Regular Expressions

We’ve seen that finite state machines can be used to model common machines such as combination locks and soda machines. They can also be used to model computers.

In general it can be difficult to analyze the behavior of a finite state machine. “Analyzing the behavior” means describing which input sequences produce which outputs. If the machine only has two possible outputs – 0 and 1 – then it is enough to understand which input sequences produce and output of 1 (that is, which sequences are recognized) because then we know that all other input sequences produce an output of 0.

Try to describe the behavior of the following finite state machine; which input sequences produce an output of 1?

Starting at the initial state (labeled A), an input of 0 or 1 will take you to the state (labeled B) that outputs 1. But then you could follow these inputs by any number of 0’s and still have an output of 1. So far we know that input sequences like the following are recognized:

0, 1, 00, 10, 000, 100, 0000, 1000, etc.

But once we reach state B we could come back to state A with a further input of 1. And then we could start all over again. We could do this last part any number of times.

It’s very awkward to describe this behavior in English words, so we’ll introduce some notation for describing patterns.

Definition If \( a \) is a string of symbols then \( a^* \) means “\( a \) any number of times” (including zero times).

Definition If \( a \) and \( b \) are strings of symbols then \( (a|b) \) means “\( a \) or \( b \)”.

Strings that can be described with this notation are called regular expressions.

For example, if our string is dog then \( (dog)^* \) describes all of the following strings:

\( dog, \quad dogdog, \quad dogdogdog, \quad etc. \)

as well as the empty string (which has no symbols at all).

If another string is cat then \( (cat|dog) \) means cat or dog, and \( (cat|dog)^* \) describes sequences like

\( cat, \quad dog, \quad catdog, \quad dogcat, \quad catcatdog, \quad dogdogcat, \quad catdogcat, \quad and \) so on.
Regular expressions can help us describe the behavior of finite state machines. Instead of using English words to analyze the behavior of the previous machine –

\[ (0|1)0^*(1(0|1)0^*)^* \]

which means (in English):

- 0 or 1 followed by 0 any number of times followed by the whole pattern (1 followed by 0 or 1 followed by 0 any number of times) any number of times.

It’s much easier to describe the behavior with regular expressions!

Here are the examples from the previous section with their behaviors described by regular expressions:

\[ 0^*(1(0|1))^* \]

\[ 01^*(01^*01^*)^* \]

\[ (0|1)1^*0(0|1)^* \]
The connection between regular expressions and finite state machines goes beyond just description:

**FACT (Kleene’s Theorem)** For every finite state machine there is a regular expression that describes its behavior and for every regular expression there is a finite state machine which has that regular expression as its behavior.

So, in some sense, regular expressions are the same thing as finite state machines.

Regular expressions are used in information technology whenever one has to look for a specific pattern. For example, many websites have a form which asks you for your email address. If you type in random letters instead of your actual email address then the website might tell you that what you typed isn’t a valid email address. To do this, they use a regular expression which describes the pattern of any email address: it should have a user name (made up of letters or numbers, and possibly the symbols _ % + - ), followed by the @ symbol, followed by at least one more letter (the domain name, which could also have a dot or hyphen), followed by a dot, followed by at least a couple more letters (the top level domain, such as com or us or info).

Writing out that regular expression using only a* and (a|b) can result in a very long regular expression, so it is common in programming languages to use **extended regular expressions**, which have additional notation as shortcuts for longer regular expressions.

Some notation for extended regular expressions (a is a string):

\[
a^+ = a \text{ one or more times (which is the same as } a a^* \text{)}
\]

\[
[ABCD] = \text{ any of the symbols inside the brackets, which for this example is the same as } (A | B | C | D)
\]

\[
0 - 9 = \text{ all numbers from 0 to 9 (which is the same as } (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) \text{ )}
\]

\[
A - Z = \text{ all letters from A to Z (which is the same as } (A | B | C | D | ... \text{ etc. } | Y | Z) \text{ )}
\]

\[
. = \text{ any symbol (if you want to indicate an actual dot in a situation where it might be taken to mean “any symbol”, you have to escape the dot by using a slash, like so: \. )}
\]

\[
a\{m, n\} = a \text{ from m to n times. For example } a\{2, 4\} \text{ means } a \text{ 2 to 4 times, which is the same as } (aa|aaa|aaaa) .
\]

Typically there are many other notations that are used in extended regular expressions, but these are enough to check if a string is a valid email address; you just check whether or not the string matches the following extended regular expression:

\[
[A-Z0-9_.%+\-]+@[A-Z0-9.-]+\.[A-Z]{2,4}
\]