

Lecture 22: Regular expressions and finite state machines

This lecture describes regular expressions and their application in the unix search utility `egrep`, and finite state machines. Historically, a large part of the communication between humans and computers has been in the form of strings, that is, finite sequences of characters. Thus, considerable infrastructure has been developed in computer science for the specification, representation, and processing of sets of strings. Regular expressions and finite state machines are two such representations, which are used in many areas of computer science.

An `egrep` search

Imagine that I decided to find out how many times I had used the Scheme expressions `cadr`, `caddr`, etc. in my lecture notes for this course so far. The source files for my lecture notes are all in one directory, and have the extension `.tex`, for the document-preparation system `Latex`. I choose to do a search as follows.

```
argentina> egrep 'c(a|d)(a|d)(a|d)*r' *.tex
lecture-12.tex:(define get-op1 cadr)
lecture-12.tex:(define get-op2 caddr)
...
lecture-4.tex:(cadadr '(15 (d 2))) => 2
lecture-4.tex:\verb$(caddr ls)$, we can use \verb$(list-ref ls 2)$.
lecture-8.tex:to the person reading the code than \verb$(caddr inst)$
```

The command `egrep` invokes a unix utility to search for lines containing strings matching a given pattern in a file or files. The characters between the quotes specify the pattern to look for, and the expression `*.tex` directs the search to include all files with the extension `.tex`. The result of the search is printed out on the subsequent lines (I have not shown all 18 lines of the results.) Each line consists of the name of the file (eg, `lecture-12.tex`) and the line of the file that matched the pattern.

The pattern is specified by means of an “extended regular expression.” The particular extended regular expression I gave matches a `c` followed by an `a` or `d`, followed by another `a` or `d`, followed by any number (0 or more) occurrences of `a` or `d`, followed by an `r`. `Egrep` prints out any line that contains a string that matches this pattern. For example, in the first line, `cadr` matches the pattern, in the second, `caddr`, and in the last, `caddr`.

Definition of regular expressions

In this section, we define standard regular expressions and specify what strings they match. We’ll return to the extended regular expressions of `egrep` after we understand these more basic ones.

Regular expressions are defined over a finite set of symbols, for example, $\{a, b, c\}$. The definition is recursive (or inductive). The base case is that each symbol by itself is a regular expression. For the inductive cases, assume that E_1 and E_2 are regular expressions. Then E_1E_2 is a regular expression, $(E_1|E_2)$ is a regular expression, and $(E_1)^*$ is a regular expression.

Using these rules, we can build up a collection of regular expressions over the set $\{a, b, c\}$ as follows.

$$a, b, c, ab, bc, abc, (a)^*, (a|b), (a|b)c, b(a|c)^*a, \dots$$

Note that we will feel free to drop extra parentheses and write $(a|c)^*$ instead of $((a|c))^*$. Clearly, the set of regular expressions is infinite, but countably so.

In addition to being able to construct regular expressions, we'd like to know what they mean, or denote. If E is a regular expression, we'll use $L(E)$ for the set of strings matched by the expression E . This is defined inductively in parallel with the structure of regular expressions.

For the base case, the set of strings matched by a single symbol is just that symbol itself. Thus, $L(a) = \{a\}$, $L(b) = \{b\}$, and $L(c) = \{c\}$. For the inductive cases, assume that E_1 and E_2 are regular expressions denoting L_1 and L_2 respectively. (That is, $L(E_1) = L_1$ and $L(E_2) = L_2$.)

The regular expression E_1E_2 denotes the language $L_1 \cdot L_2$, where

$$L_1 \cdot L_2 = \{s_1 \cdot s_2 : s_1 \in L_1, s_2 \in L_2\}.$$

That is, we consider the set of all strings obtained by choosing a string s_1 from L_1 and a string s_2 from L_2 and concatenating the two strings to get $s_1 \cdot s_2$. The set of strings we can obtain in this way is the set of strings denoted by E_1E_2 .

The regular expression $(E_1|E_2)$ denotes the set of strings $L_1 \cup L_2$, the union of the sets L_1 and L_2 , that is, the set of strings that are in L_1 or L_2 or both.

The regular expression $(E_1)^*$ denotes the set of strings that can be obtained by choosing (with replacement) and concatenating any finite number of strings from L_1 . Since the "finite number" includes 0, we have that the empty string (with no characters), denoted ϵ , is always in $L((E_1)^*)$.

We use these definitions to figure out the denotations of the regular expressions constructed above. From the base case we have that

$$L(a) = \{a\}, L(b) = \{b\}, L(c) = \{c\}.$$

Then using the definition of $L(E_1E_2)$ we have

$$L(ab) = \{ab\}, L(bc) = \{bc\}, L(abc) = \{abc\}.$$

Thus, each string denotes the set consisting of that single string. Using the definition of $L(E_1|E_2)$ we have that

$$L(a|b) = \{a, b\}, L((a|b)c) = \{ac, bc\}.$$

Thus, $L((a|b)(a|b))$ contains exactly the four strings aa , ab , ba , bb .

Using the definition of $L((E_1)^*)$, we consider $L((a)^*)$. This set includes the empty string, ϵ . It also consists of the string obtained by choosing one a , or two a 's, or three a 's, etc. Thus,

$$L((a)^*) = \{\epsilon, a, aa, aaa, aaaa, aaaaa, \dots\}.$$

That is, $L((a)^*)$ is the set of all strings containing 0 or more a 's.

