15-150

Principles of Functional Programming

Lecture 15

March 14, 2024

Michael Erdmannn

Regular Expressions using Combinators & Staging

Recall from last time:

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

Recall from last time:

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

   REQUIRES: k is total;
             perhaps also: r is in standard form.

   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.
*)
```

Recall from last time:

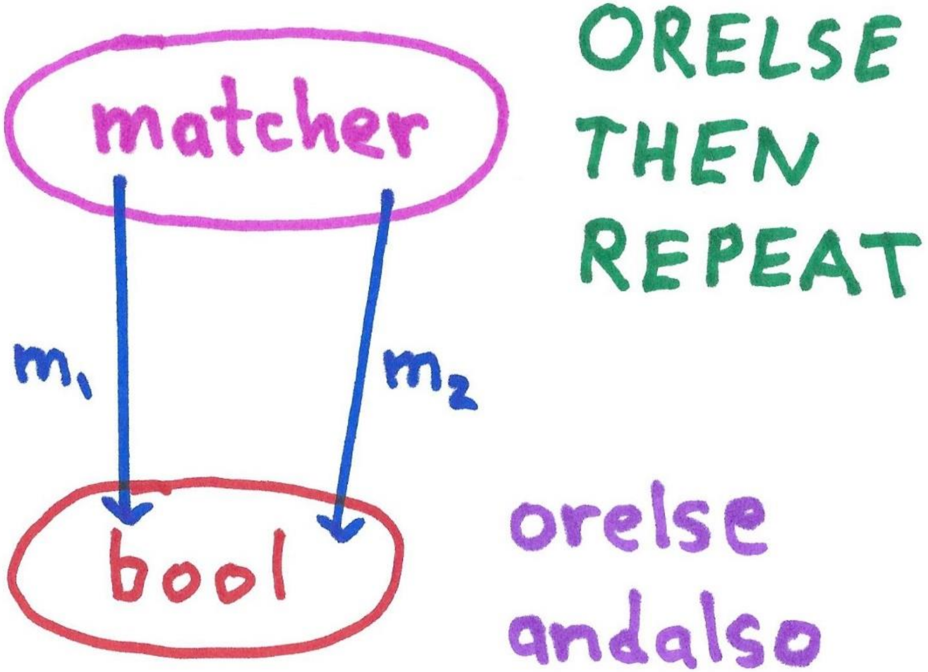
```
(* accept : regexp -> string -> bool
   REQUIRES: perhaps: r is in standard form.
   ENSURES: (accept r s) returns true if s ∈ L(r);
            (accept r s) returns false, otherwise.
*)
```

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

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

Today, we will re-implement the regular expression matcher using combinators. Doing so will disentangle the regular expression semantics from I/O (strings and continuations) by providing some staging.



m_i is a matcher for a particular regular expression r_i.

m_1 ORELSE m_2 is a matcher for $r_1 + r_2$.

$$\text{match} : \text{regexp} \to \overset{\text{conceptual}}{\underset{\text{split}}{\{}}\ \underbrace{\text{char list} \to (\text{char list} \to \text{bool}) \to \text{bool}}_{\text{matcher}}$$

$$\text{regexp} \to \text{matcher}$$

type matcher = char list → (char list → bool) → bool

Code Outline

Continuation Base Cases

ACCEPT
REJECT

These are matchers

Input Base Case

CHECK_FOR

creates a matcher from a character

Combinators

ORELSE
THEN
REPEAT

combine matchers into new matcher

Overall matcher

match
accept

```
val REJECT : matcher = fn _ => fn _ => false
val ACCEPT : matcher = fn cs => fn k => k(cs)

val CHECK_FOR (a : char) : matcher =
      fn []     => REJECT []
       | c::cs  => if a = c
                     then ACCEPT cs
                     else REJECT cs
```

Precedences of predefined infix operators

/ * mod div	7
+ - ^	6
:: @	5
= <> < > <= >=	4
:= o	3

:: & @ are right-associative.

The others are left-associative.

```
infixr 8 ORELSE
infixr 9 THEN

fun (m1 ORELSE m2) cs k = m1 cs k orelse m2 cs k

fun (m1 THEN m2) cs k = m1 cs (fn cs' => m2 cs' k)
```

If regular expressions are in standard form

```
fun REPEAT m cs k =
    let
      fun mstar cs' =
            k cs' orelse m cs' mstar
    in
          mstar cs
    end
```

More generally

```
fun  REPEAT  m  cs  k  =
      let
        fun  mstar  cs'  =
              k cs'  orelse
              m  cs'  (fn  cs'' =>
                            properSuffix (cs'', cs')
                                 andalso
                            mstar  cs'')
      in
         mstar  cs
      end
```

