15-150 Fall 2013

Stephen Brookes

Regular expressions

1 Regular expressions

A regular expression is a sequence of characters that forms a search pattern, for use in "find and replace"-like operations on strings or character lists. The concept arose in the 1950s, when the mathematician Stephen Kleene formalized the description of a regular language (and when connections were made between automata theory and regular languages). Regular expressions came into common use with the Unix operating system, being used in the text processing utilities `ed` (an editor) and `grep` (standing for "global regular expression print", a filter operation). Many programming languages provide regular expression capabilities, some built-in (e.g. Perl, Ruby, AWK, and Tcl) and others via a standard library (e.g. Java, Python and C++).

Regular expressions give us a succinct way to identify text with certain structure, involving occurrences of specific sequences of characters, or sequences with certain general properties. You can use a regular expression to search for occurrences of a specific phrase, or an occurrence of one of a set of phrases, or a sequence of phrases, or a repeatedly occurring phrase. Once you've located a suitable phrase, you may be able to delete it, or otherwise manipulate the string. For example, you might want to search through your email for messages concerning 15-150. A regular expression search could help, using the obvious search pattern. Or you might want to look for an email message that looks like a letter from your bank. Most such letters begin with the word "Dear", followed by a name, and a colon. The last line may contain "Sincerely," "Yours truly," "Love," (probably not if it's from your bank manager!) or some other standard form. Hardly any other line in any other kind of text ends with a comma. A regular expression can look for these patterns, and a successful match tells you that a file containing them is probably a letter. It's harder to do that with an ordinary text search, unless you know the exact contents of the message you want to find.

Regular expression languages all contain (at least) the following constructs (but possibly written differently!):

- a regular expression 0 denoting the empty set of strings, which matches nothing;
- a regular expression 1 denoting the set containing only the empty string, which only matches the empty string;
- a "literal" regular expression c, one for each character c in the alphabet,

matching only the string consisting of just that character;

- concatenation: $R_1 \cdot R_2$ denotes the set of strings of the form $\alpha\beta$ where α matches R_1 and β matches R_2;
- union: $R_1 + R_2$ denotes the set of strings that match either R_1 or R_2;
- iteration: R^* denotes the smallest set of strings that contains the empty string and is closed under concatenation with R.

The syntax used in different implementations of regular languages may differ drastically. For example, union is sometimes written as $R_1|R_2$, and concatenation is often written as juxtaposition, R_1R_2. You can also use parentheses in order to disambiguate between expressions. For example, $(a + b)c$ is not the same expression as $a + (bc)$.

Many implementations of regular expressions also include additional constructs such as R^+ (which behaves the same way as $R \cdot R^*$, meaning "at least one consecutive string matching R") and $\{R\}$ (which behaves the same way as $(R+1)$, meaning "an optional occurrence of a string matching R"). These two are obviously derivable from concatenation, iteration, and union.

Since there is no universal standard syntax for regular expressions, we will work with a simple ML datatype whose values serve the purpose of regular expressions. We only include the basic constructs.

To avoid confusion we will deal entirely with *lists of characters* rather than using *strings*. Concatenation will then correspond to the list append operation. There are built-in ML functions

```
explode : string -> char list
implode : char list -> string
```

that can be used to convert a string into the list of its characters, and back again.

