15–212: Fundamental Structures of Computer Science II

Notes on Regular Expression Matching

Robert Harper, Spring 1997
edited by Frank Pfenning, Fall 1997

Draft of September 23, 1997

1 Introduction

Regular expression matching is a very useful technique for describing commonly-occurring patterns in strings. For example, the Unix shell (and most other command processors) provides a mechanism (called "globbing") for describing a collection of files by patterns such as `*.sml` or `hw[1-3].sml`. The emacs text editor provides a richer, but substantially similar, pattern language for its "regexp search" mechanism. In this note we will describe a simple on-line algorithm for regular expression matching that illustrates a number of important programming concepts. By "on-line" we mean that the matching algorithm makes no attempt to pre-process the pattern before matching. Sophisticated "off-line" algorithms that perform such pre-processing (and lead to more efficient matchers) are available, but we shall not discuss these here. (These methods are covered in detail in 15-411.)

The patterns describable by regular expressions are built up from the following four constructs:

1. *Singleton*: matching a specific character.
2. *Alternation*: choice between two patterns.
3. *Concatenation*: succession of patterns.
4. *Iteration*: indefinite repetition of patterns.

Notably, regular expressions provide no concept of *nesting* of one pattern inside another. For this we require a richer formalism, called a *context-free language*, which we shall discuss later in the course.

2 Languages

To make precise the informal ideas outlined above, we introduce the concept of a *formal language*. First, we fix an *alphabet* Σ, which is any countable set of *letters*. The set Σ^* is the set of *strings* over the alphabet Σ. The *null string* is written ϵ, and string concatenation is indicated by juxtaposition. A *language* L is any subset of Σ^* — that is, any set of strings over Σ.

In practice Σ is SML type `char`, and Σ^* is the SML type `string`. We will use SML notation for operations on strings.

3 Regular Expressions

Regular expressions are a notation system for languages. The set of regular expressions over an alphabet Σ is given by the following inductive definition:

1. If $a \in \Sigma$, then $\mathbf{a}$ is a regular expression.
2. If r_1 and r_2 are regular expressions, so is $r_1\, r_2$.
3. $\mathbf{0}$ is a regular expression.
4. If r_1 and r_2 are regular expressions, so is $r_1 + r_2$.
5. $\mathbf{1}$ is a regular expression.
6. If r is a regular expression, then so is r^*.

The *language* $L(r)$ of a regular expression r is defined as follows:

$$\begin{array}{rcl} L(\mathbf{a}) & = & \{\, a \,\} \\ L(r_1\, r_2) & = & \{s_1\, s_2 \mid s_1 \in L(r_1) \quad \text{and} \quad s_2 \in L(r_2)\} \\ L(\mathbf{1}) & = & \{\, \epsilon \,\} \\ L(r_1 + r_2) & = & \{s \mid s \in L(r_1) \quad \text{or} \quad s \in L(r_2)\} \\ L(\mathbf{0}) & = & \{\,\} \\ L(r^*) & = & \{s_1 \ldots s_n \mid s_i \in L(r) \quad \text{for} \quad 1 \leq i \leq n\} \end{array}$$

By convention, $s_1 \ldots s_n$ stands for the empty string ϵ if $n = 0$.

We say that a string s *matches* a regular expression r iff $s \in L(r)$. Thus s never matches $\mathbf{0}$; s matches $\mathbf{1}$ only if $s = \epsilon$; s matches $\mathbf{a}$ iff $s = a$; s matches $r_1 + r_2$ if it matches either r_1 or r_2; s matches $r_1\, r_2$ if $s = s_1\, s_2$, where s_1 matches r_1 and s_2 matches r_2; s matches r^* iff either $s = \epsilon$, or $s = s_1\, s_2$ where s_1 matches r and s_2 matches r^*.

