# 15-150 Lectures 12 and 13 and 14: Regular Expression Matching; Staging

Lectures by Dan Licata

February 23 and 28 and March 1, 2012

In these lectures, we will discuss regular expression matching. In the process, we will encounter three broader programming techniques that you should take away from this example:

- **Proof-oriented programming**: Programming and proving correctness simultaneously can help you write your code. You can debug your code by attempting to prove it correct and failing: the proof attempt will reveal a bug.
- **Continuations and backtracking**: using higher-order functions, you can explicitly represent "what to do next," which is useful for backtracking algorithms.
- **Staging**: Using curried functions, you can *stage* a multi-argument function so that it does some work when it gets one input, and the rest of the work when it gets another.

## 1 Regular Expressions

*Regular expressions* describe patterns that match strings. Suppose you have a homework directory with files `hwA.sml`, `hwB.sml`, `hwC.sml`, .... In your Linux shell, you can type

```
a2ps hwA.sml
a2ps hw{A,B}.sml
a2ps hw*.sml
```

which will print your solutions to (1) Homework A, (2) Homeworks A and B, and (3) every homework. The reason is that the *regular expression* `hw{A,B}.sml` matches both the string `hwA.sml` and the string `hwB.sml`, while the regular expression `hw*.sml` matches `hwA.sml`, `hwB.sml`, ....

More formally, regular expressions are made up of

| **Regular Expression** | **Matches** |
|---|---|
| 1 | the empty string |
| 0 | nothing |
| $c$ | the character $c$ |
| $r_1 \cdot r_2$ (also written $r_1 r_2$) | the concatenation of a string matching $r_1$ followed by a string matching $r_2$ |
| $r_1 + r_2$ | either a string matching $r_1$ or a string matching $r_2$ |
| $r^*$ | a string made up of any number of substrings, each of which match $r$ |

For example, `hwA.sml` would be written $h \cdot w \cdot A \cdot . \cdot s \cdot m \cdot l$, or just $hwA.sml$. `hw{A,B}.sml` would be written $hw(A+B).sml$. `hw*.sml` would be written $hw(A+B+C+D\ldots+Z)^*.sml$—note that the Linux $*$ matches any sequence of any character, whereas what we just defined only matches strings of the form `hw<letters>.sml`.

### 1.1 Languages

We can formally define the *language* of a regular expression, which is the set of strings that it matches. We write $L(r)$ for the language of $r$:

$$\begin{array}{lcl} L(0) & = & \emptyset \\ L(1) & = & \{\text{``''}\} \\ L(c) & = & \{\text{``}c\text{''}\} \\ L(r_1 + r_2) & = & \{s \mid s \in L(r_1) \text{ or } s \in L(r_2)\} \\ L(r_1 r_2) & = & \{s \mid s = s_1 s_2 \text{ where } s_1 \in L(r_1) \text{ and } s_2 \in L(r_2)\} \\ L(r^*) & = & \text{the least set such that} \\ & & (1)\ \text{""} \in L(r^*), \text{ and} \\ & & (2)\ \text{if } s = s_1 s_2 \text{ where } s_1 \in L(r) \text{ and } s_2 \in L(r^*) \text{ then } s \in L(r^*) \end{array}$$

The last case, where we define $L(r^*)$ to be the least set closed under some conditions, which themselves refer to $L(r^*)$, is an *inductive definition*. This is analogous to a `datatype` definition in SML. To show that $s$ is in $L(r^*)$, you show that it satisfies one of the two conditions. To reason *from* the fact that a string is in $L(r^*)$, you use an induction principle:

> To show $L(r^*) \subseteq S$, it suffices to show (1) "" $\in S$ and (2) if $s = s_1 s_2$ where $s_1 \in L(r)$ and $s_2 \in S$ then $s \in S$

That is, because we have defined $L(r^*)$ to be the least set satisfying some conditions, any other set satisfying the same conditions is necessarily a superset of it.

Exercise: show that $aaaab \in L(a^* ab)$.

## 2 Regular Expression Matcher: First Attempt

We can transcribe the syntax of regular expressions as a datatype as follows:

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

A `char` represents a character. Literals are written like `#"a"` for the character $a$. The only other operation on characters that we will use is comparing them for equality, `chareq(c,c')`.

Our goal is to write a matcher:

```
(* accepts r s == true iff s is in L(r) *)
fun accepts (r : regexp) (s : string) : bool = ...
```

This is a subtle problem: when matching the string `aaaab` against the regexp `a*ab`, how do you know how many `a`'s you match against the `a*` before moving on? In this case it's 3, but you only know that if you know that `ab` is coming. The algorithm will work by *backtracking*: trying the possible ways of matching the first part of the string against `a*`, until one of them leaves something that matches `ab`.

Let's just try to implement it. For convenience, we'll work with lists of characters, `char list`'s, instead of strings. There is a function `explode : string -> char list` that gives you back the list of the characters in the string. Here is the definition of the language of a regular expresison, phrased in terms of character lists:

| | | |
|---|---|---|
| $L(0)$ | $=$ | $\emptyset$ |
| $L(1)$ | $=$ | $\{$`[]`$\}$ |
| $L(c)$ | $=$ | $\{$`[c]`$\}$ |
| $L(r_1 + r_2)$ | $=$ | $\{cs \mid cs \in L(r_1) \text{ or } cs \in L(r_2)\}$ |
| $L(r_1 r_2)$ | $=$ | $\{cs \mid \exists p, s \text{ such that } p@s \cong cs \text{ where } p \in L(r_1) \text{ and } s \in L(r_2)\}$ |
| $L(r^*)$ | $=$ | the least set such that<br>(1) "" $\in L(r^*)$, and<br>(2) if $\exists p, s.p@s \cong cs$ where $p \in L(r)$ and $s \in L(r^*)$ then $cs \in L(r^*)$ |

```
(* Spec: match r cs == true iff cs is in L(r) *)
fun match r (cs : char list) =
    case r of
        Zero => false
      | One => (case cs of [] => true | _ => false)
      | Char c => (case cs of
                       [c'] => chareq(c,c')
                     | _  => false)
      | Plus (r1,r2) => match r1 cs orelse match r2 cs
      | Times (r1,r2) => ???
```

In the `Zero` case, we return false, because no strings are in the language of 0. In the `One` case, we check that the string is empty, because that's the only string in $L(1)$. In the `Char` case we check that the string is exactly the character `c`, because that's the only string in $L(c)$.

In the `Plus` case: we want to check that `cs` is in $L(r_1 + r_2)$. By definition, this means that it must either be in $L(r_1)$ or $L(r_2)$. Inductively, `match r1 cs` determines whether it is in the language of $r_1$, and `match r2 cs` determines whether it is in the language of $r_2$, so we can `orelse` them together. Now that you have a few weeks' experience doing proofs, we want you to start doing *proof-oriented programming*: do the proofs as you write the code, so the spec tells you what code to write!

For the `Times` case, maybe we do something similar:

```
Times (r1,r2) => match r1 cs andalso match r2 cs
```

Inductively, `match r1 cs` and `match r2 cs` determine whether `cs` is in the language of `r1` and `r2`, so we `andalso` them together, which checks that it is in the language of both, and fortunately our definition of $L(r_1 \cdot r_2)$ is that the string is both in $L(r_1)$ and $L(r_2)$. Oh wait, it's **not**! $\cdot$ does not mean intersection, it means that the string splits into two halves, one of which matches $r_1$ and the other matches $r_2$. So this is not the right piece of code! What we just did is *proof-directed debugging*: we found a bug by thinking through the proof.

We could write a little loop to try all possible splittings, but this is silly, because it doesn't use any information from the input to guide the split. A better idea is to try matching some *prefix* of the string against $r_1$, and then match whatever is left against $r_2$.

## 3 Continuation-based Matching

To do this, we will generalize the problem, so that we keep track of *what is required of the remainder of the string, after we have matched a prefix of it against the regexp.* Then, the $\cdot$ case, we can first match a prefix against $r_1$, and then match the suffix against $r_2$.

