

CS1Ah Lecture Note 14

Describing Finite State Machines

In Lecture Notes 12 and 13 we encountered finite state machines. The notation we used there consisted of a graphical description of the individual states and transitions in the finite state machine. We also defined some operations on finite state machines that allowed us to define larger machines in terms of smaller component machines. We also saw that the representation of finite state machines becomes very cumbersome as the machines get bigger. The basic notation for finite state machines is just too low level. To make effective use of finite state machine descriptions of systems with large numbers of states we need to find a better way of talking about finite state machines.

In this note we develop a notation that has a number of uses including describing finite state acceptors. The notation comprises a set of operators for building expressions that can be interpreted as descriptions of finite state acceptors or the sets of string accepted by such acceptors. These expressions are called *regular expressions*. We then go on to identify a number of equations that relate regular expressions. This system of equations allows us to reason about regular expressions by allowing us to prove when two different expressions represent finite state machines that accept the same set of sequences of input symbols. The system of equations is usually called *regular algebra*. Regular expressions have found wide use in computing as a method for describing patterns of symbols (*e.g.* in UNIX commands, text processing and compilers).

14.1 The Syntax of Regular Expressions

The *syntax* of a class of expressions defines what are valid ways of writing down expressions in the language. For expressions we do this by defining the constants and variables in the language and then give operators that show us how to build bigger expressions up from the variables and constants.

- A symbol from the input alphabet a, b, c etc. We use typewriter font for symbols. In the interpretation we build up these represent machines that just recognise the single sequence consisting of that single symbol.
- ϵ — this represents the empty string of symbols. This is a string with length 0. This represents a one state machine whose initial state is accepting.
- \emptyset — this is the empty set of symbols. This represents a one state machine whose initial state is not accepting.

- Sometimes we use variables like R, S, T, \dots to range over regular expressions.

14.1.1 Operators

Operators are a way of combining smaller expressions to make bigger, more complex, expressions. The standard version of regular expressions uses three operators we met in Lecture Note 13. We could add all the operators from Note 13 but this would not increase the expressiveness of regular expressions. Using these three we can describe the set of sequences accepted by any finite state acceptor. These operators are:

Sequence: If R and S are regular expressions then RS is also a regular expression.

The machine described by this expression is that found by using the sequence operator on the machines defined by R and S . The other name for sequence that is commonly used is *concatenation*.

Choice: If R and S are regular expressions then $R \mid S$ is also a regular expression.

The machine described by this expression is found by using the choice operator on the machines defined by R and S . The other name for choice is *union* because the set of sequences accepted by the machine described by $R \mid S$ is the union of those accepted by the machines described by R and S .

Repeat: If R is a regular expression then R^* is also a regular expression. The machine described by R^* is found by using the repeat operator on the machine described by R . The other name for repeat that is in common use is *closure*.

Just as in normal mathematical expressions, and in Java, we use the notion of *precedence* to suppress the number of parentheses we need to indicate how an expression should be grouped. Choice has lower precedence than sequence, which has lower precedence than repeat.

14.1.2 Language

There are two ways of interpreting regular expressions. One is as a means of describing finite state machines. The other is to see a regular expression as a representation of the set of sequences of symbols accepted by the machine it describes. Thus each regular expression represents a *language*. Languages that can be described by regular expressions are called *regular languages*.

14.1.3 Examples

We can describe the set of all valid floating point constants in Java as a regular expression¹. We use some definitions to build up the definition of the regular expression:

$$S = \epsilon \mid + \mid -$$

$$D = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

¹In fact we define a subset here, but the additional features are all easily definable as regular expressions but do not add anything to this example.

$$\begin{aligned}
 N &= D^* \\
 M &= S(DN.N | .DN) \\
 E &= \mathbb{E}SDN | \epsilon \\
 F &= ME
 \end{aligned}$$

In this definition: S is an optional sign, D is the set of digits, N is a sequence of digits, M is the mantissa, E the exponent and F is the regular expression representing a floating point constant. So, for example, $+12.01\text{E}-34$ is a valid constant while $-12\text{E}+5$ is not (why?).

In computer programs we often make use of variable names to refer to certain values previously stored. Depending on the language, there are different constraints on the names allowed for variables. Let's consider a language where names can be arbitrarily long, contains letters and digits but must start with a letter. Thus `anum1` is a legal variable name but `1anum` is not.

$$\begin{aligned}
 L &= A | \dots Z | a | \dots z \\
 D &= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\
 I &= L(L | D)^*
 \end{aligned}$$

Here I is the expression representing the identifiers of our programming language.

A less straightforward example is the regular expression for strings consisting of 0s and 1s with an even number of 1s. After a little thought we arrive at $(10^*1 | 0^*)^*$. This means that we can have as many 0s as we like, but whenever a 1 is encountered, it is eventually followed by another 1.

14.2 Algebraic Laws

Having our interpretation of regular expressions as languages means that we can look for equations that capture when two seemingly different expressions represent the same language. This kind of equivalence is captured in a collection of equations that capture basic properties of the operators of regular expressions. We can group the equations according to the operator they relate to. We begin with choice:

$$\begin{aligned}
 \emptyset | R &= R = R | \emptyset & (1) \\
 R | R &= R & (2) \\
 R | S &= S | R & (3) \\
 (R | S) | T &= R | (S | T) & (4)
 \end{aligned}$$

These equations capture the idea that the choice operator is just the same as set union. Now we consider sequence:

$$\begin{aligned}
 \epsilon R &= R = R \epsilon & (5) \\
 \emptyset R &= \emptyset = R \emptyset & (6) \\
 (RS)T &= R(ST) & (7)
 \end{aligned}$$

The remaining equations involve more than one operator:

$$R(S \mid T) = RS \mid RT \quad (8)$$

$$(R \mid S)T = RT \mid ST \quad (9)$$

$$\emptyset^* = \epsilon \quad (10)$$

$$RR^* = R^*R \quad (11)$$

$$RR^* \mid \epsilon = R^* \quad (12)$$

$$(R \mid S)^* = (R^*S^*)^* \quad (13)$$

$$(RS)^*R = R(SR)^* \quad (14)$$

For each equation we can check that the language defined on the left includes that on the right and vice versa. If we examine the machines constructed by the left and right hand sides of Equation 4 we can see that in both cases the initial states of the machines for R , S , and T are connected by ϵ -transitions from the initial state and similarly the final states of each of these machines connects to the final state via ϵ -transitions.

To show the applicability of these rules let us show that $0(10)^*1 \mid (01)^* = (01)^*$. Which rules are used in each step?

$$\begin{aligned} 0(10)^*1 \mid (01)^* &= 01(01)^* \mid (01)^* \\ &= 01(01)^* \mid 01(01)^* \mid \epsilon \\ &= 01(01)^* \mid \epsilon \\ &= (01)^* \end{aligned}$$

One immediate use of these laws is that if we were asked to build a FSM to recognise patterns defined by $0(10)^*1 \mid (01)^*$, we could use the algebraic laws to build a simpler (and hence less costly) one based on $(01)^*$.

14.3 Regular Expressions and Finite State Acceptors

We know from Lecture Note 13 that the operators used in regular expressions generate finite state acceptors. Thus we know that corresponding to every regular expression we have a finite state acceptor. The proof in the other direction is beyond this course but it is true that for any finite state acceptor we can define a regular expression whose language is exactly that accepted by the machine. Thus FSAs and regular expressions give us two different ways of talking about regular languages.

Murray Cole, 1st November 2002.