Some simple examples over the alphabet $\Sigma = \{a, b\}$.

$$\begin{array}{rcl} L(\mathbf{aa}) & = & \text{singleton set containing only } aa \\ L((\mathbf{a} + \mathbf{b})^*) & = & \text{set of all strings} \\ L((\mathbf{a} + \mathbf{b})^*\mathbf{aa}(\mathbf{a} + \mathbf{b})^*) & = & \text{set of strings with two consecutive } a\text{'s} \\ L((\mathbf{a} + \mathbf{0})(\mathbf{b} + \mathbf{ba})^*) & = & \text{set of strings without two consecutive } a\text{'a} \end{array}$$

4 A Matching Algorithm

We are to define an SML function `accept` with type `regexp -> string -> bool` such that `accept` r s evaluates to `true` iff s matches r, and evaluates to `false` otherwise.

First we require a representation of regular expressions in SML. This is easily achieved as follows:

```
datatype regexp
  = Char of char
  | Times of regexp * regexp
  | One
  | Plus of regexp * regexp
  | Zero
  | Star of regexp;
```

The correspondence to the definition of regular expressions should be clear.

The matcher is defined using a programming technique called *continuation-passing*. We will define an auxiliary function `acc` of type

```
val acc : regexp -> char list -> (char list -> bool) -> bool
```

which takes a regular expression, a character list, and a *continuation*, and yields a boolean value. Informally, the continuation determines how to proceed once an initial segment of the given character list has been determined to match the given regular expression — the remaining input is passed to the continuation to determine the final outcome. In order to ensure that the matcher succeeds (yields `true`) whenever possible, we must be sure to consider *all* ways in which an initial segment of the input character list matches the given regular expression in such a way that the remaining unmatched input causes the continuation to succeed. Only if there is no way to do so may we yield `false`.

This informal specification may be made precise as follows.

1. If there exists s_1 and s_2 such that $s = s_1\,s_2$, $s_1 \in L(r)$, and $k(s_2)$ evaluates to `true`, then `acc` r s k evaluates to `true`.

2. If for every s_1 and s_2 such that $s = s_1\,s_2$ with $s_1 \in L(r)$ we have that $k(s_2)$ evaluates to `false`, then `acc` r s k evaluates to `false`

Notice that this specification determines the outcome only for continuations k that always yield either `true` or `false` on any input. This is sufficient for our purposes since the continuations that arise will always satisfy this requirement. Notice as well that the specification implies that the result should be `false` in the case that there is no way to partition the input string s such that an initial segment matches r.

Before giving the implementation of `acc`, we can define `accept` as follows:

```
fun accept r s = acc r (String.explode s) List.null;
```

Two remarks. We "explode" the string argument into a list of characters to facilitate sequential processing of the string. The *initial continuation* yields `true` or `false` according to whether the remaining input has been exhausted. Assuming that `acc` satisfies the specification given above, it is easy to see that `accept` is indeed the required matching algorithm.

Now for the code for `acc`:

```
(* val acc : regexp -> char list -> (char list -> bool) -> bool *)
fun acc (Char(c)) (c1::s) k = if (c = c1) then k s else false
  | acc (Char(c)) (nil) k = false
  | acc (Times(r1,r2)) s k = acc r1 s (fn s' => acc r2 s' k)
  | acc (One) s k = k s
  | acc (Plus(r1,r2)) s k = acc r1 s k orelse acc r2 s k
  | acc (Zero) s k = false
  | acc (Star(r)) s k = k s orelse acc r s (fn s' => acc (Star(r)) s' k);
```

Note that the case of `(Star r)` could have been written

```
  | acc (Star(r)) cs k =
      acc (Plus (One, Times (r, (Star r)))) cs k
```

but this has the disadvantage of creating a new regular expression during matching.

Does `acc` satisfy the specification given above? A natural way to approach the proof is to proceed by induction on the structure of the regular expression. For example, consider the case $r = \texttt{Times(}r_1\texttt{,}r_2\texttt{)}$. We have two proof obligations according to whether or not the input may be partitioned in such a way that an initial segment matches r and the continuation succeeds on the corresponding final segment.