2 ML implementation

```
(* A type regexp whose values represent "regular expressions" *)

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

The set of *values of type* `regexp` is inductively generated by the constructors of this datatype:

- `Zero` and `One` are values of type `regexp`
- For every value `c` of type `char`, `Char c` is a value of type `regexp`
- If $\mathtt{R}_1$ and $\mathtt{R}_2$ are values of type `regexp`, so are $\mathtt{Plus}(\mathtt{R}_1, \mathtt{R}_2)$ and $\mathtt{Times}(\mathtt{R}_1, \mathtt{R}_2)$
- If `R` is a value of type `regexp`, so is `Star R`.

Every value of type `regexp` is either `Zero`, `One`, `Char c` for some value `c:char`, or one of the forms $\mathtt{Plus}(\mathtt{R}_1, \mathtt{R}_2)$, $\mathtt{Times}(\mathtt{R}_1, \mathtt{R}_2)$ or $\mathtt{Star}(\mathtt{R}')$, where $\mathtt{R}_1, \mathtt{R}_2, \mathtt{R}'$ are values of type `regexp`.

3 Regular languages

A regular expression `R` denotes a "language" $\mathcal{L}(\mathtt{R})$, consisting of the set of character lists that "match" the regular expression. It may be convenient to refer to a character list as a *word*; using this parlance, a language is a set of words.

Here is an inductive mathematical definition of $\mathcal{L}(\mathtt{R})$, for all values `R` of type `regexp`.

$$
\begin{aligned}
&\mathcal{L}(\mathtt{Zero}) = \emptyset \\
&\mathcal{L}(\mathtt{One}) = \{[\,]\} \\
&\mathcal{L}(\mathtt{Char\ c}) = \{[\mathtt{c}]\} \\
&\mathcal{L}(\mathtt{Plus}(\mathtt{R}_1, \mathtt{R}_2)) = \mathcal{L}(\mathtt{R}_1) \cup \mathcal{L}(\mathtt{R}_2) \\
&\mathcal{L}(\mathtt{Times}(\mathtt{R}_1, \mathtt{R}_2)) = \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(\mathtt{R}_1), L_2 \in \mathcal{L}(\mathtt{R}_2)\} \\
&\mathcal{L}(\mathtt{Star\ R}) = (\mathcal{L}(\mathtt{R}))^* = \{L_1 @ \ldots @ L_n \mid n \geq 0, \text{each } L_i \in \mathcal{L}(\mathtt{R})\}
\end{aligned}
$$

The above definition is *structural*: each clause defines the language of regular expressions built using a constructor, in terms of the languages of proper subexpressions. For instance, the clause for $\mathcal{L}(\texttt{Star R})$ says that the language of `Star R` is the set of all finite concatenations of words belonging to the language of `R`. We've also used the so-called *Kleene star* operation $(\cdot)^*$ in this clause, interpreted as iterated concatenation of languages.

The language of `Star R` can also be characterized recursively, since it follows from the above definition that

$$\mathcal{L}(\texttt{Star R}) = \{[\,]\} \cup \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(\texttt{R}), L_2 \in \mathcal{L}(\texttt{Star R})\}.$$

Of course we are using some obvious facts about @ here without proof: in particular, @ is associative, and for all lists `L`, `[ ]@L = L@[ ] = L`.

In fact, $\mathcal{L}(\texttt{Star R})$ as defined above is the smallest set S satisfying the equation

$$S = \{[\,]\} \cup \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(\texttt{R}), L_2 \in S\}.$$

To show this, let S_n be $\{L_1 @ \ldots L_n \mid \text{each } L_i \in \mathcal{L}(\texttt{R})\}$, for each $n \geq 0$. So $S_0 = \{[\,]\}$, $S_1 = \mathcal{L}(\texttt{R})$, and so on. Let $S = \bigcup_{n=0}^{\infty} S_n$. We can easily see that $S = \{[\,]\} \cup \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(\texttt{R}), L_2 \in S\}$. So we could have used this recursive equation to *define* $\mathcal{L}(\texttt{Star R})$ as the least set that satisfies the equation for S. This corresponds to the way we originally described the iteration construct, adjusted to deal with character lists instead of strings: R^* denotes the smallest set containing the empty character list and closed under concatenation with (the language of) R.

The main things to remember about the language of `Star R` are:

- `[ ]` is in $\mathcal{L}(\texttt{Star R})$;
- Every non-empty list `L` belonging to $\mathcal{L}(\texttt{Star R})$ is expressible as $\texttt{L}_1 @ \texttt{L}_2$ where $\texttt{L}_1$ is a list belonging to $\mathcal{L}(\texttt{R})$ and $\texttt{L}_2$ belongs to $\mathcal{L}(\texttt{Star R})$. In fact, we can always choose the list $\texttt{L}_1$ here to be non-empty.

Examples

Let `c` be a character, i.e. a value of type `char`. ML syntax for characters uses `#"a"` for the character "a", and so on. And `"a"` has type `string`, not `char`.

$\mathcal{L}(\texttt{Star(Char c)}) = \{[\,],\ [\texttt{c}],\ [\texttt{c},\texttt{c}], \ldots\}$
$\mathcal{L}(\texttt{Star One}) = \{[\,]\} = \mathcal{L}(\texttt{One})$
$\mathcal{L}(\texttt{Star Zero}) = \{[\,]\}$
$\mathcal{L}(\texttt{Plus(Char c, One)}) = \{[\texttt{c}],\ [\,]\}$
$\mathcal{L}(\texttt{Plus(Char c, Zero)}) = \mathcal{L}(\texttt{Char c})$
$\mathcal{L}(\texttt{Times(Char \#"}a\texttt{", Char \#"}b\texttt{")}) = \{[\#"a", \#"b"]\}$
$\mathcal{L}(\texttt{Times(Char \#"}a\texttt{", Times(Char \#"}b\texttt{", Char \#"}c\texttt{"))}) = \{[\#"a", \#"b", \#"c"]\}$

Every non-empty list in $\mathcal{L}(\texttt{Star(Char c)})$ has the form `[c]@L2` where `L2` is in $\mathcal{L}(\texttt{Star(Char c)})$.

The language of `Star(Times(Char #"a", Char #"b"))` consists of all alternating lists of (the same number of) a's and b's.

The language of `Star(Plus(Char #"a", Char #"b"))` consists of all lists of a's and/or b's.

4 Implementing regular languages

We want to define an ML function

