15-150

Principles of Functional Programming

Slides for Lecture 14

Regular Expressions

March 12, 2024

Michael Erdmann

Lessons:

- Regular Expressions
- Regular Languages
- Matcher
- Correctness
  - Proof-Directed Debugging
  - Termination
  - Soundness and Completeness

Language Hierarchy

Class of Languages	Recognizers	Applications
Unrestricted	Turing Machines	General Computation
Context-Sensitive	Linear-bounded automata	Some simple type-checking
Context-Free	Nondeterministic automata with one stack	Syntax checking
Regular	Finite Automata	Tokenization

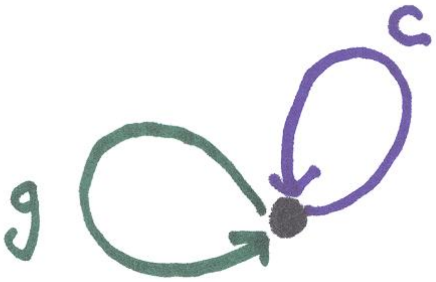
An example: Excursions from home

(home)

An example: Excursions from home

"**c**" means "go to CMU, then go home"

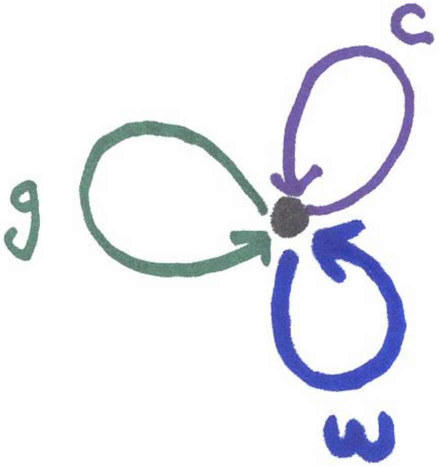
An example: Excursions from home



“**c**” means “go to CMU, then go home”

“**g**” means “get groceries, then go home”

An example: Excursions from home

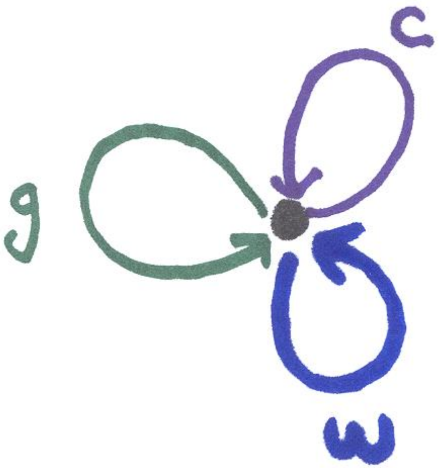


"**c**" means "go to CMU, then go home"

"**g**" means "get groceries, then go home"

"***w***" means "go for a walk, then home"

An example: Excursions from home



"**c**" means "go to CMU, then go home"

"**g**" means "get groceries, then go home"

"***w***" means "go for a walk, then home"

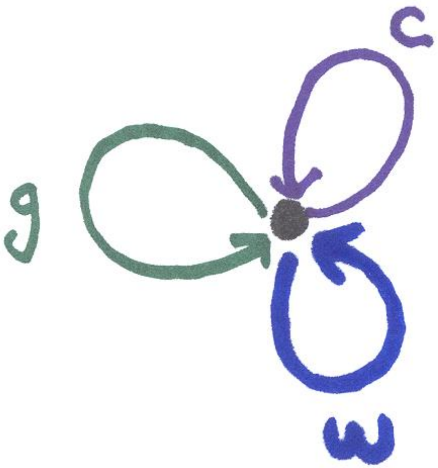
Description of excursions in a given week:

c (go to CMU once) **cc** (go to CMU twice) **ccc** (go to CMU 3 times)

c* (go to CMU zero or more times)

cgc (go to CMU, then get groceries, then go to CMU)

An example: Excursions from home



"**c**" means "go to CMU, then go home"

"**g**" means "get groceries, then go home"

"***w***" means "go for a walk, then home"

Description of excursions in a given week:

g + *w*	(get groceries **OR** go for a walk)
(g + *w*)*	(zero or more times do one of the following: get groceries **OR** go for a walk)
(g + *w*)*c	(zero or more times do one of the following: get groceries **OR** go for a walk; *after that* go to CMU once)

Notation and Definitions

Σ is an *alphabet* of *characters*. (nonempty, finite)

For example, $\Sigma = \{a, b\}$.

(Using SML , **`#"a" : char`**.)

