# A simple library for regular expressions

Claude Marché

June 20, 2002

# Contents

# Chapter 1

# Introduction

This library implements simple use of regular expressions. It provides direct definitions of regular expressions form the usual constructions, or definition of such expressions form a syntactic manner using GNU regexp syntax. It provides a simple compilation of regexp into a deterministic automaton, and use of such an automaton for matching and searching.

# Chapter 2

# Library documentation

This is the documentation for use of the library. It provides all functions in one module.

## 2.1 Module *Regexp*

### 2.1.1 The regexp datatype and its constructors

The type of regular expressions is abstract. Regular expressions may be built from the following constructors :

- *empty* is the regexp that denotes no word at all.
- *epsilon* is the regexp that denotes the empty word.
- *char c* returns a regexp that denotes only the single-character word *c*.
- *char_interv a b* returns a regexp that denotes any single-character word belonging to char interval *a*, *b*.
- *string str* denotes the string *str* itself.
- *star e* where *e* is a regexp, denotes the Kleene iteration of *e*, that is all the words made of concatenation of zero, one or more words of *e*.
- *alt e1 e2* returns a regexp for the union of languages of *e1* and *e2*.
- *seq e1 e2* returns a regexp for the concatenation of languages of *e1* and *e2*.
- *opt e* returns a regexp for the set of words of *e* and the empty word.
- *some e* denotes all the words made of concatenation of one or more words of *e*.

type *regexp*

val *empty* : *regexp*

val *epsilon* : *regexp*

val *char* : *char* → *regexp*

val *char_interv* : *char* → *char* → *regexp*

val *string* : *string* → *regexp*

val *star* : *regexp* → *regexp*

val *alt* : *regexp* → *regexp* → *regexp*

val *seq* : *regexp* → *regexp* → *regexp*

val *opt* : *regexp* → *regexp*

val *some* : *regexp* → *regexp*

| | |
|---|---|
| *char* | denotes the character *char* for all non-special chars |
| *char* | denotes the character *char* for special characters `.`, `\`, `*`, `+`, `?`, `[` and `]` |
| `.` | denotes any single-character word |
| `[`*set*`]` | denotes any single-character word belonging to *set*. Intervals may be given as in `[a-z]`. |
| `[^`*set*`]` | denotes any single-character word not belonging to *set*. |
| *regexp*`*` | denotes the Kleene star of *regexp* |
| *regexp*`+` | denotes any concatenation of one or more words of *regexp* |
| *regexp*`?` | denotes the empty word or any word denoted by *regexp* |
| $regexp_1$ `\|` $regexp_2$ | denotes any words in $regexp_1$ or in $regexp_2$ |
| $regexp_1$ $regexp_2$ | denotes any contecatenation of a word of $regexp_1$ and a word of $regexp_2$ |
| `(`*regexp*`)` | parentheses, denotes the same words as *regexp*. |

Figure 2.1: Syntax of regular expressions

### 2.1.2 Regexp matching by runtime interpretation of regexp

*match_string r s* returns true if the string *s* is in the language denoted by *r*. This function is by no means efficient, but it is enough if you just need to match once a simple regexp against a string.

If you have a complicated regexp, or if you're going to match the same regexp repeatedly against several strings, we recommend to use compilation of regexp provided by module *Automata*.

val *match_string* : *regexp* $\rightarrow$ *string* $\rightarrow$ *bool*

### 2.1.3 Syntax of regular expressions

This function offers a way of building regular expressions syntactically, following more or less the GNU regexp syntax as in egrep. This is summarized in the table of figure 2.1.

Moreover, the postfix operators `*`, `+` and `?` have priority over the concatenation, itself having priority over alternation with `|`

*from_string str* builds a regexp by interpreting syntactically the string *str*. The syntax must follow the table above. It raises exception *Invalid_argument* `"from_string"` if the syntax is not correct.

val *from_string* : *string* $\rightarrow$ *regexp*

### 2.1.4 Compilation of regular expressions

type *compiled_regexp*

val *compile* : *regexp* $\rightarrow$ *compiled_regexp*

### 2.1.5 Matching and searching functions

*search_forward e str pos* search in the string *str*, starting at position *pos* a word that is in the language of *r*. Returns a pair $(b, e)$ where $b$ is position of the first char matched, and $e$ is the position following the position of the last char matched.

Raises *Not_found* of no matching word is found.