```
accepts : regexp -> char list -> bool
```

such that `accepts R L` evaluates to `true` if `L` $\in \mathcal{L}$(`R`) and evaluates to `false` otherwise.

It's not going to be easy to define `accepts` recursively! After all, the language of a regular expression may be an infinite set, and there may be many ways to break up a list into chunks, so it will not be easy to define `accepts (Star R)` in terms of `accepts R`.

Instead we'll introduce a helper function with a more general specification, for which we can find a recursive implementation. The helper function solves a problem that we call *regular expression matching*. After we introduce the idea behind this new function we'll show that it really would be a *helpful* function.

Regular expression matching

We'll design a helper function

```
match : regexp-> char list -> (char list -> bool) -> bool
```

such that `match R L p` evaluates to `true` if `L` has a prefix belonging to $\mathcal{L}$(`R`), with a suffix that satisfies the predicate `p`; and `match R L p` evaluates to `false` otherwise. It makes sense to require that the predicate `p` is *total*, i.e. that for all character lists `A`, `p(A)` evaluates to either `true` or `false`.

In the ENSURES condition below we write `lang(R)` instead of $\mathcal{L}$(`R`), meaning the language denoted by `R`.

```
(* match : regexp-> char list -> (char list -> bool) -> bool *)
(* REQUIRES p is total *)
(* ENSURES (a) match R L p = true if there are lists L1 and L2
                   such that L = L1@L2, p(L2)=true, L1 is in lang(R);
           (b) match R L p = false, otherwise
*)
```

Note that a `match` function that meets this specification would definitely be a helper function for defining the `accepts` function, since `match R L null` would evaluate to `true` if `L` is in the language of `R` and evaluate to `false` otherwise.[1] So we would be able to define `accepts` this way, using `match`.

Note also that a `match` function that meets this specification also has the property that whenever `p` is a total function of type `char list -> bool` and `R` is a value of type `regexp`, the function `fn L' => match R L' p` is also a total function of type `char list -> bool`.

Also note that giving the specification is just the first step towards a satisfactory code design. We will still need to *prove* that the code is correct.

[1] Remember, `null L2 = true` if and only if `L2` is the empty list, and `L@[ ]=L`.

The `match` and `accepts` functions

The following ML function definition is designed with this specification in mind, paying attention to the definition of $\mathcal{L}(\texttt{R})$ from above.

```
fun match Zero L p = false
|   match One L p = p L
|   match (Char c) L p = (case L of
                              [ ] => false
                            | x::L' => (x = c) andalso p L')
|   match (Plus(R1, R2)) L p =
      match R1 L p orelse match R2 L p
|  match (Times(R1, R2)) L p =
      match R1 L (fn L' => match R2 L' p)
|  match (Star R') L p =
      p L orelse
        match R' L (fn L' => match (Star R') L' p)

(* accepts : regexp -> char list -> bool *)
(* ENSURES  accepts R L = true iff L is in lang(R) *)

fun accepts R L = match R L null
```

Correctness?

Try some examples to see if the `match` function produces results consistent with your knowledge of regular expressions. You should see that, for all values `c:char`,