```
fun match (Char(a): regexp): matcher = CHECK_FOR a
  | match Zero              =  REJECT
  | match One               =  ACCEPT
  | match (Plus(r1,r2))     =  match r1  ORELSE  match r2
  | match (Times(r1,r2))    =  match r1  THEN    match r2
  | match (Star(r))         =  REPEAT(match r)
```

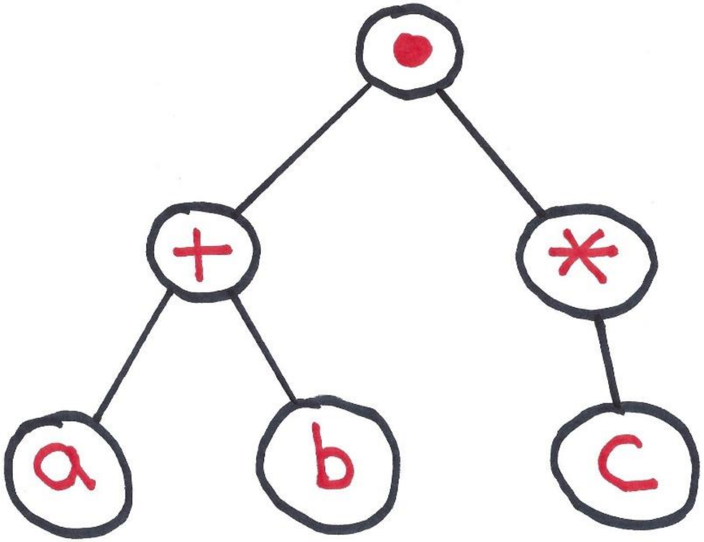
```
fun match (Char(a): regexp): matcher = CHECK_FOR a
  | match Zero             = REJECT
  | match One              = ACCEPT
  | match (Plus(r1,r2))    = match r1  ORELSE  match r2
  | match (Times(r1,r2))   = match r1  THEN  match r2
  | match (Star(r))        = REPEAT(match r)
```

Where is the staging?

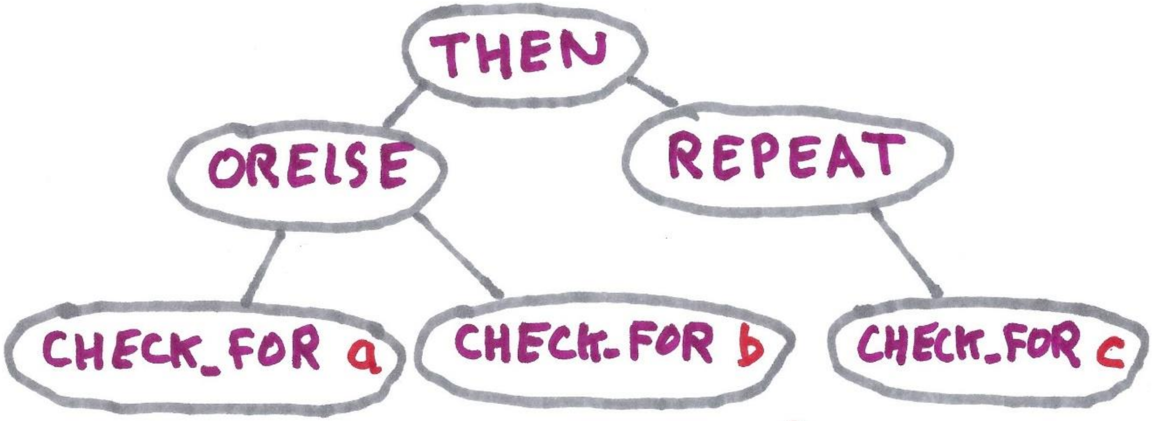
In the deconstruction of the regular expression.

(This now occurs before any string or continuation arguments are received.)

If r is $(a+b)c^*$, then as a tree r is



(match r) will effectively evaluate



producing a matcher for r from the character matchers for a, b, c.

This happens *before* any character input or continuations are specified.

We can now stage accept:

```
fun accept (r : regexp) : string → bool =
    let
      val m = match r
    in
      fn s ⇒ m (String.explode s) List.null
    end
```

Previously, evaluation of the expression

`accept (Plus (Char #"a", Char #"b"))`

would have done very little work, returning nearly instantaneously a function of type

`string → bool`.

Only after being called on a string would that function have examined the regular expression `Plus(⋯)`.

Now, with staging, `accept (Plus(⋯))` builds a matcher for `Plus(⋯)` right away. That matcher can be re-used for different strings; no need to rebuild it every time.

That is all.

Have a good weekend.

See you Tuesday, when we will start working with Modules.