Σ* means the set of all finite-length strings over alphabet **Σ**, i.e., with characters in **Σ**.

For example, aabba is in {a,b}*.

(Using SML , **`"aabba" : string`**.)

ε is the *empty string*, containing no characters.

$\boxed{\varepsilon \text{ is in } \Sigma^*.}$ (Using SML , **`"" : string`**.)

Notation and Definitions

A *language* over $\mathbf{\Sigma}$ is a subset of $\mathbf{\Sigma^*}$.

(In other words, a language is a set of finite-length strings with characters in $\mathbf{\Sigma}$.
A language may contain infinitely many strings.)

We are here interested in a particular class of languages called ***regular languages***. The languages may have infinite size, but we will describe them via a finite representation called ***regular expressions***, much like in the excursion example.

Regular Expressions

Assume we have been given some alphabet Σ.

A *regular expression* over Σ is any of the following:

Regular Expressions

Assume we have been given some alphabet Σ.

A *regular expression* over Σ is any of the following:

a for every character $\mathbf{a} \in \Sigma$,

set symbol meaning "is in"

(don't confuse with the empty string ε)

Regular Expressions

Assume we have been given some alphabet Σ.

A *regular expression* over Σ is any of the following:

a	for every character $\mathbf{a} \in \Sigma$,
0	(a special symbol),

Regular Expressions

Assume we have been given some alphabet Σ.

A *regular expression* over Σ is any of the following:

a	for every character $\mathbf{a} \in \Sigma$,
0	(a special symbol),
1	(another special symbol),

Regular Expressions

Assume we have been given some alphabet Σ.

A *regular expression* over Σ is any of the following:

a	for every character $\mathbf{a} \in \Sigma$,
0	(a special symbol),
1	(another special symbol),
$\mathbf{r_1 + r_2}$	with $\mathbf{r_1}$ and $\mathbf{r_2}$ regular expressions (called *alternation*),

Regular Expressions

Assume we have been given some alphabet Σ.

A *regular expression* over Σ is any of the following:

a	for every character $\mathbf{a} \in \Sigma$,
0	(a special symbol),
1	(another special symbol),
$\mathbf{r_1 + r_2}$	with $\mathbf{r_1}$ and $\mathbf{r_2}$ regular expressions (called *alternation*),
$\mathbf{r_1 r_2}$	with $\mathbf{r_1}$ and $\mathbf{r_2}$ regular expressions (called *concatenation*),

Regular Expressions

Assume we have been given some alphabet Σ.

A *regular expression* over Σ is any of the following:

a	for every character **a** $\in \Sigma$,
0	(a special symbol),
1	(another special symbol),
$\mathbf{r_1 + r_2}$	with $\mathbf{r_1}$ and $\mathbf{r_2}$ regular expressions (called *alternation*),
$\mathbf{r_1 r_2}$	with $\mathbf{r_1}$ and $\mathbf{r_2}$ regular expressions (called *concatenation*),
r*	with **r** a regular expression (called *Kleene star*).

Regular Expressions

Assume we have been given some alphabet Σ.

A *regular expression* over Σ is any of the following:

(And use parentheses as needed.)

$\mathbf{r_1 + r_2}$	with $\mathbf{r_1}$ and $\mathbf{r_2}$ regular expressions (called *alternation*),
$\mathbf{r_1 r_2}$	with $\mathbf{r_1}$ and $\mathbf{r_2}$ regular expressions (called *concatenation*),
$\mathbf{r^*}$	with $\mathbf{r}$ a regular expression (called *Kleene star*).

Regular Languages

Given regular expression **r** we define language **L(r)**:

L(a) = {a} (singleton set) for every character $\mathbf{a} \in \Sigma$,

L(0) = { } (the empty language, no strings),

L(1) = {ε} (the language consisting of the empty string),

$\mathbf{L(r_1 + r_2) = \{ s \mid s \in L(r_1)}$ or $\mathbf{s \in L(r_2) \}}$ (not exclusive),

$\mathbf{L(r_1 r_2) = \{ s_1 s_2 \mid s_1 \in L(r_1)}$ and $\mathbf{s_2 \in L(r_2) \}}$,

$\mathbf{L(r^*) = \{ s \mid s = s_1 s_2 \cdots s_n}$, some $\mathbf{n} \geq 0$, with each $\mathbf{s_i \in L(r) \}}$

(here we mean **s = ε** when n=0).

So: $\boldsymbol{\varepsilon} \in \mathbf{L(r^*)}$ for all regular expressions **r**.