That is, we generalize the problem to:

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) = ...
```

where `match r cs k` returns true iff `r` matches some *prefix* of `s` and the function `k` returns true on whatever is leftover. `k` is what is called a *continuation*—it is a function argument that tells you what to do *in the future*, after you have matched some of `s` against `r`. Passing a function as an argument to another function allows us to easily represent this continuation.

Once again, we will do the code and proof simultaneously.

**The Spec** How can our matcher go wrong? It might be *too easy*: that is, it might says "yes" for a string that is not really in the language (according to the mathematical definition). Or, it might be *too strict*: it might not say yes for a string that *is* really in the language. This motivates:

*Soundness* If the matcher code says yes, the string is in the language.

*Completeness* If the string is in the langauge, the matcher says yes.

A violation of soundness means the matcher is too easy; a violation of completeness means the matcher is too strict.

How do we turn this into math? Note that the "the string is in the language" isn't quite precise, because the continuation matcher really checks that there is some prefix that is in the language, such that the continuation accepts the suffix. Thus, for all $r$, we have that:

*Soundness* For all $cs, k$, if $\texttt{match}\, r\, cs\, k \cong \texttt{true}$ then there exist $p, s$ such that $p@s \cong cs$ and $p \in L(r)$ and $k\ s \cong \texttt{true}$.

*Completeness* For all $cs, k$, if (there exist $p, s$ such that $p@s \cong cs$ and $p \in L(r)$ and $k\ s \cong \texttt{true}$) then $\texttt{match}\, r\, cs\, k \cong \texttt{true}$.

We prove these simultaneously by structural induction on the regular expression $r$. This has the following template:

To show $\forall r : \texttt{regexp}, P(r)$, it suffices to show:
**Case for** `Zero`: To show: $P(\texttt{Zero})$
**Case for** `One`: To show: $P(\texttt{One})$
**Case for** `Char c`: To show: $P(\texttt{Char c})$
**Case for** `Plus(r1,r2)`: IH: $P(\texttt{r1})$ and $P(\texttt{r2})$. To show: $P(\texttt{Plus(r1,r2)})$
**Case for** `Times(r1,r2)`: IH: $P(\texttt{r1})$ and $P(\texttt{r2})$. To show: $P(\texttt{Times(r1,r2)})$
**Case for** `Star(r)`: IH: $P(\texttt{r})$. To show: $P(\texttt{Star(r)})$

In this case, we take $P$ to be the whole statement of soundness and completeness above.
Now we program and prove.

**Zero**

```
fun match r cs k =
  case r of
      Zero => false
    | ...
```

The completeness of this case is suspicious: it says that the matcher returns true, but this always returns false. What gives?

*Complete* Assume $k, cs$ and assume $\exists p, s$ such that $p$`@`$s \cong cs$ with $p \in L($`Zero`$)$ and $k\, s \cong$ `true`. We need to show that `match Zero` $s\, k \cong$ `true`. But it doesn't—it always returns false! So clearly we cannot establish the conclusion directly.

What saves us is that the assumption $p \in L($`Zero`$)$ is contradictory, because no strings are in the language of `Zero`, which is the empty set. So the result is vacuously true.

*Sound* Conversely, assume $k, cs$ such that `match Zero` $cs\, k \cong$ `true`. We need to show that $\exists p, s$ such that $p$`@`$s \cong cs$ with $p \in L($`Zero`$)$ and $k\, s \cong$ `true`

We can calculate that `match Zero` $cs\, k \cong$ `false` in two steps. Thus, by transitivity with the assumption, `true` $\cong$ `false`. This is a contradiction, so the result is vacuously true.

**One** First, we will define

```
fun match r cs k =
  case r of
      ...
    | One  => k cs
    | ...
```

For any $k$,$cs$, observe that `match One` $cs\, k \cong k\, cs$ by stepping.

Let's check soundness here. Soundness says that we need to peel off some prefix that matches the regexp, but we didn't peel anything off here. Why does that make sense?

*Sound* Assume $k$,$cs$ such that `match One` $cs\, k \cong$ `true`. We need to show that $\exists p, s$ such that $p@s \cong cs$ with $p \in L($`One`$)$ and $k\, s \cong$ `true`.

By transitivity, $k\, cs \cong$ `true` as well. So, we can take $p$ to be `[]` and $s$ to be $cs$, in which case `[]@`$cs \cong cs$, and `[]` is in the language of `One`, and $k\, cs \cong$ `true`, which establishes the three things we needed to show.

That is, we want to peel off the empty string, which is in the language of `One`, and leaves the suffix unchanged.

*Complete* Assume $k$,$cs$ and assume $\exists p@s$ such that $p@s \cong cs$ with $p \in L($`One`$)$ and $k\, s \cong$ `true`. We need to show that `match One` $cs\, k \cong$ `true`.

The assumption $p \in L($`One`$)$ entails that $p$ is the empty string, because that is the only string in the language of `One`. Thus, $s$ is all of $cs$: `[]`$@s \cong cs$ (by assumption) and `[]`$@s \cong s$ (by stepping) so $s \cong cs$. By assumption, $k\, s \cong$ `true`, so $k\, cs \cong$ `true`, so `match One` $cs\, k \cong$ `true` as well.

**Char** Define

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
  case r of
    ...
    | Char c => (case cs of
                     []  => false
                   | c' :: cs' => chareq(c,c') andalso k cs')
    ...
```

In the case for `Char`, we check that the first character is `c`, and then feed the remainder to the continuation.

Thus, we know that, for any $k, cs$, `match` (`Char` $c$) $cs\, k$ steps to

```
case cs of []  => false | c' :: cs' => chareq(c,c') andalso k cs'
```

*Complete* Assume $k$,$cs$ such that $\exists p, s$ such that $p$`@`$s \cong cs$ with $p \in L($`Char c`$)$ and $k\, s \cong$ `true`. We need to show that `match (Char c)` $cs\, k \cong$ `true`

The assumption $p \in L($`Char c`$)$ entails that $p$ is the string $[c]$, because that is the only string in the language of `Char c`. Moreover, $[c]@s \cong c :: s$, and $[c]$`@`$s \cong cs$ by assumption, so $cs \cong c :: s$ by transitivity. Thus, the case on `cs` steps to the second branch:

```
    case c::s of []  => false | c' :: cs' => chareq(c,c') andalso k cs'
 == chareq(c,c) andalso k s
```

The equality test `chareq(c,c)` returns `true` (we assume that `chareq` is implemented correctly). By assumption, $k\, s \cong$ `true`, so the whole `andalso` evaluates to true.

*Sound* Conversely, assume $k$,$cs$ such that `match (Char c)` $cs\, k \cong$ `true`. We need to show that $\exists p, s$ such that $p$`@`$s \cong cs$ with $p \in L($`Char c`$)$ and $k\, s \cong$ `true`.

By transitivity, on the assumption that `match (Char c)` $cs\, k \cong$ `true` and the calculation for the LHS above,

`(case cs of []  => false | c' :: cs' => chareq(c,c') andalso k cs')` $\cong$ `true`

`cs` is either `[]` or `c'::s'`, so we have two cases to consider.

In the first case, `case [] of [] => false | ...` $\cong$ `false`, so by transitivity `false` $\cong$ `true`, which is a contradiction.

In the second case, the `case` steps to `chareq(c,c') andalso k s'` so by transitivity this `andalso` evalaute to `true`.

We use inversion for `analso`: if `e1 andalso e2` $\cong$ `true` then `e1` $\cong$ `true` and `e2` $\cong$ `true`. By inversion for `andalso`, this means that $k\, cs \cong$ `true` and `chareq(c,c')` $\cong$ `true`. So we take $p$ to be $[c]$ and $s$ to be $s'$, in which case $[c]$`@`$s' \cong c' :: s'$ by stepping, and $c$ is in $L($`Char c`$)$ by definition, and $k\, s \cong$ `true`.

