Representing regular expressions

c | 1 | 0 | r₁ r₂ | r₁ + r₂ | r*

```
datatype regexp = Char of char
  | One
  | Zero
  | Times of regexp * regexp
  | Plus of regexp * regexp
  | Star of regexp
```
(* accept : regexp -> string -> bool

REQUIRES:  true
ENSURES: accept r s  returns true, if s is in L(r);
         accept r s  returns false, otherwise.
*)

(* 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.
*)
match and accept

fun match (Char(a)) cs k =  (case cs of
    [ ] => false
    | (c::cs') => (a=c) andalso k(cs'))
| match (One) cs k =   k(cs)
| match (Zero) _ k =   false
| match (Times (r1,r2)) cs k =  match r1 cs (fn cs' => match r2 cs' k)
| match (Plus (r1,r2)) cs k =  match r1 cs k orelse match r2 cs k
| match (Star(r)) cs k = k(cs) orelse match r cs
    (fn cs' => not (cs = cs') andalso match Star(r) cs' k)

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

accept (Plus(Char(a),Char(b)) "ab"

==> match (Plus(Char(a),Char(b)) [a,b] List.null

==> match (Char(a)) [a,b] List.null orelse match (Char(b)) [a,b] List.null

==> ((a = a) andalso List.null ([b])) orelse match (Char(b)) [a,b] List.null

==> false orelse match (Char(b)) [a,b] List.null

==> match (Char(b)) [a,b] List.null

==> (b = a) andalso List.null ([b])

==> false
Using combinators
true      false
andalso
orelse

Space of booleans

m1 THEN ORELSER m2

Space of functions that return booleans

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

true        false
andalso
orelse

Space of booleans
Code design

- match will take a regular expression and return a function (matcher) of type `char list -> (char list -> bool) -> bool`
- Combine functions of this type using combinators
  - Stage 1: Deconstructing regular expressions by pattern matching
  - Stage 2: Deal with the input string

```plaintext
type matcher = char list -> (char list -> bool) -> bool
```
Recall the staging example

\[
\textbf{fun} \ f3 \ (x:\text{int}) : \text{int} \to \text{int} = \\
\text{let} \\
\quad \textbf{val} \ z : \text{int} = \text{horrible}(x) \\
\text{in} \\
\quad \textbf{fn} \ y => z + y \\
\text{end}
\]
Continuation base cases

- **Val REJECT**: matcher = fn cs => fn k => false
- **Val ACCEPT**: matcher = fn cs => fn k => k (cs)

**Type** matcher = char list -> (char list -> bool) -> bool

- Instantly fail
- Call the continuation
Input related

```haskell
fun CHECK_FOR (a : char) : matcher =
  fn cs => fn k => (case cs of
    [ ] => false
    | (c::cs') => (a=c) andalso k(cs'))
```
fun CHECK_FOR (a : char) : matcher = 
  fn cs => fn k => (case cs of
      [ ] => false
    | c::cs' => (a=c) andalso k(cs'))

(* Alternatively, … *)

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

This is now a function that does some work (checks the character) before expecting the continuation.
ORELSE and THEN

```ml
infixr 8 ORELSE
infixr 9 THEN

fun (m1 : matcher) ORELSE (m2 : matcher) : matcher =
  fn cs => fn k => m1 cs k orelse m2 cs k

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

type matcher = char list -> (char list -> bool) -> bool
Assuming that regular expressions are in standard form

```plaintext
fun REPEAT (m : matcher) : matcher = fn cs => fn k =>
  let
    fun mstar cs' = __________________________
  in
    mstar cs
  end
```
Assuming that regular expressions are in standard form

fun REPEAT (m : matcher) : matcher = fn cs => fn k =>
  let
    fun mstar cs' = k cs' orelse m cs' mstar
  in
    mstar cs
  end
Allowing arbitrary regular expressions

```haskell
fun REPEAT (m : matcher) : matcher = fn cs => fn k =>
    let
        fun mstar cs' = k cs' orelse
            ____________________________________________
    in
        mstar cs
    end
```
Allowing arbitrary regular expressions

\textbf{fun} REPEAT (m : matcher) : matcher = \textbf{fn} cs \Rightarrow \textbf{fn} k \Rightarrow
\begin{align*}
\text{let} & \text{fun mstar cs' = k cs' \textbf{orelse} } \\
& \quad \text{m cs' (fn cs'' \Rightarrow \textbf{not} (cs' = cs'') \ \textbf{andalso} \ mstar cs'')} \\
\text{in} & \text{mstar cs} \\
\text{end}
\end{align*}
Build a matcher from a regexp

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

Write evaluation steps for accept (Plus(Char(a), Char(b)))
Staged matcher

fun accept (r : regexp) : string -> bool =

  let
    val m = match r
    in
    fn s => m (String.explode s) List.null
  end
You should be able to:

• State the grammar for regular expressions and a language associated with a given regular expression

• Write code for matching a regular expression with a string (i.e. deciding whether a given string is in the language of a given regular expression) with and without combinators

• Recognize what is being the staged in the version of the matcher code with combinators
Get some rest over the break!