

```
type state = int    (* Note: 0 is the start state *)

datatype dest =
  State of state (* Edge destination is another state *)
| Error          (* if no next state *)

type DFA = {table: dest Array2.array, accept: bool Array.array}

(* Check whether the DFA "pattern" accepts "text" *)
fun search({table, accept}: DFA, text: string): bool =
let

  (* Give the next state of the DFA, assuming that it is in
   * state "state" and the next character is text[pos]. *)
  fun next_state(pos: int, state: int): dest =
    let
      val char = Char.ord(String.sub(text, pos))
    in
      Array2.sub(table, state, char)
    end

  (* Walk the DFA down the string "text" from position "pos",
   * returning whether it accepts the rest of the string. *)
  fun search(pos: int, state: int): bool =
    if pos = String.size(text)
    then Array.sub(accept, state)
    else case next_state(pos, state) of
        Error => false
      | State(s) => search(pos + 1, s)

in
  search(0,0)
end

val nbOfStates = 6

val table = Array2.array(nbOfStates, Char.maxOrd + 1, Error)

Array2.update(table, 0, Char.ord("#a"), State 1)

Array2.update(table, 1, Char.ord("#b"), State 2)

Array2.update(table, 2, Char.ord("#b"), State 3)
Array2.update(table, 2, Char.ord("#c"), State 4)

Array2.update(table, 3, Char.ord("#b"), State 3)

Array2.update(table, 4, Char.ord("#b"), State 5)

val accept = Array.array(nbOfStates, false);

Array.update(accept, 1, true);
Array.update(accept, 2, true);
Array.update(accept, 3, true);
Array.update(accept, 5, true);

search({table = table, accept = accept}, "");

search({table = table, accept = accept}, "a");

search({table = table, accept = accept}, "b");

search({table = table, accept = accept}, "abbbbbbb");
```

```
search({table = table, accept = accept}, "abcb");
```

```
search({table = table, accept = accept}, "abcba");
```

```

datatype regexp =
  Empty                (* matches the empty string *)
| Char of char         (* Char(c) matches a string "c" iff c = c' *)
| Any                  (* Any matches a string "c" for any c *)
| Optional of regexp
| Or of regexp * regexp (* Or(r1,r2) matches s iff r1 matches s or r2
                        matches s *)
| Concat of regexp * regexp (* Concat(r1,r2) matches s iff we can break
                             s into two pieces s1 and s2 such that r1
                             matches s1 and r2 matches s2. *)
| Star of regexp        (* Star(r) matches s iff
                        Or(Empty,Concat(r,Star(r))) matches s.
                        That is, if zero or more copies of r
                        concatenated together matches s. *)

```

```

(* raised when we encounter a syntax error *)
exception SyntaxError of string

```

```

(* tokens processed by the parser *)
datatype token = Literal of char

```

```

  | VertBar
  | Asterisk
  | QMark
  | LParen
  | RParen
  | Period

```

```

(* convert list of characters into a list of tokens *)

```

```

fun tokenize (inp : char list): token list =
  (case inp of
    nil => nil
  | (#|" " :: cs) => (VertBar :: tokenize cs)
  | (#|. " :: cs) => (Period :: tokenize cs)
  | (#"? " :: cs) => (QMark :: tokenize cs)
  | (#"* " :: cs) => (Asterisk :: tokenize cs)
  | (#("(" :: cs) => (LParen :: tokenize cs)
  | (#")" :: cs) => (RParen :: tokenize cs)
  | (#" " :: cs) => tokenize cs
  | (c :: cs) => Literal c :: tokenize cs)

```

```

fun parse_rexp (ts: token list): regexp * (token list) =

```

```

  let
    val (term, rest) = parse_term ts
  in
    case rest of
      ([ | RParen::_) => (Concat(term, Empty), rest)
    | _ => let
        val (exp, rest') = parse_rexp rest
      in
        (Concat(term, exp), rest')
      end
    end
  end

```

```

and parse_term (ts: token list): regexp * (token list) =

```

```

  let
    val (factor, rest) = parse_factor ts
  in
    case rest of
      QMark::tl => (Optional factor, tl)
    | Asterisk::tl => (Star factor, tl)
    | VertBar::tl => let
        val (term, rest') = parse_term tl
      in

```

```

        (Or(factor, term), rest')
      end
    | _ => (factor, rest)
  end

and parse_factor (ts: token list): regexp * (token list) =
  case ts of
    (Literal c)::tl => (Char c, tl)
  | Period::tl => (Any, tl)
  | LParen::tl => let
      val (exp, rest) = parse_rexp tl
    in
      case rest of
        RParen::tl => (exp, tl)
      | _ => raise SyntaxError "parantheses not properly
balanced"
      end
    | _ => raise SyntaxError "incorrect regular expression"

fun match (rexp: string) (s: string): bool =
  let

    fun helper (exp: regexp) (cs: char list) (k: char list -> bool): bool
    =
      case (exp, cs) of
        (Empty, _) => k cs
      | (Char c, []) => false
      | (Char c, c'::tl) => (c = c') andalso (k tl)
      | (Any, []) => false
      | (Any, _::tl) => k cs
      | (Or(e1, e2), _) => (helper e1 cs k ) orelse (helper e2 cs k)
      | (Concat(e1, e2), _) => helper e1 cs (fn cs' => helper e2 cs' k)
      | (Star e1, _) => (k cs) orelse (helper e1 cs (fn cs' => helper exp
cs' k))
      | (Optional e, _) => (k cs) orelse (helper e cs k)

    val (exp, _): regexp * token list = parse_rexp(tokenize
(String.explode rexp))

  in

    helper exp (String.explode s) (List.null)

  end

```