Lecture 4

Inf1A: Regular Expressions

4.1 Introduction

In Lectures 1 to 3 of this *Computation* section of Inf1A we have been learning all about Finite State Machines. We have covered both Deterministic Finite State Machines (D-FSMs) and Non-deterministic Finite State Machines (N-FSMs) in some depth, and discussed the differences between them. In the tutorial for week 9 (this week), we have seen a procedure called the *Conversion procedure* for converting an acceptor-type N-FSM into an acceptor-type D-FSM. We have not formally *proved* that this conversion procedure always constructs a D-FSM to accept exactly the same language (next year), but we have given some good evidence for this fact. From now on we will take it as read that the class of languages accepted by general FSMs is exactly the same class of languages accepted by deterministic FSMs. Recall from Lecture 3 that this class of languages is called the class of *regular languages*. We are going to study this class of languages in more depth in this lecture and the next one.

Although Finite State Machines look quite appealing and it is fun to work out which language an FSM accepts, they get quite complicated as the number of states and transitions increases. It would be nice to have a shorthand for talking about Finite State Machines and the languages they accept. We will concentrate on this task in this lecture. We first introduce a mechanism for defining *certain types of languages* over a finite alphabet Σ. We will refer to the expressions that we use to represent these types of languages as *regular expressions (r.ex.s)*. We will give the *syntax* for regular expressions, and give rules for determining which language is represented by a regular expression.

So what is the connection between the languages represented by regular expressions and our Finite State Machines? Well, in fact there is a theorem, called *Kleene's theorem*, which proves that the class of languages represented by some regular expression is exactly the same class of languages accepted by some FSM. In this lecture we will see just one half of *Kleene's theorem* - we will show that for every r.ex. R, we can construct a Finite State Machine to accept the language represented by R. The other half of Kleene's theorem (for every FSM, there is some r.ex. that represents the language accepted by that machine) is harder. It will probably be proved in Informatics 2.

Independently of their relationship to Finite State Machines, regular expressions are important in Computer Science and Artificial Intelligence as a method for describing patterns of symbols (*eg* in UNIX commands, text processing and compilers).

4.2 Regular Expressions

We define *regular expressions (r.ex.s)* in two stages in this section. We will first describe the *syntax* of regular expressions, which tells us what are valid ways of writing down regular expressions. We then define the *meaning* of a regular expression R, which tells us which *language* is represented by R. We give some examples of r.ex.s in practice. Finally we give some algebraic laws relating certain regular expressions to others.

4.2.1 Syntax of Regular Expressions

We define the syntax of regular expressions in an inductive fashion. We do this by first defining the *atomic* regular expressions (these are r.ex.s that cannot be decomposed) and then giving *operators* that allow us to build large regular expressions by putting together smaller regular expressions according to the rules. The syntax of regular expressions will depend on our symbol alphabet Σ.

Atomic r.ex.s

Here are the rules telling us which are the atomic r.ex.s over Σ.

- The empty set symbol $\emptyset$ is an (atomic) r.ex..
- The symbol ϵ is a (atomic) r.ex. We will see that this represents the empty string of symbols (a string with length 0).
- Any symbol from the input alphabet Σ is a (atomic) r.ex. We use `typewriter` font to represent symbols, eg `a` or `b` (if Σ is an alphabet containing these constants).

Sometimes we will use variables like $R, S, T, \ldots$ to range over regular expressions, when we want to talk about a general r.ex... This will not necessarily mean that such a variable denotes an atomic expression.

Operators

We use operators to combine small regular expressions to make bigger, more complex, regular expressions. The standard version of regular expressions uses three operators which may look familiar (in the context of Lecture 3 of this course). These operators are:

Sequence: If R and S are regular expressions then RS is also a regular expression.
This operation is also called *concatenation*.

Choice: If R and S are regular expressions then $R \mid S$ is also a regular expression.
This operator is sometimes called *union*.

Repeat: If R is a regular expression then R^* is also a regular expression.
This is sometimes called the *Kleene star* operation.

Just as in Logic, and in Haskell, we need to have a rule to tell us how an expression should be grouped, even when some parentheses are missing. The rule for r.ex.'s is that the *Repeat* operator binds most tightly (ie. repeat is "done first"), the *Sequence* operator binds the next most tightly, and the *Choice* operator binds the least tightly (ie, choice is "done last").