Notice: even if the regular expression accepts the empty word, this function will never return $(b, e)$ with $e = b$. In other words, this function always search for non-empty words in the language of *r*.

Unpredictable results may occur if $pos < 0$.

val *search_forward* : *compiled_regexp* $\rightarrow$ *string* $\rightarrow$ *int* $\rightarrow$ *int* $\times$ *int*

*split_strings r s* extract from string *s* the subwords (of maximal size) that are in the language of *r*. For example *split_strings* (*compile* (*from_string* `"[0-9]+"`)) `"12+3*45"` returns [`"12"`;`"3"`;`"45"`].

*split_delim* *a* *s* splits string *s* into pieces delimited by *r*. For example *split_strings* (*compile* (*char* `':'`)) `"marche:G6H3a656h6g56:534:180:Claude␣Marche:/home/marche:/bin/ba` returns [ `"marche"` ; `"G6H3a656h6g56"` ; `"534"` ; `"180"` ; `"Claude` `␣␣Marche"` ; `"/home/marche"` ; `"/bin/bash"`].

val *split_strings* : *compiled_regexp* → *string* → *string list*

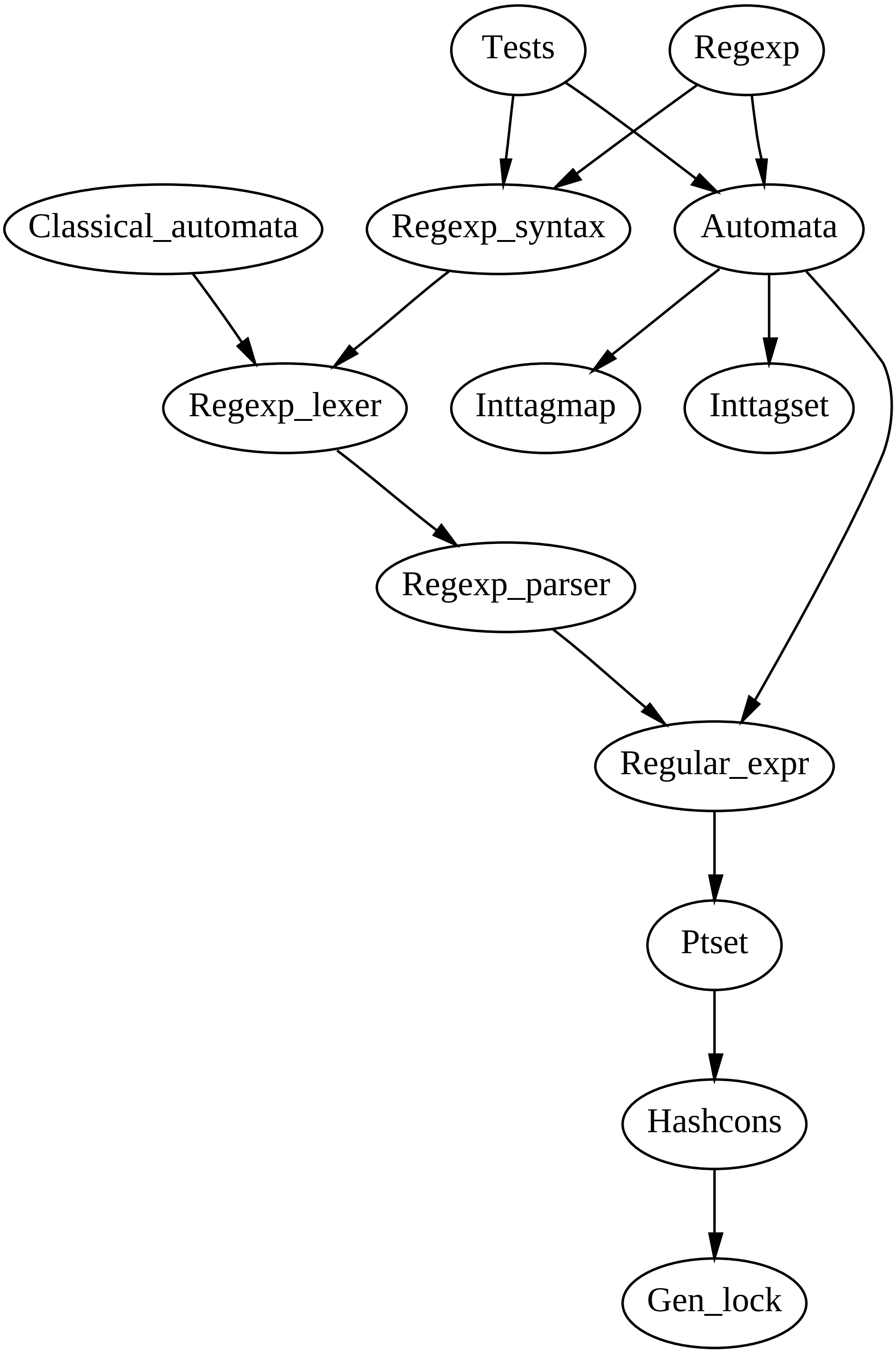
val *split_delim* : *compiled_regexp* → *string* → *string list*