**Plus** Let's try working through the proof a little before we write the code.

*Complete* Assume $k, cs$ and that $\exists p, s$ such that $p@s \cong cs$ with $p \in L(\texttt{Plus}(r_1, r_2))$ and $k\,s \cong \texttt{true}$. We need to write some code such that $\texttt{match}\,(\texttt{Plus}\,(r_1, r_2))\,cs\,k \cong \texttt{true}$.

By definition, either $p \in L(r_1)$ or $p \in L(r_2)$. In the former case, we now know that $p@s \cong cs$ with $p \in L(r_1)$ and $k\,s \cong \texttt{true}$. Our inductive hypothesis tells us that soundness and completeness hold for $r_1$, so in particular completeness holds. We have just established the premise of completenesss, so we may conclude that $\texttt{match}\,r_1\,cs\,k \cong \texttt{true}$. By analogous reasoning, in the other case $\texttt{match}\,r_2\,cs\,k \cong \texttt{true}$.

We need to find something that returns `true` in either of these cases. (Moreover, thinking through soundness, we need to find something that returns true in exactly these cases, because the prefix must either be in $L(r_1)$ or $L(r_2)$ to be in $L(r_1 + r_2)$).

Thus, we define

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
  case r of
      ...
    | Plus (r1,r2) => match r1 cs k orelse match r2 cs k
      ...
```

Observe that

```
match (Plus(r1,r2)) cs k ≅ match r1 cs k orelse match r2 cs k
```

When $\texttt{match}\,r_1\,cs\,k \cong \texttt{true}$, the whole `orelse` evaluates to true: `orelse` short-circuits, and ignores the second disjunct if the first disjunct returns true.

When $\texttt{match}\,r_2\,cs\,k \cong \texttt{true}$, the `orelse` evaluates to true *as long as the first disjunct terminates*. For this, we need to prove that the matcher terminates. We will come back to this below.

*Sound* Assume $cs, k$ such that $\texttt{match}\,(\texttt{Plus}\,(r_1, r_2))\,cs\,k \cong \texttt{true}$. We need to show that $\exists p, s$ such that $p@s \cong cs$ with $p \in L(\texttt{Plus}(r_1, r_2))$ and $k\,s \cong \texttt{true}$.

By transitivity, $\texttt{match}\,r_1\,cs\,k$ `orelse` $\texttt{match}\,r_2\,cs\,k \cong \texttt{true}$. By inversion for `orelse`, this means that either the left-hand side evaluates to true, or it evaluates to false and the right-hand side evaluates to true.

In the former case, we know that $\texttt{match}\,r_1\,cs\,k \cong \texttt{true}$. Thus, by the soundness IH on $r_1$, $\exists p, s$ such that $p@s \cong cs$ with $p \in L(r_1)$ and $k\,s \cong \texttt{true}$. By definition, $p$ is also in $L(\texttt{Plus}(r_1, r_2))$. So we have shown what we needed to show.

In the latter, the soundness IH on $r_2$ gives that $\exists p, s$ such that $p@s \cong cs$ with $p \in L(r_2)$ and $k\,s \cong \texttt{true}$. $p$ is also in $L(\texttt{Plus}(r_1, r_2))$, so we have shown what we needed to show.

**Times** We generalized the problem to the continuation-based matcher to make times work as follows:

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
  case r of
      ...
    | Times (r1,r2) => match r1 cs (fn cs' => match r2 cs' k)
    | ...
```

To match `cs` against `Times(r1,r2)`, we first match some prefix of `cs` against `r1`, with a continuation `(fn cs' => match r2 cs' k)`. When whatever is leftover after matching `r1` gets plugged in for `cs'`, this continuation will match some prefix of *it* against `r2` and then feed the remainder to `k`. Thus, we will have peeled off part of the string that matches `r1`, then part that matches `r2`, and then fed the rest to `k`.

Observe that for all $k, cs$,

$$\texttt{match}\,(\texttt{Times}(r_1, r_2))\; cs\; k \cong \texttt{match}\, r_1\; cs\; (\texttt{fn}\; cs' \;\texttt{=>}\; \texttt{match}\; r_2\; cs'\; k)$$

*Sound* Assume $cs, k$ such that $\texttt{match}\,(\texttt{Times}\,(r_1, r_2))\; cs\; k \cong \texttt{true}$. We need to show that $\exists p, s$ such that $p\texttt{@}s \cong cs$ with $p \in L(\texttt{Times}(r_1, r_2))$ and $k\, s \cong \texttt{true}$.

By transitivity,

$$\texttt{match}\, r_1\; cs\; (\texttt{fn}\; cs' \;\texttt{=>}\; \texttt{match}\; r_2\; cs'\; k) \cong \texttt{true}$$

Thus, we can apply the soundness part of the IH on $r_1$, taking the continuation to be $(\texttt{fn}\; cs' \;\texttt{=>}\; \texttt{match}\; r_2\; cs'\; k)$. Thus, we learn that there exist $p_1, s_1$ such that $p_1@s_1 \cong cs$ and $p_1 \in L(r_1)$ and

$$(\texttt{fn}\; cs' \;\texttt{=>}\; \texttt{match}\; r_2\; cs'\; k)s_1 \cong \texttt{true}$$

Observe that $(\texttt{fn}\; cs' \;\texttt{=>}\; \texttt{match}\; r_2\; cs'\; k)s_1 \mapsto \texttt{match}\; r_2\; s_1\; k$, so

$$\texttt{match}\; r_2\; s_1\; k \cong \texttt{true}$$

Thus, we can apply the soundness part of the IH again, taking the string to be $s_1$, and concluding that there exist $p_2, s_2$ such that $p_2@s_2 \cong s_1$ and $p_2 \in L(r_2)$ and $ks_2 \cong \texttt{true}$.

Thus, the two inductive calls first divide $cs$ into $p_1@s_1$, and then $s_1$ into $p_2@s_2$, peeling off first a prefix $p_1 \in L(r_1)$, and then a prefix of the suffix $p_2 \in L(r_2)$. Putting these facts together shows that $p_1@(p_2@s_2) \cong cs$.

We use (without proof) a lemma that append is associative, which means that $(p_1@p_2)@s_2 \cong cs$. Thus, we can think of the division of $cs$ as a prefix $(p_1@p_2)$ and a suffix $s_2$.

By definition, $(p_1@p_2) \in L(\texttt{Times}(r_1, r_2))$, because the specified split is a division such that $p_1 \in L(r_1)$ and $p_2 \in L(r_2)$.

Moreover, we have concluded from the second IH that $ks_2 \cong \texttt{true}$.

Thus, we can take $p$ to be $(p_1@p_2)$ and $s$ to be $s_2$ to establish the conclusion.

*Complete* Assume $cs, k$ and that $\exists p, s$ such that $p\texttt{@}s \cong cs$ with $p \in L(\texttt{Times(}r_1, r_2\texttt{)})$ and $k\, s \cong \texttt{true}$. We need to show that $\texttt{match}\,(\texttt{Times}\,(r_1, r_2))\, cs\, k \cong \texttt{true}$. To show this, it suffices to show that $\texttt{match}\, r_1\, cs\, (\texttt{fn}\, cs' \texttt{=>}\, \texttt{match}\, r_2\, cs'\, k) \cong \texttt{true}$.

Thus, we will apply the completeness IH on $r_1$ to show that $\texttt{match}\, r_1\, cs\, (\texttt{fn} cs' \texttt{=>} \texttt{match} r_2 cs' k) \cong \texttt{true}$. To satisfy the premise of the IH, we need to show three things:

First, we show that $p_1@(p_2@s) \cong cs$. By definition, there exist $p_1, p_2$ such that $p_1@p_2 \cong p$ where $p_1 \in L(r_1)$ and $p_2 \in L(r_2)$. Since $(p_1\texttt{@}p_2)\texttt{@}s \cong cs$ by assumption, $(p_1@p_2)@s \cong cs$ by associativity.

Second, we know that $p_1 \in L(r_1)$.