Considering $(a|c)^*$, we get the empty string, ϵ , any string consisting of one string from $L(a|c)$, that is, a or c , any string consisting of a concatenation of two strings drawn (with replacement) from $L(a|c)$, that is, aa , or ac , or ca , or cc , any string consisting of a concatenation of three strings drawn (with replacement) from $L(a|c)$, namely, aaa , aac , aca , acc , caa , cac , cca , ccc , and so on. That is, $L((a|c)^*)$ is any string consisting of a 's and c 's.

Finally, considering $b(a|c)^*a$, we see that this denotes the set of strings that begin with b , have any string of a 's and c 's, and end with a . This includes strings such as ba , baa , bca , $baaa$, $baca$, $bcaa$, $bcca$, and so on.

Thus, the expression $c(a|d)(a|d)(a|d)^*r$ does, in fact, match any string that begins with c , has an a or a d followed by another a or d , followed by any string of 0 or more a 's and d 's, ending with an r . Or, in other words,

$$L(c(a|d)(a|d)(a|d)^*r) = \{caar, cadr, cdar, cddr, caaar, caadr, \dots\}.$$

Back to egrep

Egrep incorporates certain convenient extensions to the regular expression notation defined in the preceding subsection including the following. The symbol `.` matches any single character. The expression `[aeiou]` matches any single character in the set of characters between the brackets, that is, it is equivalent to `(a|e|i|o|u)`. In a set, ranges of characters such as `A-Z` or `a-z` or `0-9` can be used. Thus, `[A-Za-z]` matches any single upper or lower case letter. Also, the expression `[^aeiou]` matches any single character NOT in the set containing `a`, `e`, `i`, `o`, and `u`. Using `+` in place of `*` means one or more occurrences of strings from the set, rather than zero or more occurrences of strings from the set. Thus, `(ab)+` denotes the set of strings obtained by one or more repetitions of `ab`. Note that we could get the same effect with `ab(ab)*`. Using `?` in place of `*` means zero or one occurrences of strings from the set. Thus, `yog(h)?urt` will match the two strings `yogurt` and `yoghurt`. This is by no means all of what is available in extended regular expressions, but enough to indicate their potential usefulness.

As an exercise, we define a regular expression to denote the set of identifiers, where an identifier is any string of characters that begins with an upper or lower case letter and has 0 or more upper or lower case letters or digits following that. Thus, the expression should match such things as `A`, `Start`, `v223`. Taking advantage of sets and ranges, we can write the following egrep expression.

```
[A-Za-z][A-Za-z0-9]*
```

Deterministic finite acceptors

Another method of describing sets of strings is via deterministic finite acceptors. These are automata that are like Turing machines with no tape. A deterministic finite acceptor (or dfa) has a finite set of input symbols, a finite set of states, a start state, a set of accepting states, and a transition function. The transition function is defined for pairs consisting of a state and a symbol, and produces a state.

As an example, we specify a dfa over the finite set of symbols $\{a, b\}$ that has states $\{1, 2\}$, with the start state being state 1 and the set of accepting states being $\{1\}$, and a transition function δ given by

$$\begin{aligned}\delta(1, a) &= 2 \\ \delta(1, b) &= 1 \\ \delta(2, a) &= 1 \\ \delta(2, b) &= 2\end{aligned}$$

This information can also be conveyed by a diagram in which each state is represented by a circle and each transition by an arrow. For example, the transition for $\delta(1, a) = 2$ would be shown as an arrow from state 1 to state 2 labelled by a , and the transition for $\delta(1, b) = 1$ would be shown as an arrow from state 1 to state 1 labelled by b . The initial state is indicated by a single incoming arrow, (not coming from another state) and the accepting states are represented by placing another circle around the state symbol. (Breathe easy, there will be no ASCII drawings of deterministic finite state acceptors.)

For each string of input symbols, we determine whether the dfa accepts the string as follows. Starting at the start state, we follow the transition corresponding to each symbol in the string, from left to right, until we have used up all the symbols in the string. If this process ends in an accepting state, the string is accepted. Otherwise, the string is not accepted.

For example, for the dfa we defined above, if we consider the string `abbab`, we start in state 1, follow the transition to state 2 for the first symbol (a), follow the transition to the state 2 for the next symbol (b), follow the transition to the state 2 for the next symbol (b), follow the transition to the state 1 for the next

symbol (a), and follow the transition to the state 1 for the final symbol (b). Since this process ends in the state 1, which is an accepting state of this dfa, the string $abbab$ is accepted. Similar reasoning leads us to the conclusion that the string $ababab$ is not accepted. Experimenting with other strings, we come to the conclusion that this dfa accepts a string of a 's and b 's if and only if it contains an even number of a 's (which includes the possibility of all b 's, that is, zero a 's.)

Regular sets

A set of strings is called *regular* if it can be denoted by a regular expression. It is a theorem that a set is regular if and only if it is accepted by a deterministic finite acceptor. The proof is not difficult, though we do not have time for it in this course. This means that for every regular expression, there is an equivalent dfa and vice versa. As an exercise, think of a regular expression to denote the set of strings of a 's and b 's that contain an even number of a 's.

(Answers to this exercise include $b^*(ab^*ab^*)^*$, $(b^*|(b^*ab^*ab^*))$, and $(b^*|ab^*a)^*$.)