Regular Languages

Let **Σ** be a given alphabet and **L** a subset of **Σ***.

We say that language **L** is *regular* if **L = L(r)** for some regular expression **r**.

(Fact: The class of regular languages over **Σ** is the minimal class containing the empty set and all singleton subsets of **Σ**, and that is closed under union, concatenation, and Kleene star.)

(The class is also closed under complement:

L is regular iff **Σ* \ L** is regular.)

Examples (assume $\Sigma = \{a, b\}$)

L(a) = {a} (singleton set consisting of the string **a**)

L(aa) = {aa} (singleton set consisting of the string **aa**)

L((a + b)*) = Σ* (all finite-length strings with **a**s and **b**s)

L((a + b)*aa(a + b)*) = all strings in **Σ*** containing at least two consecutive **a**s.

L((a + 1)(b + ba)*) = **?????**

Examples (assume $\Sigma = \{a, b\}$)

L(a) = {a} (singleton set consisting of the string **a**)

L(aa) = {aa} (singleton set consisting of the string **aa**)

L((a + b)*) = Σ* (all finite-length strings with **a**s and **b**s)

L((a + b)*aa(a + b)*) = all strings in **Σ*** containing at least two consecutive **a**s.

L((a + 1)(b + ba)*) = all strings in **Σ*** that do *not* contain two consecutive **a**s.

Examples (assume $\Sigma = \{a, b\}$)

Comment: Different regular expressions can give rise to the same regular language.

For instance:

L(ab + b*ab)
= L((1 + b*)ab)
= L((1 + bb*)ab)
= L(b*ab)
= L(b*ab + 0)
= all strings in **Σ*** consisting of zero or more **b**s followed by **ab** (and nothing thereafter).

Examples (assume $\Sigma = \{a, b\}$)

Comment: Different regular expressions can give rise to the same regular language.

For instance:

In particular, for any reg exp **r**:

$$L(r^*) = L(1 + rr^*)$$

$$
\begin{aligned}
& L(ab + b^*ab) \\
= {} & L((1 + b^*)ab) \\
= {} & L((1 + bb^*)ab) \\
= {} & L(b^*ab) \\
= {} & L(b^*ab + 0)
\end{aligned}
$$

= all strings in Σ^* consisting of zero or more **b**s followed by **ab** (and nothing thereafter).

An Acceptor

We would like to implement a function that decides whether a given string **s** is in the language **L(r)** of a given regular expression **r**.

```
(* accept : regexp -> string -> bool
   REQUIRES: true (may change this later).
   ENSURES: (accept r s) returns true if s ∈ L(r);
            (accept r s) returns false, otherwise.
*)
```

Think of `accept` as a simple parser/compiler.

(Still need to define the `regexp` type.)

Matching

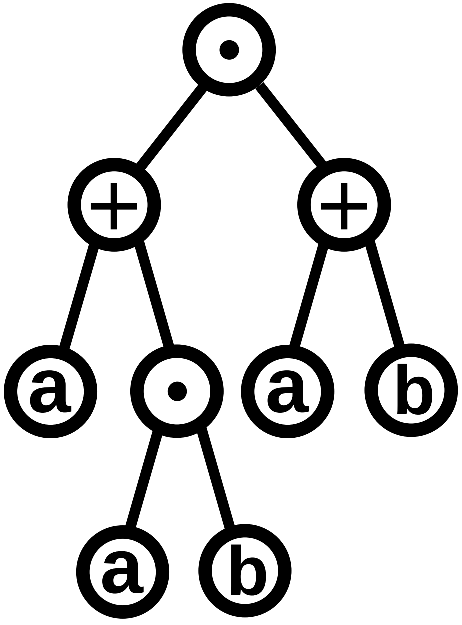
Suppose **r = (a + ab)(a + b).**

Then **L(r) = {aa, ab, aba, abb}.**

How does the acceptor recognize that **aba ∈ L(r)** ?

By backtracking search.

View **r** as a tree.



Use up characters in **aba** matching tree operations determined by **r**.