Third, we need to show that the continuation accepts $(p_2@s)$:

$$(\texttt{fn}\, cs' \texttt{=>}\, \texttt{match}\, r_2\, cs'\, k)(p_2\texttt{@}s) \cong \texttt{true}$$

This steps to $\texttt{match}\, r_2\, (p_2\texttt{@}s)\, k$ (note that @ is total, so $(p_2\texttt{@}s)$ is valuable). So it suffices to show that $\texttt{match}\, r_2\, (p_2\texttt{@}s)\, k \cong \texttt{true}$. To do so, we apply the completeness IH on $r_2$, observing that $p_2@s \cong p_2@s$, that $p_2 \in L(r_2)$, and that $k\ s \cong \texttt{true}$ was assumed above. Thus, the IH shows that $\texttt{match}\, r_2\, (p_2@s)\, k \cong \texttt{true}$.

**Star** As with plus, let's start thinking through completeness to see how to write the code. We assume $cs, k$ such that $\exists p, s$ such that $p\texttt{@}s \cong cs$ with $p \in L(\texttt{Star(}r\texttt{)})$ and $k\, s \cong \texttt{true}$. We want to show that $\texttt{match}\,(\texttt{Star}\, r)\, cs\, k \cong \texttt{true}$.

What does it mean for $p \in L(\texttt{Star}\, r)$. Expanding the definition, there are exactly two possibilities: either $p$ is the empty string, or $p$ splits as $p_1@p_2$ where $p_1 \in L(r)$ and $p_2 \in L(\texttt{Star}\, r)$. We need the matcher to return true in either of these two cases, which suggests the code will involve an `orelse` as in the case for `Plus`. Moreover, this definition says that to check that something is in the language of $r^*$, we need to check that something (else) is in the language of $r^*$—recursively! So we will define a recursive helper function for this case:

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
  case r of
      ...
    | Star r =>
        let fun matchstar cs’ = k cs’ orelse match r cs’ matchstar
        in
            matchstar cs
        end
```

We write a local, recursive, helper function `matchstar`. Note that this helper function mentions the `r` and `k` bound in the arguments to `match`, which is why it is helpful to define it inside of `match`, rather than in a `local` block before it (if you wanted to do this, you could parametrize `matchstar` by `r` and `k` and pass them in).

The idea with `matchstar` is this: to account for the empty prefix, we check $k$ on the whole string, just like for `One`. Otherwise, to check that $cs'$ has a prefix that matches $r$ and a suffix that matches $r^*$, we make a recurisve recursive call to `match r`, using the local helper function `matchstar` *itself* as the continuation. This way, the continuation also checks the empty prefix, and then tries peeling off something matching `r`, ..., and so on.

It's important to wrap your head around the idea that you can define a recursive function that passes itself as an argument to another function. There is no way to do this with a for-loop or a while-loop, where the structure of the loop has to be explicit in the code. Here, `match` is inside the "body" of the loop defined by `matchstar`, as `match` may eventually call the continuation, which will take you back to the "top" of the loop defined by `matchstar`. In this way, recursion with higher-order functions enables more general patterns of control flow than loops do.

Let's check soundness:

*Sound* Soundness of `matchstar` is expressed by the following lemma: If $\texttt{matchstar}\, cs \cong \texttt{true}$ then $\exists p@s \cong cs$ with $p \in L(\texttt{Star}\, r)$ and $k\, s \cong \texttt{true}$.

Proof: Suppose $\texttt{matchstar}\, cs \cong \texttt{true}$. By inversion for `orelse`, we have two cases:

If $k\ cs \cong \texttt{true}$, then we can choose $p$ to be empty, and $s$ to be $cs$, and observe additionally that $[]@cs \cong cs$ and $[] \in L(\texttt{Star}\, r)$.

Otherwise, we have that $\texttt{match}\, r\, cs\, \texttt{matchstar} \cong \texttt{true}$. By the IH on $r$, this means that there exists $p@s \cong cs$, such that $p \in L(r)$ and $\texttt{matchstar}\, s \cong \texttt{true}$.

*If we had an inductive hypothesis for `matchstar` applied to cs*, we could finish the proof: this would say that we can split $s$ into a prefix $s_1 \in L(\texttt{Star}\, r)$ and a suffix $s_2$ accepted by $k$, and then as in the `Times` case we could choose the prefix to be $p@s_1$ and the suffix to be $s_2$. However, it's not clear what justifies this inductive call. We'll sort this out next time.

We'll also discuss completeness next time.

**Summary** The complete matcher is in Figure 1.

Using `match`, we write `accepts` by passing the function that accepts only the empty list as the initial continuation. Thus, `match` checks that $\exists p', s'$ such that $p'@s' \cong$ `explode s` where $p' \in L(r)$ and $s'$ is the empty list. However, if the suffix is the empty list, then the prefix is the whole list, so this show that `explode s` is in the language of $r$. Thus, assuming `match` is correct, `accepts r s` $\cong$ `true` iff $s \in L(r)$.

Next time, we will sort out the issues with the case of soundness for $r^*$, and finish completeness. Completeness relies on termination, which we have not yet established! In fact, when we look into termination next time, we will find a subtle bug: there are cases in which the above matcher doesn't terminate! However, the attempted proof of termination will also suggest how to fix that bug.

# 4 Proof-Directed Debugging

The above matcher has a termination bug. You could try to find it by testing, or pouring over the code and being really clever. However, here is a systematic way to find the bug: try to prove the matcher terminates, and see where the proof breaks down. We will find a counterexample to the proof, which will turn out to be an input that violates the specification.

**Theorem 1** (Termination, Take 1). *For all $r$ : `regexp`, $cs$ : `char list`, $k$ : `char list` $\rightarrow$ `bool`, $\texttt{match}\, r\, cs\, k$ is valuable.*

```
fun match (r : regexp) (cs : char list) (k : char list -> bool) : bool =
  case r of
      Zero => false
    | One => k cs
    | Char c => (case cs of
                    []  => false
                  | c' :: cs' => chareq(c,c') andalso k cs')
    | Plus (r1,r2) => match r1 cs k orelse match r2 cs k
    | Times (r1,r2) => match r1 cs (fn cs' => match r2 cs' k)
    | Star r =>
        let fun matchstar cs' = k cs' orelse match r cs' matchstar
        in
            matchstar cs
        end

fun accepts (r : regexp) (s : string) =
   match r (String.explode s) (fn l => case l of [] => true | _ => false)
```

Figure 1: Regular Expression Matcher

Let's try the `Star` case first, since, as we discussed above, that's the one that isn't structurally recursive.

*Attempted proof.* **Case for `Star r`**: IH: For all $cs'$,$k'$, `match` $r\ cs'\ k'$ is valuable.
Proof: Assume $k, cs$. To show: `match` (`Star` $r$) $cs\ k$ is valuable. Observe that

```
     match (Star r) cs k
==   let fun matchstar cs' = k cs' orelse match r cs' matchstar
     in matchstar cs end
==   matchstar cs
==   k cs  orelse  match r cs matchstar
```

To show that this is valuable, the first thing we need to do is show that $k\ cs$ is valuable. But we haven't assumed anything about $k$! In particular, $k$ might be `fn _ => <infinite loop>`. So the theorem, as phrased above, is definitely not true. □

However, this a bug in the spec, not the code: we only care about the behavior of the matcher on terminating continuations. So we can revise the spec to assume that $k$ is total.

**Theorem 2** (Termination, Take 2)**.** *For all $r$ : `regexp`, $cs$ : `char list`, $k$ : `char list` → `bool`, if $k$ is total then `match` $r\ cs\ k$ is valuable.*

Let's return to the

*Attempted proof.* **Case for `Star r`**: As above, it suffices to show that

