Perl regular expression puzzle: even zeros and odd ones

Greg Bacon
gbacon@hiwaay.net

The challenge was to write a regular expression — subject to seemingly tight constraints: only literals, concatenation, Kleene star, grouping, and alternation — for binary strings whose total number of zeros is even and whose total number of ones is odd.

Expressed in Perl, we're after a regular expression equivalent to the following test:

```
sub meets_criteria {
    local $_ = shift;
    my $all01  = /^[01]*$/;
    my $odd1s  =     tr/1// & 1;
    my $even0s = not tr/0// & 1;

    $all01 && $odd1s && $even0s;
}
```

Formally, we want a regular expression that recognizes the following language:

$$\mathcal{L} = \{x \in \{0,1\}^* : \text{even}(\text{zeros}(x)) \wedge \text{odd}(\text{ones}(x))\}$$

In "How Regexes Work", Mark-Jason Dominus demonstrates the equivalence between regular expressions and non-deterministic finite automata (NFAs). In fact, NFAs and their equivalent cousins deterministic finite automata (DFAs) are the underlying models for regular expression engines. Constructing an NFA that recognizes $\mathcal{L}$ is straightforward; see Figure 1.

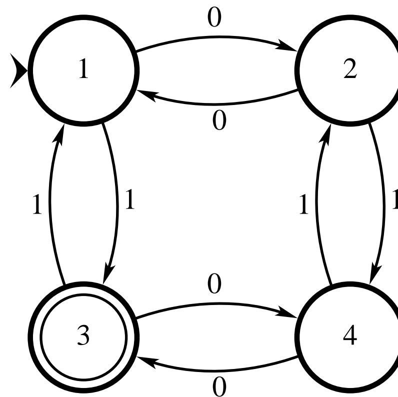

Figure 1: NFA that recognizes $\mathcal{L}$

Now the problem is to convert our NFA to an equivalent regular expression.

As a simple example, the NFA in Figure 2 recognizes the same inputs as the Perl regular expression `/^0*$/`:

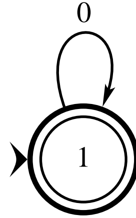

Figure 2: NFA for `/^0*$/`

Both `/^(11)*$/` and Figure 3 recognize strings with an even number of ones:

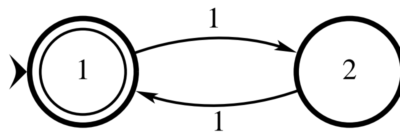

Figure 3: NFA for `/^(11)*$/`

A generalized two-state NFA, thanks to Jan Daciuk, is depicted in Figure 4.

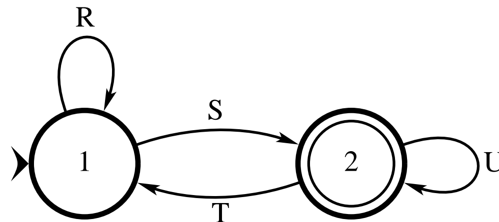

Figure 4: Generalized two-state NFA

A corresponding Perl regular expression is `/^(R|SU*T)*SU*$/`. The `(R|SU*T)*` portion allows the input to run around the loop between state 1 and state 2 however it it likes, taking the R and U diversions if necessary and then finally to the accept state to stay — captured by `SU*`.

It's obvious that we'd like to go from the four-state NFA in Figure 1 to this easy template. One way to do this is to use a method called state elimination in which we recursively remove states other than the start state and the accept state. If our NFA had multiple accept states, we'd have to repeat this process for each accept state and then compute a union.

Each state in an NFA encodes a certain amount of information. For example, in state 2 of Figure 1, we know that the machine has seen an odd number of zeros and an even number of ones. To be in state 2 of the generalized two-state NFA, the machine *must* have seen at least one copy of S. We

don't want to lose information, so the tradeoff in canning a state is that the labels on the transitions (the labels on the arrows between the states) become regular expressions rather than single symbols.

Eliminating state 2 from Figure 1 produces Figure 5, shown below.

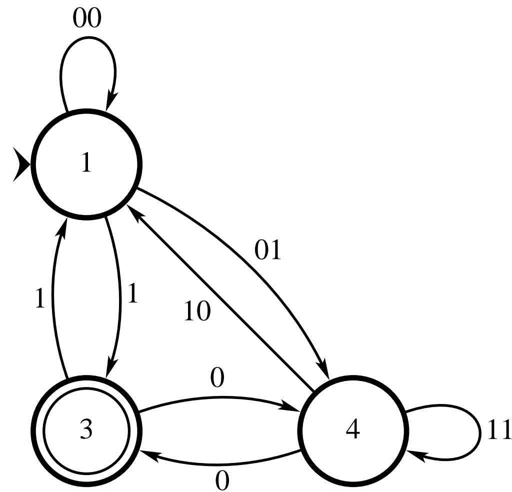

Figure 5: After eliminating state 2

To go from state 1 to state 2 and back to state 1, the NFA must see `00`, so in Figure 5, state 1 loops back on itself with a `00` transition. State 4 receives similar treatment but for input of `11`. Going from state 1 to state 4 requires input of `01`, and a return to state 1 would require `10`.

Dropping state 4 will bring us to victory! Starting in state 1 and looping back via state 4 requires input matching `01(11)*10`. From 1 to 3 by way of 4 requires `01(11)*0`. Roundtrip from state 3 through state 4 takes `0(11)*0`. The last leg, 3 to 4 to 1, needs `0(11)*10`. These changes give Figure 6, shown below (where, for reasons of typography, `+` denotes alternation, i.e., `|` in Perl regular expressions).

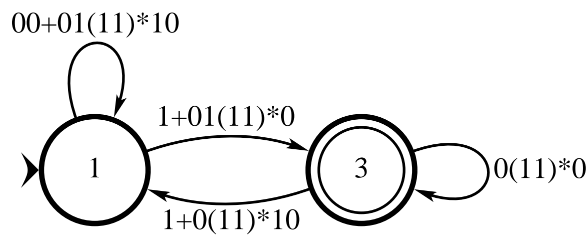

Figure 6: Original NFA reduced to two states

Plugging in to the template from Figure 4 produces the following Perl regular expression that meets the stated criteria:

```
qr/
    ^
    #(      R         +      S           U*          T       )*
     ( (00|01(11)*10) | (1|01(11)*0) (0(11)*0)* (1|0(11)*10) )*
    #     S           U*
     (1|01(11)*0) (0(11)*0)*
    $
/x
```

At least one other equivalent regular expression exists and results from eliminating state 4 first. This problem is left as an exercise for the reader.

Update

Roy Johnson sent an equivalent but much quicker regular expression:

```
qr/^
    (00        # Round-trip 1->2
    |11        # or round-trip 1->3
    |          # or:
     (01|10)   # 1->4 through either 2 or 3
     (00|11)*  # any roundtrips from 4->2 or 4->3
     (10|01)   # back to 1
    )*         # as often as you like.
    (1         # straight from 1->3
    |          # or:
     (01|10)   # 1->4 through either 2 or 3...
     (11|00)*  # any round-trips from 4->2 or 4->3
     0)        # then finish on 3
$/x
```

References

Dominus, Mark-Jason. “How Regexes Work”. *The Perl Journal*. Issue 9 (Spring 1998). http://perl.plover.com/Regex/

Daciuk, Jan. “Formal Language Theory (computation theory)”. http://odur.let.rug.nl/~daciuk/FLT/fltnew2.pdf

Kleiweg, Peter. “Finite State Automata”. http://odur.let.rug.nl/~kleiweg/automata/