```
match (Star (Char c)) [ ] null =>* true
match (Star (Char c)) [c] null =>* true
match (Star (Char c)) [c,c] null => true
match (Star (Char c)) [c,c] (not o null) => true
```

Try some other examples. Make sure you see how the results are consistent with the specification. For instance, in the 4th example above, `[c,c]` has a split `[c]@[c]` with the prefix `[c]` in the language of `Star(Char c)` and the suffix satisfying `(not o null) [c] = true`.

So the preliminary results of testing look favorable. However, we've warned you that testing is not an adequate substitute for a complete proof.

Let's try to prove that this code is correct.

Let `P(R)` be the statement:

For all values `L` and all total functions `p`,

(a) `match R L p = true` if there are `L1,L2` such that `L = L1@L2`, `p L2 = true`, and `L1` is in $\mathcal{L}$(`R`);

(b) `match R L p = false` otherwise.

We want to prove:

Claim
For all values `R : regexp`, `P(R)` holds.

Proof development

Since the `match` function is defined using structural induction on regular expressions, let's try for a proof by structural induction. Here is what we need to do:

- Base cases:
  - Prove that `P(Zero)` and `P(One)` hold.
  - Prove that `P(Char c)` holds, for all values `c:char`.
- Inductive steps:
  - Show that `P(`$\mathtt{R}_1$`)` and `P(`$\mathtt{R}_2$`)` imply `P(Plus(`$\mathtt{R}_1, \mathtt{R}_2$`))`.
  - Show that `P(`$\mathtt{R}_1$`)` and `P(`$\mathtt{R}_2$`)` imply `P(Times(`$\mathtt{R}_1, \mathtt{R}_2$`))`.
  - Show that `P(`$\mathtt{R}'$`)` implies `P(Star `$\mathtt{R}'$`)`.

Here are the details.

(i) For `Zero`, we prove `P(Zero)`.
Let `L` be a character list and `p` be total. $\mathcal{L}$(`Zero`) = $\emptyset$.
There is no way to split `L` to get a suitable prefix-suffix combination. By definition, `match Zero L p = false`. So `P(Zero)` holds. (In this case, property (a) of `P(Zero)` holds trivially, because there are no relevant splits of `L`, and (b) holds always.)

(ii) For `One`, we prove `P(One)`.
Let `L` be a character list and `p` be total. $\mathcal{L}$(`One`) = {[ ]}.
(a) If `L` has a split `L1@L2` with `L1` $\in \mathcal{L}$(`One`) and `p L2 = true`,
then `L1` must be `[ ]` and `L2` must be `L`. By the function definition,
`match One L p = p(L)`. So we get `match One L p = true`, as required.

(b) Otherwise, if `L` has no such split, we must have `p L = false`.
This is because `[ ]` is in $\mathcal{L}$(`One`), `L = [ ]@L`, and `p` is total.
So `match One L p = p(L) = false`, as required.

(iii) For `Char`, let `c` be a value of type `char`; we prove `P(Char c)`.
Let `L` be a character list and `p` be total. $\mathcal{L}$(`Char c`) = {[`c`]}.

(a) If `L` has a split `L1@L2` with `L1` $\in \mathcal{L}$(`Char c`) and `p L2 = true`, then `L1` must be `[c]` and `L=c::L2`. By definition of `match` we get

```
match (Char c) L p = match (Char c) (c::L2) p = p L2
```

so `match (Char c) L p = true`, as required.

(b) Otherwise, if `L` has no such split, there are three possibilities: either `L` is empty, or `L` does not begin with `c`, or `L` begins with `c` but the rest of `L` does not satisfy `p`. We consider these three possibilities:

- If `L` is `[ ]`, we have

  ```
  match(Char c) L p = match (Char c) [ ] p = false
  ```

  as required.
- If `L=x::L'` for some `x:char` and some character list `L'`, but `x` is not the same character as `c`, we have

  ```
  match (Char c) L p = match (Char c) (x::L') p
                     = (x = c) andalso p L'
                     = false
  ```

  as required.
- If `L = c::L'` for some character list `L'` with `p(L') = false`, we have