```
k cs  orelse  match r cs matchstar
```

is valuable. Because $k$ is total, $k\,cs$ is valuable. So it suffices to show that `match` $r\ cs$ `matchstar` is valuable.

This follows from the IH on $r$, right? *No!* Given the revision to the spec, the IH now says that *if $k'$ is total, then* `match` $r\ cs'\ k'$ *is valuable.* So to use the IH to conclude that `match` $r\ cs$ `matchstar` is valuable, we need to show that `matchstar` is total. But that's exactly what we're trying to show in this case! □

It's possible that the theorem is true and we just haven't found the right proof yet. One thing we can try is to prove `matchstar` total by an *inner induction* on *cs*: inside of the "outer" induction on the regular expression, we do an induction on the string.

**Lemma 1.** *For all cs,* `matchstar` *cs is valuable.*

*Attempted proof.* Structural induction on *cs*. For this to work, it needs to be the case that whenever `matchstar` is called recursively, the call is made on a smaller string. However, the recursive calls are not readily apparent: they happen whenever `match r cs' matchstar` calls its continuation argument. Thus, we need to look at the calls to `k` in `match` to see where the recursive calls happen.

Maybe it is the case that

> Whenever `match r cs k` calls `k cs'`, `cs'` is a strict suffix of `cs`.

We can prove this by inspecting the code. For example, in the `One` case... uh oh! In this case `match One cs k` calls `k cs` directly, so the string is not smaller. □

**Counterexample** This failure suggests a concrete counterexample: consider the behavior of `matchstar` when $r$ is the regular expression `Star One`. It is easy to verify that for any string *cs* not accepted by $k$, `matchstar` *cs* diverges (goes into an infinite loop)!

```
     matchstar cs
==   k cs orelse match One cs matchstar
==   match One cs matchstar              [ because k cs is false ]
==   matchstar cs
```

So, in fact, `matchstar` is *not* total after all! And therefore `match` does not terminate. Termination is false, and by attempting the proof, we found the bug: the `Star` case loops when it attempts to continually peel off the empty string and recursively match the rest.

### 4.1 Solution 1: Checks

What to do? The theorem is *false*, so it is no wonder that the proof attempt breaks down! But what now? Following Imré Lakatos, we observe that *the proof attempt proves something, just not the theorem we stated.* So what does the proof prove? To ensure that `matchstar` is total, it is enough to ensure that each match of $r$ consumes some non-empty portion of its input. For then the subsequent calls to `matchstar` are indeed on shorter strings, and we may use an inner induction on the length of the input to show that `matchstar` is total.

There are at least two ways to do this. The obvious method is to insist in the definition of `matchstar` that $r$ must match some non-empty initial segment of the input, by explicitly checking that the final segment passed to `matchstar` is, in fact, a proper suffix. We do this by inserting a run-time check:

```
fun matchstar cs' =
   k cs' orelse
   match r cs' (fn cs'' => suffix cs'' cs' andalso matchstar cs'')
```

The function `suffix` checks that its first argument is a proper suffix of the second. Recall from a few lectures ago: the purpose of a run-time check is to establish a spec. The spec tells us what to check to do termination.

Now, we can prove, by an inner induction, that:

**Lemma 2.** *For all cs,* `matchstar` *cs is valuable*

*Proof.* By complete induction on *cs*:
IH: For all strict suffices $cs'$ of $cs$, `matchstar` $cs'$ is valuable.
TS: `matchstar` $cs$ is valuable.

As above,

```
     match (Star r) cs k
==   k cs  orelse  match r cs (fn cs'' => suffix cs'' cs andalso matchstar cs'')
```

Because $k$ is assumed to be total, the $k\ cs$ is valuable.

To show that

```
match r cs (fn cs'' => suffix cs'' cs andalso matchstar cs'')
```

is valuable, we can appeal to the outer inductive hypothesis on $r$, provided that the continuation is total. Thus, we assume some `cs''` and show that

```
suffix cs'' cs  andalso  matchstar cs''
```

is valuable. The spec for `suffix` is that it is total, and that it returns `true` iff `cs''` is in fact a strict suffix of `cs`. Thus, the first conjunct is valuable, we only run the second conjunct when `cs''` is a suffix of `cs`. Our inner IH says that `matchstar` is valuable on all suffices of `cs`, so it is valuable on `cs''`. Thus, the continuation is total. □

However, this solution has some drawbacks: First of all, why is it complete? It could potentially be necessary to pull off an empty prefix sometimes. It turns out that it is complete, because the initial check for whether the continuation accepts the whole string is sufficient to catch the case where $r$ may accept the empty string as well. But this is harder to show.

The second drawback of this method is that it imposes an additional run-time check during matching.

### 4.2 Solution 2: Specs

A less obvious approach is to change the spec, to preclude cases cases where the matcher might peel off an empty prefix. This is called *monster-baring*: change the statement of the theorem to rule out the counterexamples.

Specifically, we can impose the requirement that the regular expression be in *standard form*, which means that whenever `Star`$(r_1)$ occurs within it, the regular expression $r_1$ must not accept the empty string. So, in particular, the counterexample `Star(One)` is not in standard form. Moreover, there is no loss of generality in imposing this restriction, because *every regular expression is equivalent to one in standard form*, in the sense that they accept the same language. (We will not give a proof of this fact here.)

### Termination

You need to set up the theorem statement up carefully to allow this proof to go through. We write $s < cs$ to means $s$ is a strict suffix of $cs$, and $s \leq cs$ to mean $s$ is a suffix or equal to $cs$.

**Lemma 3** (Termination, Inductively). *For all $r$ : `regexp`, if $r$ is standard, then:*

1. *For all $cs$ : `char list`, $k$ : `char list` $\rightarrow$ `bool`, if (for all $cs' \leq cs$, $k\, cs'$ is valuable) then `match` $r\, cs\, k$ is valuable.*
2. *For all $cs$ : `char list`, $k$ : `char list` $\rightarrow$ `bool`, if $[] \notin L(r)$ and (for all $cs' < cs$, $k\, cs'$ is valuable) then `match` $r\, cs\, k$ is valuable.*

The first clause that the matcher is valuable when given a continuation that may be applied to $cs$ or any suffix. The second says that the matcher is valuable when given any continuation that can be applied *only* to a strict suffix, provided that $r$ does not accept the empty string. We sometimes say "$f$ is valuable on ¡certain lists¿" to mean that $f\, l$ is valuable for each of those lists.

*Proof.* We sketch the easy cases and then show the interesting ones in detail.

`Zero` `match Zero` $cs\, k \cong$ `false` independently of the assumptions, so both (1) and (2) hold.

`One`
1. Observe that `match One` $cs\, k \cong\; k\, cs$, and that $cs \leq cs$, so the assumption about $k$ tells us that $k\, cs$ is valuable.
2. We cannot apply the assumption about $k$ in this case, because $cs$ is not a strict suffix of itself. Forunately, the assumption that $[] \notin L(1)$ is contradictory, so the case holds vacuously.

`Char` $c$ Observe that, for any $k, cs$, `match` (`Char` $c$) $cs\, k$ steps to

```
case cs of []  => false | c' :: cs' => chareq(c,c') andalso k cs'
```

`cs` must be either [], in which case `false` is valuable, or a cons, in which case we step to `chareq(c,c') andalso k cs'`. `chareq` is valuable, so it suffices to show that $k\, cs'$ is valuable. In both (1) and (2), the assumption about the continuation says that it is valuable on strict suffices, and $cs' < (c :: cs')$, so the assumption gives the result.

`Plus` $(r_1, r_2)$ Observe that