# Index of Identifiers

# Chapter 3

# Documentation of implementation

This part describe the implementation of the library. It provides several modules which depends each other as shown by the graph below.




## 3.1 Module *Hashcons*

See the GNU Library General Public License version 2 for more details (enclosed in the file LGPL).

Hash tables for hash consing. Hash consed values are of the following type *hash_consed*. The field *tag* contains a unique integer (for values hash consed with the same table). The field *hkey* contains the hash key of the value (without modulo) for possible use in other hash tables (and internally when hash consing tables are resized). The field *node*} *contains the value itself*.

```
type α hash_consed = {
  hkey : int;
  tag : int;
  node : α }
```

Functorial interface.

```
module type HashedType =
  sig
    type t
    val equal : t × t → bool
    val hash : t → int
  end

module type S =
  sig
    type key
    type t
    val create : int → t
    val hashcons : t → key → key hash_consed
  end

module Make(H : HashedType) : (S with type key = H.t)
```

## 3.2 Module *Regular_expr*

This module defines the regular expressions, and provides some simple manipulation of them.

### 3.2.1 The regexp datatype and its constructors

The type of regular expressions is abstract. Regular expressions may be built from the following constructors :

- *empty* is the regexp that denotes no word at all.
- *epsilon* is the regexp that denotes the empty word.
- *char c* returns a regexp that denotes only the single-character word *c*.
- *chars s* returns a regexp that denotes any single-character word belonging to set of chars *s*.
- *string str* denotes the string *str* itself.
- *star e* where *e* is a regexp, denotes the Kleene iteration of *e*, that is all the words made of concatenation of zero, one or more words of *e*.
- *alt e1 e2* returns a regexp for the union of languages of *e1* and *e2*.
- *seq e1 e2* returns a regexp for the concatenation of languages of *e1* and *e2*.
- *opt e* returns a regexp for the set of words of *e* and the empty word.
- *some e* denotes all the words made of concatenation of one or more words of *e*.

type *regexp*

val *uniq_tag* : *regexp* $\to$ *int*

val *empty* : *regexp*

val *epsilon* : *regexp*

val *char* : *char* $\to$ *regexp*

val *char_interv* : *char* $\to$ *char* $\to$ *regexp*

val *string* : *string* $\to$ *regexp*

val *star* : *regexp* $\to$ *regexp*

val *alt* : *regexp* $\to$ *regexp* $\to$ *regexp*

val *seq* : *regexp* $\to$ *regexp* $\to$ *regexp*

val *opt* : *regexp* $\to$ *regexp*

val *some* : *regexp* $\to$ *regexp*

### 3.2.2 Simple regexp operations

The following three functions provide some simple operations on regular expressions:

- *nullable* $r$ returns true if regexp $r$ accepts the empty word.
- *residual* $r$ $c$ returns the regexp $r$' denoting the language of words $w$ such that $cw$ is in the language of $r$.
- *firstchars* $r$ returns the set of characters that may occur at the beginning of words in the language of $e$.

val *nullable* : *regexp* $\to$ *bool*

val *residual* : *regexp* $\to$ *int* $\to$ *regexp*

val *firstchars* : *regexp* $\to$ $(int \times int \times regexp)$ *list*

### 3.2.3 Regexp matching by runtime interpretation of regexp

*match_string* $r$ $s$ returns true if the string $s$ is in the language denoted by $r$. This function is by no means efficient, but it is enough if you just need to match once a simple regexp against a string.