  ```
  match (Char c) L p = match (Char c) (c::L') p = (c = c) andalso p L' =
  ```

  as required.

In all three of these possibilities, `match (Char c) L p = false`.
So `P(Char c)` holds.

(iv) For $\texttt{Plus}(\texttt{R}_1, \texttt{R}_2)$, assume the induction hypotheses $\texttt{P}(\texttt{R}_1)$ and $\texttt{P}(\texttt{R}_2)$.
We must show $\texttt{P}(\texttt{Plus}(\texttt{R}_1, \texttt{R}_2))$. Recall that $\mathcal{L}(\texttt{Plus}(\texttt{R}_1, \texttt{R}_2)) = \mathcal{L}(\texttt{R}_1) \cup \mathcal{L}(\texttt{R}_2)$.
By definition,

$$\texttt{match}\ (\texttt{Plus}(\texttt{R}_1, \texttt{R}_2))\ \texttt{L}\ \texttt{p} = (\texttt{match}\ \texttt{R}_1\ \texttt{L}\ \texttt{p})\ \texttt{orelse}\ (\texttt{match}\ \texttt{R}_2\ \texttt{L}\ \texttt{p}).$$

Let `p` be total.

(a) Suppose `L` has a split `L1@L2` with `L1` in $\mathcal{L}(\texttt{Plus}(\texttt{R}_1, \texttt{R}_2))$ and `p(L2)=true`.
We must show that $\texttt{match}(\texttt{Plus}(\texttt{R}_1, \texttt{R}_2)) = \texttt{true}$.
Clearly, either $\texttt{L1} \in \mathcal{L}(\texttt{R}_1)$, or $\texttt{L1} \in \mathcal{L}(\texttt{R}_2)$.

(1) If $\texttt{L1} \in \mathcal{L}(\texttt{R}_1)$, then `L` has a split `L1@L2` with `L1` in $\mathcal{L}(\texttt{R}_1)$ and `p(L2)=true`. By induction hypothesis $\texttt{P}(\texttt{R}_1)$ (part (a)), we have $\texttt{match}\ \texttt{R}_1\ \texttt{L}\ \texttt{p} = \texttt{true}$. So

$$\begin{aligned}&\texttt{match}(\texttt{Plus}(\texttt{R}_1, \texttt{R}_2))\ \texttt{L}\ \texttt{p}\\ &\quad = (\texttt{match}\ \texttt{R}_1\ \texttt{L}\ \texttt{p})\ \texttt{orelse}\ (\texttt{match}\ \texttt{R}_2\ \texttt{L}\ \texttt{p})\\ &\quad = \texttt{true}\ \texttt{orelse}\ (\texttt{match}\ \texttt{R}_2\ \texttt{L}\ \texttt{p})\\ &\quad = \texttt{true}\end{aligned}$$

as required.

(2) Otherwise, if `L` has no split as in (1), but it has a split `L1@L2` with $\texttt{L1} \in \mathcal{L}(\texttt{R}_2)$ and `p(L2)=true`, we argue as follows.
By induction hypothesis $\texttt{P}(\texttt{R}_1)$ (part (b)), $\texttt{match}\ \texttt{R}_1\ \texttt{L}\ \texttt{p} = \texttt{false}$.
By induction hypothesis $\texttt{P}(\texttt{R}_2)$ (part (a)), $\texttt{match}\ \texttt{R}_2\ \texttt{L}\ \texttt{p} = \texttt{true}$.
So

$$\begin{aligned}&\texttt{match}(\texttt{Plus}(\texttt{R}_1, \texttt{R}_2))\ \texttt{L}\ \texttt{p}\\ &\quad = (\texttt{match}\ \texttt{R}_1\ \texttt{L}\ \texttt{p})\ \texttt{orelse}\ (\texttt{match}\ \texttt{R}_2\ \texttt{L}\ \texttt{p})\\ &\quad = \texttt{false}\ \texttt{orelse}\ \texttt{true}\\ &\quad = \texttt{true}\end{aligned}$$

as required.

So in both cases (1) and (2) we get $\texttt{match}\ (\texttt{Plus}(\texttt{R}_1, \texttt{R}_2))\ \texttt{L}\ \texttt{p} = \texttt{true}$.

We have shown that if `L` has a split `L1@L2` with `L1` in $\mathcal{L}(\texttt{Plus}(\texttt{R}_1, \texttt{R}_2))$ and `p(L2)=true`, then $\texttt{match}\ (\texttt{Plus}(\texttt{R}_1, \texttt{R}_2))\ \texttt{L}\ \texttt{p} = \texttt{true}$.
So $\texttt{P}(\texttt{Plus}(\texttt{R}_1, \texttt{R}_2))$ (part (a)) holds.

(b) Now suppose that L has no split L1@L2 with L1 in $\mathcal{L}$(Plus(R_1, R_2)) and p(L2)=true. Then since $\mathcal{L}$(Plus(R_1, R_2) = $\mathcal{L}(R_1) \cup \mathcal{L}(R_2)$,

(3) L has no split L1@L2 with L1 in $\mathcal{L}(R_1)$ and p(L2)=true; and

(4) L has no split L1@L2 with L1 in $\mathcal{L}(R_2)$ and p(L2)=true.

By the induction hypotheses for R_1 and R_2 (part (b) in both cases), we have match R_1 L p = false and match R_2 L p = false. Hence

match (Plus(R_1, R_2)) L p = (match R_1 L p) orelse (match R_2 L p) = false.

So P(Plus(R_1, R_2)) (part (b)) holds, as required.

(v) For Times(R_1, R_2), assume the induction hypotheses P(R_1) and P(R_2). We must show P(Times(R_1, R_2)).
Recall that $\mathcal{L}$(Times(R_1, R_2)) = $\{L_1@L_2 \mid L_1 \in \mathcal{L}(R_1), L_2 \in \mathcal{L}(R_2)\}$.

By definition,

match (Times(R_1, R_2)) L p = match R_1 L (fn L′ => match R_2 L′ p).

Let p be total. It follows from induction hypothesis P(R_2) that the function value

(fn L′ => match R_2 L′ p)

is also total.

(a) Suppose L has a split with a prefix in $\mathcal{L}$(Times(R_1, R_2)) and suffix satisfying p. Then there are lists L1,L2,L3 such that L=(L1@L2)@L3, L1 in $\mathcal{L}(R_1)$, L2 in $\mathcal{L}(R_2)$, and p(L3)=true.

Since @ is associative, we also have L=L1@(L2@L3). Let L23 be the value of L2@L3. By the properties given above, L2 $\in \mathcal{L}(R_2)$ and pL3 = true. So by induction hypothesis P(R_2) (part (a)),

match R_2 L23 p = true.

Thus, again by the properties given above, L has a split L1@L23 with L1 $\in \mathcal{L}(R_1)$ and (fn L′ => match R_2 L′ p)(L23) = true. Since this function is also total (noted above), it follows by induction hypothesis P(R_1) (part (a)) that

match R_1 L (fn L′ => match R_2 L′ p) = true.

So `match` (`Times`($\mathtt{R}_1, \mathtt{R}_2$)) `L p` = `true`, as required.

(b) Now suppose `L` does not have a split `L1@L2` with `L1` $\in \mathcal{L}$(`Times`($\mathtt{R}_1, \mathtt{R}_2$)) and `p`(`L2`) = `true`. We need to show that

$$\texttt{match R}_1\ \texttt{L (fn L}' \texttt{ => match R}_2\ \texttt{L}'\ \texttt{p)} = \texttt{false}.$$

By the induction hypothesis for $\mathtt{R}_2$, the function (`fn L'` => `match` $\mathtt{R}_2$ `L' p`) is total. So by the induction hypothesis for $\mathtt{R}_1$, the expression

$$\texttt{match R}_1\ \texttt{L (fn L}' \texttt{ => match R}_2\ \texttt{L}'\ \texttt{p)}$$

evaluates either to `true` or to `false`. To show that it evaluates to `false`, let's suppose it evaluates to `true` and derive a contradiction. Assume this expression evaluates to `true`. By the induction hypothesis for $\mathtt{R}_1$, `L` has a split `L = A@B` such that `A` $\in \mathcal{L}$($\mathtt{R}_1$) and

$$\texttt{(fn L}' \texttt{ => match R}_2\ \texttt{L}'\ \texttt{p)(B)} = \texttt{true}.$$

But then we also have `match` $\mathtt{R}_2$ `B p` = `true`.
By induction hypothesis for $\mathtt{R}_2$, `B` must have a split `B` = `B1@B2` with `B1` $\in \mathcal{L}$($\mathtt{R}_2$) and `p`(`B2`) = `true`. So we have `L` = `A@(B1@B2)` = `(A@B1)@B2`, and `A@B1` $\in \mathcal{L}$(`Times`($\mathtt{R}_1, \mathtt{R}_2$), and `p`(`B2`) = `true`. This contradicts the original assumption that `L` had no suitable split.

Assuming that the above expression evaluates to `true` leads to a contradiction, and the expression has a value of type `bool`, so it follows that the expression evaluates to `false`. So

$$\texttt{match R}_1\ \texttt{L (fn L}' \texttt{ => match R}_2\ \texttt{L}'\ \texttt{p)} = \texttt{false}$$

as required.

(vi) For `Star` $\mathtt{R}'$, assume as induction hypothesis that `P`($\mathtt{R}'$) holds. We must prove `P`(`Star` $\mathtt{R}'$). Recall that

$$\mathcal{L}(\texttt{Star R}') = \{[\,]\} \cup \{L_1 @ L_2 \mid L_1 \in \mathcal{L}(\texttt{R}), L_2 \in \mathcal{L}(\texttt{Star R}')\}$$

and that we defined