```
match (Plus(r1,r2)) cs k == match r1 cs k orelse match r2 cs k
```

Thus, it suffices to show that both disjuncts are valuable. Moreover, `Plus` $(r_1, r_2)$ is standard, so $r_1$ and $r_2$ are.

1. Assume that $k$ is valuable on $cs$ and its suffices. To use IH part 1 for $r_1$ to prove that $\texttt{match}\, r_1\, cs\, k$ is valuable, it suffices to show that $r_1$ is standard (which we concluded above) and that $k$ is valuable on $cs$ and its suffices (which is exactly the assumption). Analogously, the IH Part 1 for $r_2$ says that the other disjunct is valuable.
2. Assume that $[] \notin L(\texttt{Plus}(r_1, r_2))$ and that $k$ is valuable on strict suffices of $cs$. By definition of $L(+)$, $[] \notin L(r_1)$ and $[] \notin L(r_2)$. Thus, we can appeal to the IH part 2 in for both $r_1$ and $r_2$ to get the results.

$\texttt{Times}\,(r_1, r_2)$ Observe that for all $k, cs$,

$$\texttt{match}\,(\texttt{Times}(r_1, r_2))\, cs\, k \mapsto^* \texttt{match}\, r_1\, cs\,(\texttt{fn}\, cs' \texttt{ => match}\, r_2\, cs'\, k)$$

Moreover, $\texttt{Times}\,(r_1, r_2)$ is standard, so $r_1$ and $r_2$ are.

This one is slightly tricky:

1. Assume $cs$, $k$ and that (for all $cs' \leq cs$, $k\, cs'$ is valuable).
   Suffices to show: $\texttt{match}\, r_1\, cs\,(\texttt{fn}\, s \texttt{ => match}\, r_2\, s\, k)$ is valuable.
   We will use the IH Part 1 on $r_1$ to give the result. So we must show that $r_1$ is standard (check) and that

   $$\forall cs' \leq cs, (\texttt{fn}\, s \texttt{ => match}\, r_2\, s\, k)\, cs' \text{ is valuable}$$

   So assume some $cs' \leq cs$.

   $$(\texttt{fn}\, s \texttt{ => match}\, r_2\, s\, k)\, cs' \cong \texttt{match}\, r_2\, cs'\, k$$

   so it suffices to show that $\texttt{match}\, r_2\, cs'\, k$ is valuable.
   We will use the IH Part 1 on $r_2$ to prove this. So we must show that $r_2$ is standard (check) and that

   $$\forall cs'' \leq cs', k\, cs'' \text{ is valuable}$$

   We know that $k$ is valuable on $cs$ and its suffices. Moreover, we know that $cs' \leq cs$, so any $cs'' \leq cs'$ is also $\leq cs$. Thus, the assumption that $k$ is valuable on $cs$ and its suffices suffices to justify the appeal to the IH.
2. Assume $cs$, $k$, that $[] \notin L(\texttt{Times}(r_1, r_2))$, and that (for all $cs' < cs$, $k\, cs'$ is valuable).
   Suffices to show: $\texttt{match}\, r_1\, cs\,(\texttt{fn}\, s \texttt{ => match}\, r_2\, s\, k)$ is valuable.

   $[] \notin L(\texttt{Times}(r_1, r_2))$, we have two cases: either $[] \notin L(r_1)$ or $[] \notin L(r_2)$. (because if it was in the language of both, it would also be in the language of $\texttt{Times}$, because $[]@[] \cong \texttt{Empty}$).
   Case 1: Suppose $[] \notin L(r_1)$. Then we will use the IH Part 2, observing that $r_1$ is standard, and that

   $$\forall cs' < cs, (\texttt{fn}\, s \texttt{ => match}\, r_2\, s\, k)\, cs' \text{ is valuable}$$

   Assume some $cs' < cs$. As above, it suffices to show that $\texttt{match}\, r_2\, cs'\, k$ is valuable. To do this, we need to use IH *Part 1*, because we don't know that $[] \notin L(r_2)$. So, we observe that $r_2$ is standard, and that

   $$\forall cs'' \leq cs', k\, cs'' \text{ is valuable}$$

We only know that $k$ is valuable on strict suffices of $cs$, but forunately $cs'$ is a strict suffix, so any $cs'' \leq cs'$ is also $< cs$. Thus, the assumption is sufficient for the call to the IH.

Case 2: Suppose $[] \notin L(r_2)$. Then we will use the IH *Part 1*, observing that $r_1$ is standard, and that

$$\forall cs' \leq cs, (\texttt{fn}\ s\ \texttt{=>}\ \texttt{match}\ r_2\ s\ k)\ cs' \text{ is valuable}$$

Assume some $cs' \leq cs$. As above, it suffices to show that $\texttt{match}\ r_2\ cs'\ k$ is valuable. To do this, we may use IH Part 2, because we know that $[] \notin L(r_2)$. So, we observe that $r_2$ is standard, and that

$$\forall cs'' < cs', k\ cs'' \text{ is valuable}$$

We know that $k$ is valuable on strict suffices of $cs$, but forunately $cs''$ is a strict suffix of $cs'$, and that $cs' \leq cs$, so $cs'' < cs$. Thus, the assumption is sufficient for the call to the IH.

At a high level, in this case, we know that $k$ is valuable on strict suffices. We know that the empty string is not in the language of one of the two regexps. So by the time the code calls $k$, one of them has consumed a non-empty prefix—but we don't know whether it's $r_1$ or $r_2$. The two cases above consider both possibilities.

$\texttt{Star}\ r$ Part 2 is vacuously true, because the assumption that $[] \notin L(\texttt{Star}\ r)$ is contradictory. So it remains to prove Part 1.

Outer IH: termination holds for $r$.

Assume $cs$ and $k$ such that (for all $cs' \leq cs$, $k\ cs'$ is valuable).

To show: $\texttt{match}\ (\texttt{Star}\ r)\ cs\ k$ is valuable.

It suffices to prove that $\texttt{matchstar}\ cs$ is valuable, which we do using the following lemma:

For all $cs' \leq cs$, $\texttt{matchstar}\ cs'$ is valuable.

The proof is by *complete induction on $cs'$*: this means we assume the theorem for all strict suffices of $cs'$, and prove it for $cs'$.

Assume some $cs' \leq cs$.

Inner IH: For all $cs'' < cs'$, if $cs'' \leq cs$ then $\texttt{matchstar}\ cs''$ is valuable.

To show: $\texttt{matchstar}\ cs'$ is valuable.

Proof: By calculation, it suffices to show that