Matching

(a + ab)(a + b)

a **ba**

Suppose **r = (a + ab)(a + b).**

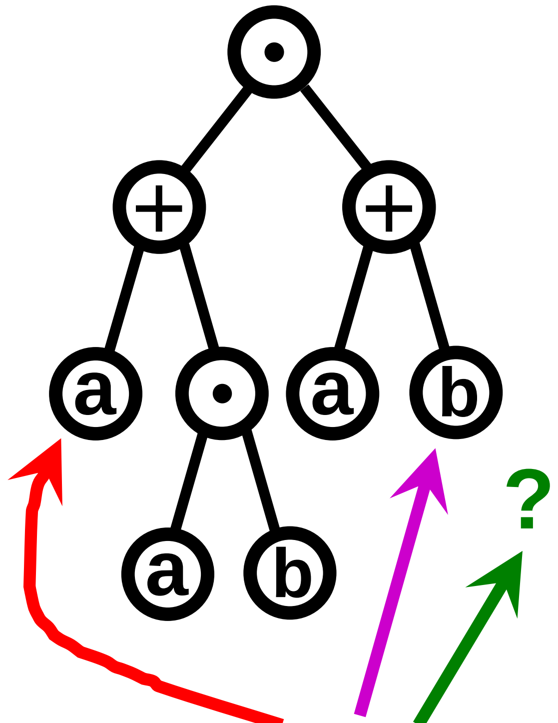
Then **L(r) = {aa, ab, aba, abb}.**

How does the acceptor recognize that **aba** $\in$ **L(r)** ?

By backtracking search.

View **r** as a tree.

Use up characters in **aba** matching tree operations determined by **r**.



First split of **aba** as **a ba** fails on last character.

Matching

(a + ab)(a + b)

ab **a**

Suppose **r = (a + ab)(a + b).**

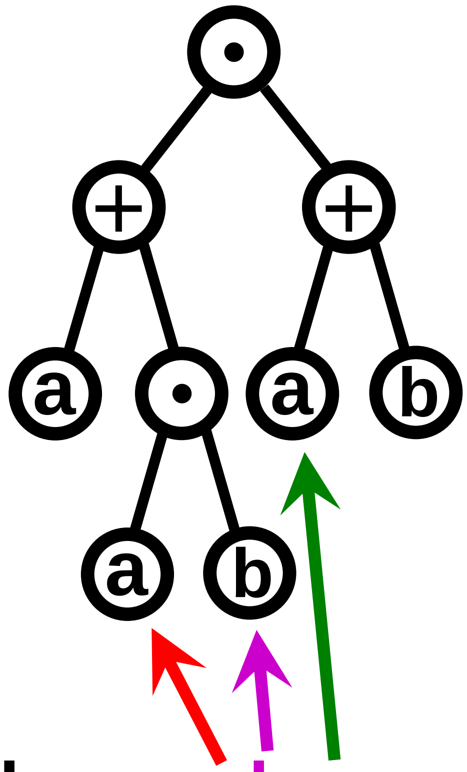
Then **L(r) = {aa, ab, aba, abb}.**

How does the acceptor recognize that **aba ∈ L(r)** ?

By backtracking search.

View **r** as a tree.

Use up characters in **aba** matching tree operations determined by **r**.



Second split of **aba** as ab a succeeds.

Matching

Suppose **r = (a + ab)(a + b).**

Then **L(r) = {aa, ab, aba, abb}.**

How does the acceptor recognize that **aba ∈ L(r)** ?

By backtracking search.

Tonight, do an evaluation trace on this example of the code we are about to write.

(Check yourself using today's lecture page.)

A Matcher

We will implement the backtracking search using a Boolean-specific continuation.

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

   REQUIRES: k is total (aside: weaker condition
                      simplifies termination proof).

   ENSURES: (match r cs k) returns true if
                 cs can be split as cs≅p@s, with
                 p representing a string in L(r)
                 and k(s) evaluating to true;
          (match r cs k) returns false, otherwise.
*)
```

A Matcher

We will implement the backtracking search using a Boolean-specific continuation.

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

   REQUIRES: k is total.

   ENSURES: (match r cs k) returns true if
                 cs can be split as cs≅p@s, with
                 p representing a string in L(r)
                 and k(s) evaluating to true;
            (match r cs k) returns false, otherwise.
*)
```

We use character lists instead of strings here for simplicity.
In discussions/proofs we sometimes treat them as identical.