```
match (Star R') L p =
   p L orelse match R' L (fn L' => match (Star R) L' p)
```

It is obvious that this definition of `match (Star R')` is recursive, and based on the way we defined `match` for `Times`. We will need to find an inductive way to reason about this piece of code. Probably the only hope is to try using induction on the length of the character list, but then how could we show that the length decrease in all recursive calls? So now we realize there's a problem! If we try to adapt the proof argument from the `Times` case we'll need to use the structural induction hypothesis for $\texttt{P}(\texttt{R}')$ at some point, but that's only justifiable if we can prove that the function value

$$(\texttt{fn L}' \texttt{=>} \texttt{match}(\texttt{Star R}')\ \texttt{L}'\ \texttt{p})$$

is total. Even though we'll be assuming that `p` is total, there's no way to deduce that this function value is total. In fact it won't always be total!

After further thought you may be able to find an example of an $\texttt{R}', \texttt{L}$ and `p` in which, even though `p` is total, `match` $(\texttt{Star R}')$ `L p` fails to terminate. This happens if `p(L)=false` and $\mathcal{L}(\texttt{R}')$ contains the empty list. For example, `match` (`Star One`) [`c`] `null` fails to terminate.

At this stage we must abandon the correctness proof and revise the function definition! At least we have some insights into why the `Star` clause is broken.

Lesson

Our function definition seemed to be working out well, and we completed almost the entire correctness proof before discovering that the case for `Star` does not work as intended. But having failed to complete the proof we have insight into why the code doesn't work and we can see how to fix it.

The `match` and `accepts` functions, revisited

There's an easy way to fix the function: insist that `match (Star R′) L p` only ever makes a recursive call to `match(Star R′) L′ p` when `L′` is a shorter list than `L`. Here is the revised definition of `match`. Only the clause for `Star` has been changed from the original flawed definition.

