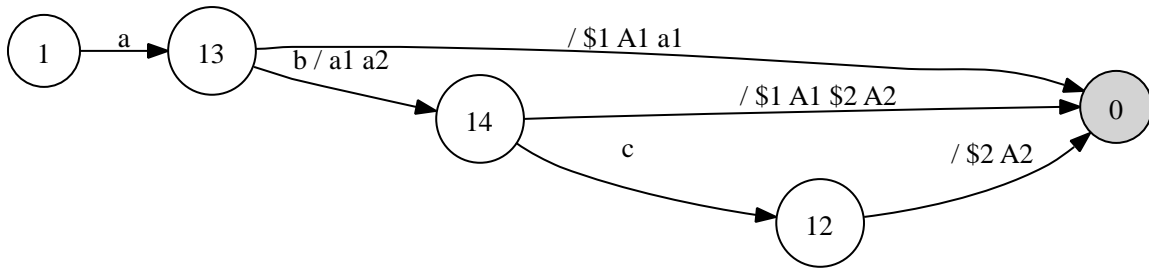


5.3. Example 3

Note that if the input is ab both expressions match.

a(b?)

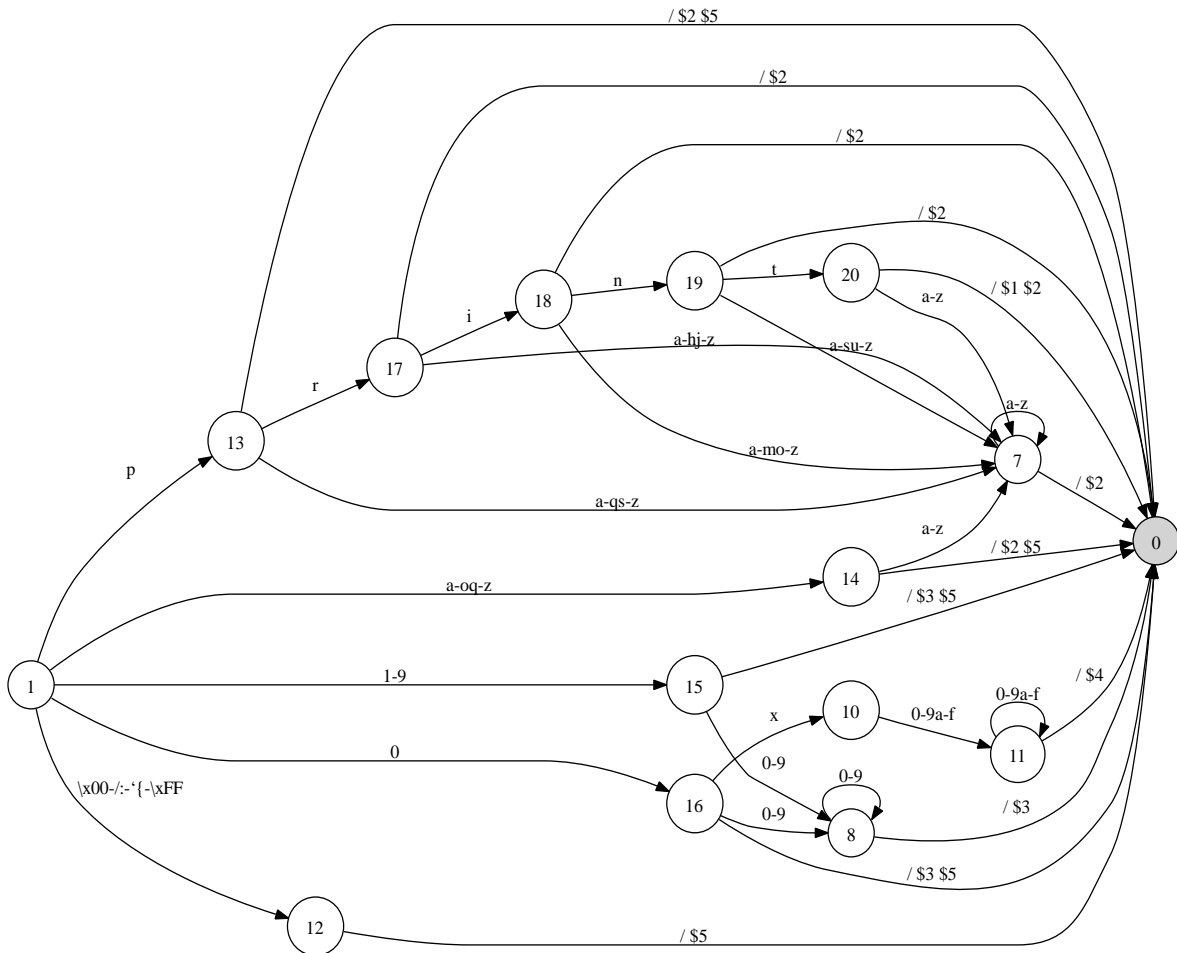
a(bc?)



5.4. Example 4

The example from the re2c report.

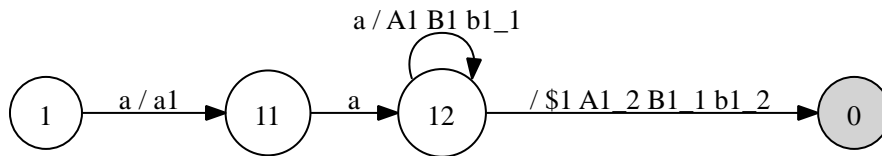
```
print
[a-z]+
[0-9]+
0x[0-9a-f]+
[\\000-\\377]
```



5.5. Example 5

Avoiding infinite loop.

$(a^*)(a)a$



5.6. Example 6

a function f with a variable number of decimal arguments.

$f\(((\backslash d+(\backslash d+)^*)?)\backslash)$