Acceptor Based on Matcher Specs

```
(* match : regexp -> char list ->
                    (char list -> bool) -> bool
   REQUIRES: k is total.
   ENSURES: (match r cs k) ≅ true if
               cs ≅ p@s, with p ∈ L(r) & k(s) ≅ true;
            (match r cs k) ≅ false, otherwise.

   accept : regexp -> string -> bool
   REQUIRES: true
   ENSURES: (accept r s) ≅ true if s ∈ L(r);
            (accept r s) ≅ false otherwise.
*)

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

Acceptor Based on Matcher Specs

```
(* match : regexp -> char list ->
                    (char list -> bool) -> bool
   REQUIRES: k is total.
   ENSURES: (match r cs k) ≅ true if
                 cs ≅ p@s, with p ∈ L(r) & k(s) ≅ true;
            (match r cs k) ≅ false, otherwise.

   accept : regexp -> string -> bool
   REQUIRES: true
   ENSURES: (accept r s) ≅ true if s ∈ L(r);
            (accept r s) ≅ false otherwise.
*)
```

turns a string into a char list

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

`List.null : 'a list -> bool` decides whether a list is empty.

Implementation

We will define a datatype that mirrors the mathematical definition of regular expressions.

We will implement a matcher that mirrors the definition of a regular expression's language.

Implementation

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

Implementation

```
fun match
```

Implementation

```
fun match (Char a) cs k =
```

Implementation

```
fun match (Char a) cs k =
      (case cs of
           [] =>
         | c::cs' =>                                  )
```

Implementation

```
fun match (Char a) cs k =
      (case cs of
           [] => ?????
         | c::cs' =>                                    )
```

Recall:

$$\texttt{(match r cs k)} \cong \texttt{true}$$

if $\texttt{cs} \cong \texttt{p@s}$, with $\texttt{p} \in \texttt{L(r)}$ & $\texttt{k(s)} \cong \texttt{true}$

L(a) = {a}

Implementation

```
fun match (Char a) cs k =
      (case cs of
          [] => false
        | c::cs' =>          ?????          )
```

Recall:

(match r cs k) ≅ true

if cs ≅ p@s, with p ∈ L(r) & k(s) ≅ true

L(a) = {a}

Implementation

```
fun match (Char a) cs k =
     (case cs of
          [] => false
        | c::cs' => (a=c) andalso (k cs'))
```

Implementation

```
fun match (Char a) cs k =
      (case cs of
          [] => false
        | c::cs' => (a=c) andalso (k cs'))
  | match Zero _ _ = ?????
```

Recall:

$(\texttt{match r cs k}) \cong \texttt{true}$

if $\texttt{cs} \cong \texttt{p@s}$, with $\texttt{p} \in \texttt{L(r)}$ & $\texttt{k(s)} \cong \texttt{true}$

L(0) = { }

Implementation

```
fun match (Char a) cs k =
      (case cs of
           [] => false
         | c::cs' => (a=c) andalso (k cs'))
  | match Zero _ _ = false
```

Implementation

```
fun match (Char a) cs k =
      (case cs of
           [] => false
         | c::cs' => (a=c) andalso (k cs'))
  | match Zero _ _ = false
  | match One cs k = ?????
```

Recall:

(match r cs k) ≅ true

if cs ≅ p@s, with p ∈ L(r) & k(s) ≅ true

L(1) = {ε}

Implementation

```
fun match (Char a) cs k =
      (case cs of
          [] => false
        | c::cs' => (a=c) andalso (k cs'))
  | match Zero _ _ = false
  | match One cs k = k cs
```

Implementation

```
fun match (Char a) cs k =
      (case cs of
          [] => false
        | c::cs' => (a=c) andalso (k cs'))
  | match Zero _ _ = false
  | match One cs k = k cs
  | match (Plus(r1,r2)) cs k =
```

Implementation

`(match r cs k)` ≅ `true`

if `cs ≅ p@s`, with `p ∈ L(r)` & `k(s) ≅ true`

$L(r_1 + r_2) = \{ s \mid s \in L(r_1) \text{ or } s \in L(r_2) \}$

```
| match (Plus(r₁,r₂)) cs k =
    (match r₁ cs k)  ?????
```

Implementation

```
fun match (Char a) cs k =
      (case cs of
           [] => false
         | c::cs' => (a=c) andalso (k cs'))
  | match Zero _ _ = false
  | match One cs k = k cs
  | match (Plus(r1,r2)) cs k =
      (match r1 cs k) orelse (match r2 cs k)
```

Implementation

```
fun match (Char a) cs k =
     (case cs of
          [] => false
        | c::cs' => (a=c) andalso (k cs'))
  | match Zero _ _ = false
  | match One cs k = k cs
  | match (Plus(r1,r2)) cs k =
     (match r1 cs k) orelse (match r2 cs k)
  | match (Times(r1,r2)) cs k =
```

Implementation

`(match r cs k) ≅ true`

if `cs ≅ p@s,` with `p ∈ L(r)` & `k(s) ≅ true`

$$L(r_1 r_2) = \{\, s_1 s_2 \mid s_1 \in L(r_1) \text{ and } s_2 \in L(r_2) \,\}$$