```
fun match Zero L p = false
|   match One L p = p L
|   match (Char c) L p = (case L of
                              [ ] => false
                            | x::L' => (x = c) andalso p L')
|   match (Plus(R1, R2)) L p =
      match R1 L p orelse match R2 L p
|  match (Times(R1, R2)) L p =
      match R1 L (fn L2 => match R2 L2 p)
|  match (Star R') L p =
      p L orelse
        match R' L (fn L' => length L' < length L andalso match (Star R') L' p)

(* accepts : regexp -> char list -> bool *)
(* ENSURES  accepts R L = true iff L is in lang(R) *)

fun accepts R L = match R L null
```

We can prove that this function satisfies the specification above, i.e. we can prove the claim that "For all values `R:regexp`, `P(R)` holds", by repeating the proofs of (i) through (v) from above but with a new inductive step (vi) for the revised `Star` clause.

(vi) For `Star R′`. Recall that

$$\mathcal{L}(\texttt{Star R}') = \{[\,]\} \cup \{L_1@L_2 \mid L_1 \in \mathcal{L}(\texttt{R}), L_2 \in \mathcal{L}(\texttt{Star R}')\}.$$

We can rewrite this language description as

$$\mathcal{L}(\texttt{Star R}') = \{[\,]\} \cup \{L_1@L_2 \mid L_1 \in \mathcal{L}(\texttt{R}), L_1 \neq [\,],\ L_2 \in \mathcal{L}(\texttt{Star R}')\}.$$

(Make sure you understand why this is an equivalent description.) This formula is better suited to help analyze the correctness of the revised function definition, which is:

```
match (Star R') L p =
  p L orelse
   match R' L (fn L' => length L' < length L andalso match (Star R') L' p)
```

We want to show that P(R′) implies P(Star R′).

So assume P(R′) as induction hypothesis. We need to prove P(Star R′).

Since `match(Star R')` may make recursive calls to `match(Star R')`, we need an inductive proof for P(Star(R′)), and remember that we can use P(R′) as a hypothesis. In fact, even though we are *inside* a proof by structural induction on regular expression values, we need to use a different form of induction to deal with the `Star` case. It should be fairly obvious that we can use induction on the length of `L`, to prove (assuming that P(R′) holds):

> For all values `L:char list` and all total values `p`,
> (a) `match (Star R') L p = true` if there are `L1,L2` such that `L = L1@L2`, `p L2 = true`, and `L1` $\in \mathcal{L}$(`Star R'`);
> (b) `match (Star R') L p = false` otherwise.

(This is just P(Star R′), rewritten.)

Proof: by induction on the length of `L`.

- Base case: for `L = [ ]`.
  (a) If `[ ]` has a split `L1@L2` with `L1` $\in \mathcal{L}$(`Star R'`) and `p`(`L2`) = `true`, we have `L1 = [ ]` and `L2 = [ ]`, and `p[ ] = true`. By the function definition we have

  $$\texttt{match (Star R')}\ [\,]\ \texttt{p} = \texttt{p}\ [\,]\ \texttt{orelse}\ (\ldots) = \texttt{true}$$

  as required.
  (b) If `[ ]` has no split `L1@L2` with `L1` $\in \mathcal{L}$(`Star R'`) and `p`(`L2`) = `true`, then `p[ ]` does not evaluate to `true`; since `p` is total, it follows that `p[ ] = false`. So we have

  $$\begin{aligned}&\texttt{match (Star R')}\ [\,]\ \texttt{p}\\ &\quad = \texttt{p}\ [\,]\ \texttt{orelse}\ (\texttt{match R'}\ [\,]\ \texttt{p'})\\ &\quad = \texttt{match R'}\ [\,]\ \texttt{p'}\end{aligned}$$

where p′ is the function value

(fn L′ => length L′ < length [ ] andalso match (Star R′) L′ p).

Note that p′ is extensionally equivalent to (fn L′ => false), and that p′ is total. Also there is no split of [ ] with a suffix that satisfies p′. So by hypothesis P(R′), match R′ [ ] p′ = false. So we've shown that

$$\texttt{match (Star R')}\ [\,]\ \texttt{p} = \texttt{false},$$

as required.

– Inductive step: Let L be a non-empty list, and assume that for all lists L′ shorter than L,

(a') match (Star R') L' p = true if there are L1,L2 such that L' = L1@L2, p L2 = true, and L1 ∈ $\mathcal{L}$(Star R′);

(b') match (Star R') L' p = false otherwise.

Need to show (a) and (b) for L.

(a) Suppose L has a split L1@L2 with L1 ∈ $\mathcal{L}$(Star R′) and p(L2) = true.
If p L = true we can use the split [ ]@L of L as in the base case, so again we get match (Star R′) L p = true as required. Otherwise, we have p L = false and we have a split with L1 non-empty, L2 shorter than L, satisfying the above properties. Hence

$$\begin{aligned}&\texttt{match (Star R')}\ \texttt{L}\ \texttt{p}\\ &\quad = \texttt{p L orelse (match R' L p')}\\ &\quad = \texttt{match R' L p'}\end{aligned}$$

where p′ is the function value

(fn L′ => length L′ < length L andalso match (Star R′) L′ p).

By the induction hypotheses (a') and (b'), p′ is a total function. But L1 is a non-empty list belonging to $\mathcal{L}$(Star R′), so must have a split L1=A@B with A non-empty, A ∈ $\mathcal{L}$(R′) and B ∈ $\mathcal{L}$(Star R′).

Let `B2` be the value of `B@L2`. Then `length B2` $<$ `length L`, so

$$\texttt{p}'(\texttt{B@L2}) = \texttt{p}'(\texttt{B2}) = \texttt{match (Star R}')\ \texttt{B2 p}.$$

But, from above, we know that the list `B2` is shorter than `L` and has a split `B@L2` with `B` $\in \mathcal{L}$(`Star R`$'$) and `p(L2) = true`. So by induction hypothesis `(a')`, `match (Star R`$'$`) B2 p` = `true`. Hence `p`$'$`(B2)` = `true`. So `L` has a split `A@B2` with `A` $\in \mathcal{L}$(`R`$'$) and `p`$'$`(B2)` = `true`. It follows from `P(R`$'$`)` that `match (Star R`$'$`) L p` = `true`, as required.

(b) We leave the details for this part as an exercise. Try adapting the argument given above for the case where `L` is `[ ]`.