4.2.2 Language represented by a regular expression

We now give a set of rules to associate every well-formed regular expression with the language that it represents. Recall from our earlier lectures that a *language* over the alphabet Σ is just some set of strings from Σ^*. We now give an *inductive definition* of the language $L(R)$ represented by a well-formed regular expression R:

Empty set: $L(\emptyset) = \emptyset$. So the language represented by the $\emptyset$ symbol is the empty language (the language containing no elements).

Empty string: $L(\epsilon) = \{\epsilon\}$. The language represented by the ϵ symbol is the language containing just a single string, which is the empty string ϵ.

Symbol a: For any symbol $\mathsf{a} \in \Sigma$, we define $L(\mathsf{a}) = \{\mathsf{a}\}$ (the language containing just a single string, which is the string a).

Sequence: For any r.ex. R which is of the form $R = ST$, we define

$$L(R) = L(S)L(T) = \{xy \mid x \in L(S), y \in L(T)\}.$$

Choice: For any r.ex. R which is of the form $R = S \mid T$, we define

$$L(R) = L(S) \cup L(T) = \{x \mid x \in L(S)\} \cup \{y \mid y \in L(T)\}.$$

Repeat: For any r.ex. R which is of the form $R = S^*$, we define

$$\begin{array}{rcl} L(R) \;=\; L(S^*) & = & \{\epsilon\} \cup \{x \mid x \in L(S)\} \cup \{x_1x_2 \mid x_1, x_2 \in L(S)\} \cup \ldots \\ & = & \{x_1 \ldots x_n \mid n \in \mathbb{N}, x_1, \ldots, x_n \in L(S)\}. \end{array}$$

In this case we will usually write $L(R) = L(S^*) = L(S)^*$.

It is important to remember that a regular expression does not describe a *string* (even for the atomic cases), but a language, that is, a set of strings.

4.2.3 Examples

We can describe the set of all valid floating point constants in Java (which you will all study next semester) as a regular expression[1]. We use some definitions to build up the definition of the regular expression:

$$\begin{array}{rcl} S & = & \epsilon \mid + \mid - \\ D & = & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9} \\ N & = & D^* \\ M & = & S(DN.N \mid .DN) \\ E & = & \mathtt{E}SDN \mid \epsilon \\ F & = & ME \end{array}$$

[1] In 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.

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. For example, `+12.01E-34` is a valid constant while `-12E+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.

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

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

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

4.2.4 Algebraic Laws

It is interesting to consider the question of *simplifying* regular expressions. There are usually many regular expressions to represent the same language. Therefore it is helpful to have some equations to capture basic properties of the operators of regular expressions. We begin with *choice*:

$$L(\emptyset \mid R) = L(R) = L(R \mid \emptyset) \tag{4.1}$$
$$L(R \mid R) = L(R) \tag{4.2}$$
$$L(R \mid S) = L(R) \cup L(S) = L(S \mid R) \tag{4.3}$$
$$L((R \mid S) \mid T) = L(R \mid (S \mid T)) \tag{4.4}$$

These equations capture the idea that the choice operator is a way of representing set union. Now we consider *sequence*:

$$L(\epsilon R) = L(R) = L(R\epsilon) \tag{4.5}$$
$$L(\emptyset R) = L(\emptyset)L(R) = L(\emptyset) = L(R\emptyset) \tag{4.6}$$
$$L((RS)T) = L(R(ST)) \tag{4.7}$$

The first two of the sequence equations above depend on the definition of sequence, and on the particular properties of $\emptyset$ and ϵ.

The remaining equations involve more than one operator:

$$L(R(S \mid T)) = L(R)L(S \mid T) = L(RS \mid RT) \tag{4.8}$$
$$L((R \mid S)T) = L(R \mid S)L(T) = (L(R) \cup L(S))L(T) = L(RT \mid ST) \tag{4.9}$$
$$L(\emptyset^*) = L(\emptyset)^* = \{\epsilon\} = L(\epsilon) \tag{4.10}$$
$$L(RR^*) = L(R)L(R^*) = L(R^*R) \tag{4.11}$$
$$L(RR^* \mid \epsilon) = L(R^*) \tag{4.12}$$
$$L((R \mid S)^*) = L((R^*S^*)^*) \tag{4.13}$$
$$L((RS)^*R) = L(R(SR)^*) \tag{4.14}$$

Some of these equivalences are a bit more complicated than the earlier ones. It is possible to show the equivalences hold by using the definitions for $L(R)$ given in Subsection 4.2.2. For each equation it is possible to check that the language defined on the left includes that on the right and vice versa.

