Regular expressions

Historically, a large part of the communication between humans and computers has been in the form of strings, that is, finite sequences of characters. Considerable infrastructure has been developed in computer science for the specification, representation, and processing of sets of strings. Regular expressions are one such representation. We start with an application of regular expressions: the linux search utility `egrep`.

An egrep search

Imagine that I decided to find out how many times I had used `car` in my solution for homework 1, which is in a file called `hw1-solution.scm`. I choose to do a search as follows.

```
termite> egrep 'car' hw1-solution.scm
     ((equal? itm (car lst)) (remove itm (cdr lst)))
     (else (cons (car lst) (remove itm (cdr lst)))))))
      ...
```

The command `egrep` invokes a linux utility to search for lines that match a given pattern in a file (or files.) The characters between the quotes specify the pattern to look for, in this case, the literal occurrence of `car` somewhere on the line. The results of the search are to print out each line of the file that matched the pattern. (I have not shown all 16 lines of the results.) The pattern is specified by means of an "extended regular expression."

As an aside, we describe some additional linux functionality that may make `egrep` more useful to you. To put the results of a search in a file instead of on the screen, we can use the "redirect" capability. For example the following command puts the 16 lines of `egrep` output in the file named `temp.txt`.

```
termite> egrep 'car' hw1-solution.scm > temp.txt
termite>
```

The greater than (>) character indicates that the output of the command should be placed in the indicated file. I could then choose to look at this file in a text editor; I could also use the word-count utility (`wc`) on it as follows.

```
termite> wc temp.txt
 16 146 862 temp.txt
termite>
```

This shows that the file contains 16 lines, 146 words and 862 characters. Thus, the `egrep` command found 16 lines of the file that contained the substring `car` at least once. If all I was interested in was the word count of the results of the `egrep`, I could "pipe" the output of the `egrep` command to the `wc` command as follows. Here the vertical bar (|) indicates that the output of the first command becomes the input of the second command.

```
termite> egrep 'car' hw1-solution.scm | wc
     16     146     862
termite>
```

These facilities to redirect and search output can help you deal with the occasional verbosity of the `autograde` program.

Strings

We now briefly describe the mathematical treatment of strings. Strings are defined over some finite alphabet of symbols, for example, $\{a, b, c\}$. Traditionally, the alphabet is denoted by Σ and an arbitrary symbol by σ. A string is a finite sequence of symbols from the alphabet, for example *aaab* or *abaccabb*. There is an empty string, of no symbols, which we'll denote by ε. (The empty string may also be denoted by λ.) A fundamental operation on a string is to add a symbol at the start of the string (analogous to cons'ing an element onto a list). Others are to identify the first symbol of a non-empty string (analogous to car) and remove the first symbol of a non-empty string (analogous to cdr). Two strings s_1 and s_2 may be concatenated, which consists of forming the string consisting of the symbols of the first string followed by the symbols of the second string (analogous to appending two lists.) The operation of concatenation is denoted by $\cdot$. For example, concatenating *aabca* with *bbacaaa* results in the string *aabcabbacaaa*. That is

$$aabca \cdot bbacaaa = aabcabbacaaa.$$

Definitions and proofs concerning strings are generally inductive. For example, we can define the concatenation of two strings s_1 and s_2 inductively as follows. If s_1 is the empty string, then the concatenation of s_1 and s_2 is just s_2. If s_1 is not empty, then it consists of a symbol σ followed by a string t, and the concatenation of s_1 and s_2 is obtained by concatenating t and s_2 and adding σ as the first element of the result. (Compare this definition with our recursive procedure to append two lists in Racket.) Similarly, the length of a string (the number of symbols it contains) can be defined inductively as follows. The length of ε (the empty string) is 0. If s is a nonempty string, then s consists of a symbol σ followed by a string t, and the length of s is 1 more than the length of t. For example, the length of the string *aabca* is 5.

A set of strings is called a language. This terminology comes from the use of formal languages and grammars as models of natural languages and grammars in linguistics.

Definition of basic regular expressions

In this section, we define "basic" regular expressions and specify what sets of strings they denote. We'll return to the extended regular expressions of `egrep` after we understand these.

Regular expressions are defined over a finite alphabet of symbols, for example, $\{a, b, c\}$. The definition is recursive (or inductive). (1) The base cases are that ε is a regular expression, and each symbol σ by itself is a regular expression. (2) For the inductive cases, assume that E_1 and E_2 are regular expressions. Then $(E_1 \cdot E_2)$ is a regular expression, $(E_1|E_2)$ is a regular expression, and $(E_1)^*$ is a regular expression.

The centered dot $\cdot$ is used to indicate concatenation, and will be omitted when possible. Extra parentheses will also be omitted, using an operator precedence that says $*$ has higher precedence than $\cdot$, which has higher precedence than $|$. Thus, putting the parentheses and centered dots back into $a|bc^*$, we get $(a|(b \cdot (c)^*))$.

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

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

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 "language of E", that is, the set of strings denoted by the expression E. This is defined inductively, paralleling the syntactic structure of regular expressions.

For the base cases, the set of strings denoted by ε is just $\{\varepsilon\}$, and the set of strings denoted by a single symbol σ is just $\{\sigma\}$. 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 $E_1 \cdot E_2$ 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 $E_1 \cdot E_2$.

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. Here $\cup$ just denotes the usual operation of taking the union of two sets.

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 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

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

Then using the definition of $L(E_1 \cdot E_2)$ 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,

$$L((a)*) = \{\varepsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}.$$

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, over the alphabet $\{a, c, d, r\}$, the expression $c(a|d)(a|d)^*r$ matches any string that begins with c, has an a or a d followed by any string of 0 or more a's and d's, and ending with an r. Or, in other words,

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

Back to egrep

Note that the `egrep` utility takes a regular expression and for each line determines whether a *substring* of the characters on that line is denoted by the regular expression. That is, only part of the line has to match the expression to produce a match.

`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.

Suppose we define an "identifier" to be 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. Then examples of identifiers are `A`, `Start` and `v223x`. The following `egrep` expression denotes the set of all identifiers.

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