# 15–212: Fundamental Structures of Computer Science II

## *Notes on Regular Expression Matching*

Robert Harper

edited by Frank Pfenning, Fall 1997
edited by John Lafferty, Fall 1998

Draft of September 30, 1998

## 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* $\Sigma$, which is any countable set of *letters*. The set $\Sigma^*$ is the set of *strings* over the alphabet $\Sigma$. The *null string* is written $\epsilon$, and string concatenation is indicated by juxtaposition. A *language* $L$ is any subset of $\Sigma^*$ — that is, any set of strings over $\Sigma$.

In practice $\Sigma$ is SML type `char`, and $\Sigma^*$ 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 $\Sigma$ is given by the following inductive definition:

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

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

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

By convention, $s_1 \ldots s_n$ stands for the empty string $\epsilon$ 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 $r_1 + r_2$ iff it matches either $r_1$ or $r_2$; $s$ matches $r_1\, r_2$ iff $s = s_1\, s_2$, 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 $r^*$. The parentheses are often omitted in a regular expression. If so, the evaluation is done using the precedence rules: star precedes times precedes plus.

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

$$\begin{array}{rcl}
L(\mathbf{aa}) & = & \text{singleton set containing only } aa \\
L((\mathbf{a} + \mathbf{b})^*) & = & \text{set of all strings} \\
L((\mathbf{a} + \mathbf{b})^*\mathbf{aa}(\mathbf{a} + \mathbf{b})^*) & = & \text{set of strings with two consecutive } a\text{'s} \\
L((\mathbf{a} + \mathbf{1})(\mathbf{b} + \mathbf{ba})^*) & = & \text{set of strings without two consecutive } a\text{'a}
\end{array}$$

## 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
  | One
  | Zero
  | Plus of regexp * regexp
  | Times of regexp * regexp
  | 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 is no way to write $s = s_1\, s_2$ such that $s_1 \in L(r)$, then acc $r$ $s$ $k$ evaluates to false.
2. 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.
3. 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.

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)) (nil) k = false
  | acc (Char(c)) (c1::s) k = if (c = c1) then k s else false
  | acc (One) s k = k s
  | acc (Zero) s k = false
  | acc (Plus(r1,r2)) s k = acc r1 s k orelse acc r2 s k
  | acc (Times(r1,r2)) s k = acc r1 s (fn s' => acc r2 s' k)
  | 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)) s k =
      acc (Plus (One, Times (r, (Star r)))) s 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 $r$ = `Times(`$r_1$`,`$r_2$`)`. We will structure the proof according to whether or not the input may be partitioned in such a way that an initial segment matches $r$, and refer to the specific case of the specification (1., 2., or 3.) that we are appealing to in the structural induction hypothesis.

1. Suppose that there is no way to write $s = s_1\, s_2$ such that $s_1 \in L(r)$. We need to show that `acc` $r$ $s$ $k \Rightarrow$ `false`. First suppose that we can write $s = s_{1,1}\, s'_2$, with $s_{1,1} \in L(r_1)$. Then by assumption, there is no way to write $s'_2 = s_{1,2}\, s_2$, with $s_{1,2} \in L(r_2)$. Thus, by the structural induction hypothesis (case 1.) applied to $r_2$, `acc` $r_2$ $(s_{1,2}\, s_2)$ `k` $\Rightarrow$ `false`, and so by the structural induction hypothesis (case 3.) applied to $r_1$, `acc` $r_1$ $s$ `(fn s' => acc` $r_2$ `s' k)` $\Rightarrow$ `false`. Now suppose that there is no way to write $s = s_{1,1}\, s'_2$ with $s_{1,1} \in L(r_1)$. Then by the structural induction hypothesis (case 1.) applied to $r_1$, `acc` $r_1$ $s$ `(fn s' => acc` $r_2$ `s' k)` $\Rightarrow$ `false`.

2. Suppose that $s = s_1\, s_2$ with $s_1$ matching $r$ and $k$`(`$s_2$`)` $\Rightarrow$ `true`. We are to show that `acc` $r$ $s$ $k \Rightarrow$ `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 (case 2.) applied to $r_2$, we have that `acc` $r_2$ $(s_{1,2}\, s_2)$ $k \Rightarrow$ `true`. Therefore the application `(fn s' => acc` $r_2$ `s'` $k$`)` $(s_{1,2}\, s_2) \Rightarrow$ `true`, and hence by the inductive hypothesis (case 2.) applied to $r_1$, we have that `acc` $r_1$ $s$ `(fn s' => acc` $r_2$ `s'` $k$`)` $\Rightarrow$ `true`.

3. Finally, 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 $k$`(`$s_2$`)` $\Rightarrow$ `false`. We need to show that `acc` $r_1$ $s$ `(fn s' => acc` $r_2$ `s'` $k$`)` $\Rightarrow$ `false`. By the inductive hypothesis (case 3.) 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` $r_2$ $s'_2$ $k \Rightarrow$ `false`. To show this, it suffices to show, by the inductive hypothesis (case 3.) applied to $r_2$, 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 $k$`(`$s_2$`)` $\Rightarrow$ `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, $r_1$ might be $\mathbf{1}^*$, in which case our matcher loops forever.

One solution is to change the code 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.

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.