If you have a complicated regexp, or if you're going to match the same regexp repeatedly against several strings, we recommend to use compilation of regexp provided by module *Automata*.

val *match_string* : *regexp* $\to$ *string* $\to$ *bool*

pretty-printer

val *fprint* : *Format.formatter* $\to$ *regexp* $\to$ *unit*

val *print* : *regexp* $\to$ *unit*

## 3.3 Implementation of module *Regular_expr*

### 3.3.1 regexp datatype and simplifying constructors

open *Hashcons*

type *regexp* = *regexp_struct Hashcons.hash_consed*

```
and regexp_struct =
  | Empty
  | Epsilon
  | Char_interv of int × int
  | String of string                                    (* length at least 2 *)
  | Star of regexp
  | Alt of regexp_struct Ptset.t
  | Seq of regexp × regexp

let uniq_tag r = r.tag

let regexp_eq (r1, r2) =
  match (r1, r2) with
    | Empty, Empty → true
    | Epsilon, Epsilon → true
    | Alt s1, Alt s2 → Ptset.equalq s1 s2
    | Star r1, Star r2 → r1 ≡ r2
    | Seq(s11, s12), Seq(s21, s22) → s11 ≡ s21 ∧ s12 ≡ s22
    | String s1, String s2 → s1 = s2
    | Char_interv(a1, b1), Char_interv(a2, b2) → a1 ≡ a2 ∧ b1 ≡ b2
    | _ → false

module Hash = Hashcons.Make(struct type t = regexp_struct
                                   let equal = regexp_eq
                                   let hash = Hashtbl.hash end)

let hash_consing_table = Hash.create 257

let hash_cons = Hash.hashcons hash_consing_table

let empty = hash_cons Empty

let epsilon = hash_cons Epsilon

let star e =
  match e.node with
    | Empty | Epsilon → epsilon
    | Star _ → e
    | _ → hash_cons (Star e)

let alt e1 e2 =
  match e1.node, e2.node with
    | Empty, _ → e2
    | _, Empty → e1
    | Alt(l1), Alt(l2) → hash_cons (Alt(Ptset.union l1 l2))
    | Alt(l1), _ → hash_cons(Alt(Ptset.add e2 l1))
    | _, Alt(l2) → hash_cons(Alt(Ptset.add e1 l2))
    | _ →
        if e1 ≡ e2 then e1
        else hash_cons(Alt(Ptset.add e1 (Ptset.singleton e2)))

let seq e1 e2 =
  match e1.node, e2.node with
    | Empty, _ → empty
    | _, Empty → empty
    | Epsilon, _ → e2
    | _, Epsilon → e1
    | _ → hash_cons(Seq(e1, e2))

let char c = let c = Char.code c in hash_cons(Char_interv(c, c))
```

```
let char_interv a b = hash_cons(Char_interv(Char.code a, Char.code b))

let string s =
  if s = ""
  then
    epsilon
  else
    if String.length s = 1
    then let c = String.get s 0 in char c
    else hash_cons(String s)
```

### 3.3.2 extended regexp

```
let some e = (seq e (star e))

let opt e = (alt e epsilon)
```

### 3.3.3 Regexp match by run-time interpretation of regexp

```
let rec nullable r =
  match r.node with
    | Empty → false
    | Epsilon → true
    | Char_interv(n1, n2) → false
    | String _ → false                                        (* cannot be "" *)
    | Star e → true
    | Alt(l) → Ptset.exists nullable l
    | Seq(e1, e2) → nullable e1 ∧ nullable e2

let rec residual r c =
  match r.node with
    | Empty → empty
    | Epsilon → empty
    | Char_interv(a, b) → if a ≤ c ∧ c ≤ b then epsilon else empty
    | String s →                                              (* s cannot be "" *)
        if c = Char.code(String.get s 0)
        then string (String.sub s 1 (pred (String.length s)))
        else empty
    | Star e → seq (residual e c) r
    | Alt(l) →
        Ptset.fold
          (fun e accu → alt (residual e c) accu)
          l
          empty
    | Seq(e1, e2) →
        if nullable(e1)
        then alt (seq (residual e1 c) e2) (residual e2 c)
        else seq (residual e1 c) e2
```

```
let match_string r s =
  let e = ref r in
  for i = 0 to pred (String.length s) do
    e := residual !e (Char.code(String.get s i))
  done;
  nullable !e
```