```
| match (Times(r₁,r₂)) cs k =
   match r₁ cs
```

?????

Implementation

```
(match r cs k) ≅ true
```

if `cs ≅ p@s`, with `p ∈ L(r)` & `k(s) ≅ true`

$$L(r_1 r_2) = \{\, s_1 s_2 \mid s_1 \in L(r_1) \text{ and } s_2 \in L(r_2) \,\}$$

```
| match (Times(r1,r2)) cs k =
   match r1 cs (fn cs' =>  ?????      )
```

Implementation

```
fun match (Char a) cs k =
      (case cs of
           [] => false
         | c::cs' => (a=c) andalso (k cs'))
  | match Zero _ _ = false
  | match One cs k = k cs
  | match (Plus(r1,r2)) cs k =
      (match r1 cs k) orelse (match r2 cs k)
  | match (Times(r1,r2)) cs k =
      match r1 cs (fn cs' => match r2 cs' k)
```

Implementation – Star clause

```
| match (Star r) cs k =
```

Implementation – Star clause

```
| match (Star r) cs k =
```

Recall: **L(r*) = L(1 + rr*)**

We could make calls to previous clauses, but let's implement this equation directly.

Implementation – Star clause

```
| match (Star r) cs k =
   k cs    orelse
```

Recall: **L(r*) = L(1 + rr*)**

We could make calls to previous clauses, but let's implement this equation directly.

Implementation – Star clause

```
| match (Star r) cs k =
    k cs    orelse
    match r cs (fn cs' =>
                                          )
```

Recall: **$L(r^*) = L(1 + rr^*)$**

We could make calls to previous clauses, but let's implement this equation directly.

Implementation – Star clause

```
| match (Star r) cs k =
    k cs   orelse
    match r cs (fn cs' =>
                      match (Star r) cs' k)
```

Recall: **$L(r^*) = L(1 + rr^*)$**

We could make calls to previous clauses,
but let's implement this equation directly.

There is a potential bug.

```
| match (Star r) cs k =
  (k cs) orelse (match r cs (fn cs' => match (Star r) cs' k))
```

Proof-Directed Debugging

```
| match (Star r) cs k =
   (k cs) orelse (match r cs (fn cs' => match (Star r) cs' k))
```

Imagine trying to prove that **`(match (Star r) cs k)`** reduces to a value as part of some larger induction proof that **`match`** always *terminates* (returns a value) when given input satisfying the specs.

In the Induction Hypothesis we may assume that **`(match r cs k)`** reduces to a value whenever **`k`** is total. So we need to establish that **`(fn cs' => match (Star r) cs' k)`** is total. Now we are in a circular argument!

Proof-Directed Debugging

```
| match (Star r) cs k =
   (k cs) orelse (match r cs (fn cs' => match (Star r) cs' k))
```

A possible way out: We don't really need to establish that

```
      (fn cs' => match (Star r) cs' k)
```

is total, merely that it returns values when called on suffixes **`cs'`** of the given **`cs`**. Maybe a second induction on **`cs`** will help.

If we could show that **`cs'`** is a *proper suffix* of **`cs`**, we could perhaps establish eventual termination.

Proof-Directed Debugging

```
                                                  k'
| match (Star r) cs k =
   (k cs) orelse (match r cs (fn cs' => match (Star r) cs' k))
```

A possible way out: We don't really need to establish that

```
(fn cs' => match (Star r) cs' k)
```

is total, merely that it returns values when called on suffixes **cs'** of the given **cs**. Maybe a second induction on **cs** will help.

If we could show that **cs'** is a *proper suffix* of **cs**, we could perhaps establish eventual termination.

ALAS, that need not be true:

```
match (Star One) [#"a"] List.null
```

will loop forever since **List.null [#"a"] ≅ false** and since **match One cs k'** will pass all of **cs** to **k'**.

Proof-Directed Debugging

k'

```
| match (Star r) cs k =
   (k cs) orelse (match r cs (fn cs' => match (Star r) cs' k))
```

This issue arises when the empty string is in **L(r)**.

If we could show that **cs'** is a *proper suffix* of **cs**, we could perhaps establish eventual termination.

ALAS, that need not be true:

