Lec 15

Regex continued

Recall :

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

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
```

But we can get better CPS

rewrite

```
  | match (Star r) cs k =
    let
      fun mstar cs' = (k cs') orelse
          (match r cs' => mstar)
    in
      mstar cs
    end
```

Refactoring using combinators

Notice `regexp -> char list -> (char list -> bool) -> bool`

← WT break type here

↑ Staging → do work here

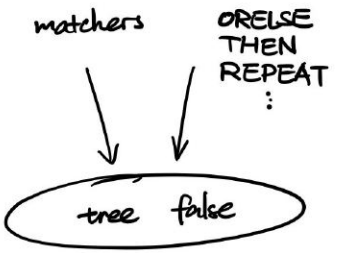
Want:

```
type matcher : char list -> (char list -> bool) -> bool

(* match : regexp -> matcher *)

(* accept : regexp -> string -> bool *)
```

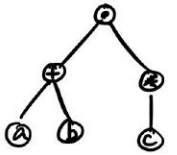
Some combo to make modules that recognise particular regex.

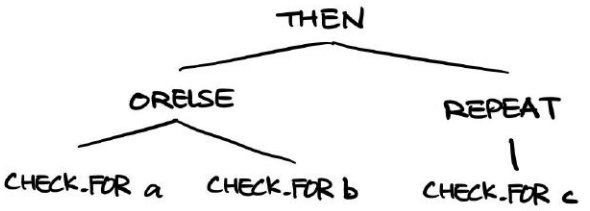


Some matchers

```
ACCEPT : matcher
REJECT : matcher
CHECK_FOR : matcher
THEN
REPEAT
```

(a + b) c*





Code

reject whatever ↓

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

fun CHECK_FOR (a:char) : matcher =
  (fn [] => REJECT []
    | c::cs' => if a = c then ACCEPT cs' else REJECT c::cs' )
  char list -> ((char list -> bool) -> bool)
```

← move on once happy with local position

← some staging here : knowing some of what to do before seeing actual character

```
infixr 8 ORELSE
infixr 9 THEN
```

infixr is right associative
infix is left ... by default
higher priority

```
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)

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

(* match : regexp -> matcher *)
fun match (Char(a))      = 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 accept r =
  let
    val m = match r
  in
    fn s => m (String.explode s) List.null
  end
```