We will say that two regular expressions R and S are *equivalent* if and only if $L(R) = L(S)$. Sometimes we will be more careless and write $R = S$ for equivalent regular expressions.

To show the applicability of these rules let us show that $L(\mathtt{0(10)^*1} \mid \mathtt{(01)^*}) = L(\mathtt{(01)^*})$.

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

Can you work out which rules are used in each step?

4.3 Kleene's theorem (one half)

Our initial goal in introducing regular expressions was to have a shorthand for talking about the languages accepted by Finite State Machines. We have now presented regular expressions in some detail. Our final step is to relate the class of languages represented by regular expressions to the class of languages accepted by Finite State Machines.

Theorem 4.1: (Kleene's theorem) *A language L is a* regular language *(that is, a language accepted by* some *Finite State Machine),* if and only if *there is some regular expression R such that $L(R) = L$.*

Notice that in the statement of this Theorem, it is not necessary to decide which type of Finite State Machines we are talking about. That is because the class of languages accepted by N-FSMs is the same as the class of languages accepted by D-FSMs.

There are two steps involved in proving Kleene's theorem. It is necessary to show:
• For every regular expression R, it is possible to construct a Finite state Machine M such that $L(M) = L(R)$, and
• For every Finite State machine M, it is possible to construct a regular expression R such that $L(R) = L(M)$.

In *Informatics 1*, we will restrict ourselves to showing the first part of the proof. In fact, you will see that we have done most of the work already. You will probably see the second half of the proof next year.

Theorem 4.2: For every regular expression R, it is possible to construct a Finite state Machine M such that $L(M) = L(R)$.

Proof: We will prove Theorem 4.2 by induction on *the number of operators* ("sequence", $\mid$ and *) of the regular expression.

For our base case, we will show that every regular expression which has 0 operators in it (ie, is an atomic expression) can be represented by a Finite State Machine. This step is relatively straightforward, because we know from section 4.2.1 that there are only three types of atomic regular expressions, $\emptyset, \epsilon$ and $\mathtt{a}$.

Finite State Machines to recognise $L(\emptyset)$, $L(\epsilon)$, and $L(\mathsf{a})$ are given below in Figure 4.1. Notice the important difference between the machine for $L(\emptyset)$ and for $L(\epsilon)$.

Figure 4.1: N-FSMs to recognise $L(\emptyset)$, $L(\epsilon)$, and $L(\mathsf{a})$ (in that order).

For the inductive step, suppose that we have a regular expression R which has $k+1$ operators, and that that we already know how to build an FSM to recognise the language of any regular expression with *at most* k operators in it. We will see that this is enough to allow us to construct an FSM to recognise $L(R)$. Consider the operator at the top level of R. Since R is not atomic, it has the form $R = ST$ (then $L(R) = L(S)L(T)$), or $R = S \mid T$ (then $L(R) = L(S) \cup L(T)$), or $R = S^*$ (then $L(R) = L(S)^*$). For each of these cases, both S and T can have at most k operators. So we know that there exist FSMs to accept the languages $L(S)$ and $L(T)$.

In the case of $R = ST$, we use the machines for $L(S)$ and $L(T)$, and the construction of 3.3, to obtain a machine to recognise $L(R) = L(S)L(T)$. If $R = S \mid T$, then we use the construction of Figure 3.4. If $R = S^*$, then we use the construction of Figure 3.5. In all cases we obtain a FSM to accept the language $L(R)$, as required. □

4.4 Summary

The advantage of Kleene's theorem is that in order to show that a language is a regular language, we only need to write down a r.ex. for that language. Also the r.ex. can be simplified, using the laws from Subsection 4.2.4.

4.5 Notes

The `Wikipedia` link for regular expressions is very good, for the basics and also for giving details about r.ex.'s in operating systems (eg `Unix`):

`http://en.wikipedia.org/wiki/Regular_expressions`

Change to schedule: On Thursday (Lecture 5) we will be covering the material originally planned for Lecture 6 (in the "Plan of lectures" handed out at the beginning). This is the "Limits of FSMs" lecture.

These notes have been adapted from two sets of earlier notes, the first due to Stuart Anderson, Murray Cole and Paul Jackson, and the second due to Martin Grohe and Don Sannella.

Mary Cryan, November 2004