```
k cs' orelse match r cs' matchstar
```

is valuable. We have assumed that $k$ is valuable on $cs$ and its suffices, so it is valuable on $cs'$. Thus, it suffices to show that $\texttt{match}\ r\ cs'\ \texttt{matchstar}$ is valuable.

To do so, we use the IH Part 2: because $\texttt{Star}r$ is assumed to be standard, $r$ is standard, and moreover $[] \notin L(r)$. This is exactly the inner IH! Well, almost exactly: the premise that $cs'' \leq cs$ is unnecessary because $cs'' < cs' \leq cs$ by transitivity. So the outer IH Part 2 gives the result.

Note the similarities between the proof and the program: Just as the code passes `matchstar` as the continuation, the proof uses the inner IH to justify termination of the continuation! □

Whew, hard work. As a corollary, we obtain a more readable statement of termination:

**Theorem 3** (Termination)**.** *For all $r, cs, k$, if $r$ is standard and $k$ is total, then $\mathtt{match}\, r\, cs\, k$ is valuable.*

Now that we have established termination, we should go back and formally check soundness and completeness:

### 4.2.1 Soundness

**Theorem 4** (Soundness)**.** *For all $r, cs, k$, if $r$ is standard, then if $\mathtt{match}\, r\; cs\; k \cong \mathtt{true}$ then there exist $p, s$ such that $p@s \cong cs$ and $p \in L(r)$ and $k\; s \cong \mathtt{true}$.*

*Proof.* The above cases for `One`, `Zero`, `Char`, `Plus`, and `Times` are easily adapted to this theorem statement. In each case, we assume that $r$ is standard, and must prove that the subjects of the recursive calls are standard, to satisfy the premise of the IH. But any subexpression of a standard regular expression is standard, so this is possible.

We show that case for `Star r`:
Outer IH: For all $cs, k$, if $r$ is standard, then if $\mathtt{match}\, r\; cs\; k \cong \mathtt{true}$
then there exist $p, s$ such that $p@s \cong cs$ and $p \in L(r)$ and $k\; s \cong \mathtt{true}$.
Assume $cs$, $k$, that $\mathtt{Star}\, r$ is standard, and that $\mathtt{match}\, r\; cs\; k \cong \mathtt{true}$.
To show: there exist $p, s$ such that $p@s \cong cs$ and $p \in L(\mathtt{Star}\, r)$ and $k\; s \cong \mathtt{true}$.
Observe that $\mathtt{match}\, r\; cs\; k \cong \mathtt{matchstar}\; cs$, so so $\mathtt{matchstar}\; cs \cong \mathtt{true}$.
We prove a lemma about `matchstar`:

> For all $cs'$, if $\mathtt{matchstar}\; cs' \cong \mathtt{true}$, then there exist $p, s$ such that $p@s \cong cs'$ and $p \in L(\mathtt{Star}\, r)$ and $k\; s \cong \mathtt{true}$.

*Proof of lemma.* The proof is by *well-founded induction on cs'*: this means we assume the theorem for all strict suffices of $cs'$, and prove it for $cs'$:

Inner IH: For all $cs'' < cs'$, if $\mathtt{matchstar}\; cs'' \cong \mathtt{true}$,
then there exist $p, s$ such that $p@s \cong cs$ and $p \in L(\mathtt{Star}\, r)$ and $k\; s \cong \mathtt{true}$.
Proof: Assume $cs'$ such that $\mathtt{matchstar}\; cs' \cong \mathtt{true}$.
To show: there exist $p, s$ such that $p@s \cong cs'$ and $p \in L(\mathtt{Star}\, r)$ and $k\; s \cong \mathtt{true}$.
Observe that $\mathtt{matchstar}\; cs' \cong k\; cs'\; \mathtt{orelse}\; \mathtt{match}\; r\; cs'\; \mathtt{matchstar}$, so this evaluates to `true`. By inversion, we have two cases two consider.

In the first, $k\; cs' \cong \mathtt{true}$. In this case, we take $p$ to be $[]$, $s$ to be $cs'$, and observe that $[]@cs' \cong cs'$, $[] \in L(\mathtt{Star}\, r)$, and $k\; cs' \cong \mathtt{true}$.

In the second, $\mathtt{match}\; r\; cs'\; \mathtt{matchstar} \cong \mathtt{true}$. Note that $r$ is standard because $\mathtt{Star}\, r$ was assumed to be. Thus, by the outer IH on $r$, there exist $p_1, s_1$ such that $p_1@s_1 \cong cs'$ and $p_1 \in L(r)$ and $\mathtt{matchstar}\; s_1 \cong \mathtt{true}$.

Because $\mathtt{Star}\, r$ is standard, $[] \notin L(r)$. Thus $p_1$ is not empty, and since $p_1@s_1 \cong cs'$, $s_1$ is a strict suffix of $cs'$. Therefore, we can appeal to the inner IH on the fact that $\mathtt{matchstar}\; s_1 \cong \mathtt{true}$. to conclude that there exist $p_2, s_2$ such that $p_2@s_2 \cong s_1$ and $p_2 \in L(\mathtt{Star}\, r)$ and $k\; s_2 \cong \mathtt{true}$.

Thus, we take $p$ to be $p_1@p_2$ and $s$ to be $s_2$, and observe that $(p_1@p_2)@s_2 \cong cs'$ (using associativity, as in the times case), that $p_1@p_2 \in L(\mathtt{Star}\, r)$ (because $p_1 \in L(r)$ and $p_2 \in L(\mathtt{Star}\, r)$), and that $k\; s_2 \cong \mathtt{true}$ (as a result of the inner IH). □

Because $\mathtt{matchstar}\; cs \cong \mathtt{true}$, the lemma immediately implies what we need to show to finish the case. □

### 4.2.2 Completeness

**Theorem 5** (Completeness). *For all $r, cs, k$, if $r$ is standard and $k$ is total, then if (there exist $p, s$ such that $p@s \cong cs$ and $p \in L(r)$ and $k\ s \cong$ `true`) then `match` $r\ cs\ k \cong$ `true`.*

*Proof.* The theorem statement has not changed very much from above: we have added the preconditions that $r$ be standard and $k$ be total. As we have seen, it is easy to satisfy the obligation that $r$ be standard at each recursive call, because subexpressions of a standard regexp are standard. Similarly, because we have proved termination, we can prove that the continuations passed in in each recursive call remain total. For example, in the times case, we need to show that `fn cs' => match r2 cs' k` is total, under the assumption that `k` is total. This follows from termination of `match`.

Thus, we just show the $r^*$ case:
Outer IH: For all $cs, k$, if $r$ is standard and $k$ is total, then
if there exist $p, s$ such that $p@s \cong cs$ and $p \in L(r)$ and $k\ s \cong$ `true`
then `match` $r\ cs\ k \cong$ `true`.

Assume $cs, k$ and that `Star` $r$ is standard and $k$ is total and
there exist $p, s$ such that $p@s \cong cs$ and $p \in L($`Star` $r)$ and $k\ s \cong$ `true`.

To show: `match` (`Star` $r$) $cs\ k \cong$ `true`.
It suffices to show that `matchstar` $cs \cong$ `true`, which is a consequence of the following lemma:

For all $p, s, cs$, if $p@s \cong cs$ and $p \in L(r^*)$ and $k\ s \cong$ `true` then `matchstar` $cs \cong$ `true`.

*Proof of lemma.* Assume $p, s, cs$ such that $p@s \cong cs$ and $k\ s \cong$ `true`. The proof uses an inner induction on the fact that $p \in L(r^*)$ (recall that this was defined inductively). We have two cases:

In the first case, $p$ is [], so $s$ is $cs$, so $k\ cs \cong$ `true`, and therefore

```
   matchstar cs
== k cs orelse match r cs matchstar
== true
```

In the second case, there exists $p_1@p_2 \cong p$, where $p_1 \in L(r)$, and $p_2 \in L($`Star`$r)$, and we have an inner inductive hypothesis about the fact that $p_2 \in L($`Star`$r)$.

We use the outer IH on $r$ to prove that `match` $r\ cs$`matchstar` $\cong$ `true`, which, because $k$ is total, implies the result.

To use the IH we prove that (1) $r$ is standard: it is, because `Star`$r$ is assumed to be standard. (2) `matchstar` is total: because $k$ is total, this follows from termination. (3) $p_1@(p_2@s) \cong cs$: this is a consequence of associativity. (4) $p_1 \in L(r)$: assumed for this case. (5) `matchstar` $(p_2@s) \cong$ `true`: the inner inductive hypothesis gives the result. □

□

# 5 Introduction to Staging

Suppose you want to write a function to raise a base to a fixed power, because you plan to raise many numbers to the same power. If you were writing them by hand, you would write:

```
val square = fn b => b * b
val cube   = fn b => b * b * b
```

Can we define square and cube using the exponentation function we defined earlier in the semester? Let's curry it:

```
fun exp (e : int) : int -> int = fn b =>
    case e of
        0 => 1
      | _ => b * (exp (e-1) b)
```

Because it is curried, we can *partially apply* exp to an exponent, which generates a function that raises any base to that power.

E.g.

```
val square : int -> int = exp 2
```

However, what is the value of square?