```
match (Star One) [#"a"] List.null
```

will loop forever since **List.null [#"a"] ≅ false** and since **match One cs k'** will pass all of **cs** to **k'**.

Two possible fixes to avoid infinite loops

1. Change the specs:
   - Require regular expressions to be in *standard form* (definition shortly).

2. Change the code:
   - Explicitly check that `cs'` is a proper suffix of `cs`.

Two possible fixes to avoid infinite loops

1. Change the specs

Definition:

A regular expression **`r`** is in *standard form* iff for any subexpression **`Star(r')`** of **`r`**, **L(`r'`)** does *not* contain the empty string **ε**.

Fact: It is possible to convert any regular expression **`r`** into a regular expression **`q`** that is in standard form such that **L(`r`) = L(`q`)**.

Consequently, if we **`REQUIRE`** regular expressions to be in standard form we avoid infinite loops without losing any regular languages.

(Preprocess **`r`** into standard form, then call **`match`**.)

Two possible fixes to avoid infinite loops

2. Change the code

```
| match (Star r) cs k =
    k cs   orelse
    match r cs (fn cs' =>
                    properSuffix (cs',cs)
                          andalso
                    match (Star r) cs' k)
```

Two possible fixes to avoid infinite loops

2. Change the code

```
| match (Star r) cs k =
    k cs   orelse
    match r cs (fn cs' =>
                properSuffix (cs',cs)
                      andalso
                match (Star r) cs' k)
```

This is new.

The code checks that `cs'` is a proper suffix of `cs`.

Sketch of a Proof of Correctness

1. <u>Prove Termination</u>

   Show that **`(match r cs k)`** returns a value
   for all arguments **`r`**, **`cs`**, **`k`** satisfying **`REQUIRES`** specs.
   (This proof is surprisingly difficult. We assume it here.)

2. <u>Prove Soundness and Completeness</u>

   Given termination, we can simplify the **`ENSURES`** specs
   in a convenient way, then perform structural induction.
   (We will write out one of the recursive cases here.)

Soundness & Completeness, Assuming Termination

Here are the given **`ENSURES`** specs for **`match`**:

```
(match r cs k) ≅ true if cs≅p@s,
                         with p∈L(r) and k(s)≅true;
(match r cs k) ≅ false, otherwise.
```

Given termination, we can rephrase the specs as:

```
(match r cs k) ≅ true if and only if there exist p and s
     such that cs≅p@s, p∈L(r), and k(s)≅true.
```

That is the theorem we must prove.

The "if" part is sometimes called "completeness".

The "only if" part is sometimes called "soundness".

Theorem

For all values

r : regexp, **cs : char list**, **k : char list -> bool**, with **k** total,

(match r cs k) ≅ true

if and only if

there exist **p** and **s** such that

cs ≅ p@s, **p** ∈ **L(r)**, and **k(s) ≅ true**.

Theorem

For all values

r : regexp, **cs : char list**, **k : char list -> bool**, with **k** total,

$$(\texttt{match r cs k}) \cong \texttt{true}$$

if and only if

there exist **p** and **s** such that

$\texttt{cs} \cong \texttt{p@s}$, $\texttt{p} \in L(\texttt{r})$, and $\texttt{k(s)} \cong \texttt{true}$.

(We are assuming termination as a lemma.)

Theorem

For all values

r : **regexp**, **cs** : **char list**, **k** : **char list -> bool**, with **k** total,

(match r cs k) ≅ **true**

if and only if

there exist **p** and **s** such that

cs ≅ **p@s**, **p** ∈ **L(r)**, and **k(s)** ≅ **true**.

(We are assuming termination as a lemma.)

Proof

By structural induction on **r**.

Theorem For all values

r : regexp, **cs : char list**, **k : char list -> bool**, with **k** total,

(match r cs k) ≅ true

if and only if

there exist **p** and **s** such that

cs ≅ p@s, **p ∈ L(r)**, and **k(s) ≅ true**.

(We are assuming termination as a lemma.)

Proof By structural induction on **r**.

Base Cases: **Zero**, **One**, **Char(a)** for every **a:char**.

Theorem For all values

r : **regexp**, **cs** : **char list**, **k** : **char list -> bool**, with **k** total,

(match r cs k) $\cong$ **true**

if and only if

there exist **p** and **s** such that

cs $\cong$ **p@s**, **p** $\in$ **L(r)**, and **k(s)** $\cong$ **true**.

(We are assuming termination as a lemma.)

Proof By structural induction on **r**.

Base Cases: **Zero**, **One**, **Char(a)** for every **a:char**.

Inductive Cases:

Plus(r_1,r_2), **Times**(r_1,r_2), **Star(r)**.

Theorem For all values

`r : regexp`, `cs : char list`, `k : char list -> bool`, with `k` total,

`(match r cs k)` $\cong$ `true`

if and only if

there exist `p` and `s` such that

`cs` $\cong$ `p@s`, `p` $\in$ L(`r`), and `k(s)` $\cong$ `true`.

(We are assuming termination as a lemma.)

Proof By structural induction on `r`.

Base Cases: `Zero`, `One`, `Char(a)` for every `a:char`.

Inductive Cases:

`Plus(`r_1`,`r_2`)`, `Times(`r_1`,`r_2`)`, `Star(r)`.

We will discuss only the `Plus` case here, as an example.

(See also today's online notes, including another proof technique.)

Inductive Case `r = Plus(r`$_1$`,r`$_2$`)`, for some `r`$_1$`,r`$_2$:

IH: For `i=1,2` and for all values `cs` & `k`, with `k` total,

`(match r`$_i$` cs k)` $\cong$ `true` iff there exist `p`&`s` such that `cs` $\cong$ `p@s`, `p` $\in$ L(`r`$_i$), & `k(s)` $\cong$ `true`.

NTS: For all values `cs` & `k`, with `k` total,

`(match (Plus(r`$_1$`,r`$_2$`)) cs k)` $\cong$ `true` iff there exist `p`&`s` such that `cs` $\cong$ `p@s`, `p` $\in$ L(`Plus(r`$_1$`,r`$_2$`)`), & `k(s)` $\cong$ `true`.

(We will prove the two parts of the "iff" separately.)

I. Suppose `(match (Plus(`r_1`,`r_2`)) cs k)` $\cong$ `true`.

NTS: There exist `p`&`s` such that `cs` $\cong$ `p@s`,
`p` $\in$ L(`Plus(`r_1`,`r_2`)`), & `k(s)` $\cong$ `true`.

Showing:

		`true`
[assumption]	$\cong$	`(match (Plus(`r_1`,`r_2`)) cs k)`
[`Plus`]	$\cong$	`(match` r_1 `cs k) orelse (match` r_2 `cs k)`

∴ One or both of the arguments to `orelse` must be `true`.
Let us suppose it is the first argument (proof similar for second).
So `(match` r_1 `cs k)` $\cong$ `true`.
By IH for r_1,
there exist `p`&`s` s.t. `cs` $\cong$ `p@s`, `p` $\in$ L(r_1), & `k(s)` $\cong$ `true` .
Then also `p` $\in$ L(`Plus(`r_1`,`r_2`)`), by language definition for `Plus`.

That finishes this part of the proof (soundness).

II. Suppose there exist `p`&`s` such that $\texttt{cs} \cong \texttt{p@s}$, $\texttt{p} \in L(\texttt{Plus}(\texttt{r}_1,\texttt{r}_2))$, & $\texttt{k(s)} \cong \texttt{true}$.

NTS: $\texttt{(match (Plus(}\texttt{r}_1\texttt{,}\texttt{r}_2\texttt{)) cs k)} \cong \texttt{true}$.

Showing:

$$
\begin{array}{lll}
 & & \texttt{(match (Plus(}\texttt{r}_1\texttt{,}\texttt{r}_2\texttt{)) cs k)} \\
[\texttt{Plus}] & \cong & \texttt{(match } \texttt{r}_1 \texttt{ cs k) orelse (match } \texttt{r}_2 \texttt{ cs k)} \\
\text{[see below]} & \cong & \texttt{true}
\end{array}
$$

By supposition, there exist `p`&`s` such that $\texttt{cs} \cong \texttt{p@s}$, $\texttt{p} \in L(\texttt{Plus}(\texttt{r}_1,\texttt{r}_2))$, & $\texttt{k(s)} \cong \texttt{true}$. By the language definition for `Plus`, $\texttt{p} \in L(\texttt{r}_1)$ and/or $\texttt{p} \in L(\texttt{r}_2)$.

If $\texttt{p} \in L(\texttt{r}_1)$, then $\texttt{(match } \texttt{r}_1 \texttt{ cs k)} \cong \texttt{true}$ by IH for $\texttt{r}_1$.

Otherwise, $\texttt{(match } \texttt{r}_1 \texttt{ cs k)} \cong \texttt{false}$ by termination, $\texttt{p} \in L(\texttt{r}_2)$, and $\texttt{(match } \texttt{r}_2 \texttt{ cs k)} \cong \texttt{true}$ by IH for $\texttt{r}_2$.

That finishes this part of the proof (completeness), and so the `Plus` case.

That is all.

Please have a good lab.

See you Thursday.

We will discuss another matcher, inspired by staging and combinators.