First, suppose that $s = s_1\,s_2$ with s_1 matching r and $k(s_2)$ evaluates to `true`. We are to show that `acc` r s k evaluates to `true`. Now since s_1 matches r, we have that $s_1 = s_{1,1}\,s_{1,2}$ with $s_{1,1}$ matching r_1 and $s_{1,2}$ matching r_2. Consequently, by the inductive hypothesis applied to r_2, we have that `acc` r_2 $(s_{1,2}\,s_2)$ k evaluates to `true`. Therefore the application `(fn cs' => acc` r_2 `cs' k)` $(s_{1,2}\,s_2)$ evaluates to `true`, and hence by the inductive hypothesis applied to r_1, the expression `acc` r_1 s `(fn cs' => acc` r_2 `cs' k)` evaluates to `true`, which is enough for the result.

Second, suppose that no matter how we choose s_1 and s_2 such that $s = s_1\,s_2$ with $s_1 \in L(r)$, we have that $k(s_2)$ evaluates to `false`. It suffices to show that `acc` r_1 s `(fn cs' => acc` r_2 `cs' k)` evaluates to `false`. By the inductive hypothesis (applied to r_1) it suffices to show that for every $s_{1,1}$ and s_2' such that $s = s_{1,1}\,s_2'$ with $s_{1,1} \in L(r_1)$, we have that `acc` r_2 s_2' k evaluates to `false`. By the inductive hypothesis (applied to r_2) it suffices to show that for every $s_{1,2}$ and s_2 such that $s_2' = s_{1,2}\,s_2$ with $s_{1,2} \in L(r_2)$, we have that $k(s_2)$ evaluate to `false`. But this follows from our assumptions, taking $s_1 = s_{1,1}\,s_{1,2}$.

This completes the proof for $r = r_1\,r_2$; you should carry out the proof for $\mathbf{0}$, $\mathbf{1}$, $\mathbf{a}$, and $r_1 + r_2$.

What about iteration? Let r be `Star` r_1, and suppose that $s = s_1\,s_2$ with s_1 matching r and $k(s_2)$ evaluates to `true`. By our choice of r, there are two cases to consider: either $s_1 = \epsilon$, or $s_1 = s_{1,1}s_{1,2}$ with $s_{1,1}$ matching r_1 and $s_{1,2}$ matching r. In the former case the result is the result of $k(s)$, which is $k(s_2)$, which is `true`, as required. In the latter case it suffices to show that `acc` r_1 s `(fn cs' => acc` r `cs' k)` evaluates to `true`. By inductive hypothesis it suffices to show that `acc` r s_2 k evaluates to `true`. It is tempting at this stage to appeal to the inductive hypothesis to complete the proof — but we cannot because the regular expression argument is the *original* regular expression r, and not some sub-expression of it!

What to do? In general there are two possibilities: fix the proof or find a counterexample to the theorem. Let's try to fix the proof — it will help us to find the counterexample (the theorem as stated is false). A natural attempt is to proceed by an "outer" induction on the structure of the regular expression (as we've done so far), together with an "inner" induction on the length of the string, the idea being that in the case of iteration we appeal to the *inner* inductive hypothesis to complete the proof — provided that the string $s_{1,2}\,s_2$ can be shown to be shorter than the string s. This is equivalent to showing that $s_{1,1}$ is non-empty — which is false! For example, r might be $\mathbf{1}^*$, in which case our matcher loops forever.

What to do? Change the specification. The proof goes through provided that the regular expression is in *standard form*, by which we mean that if r^* occurs within the regular expression, then r does not accept the null string. Given this condition, the proof goes through as outlined. You should carry out the proof for the case of iteration to check this claim. Remember that there are two parts to the proof: complete the proof for the case that there is a partitioning that leads to the continuation yielding `true` (sketched above), and for the case that there is no such partitioning. Consult the proof for the concatenation of regular expressions as a guide.

In the next section we see how to convert any regular expression into standard form so we can apply the algorithm as given above.