```
    exp 2
|-> fn b =>
    case 2 of
        0 => 1
      | _ => b * (exp (2-1) b)
```

Because functions are values, it's done evaluating. Notice that there is some *interpretive overheaded* in exp 2 that was not present when we defined square directly: the value of exp 2 still has to do the recursion to determine how many multiplications to do, whereas square does not.

**The difference between evaluation and equivalence.** Note that exp 2 $\cong$ fn b => b * b. The reason is that equivalence can proceed into a function, whereas evaluation does not:

Function extensionality: If (for all v, f v $\cong$ g v) then f $\cong$ g.

This reinforces the point that equivalence says nothing about running-time: equivalent expressions behave the same in terms of what they do, but not how long they take to do it.

So to show

```
   fn b =>
    case 2 of
        0 => 1
      | _ => b * (exp (2-1) b)
== fn b => b * b
```

It suffices to show that for all b

```
    case 2 of
        0 => 1
      | _ => b * (exp (2-1) b)
```

```
==  b * exp (2 - 1) b            [step]
==  b * exp 1 b                  [step]
    ...
==  b * b                        [...analogously...]
==  b * b
```

**Staging** To make pancakes, you mix the dry ingredients (flour, sugar, baking powder, salt), then mix the wet ingredients (oil, egg, milk), and then mix the two together. This means that if someone gives you just the dry ingredients, you can do useful work, mixing the dry stuff, before you ever get the wet ingredients. A *multi-staged* function does useful work when applied to only some of its arguments. Applying to these arguments *specializes* a multi-staged function, generating code specific to those arguments. This can improve efficiency when the specialized function is used many times. *Staging* is the programming technique of writing multi-staged functions.

One application of staging is reducing interpretative overhead. We can write a staged exponentiation function that no longer needs to recur on `2` every time it is called. The idea is to delay asking for the base until we have entirely processed the exponent:

```
fun staged_exp (e : int) : int -> int =
    case e of
        0 => (fn _ => 1)
      | _ => let val oneless = staged_exp (e-1)
              in
                 fn b => b * oneless b
             end
```

Then, letting `f` stand for the expression `fn b => b * (oneless b)`.

```
     staged_exp 2
|->* let val oneless = (staged_exp (2-1)) in fn b => b * oneless b end
|->* let val oneless = (let val oneless = staged_exp (1-1) in f end) in f end
|->* let val oneless = (let val oneless = (fn _ => 1) in f end) in f end
|->  let val oneless = (fn b => b * ((fn _ => 1) b)) in f end
|->  fn b => b * ((fn b => b * ((fn _ => 1) b)) b)
```

There is no interpretative overhead left! A smart compiler might optimize this, using contextual equivalence, to

```
fn b => b * b
```

by reducing a known function applied to a variable. Thus, we'd get out exactly the code we wanted, which directly does the specfied number of multiplications.

You can think of the exponent as a program—do $e$ multiplications, and of staging as *compiling* the program to an ML function. The compiled version no longer needs to recursively traverse the input program.

Subtlety: The compiler is free to apply any equivalences as optimizations, without changing the meaning of your program, so in principle it could transform the original `exp 2` into this form as well. Why don't you want it to do this? For one, it's good to have a predictable cost model, and letting the compiler do arbitrary things makes it hard to predict performance. Secondly, optimizations that require expanding recursive calls are tricky to apply, because there is a termination worry: when do you stop optimizing? Additionally, there's a tradeoff, because by unrolling the recursion, you're increasing the size of the code. The nice thing about staging is that it lets you express the optimization yourself, modulo some harmless optimizations like applying a function to a variable.

# 6 Kleene Algebra Homomorphisms

There is a nice way to rewrite the regular expression matcher, to draw out a pattern in the structure of the code. By definining a helper function for each case, we see that what we're doing is interpreting the syntax of regular expressions as operations on *matcher*s:

```
fun match (r : regexp) : matcher =
    case r of
        Zero => FAIL
      | One => NULL
      | Char c => LITERALLY c
      | Plus (r1,r2) => match r1 OR match r2
      | Times (r1,r2) => match r1 THEN match r2
      | Star r => REPEATEDLY (match r)
```

This is a *Kleene algebra homomorphism*—"Kleene algebra" is the name for an algebraic structure with plus, times, and star satisfying the properties you expect for regexps. The role of this function is to interpret the syntax of regular expressions as corresponding operations on matchers: `FAIL` corresponds to `Zero`, `OR` to `Plus`, etc.

The hard work gets moved to defining matchers, which is just moving the above code into helper functions:

```
type matcher = char list -> (char list -> bool) -> bool
val FAIL : matcher = fn _ => fn _ => false
val NULL : matcher = fn cs => fn k => k cs
fun LITERALLY (c : char) : matcher =
    fn cs => fn k => (case cs of
                          []  => false
                        | c' :: cs' => c = c' andalso k cs')
infixr 8 OR
infixr 9 THEN
fun m1 OR m2 = fn cs => fn k => m1 cs k orelse m2 cs k
fun m1 THEN m2 = fn cs => fn k => m1 cs (fn cs' => m2 cs' k)
fun REPEATEDLY m = fn cs => fn k =>
    let fun repeat cs' = k cs' orelse m cs' repeat
    in
        repeat cs
    end
```

(Note that these definitions must of course come before the definition of `match` in your SML file.)

There are a couple of advantages of this way of writing the code. First, it makes it clear to the reader that the outer loop is a homomorphism, which helps you understand the code—you know that you can consider each clause independently.

Second, the type `matcher` and the helper functions constitute what is called a *combinator library*—a collection of higher-order functions for solving problems in a particular domain. For example, you could ignore the syntax of `regexp`s and just write your regular expressions down as combinators: e.g. `REPEATEDLY (CHAR #''a'' THEN CHAR #''b'')` for $(ab)^*$. An advantage of the combinators is that they are *open-ended*: you can add new ones after the fact and mix them in with the old ones. On the other hand, the advantage of the syntax of `regexp`s is that they are closed-ended: if you want to define transformations on `regexp`s, like standardization, it is necessary to know what all of them are.

A third advantage of this code is that it is *staged.* For example, suppose you are going to match many strings against the same regular expression. It would be nice to process the regular expression once, and generate the code that you would have written if you were matching against a specific regular expression by hand, so that there is no interpretive overhead left.

For example, for `Star(Char #''a'')`, you might write

```
fun matchastar cs k =
  let fun matchstar cs = k cs orelse
    (case cs of [] => false
             | c' :: cs' => #''a'' = c' andalso matchstar cs')
  in
    matchstar cs
  end
```

to match any number of occurrences of $a$.

However, if you try taking the first version of `match` and applying it only to a `regexp` $r$, it gets stuck:

```
    match (Star(Char #''a''))
|-> fn cs => fn k => case (Star(Char #''a'')) of ...
```

Because we do not continue computing under functions, we do not reduce the `case` until a string is given, even though we have enough information to do so right here.

However, the combinator version of the matcher is well-staged: it processes the regexp entirely before a string is given. This is obvious from the code: in ML, you evaluate the arguments to a function before applying a function, and `r` only gets used in recursive calls to `match`, which appear in the arguments to functions like `OR` and `THEN`.

For example:

```
    match (Star(Char #"a"))
|-> REPEATEDLY (match (Char #"a"))
|-> REPEATEDLY (LITERALLY #"a")
|-> REPEATEDLY (fn cs => fn k =>
       (case cs of [] => false
                 | c' :: cs' => #"a" = c' andalso k cs'))
|-> fn cs => fn k =>
    let fun repeat cs' = k cs' orelse
      (fn cs => fn k =>
       (case cs of [] => false
                 | c' :: cs' => #"a" = c' andalso k cs'))
      cs' repeat
    in
        repeat cs
    end
==
   fn cs => fn k =>
    let fun repeat cs' = k cs' orelse
       (case cs' of [] => false
                  | c' :: cs' => #"a" = c' andalso repeat cs')
    in
        repeat cs
    end
```

The final step involves applying a known function to variables, which is an optimization that is safe for your compiler to perform, though it is not required. If it does, we get exactly the code we would have written by hand! We can specialize the general solution to specific instances without any performance cost whatsoever.

Note that staging depends crucially on Currying: if you write a multi-argument function using tuples, such as

```
fun match (r : regexp, cs : char list, k : char list -> bool) = ...
```

there is no room between the arguments to do any work. However, if you have "functions that return functions" as a language concept, you can express staging without any specific support from your language.

It takes a little care to make sure that the initial call to `match` from `accepts` maintains staging:

```
 fun accepts (r : regexp) : string -> bool =
     let
         val m = match r
     in
         fn s => m (String.explode s) isnil
     end
```

We need to be sure to Curry the function, and to evaluate `match r` before abstracting over the string `s`—otherwise we would not be exploiting the staging of `match`.