Another solution is to rule out matches of r against the empty string when matching r^* against any string. This never rules out a valid solution, but we can use the idea to force termination

(which is the only the problem with the algorithm above). All cases remain the same, except for the case of `Star(r)`.

```
| acc (r as Star(r1)) s k =
    k s orelse
    acc r1 s (fn s' => if s = s' then false
                       else acc r s' k);
```

We fail if $s = s'$, which means that the initial segment of s matched by r must have been empty.

The correctness proof now works by two nested structural inductions: one on the structure of the regular expression r and one on structure of the string we match against. This means that either (a) the regular expression has to get smaller (which it does in all cases except the last), or (b) if the regular expression stays the same, then the string has to get smaller. Once can see that this is guaranteed in the modified program above since the continuation will always be called on a substring of the original string.

5 Standardization

If we do not want to change the program, we can restrict the input to be in standard form. But have we lost something by making this restriction? Are there *non-standard* regular expressions of interest? What have we left out? The answer is nothing, because *any regular expression can be brought into standard form*! The idea is that before matching commences we rewrite the regular expression into another regular expression that accepts the same language and which is in standard form.

Here is how it is done. We rely on the equation $r = \delta(r) + r^-$, where $\delta(r)$ is either $\mathbf{1}$ or $\mathbf{0}$ according to whether or not r accepts the null string, and where $L(r^-) = L(r) \setminus \{\epsilon\}$. Check for yourself that $L(r) = L(\delta(r) + r^-)$, as required.

The functions $\delta(r)$ and r^- are defined as follows.

$$\begin{array}{rcl} \delta(\mathbf{0}) & = & \mathbf{0} \\ \delta(\mathbf{1}) & = & \mathbf{1} \\ \delta(\mathbf{a}) & = & \mathbf{0} \\ \delta(r_1 + r_2) & = & \delta(r_1) \oplus \delta(r_2) \\ \delta(r_1\, r_2) & = & \delta(r_1) \otimes \delta(r_2) \\ \delta(r^*) & = & \mathbf{1} \end{array}$$

We define $r_1 \oplus r_2$ to be $\mathbf{1}$ if either r_1 or r_2 is $\mathbf{1}$, and $\mathbf{0}$ otherwise. We define $r_1 \otimes r_2$ to be $\mathbf{0}$ if either r_1 or r_2 is $\mathbf{0}$, and is $\mathbf{1}$ otherwise.

Finally, we define r^- as follows:

$$\begin{array}{rcl} \mathbf{0}^- & = & \mathbf{0} \\ \mathbf{1}^- & = & \mathbf{0} \\ \mathbf{a}^- & = & \mathbf{a} \\ (r_1 + r_2)^- & = & r_1^- + r_2^- \\ (r_1\, r_2)^- & = & \delta(r_1)\, r_1^- + r_1\, \delta(r_2) + r_1^-\, r_2^- \\ (r^*)^- & = & r^-(r^-)^* \end{array}$$

The last two deserve comment. The non-empty strings matching $r_1\, r_2$ are (1) the non-empty strings in r_2, in the case that r_1 contains the empty string, (2) the non-empty strings in r_1, in the case

that r_2 contains the empty string, and (3) the concatenation of a non-empty string in r_1 followed by a non-empty string in r_2. Check that the given equation expresses these three conditions as a regular expression! The clause for iteration is motivated by the observation that the non-empty strings in the iteration r^* are simple the *non-zero* iterations of the *non-empty* strings in r.

As an exercise, prove that $\delta(r)$ and r^- have the required properties (stated above) and that r^- is in standard form.

6 Conclusion

The example of regular expression matching illustrates a number of important programming concepts:

1. *Continuation-passing*: the use of higher-order functions as continuations.
2. *Proof-directed debugging*: the use of a breakdown in a proof attempt to discover an error in the code.
3. *Change of specification*: once we isolated the error, we had the choice of changing the *code* or changing the *specification*. Moral: debugging isn't always a matter of hacking!
4. *Pre-processing*: to satisfy the more stringent specification we pre-processed the regular expression so that it satisfies the additional assumption required for correctness, without losing the generality of the matcher.