*firstchars* $r$ returns the set of characters that may start a word in the language of $r$

```
let add a b r l = if a > b then l else (a, b, r) :: l
```

```
let rec insert a b r l =
  match l with
    | [] → [(a, b, r)]
    | (a1, b1, r1) :: rest →
        if b < a1
        then
          (* a <= b < a1 <= b1 *)
          (a, b, r) :: l
        else
          if b ≤ b1
          then
            if a ≤ a1
            then
              (* a <= a1 <= b <= b1 *)
              add a (a1 − 1) r ((a1, b, alt r r1) :: (add (b + 1) b1 r1 rest))
            else
              (* a1 < a <= b <= b1 *)
              (a1, a − 1, r1) :: (a, b, alt r r1) :: (add (b + 1) b1 r1 rest)
          else
            if a ≤ a1
            then
              (* a <= a1 <= b1 < b *)
              add a (a1 − 1) r ((a1, b1, alt r r1) :: (insert (b1 + 1) b r rest))
            else
              if a ≤ b1
              then
                (* a1 < a <= b1 < b *)
                (a1, a − 1, r1) :: (a, b1, alt r r1) :: (insert (b1 + 1) b r rest)
              else
                (* a1 <= b1 < a <= b *)
                (a1, b1, r1) :: (insert a b r rest)
```

```
let insert_list l1 l2 =
  List.fold_right
    (fun (a, b, r) accu → insert a b r accu)
    l1
    l2
```

```
let rec firstchars r =
  match r.node with
    | Empty → [ ]
    | Epsilon → [ ]
    | Char_interv(a, b) → [(a, b, epsilon)]
    | String s →
        let c = Char.code(String.get s 0) in
        [(c, c,string (String.sub s 1 (pred (String.length s))))]
    | Star e →
        let l = firstchars e in
        List.map (fun (a, b, res) → (a, b, seq res r)) l
    | Alt(s) →
        Ptset.fold
          (fun e accu → insert_list (firstchars e) accu)
          s
          [ ]
    | Seq(e1, e2) →
        if nullable e1
        then
          let l1 = firstchars e1 and l2 = firstchars e2 in
          insert_list
            (List.map (fun (a, b, res) → (a, b, seq res e2)) l1)
            l2
        else
          let l1 = firstchars e1 in
          List.map (fun (a, b, res) → (a, b, seq res e2)) l1

let _ =
  let r = seq(star (alt (char 'a') (char 'b'))) (char 'c') in
  firstchars r

open Format

let rec fprint fmt r =
  match r.node with
    | Empty → fprintf fmt "(empty)"
    | Epsilon → fprintf fmt "(epsilon)"
    | Char_interv(a, b) → fprintf fmt "['%c'-'%c']" (Char.chr a) (Char.chr b)
    | String s → fprintf fmt "\"%s\"" s
    | Star e → fprintf fmt "(%a)*" fprint e
    | Alt(l) →
        fprintf fmt "(";
        Ptset.iter (fun e → fprintf fmt "|%a" fprint e) l;
        fprintf fmt ")"
    | Seq(e1, e2) → fprintf fmt "(%a %a)" fprint e1 fprint e2

let print = fprint std_formatter
```

# Module Regexp_parser (Yacc)

## Header

```
open Regular_expr
```

### Token declarations

```
%token <char> CHAR
%token <Regular_expr.regexp> CHARSET
%token STAR ALT PLUS QUESTION OPENPAR CLOSEPAR EOF

%start regexp_start
%type <Regular_expr.regexp> regexp_start

%left ALT
%left CONCAT CHAR CHARSET OPENPAR
%nonassoc STAR PLUS QUESTION
```

### Grammar rules

```
regexp_start ::=
  | regexp EOF
      { $1 }


regexp ::=
  | CHAR
      { char $1 }
  | CHARSET
      { $1 }
  | regexp STAR
      { star $1 }
  | regexp PLUS
      { some $1 }
  | regexp QUESTION
      { opt $1 }
  | regexp ALT regexp
      { alt $1 $3 }
  | regexp regexp %prec CONCAT
      { seq $1 $2 }
  | OPENPAR regexp CLOSEPAR
      { $2 }
```

## Module Regexp_lexer (Lex)

```
{

  open Regular_expr

  open Regexp_parser

  open Lexing
```

```
let add_inter a b l =
  let rec add_rec a b l =
    match l with
    | [] → [(a, b)]
    | (a1, b1) :: r →
        if b < a1
        then (a, b) :: l
        else
          if b1 < a
          then (a1, b1) :: (add_rec a b r)
          else
            (* intervals a, b and a1, b1 intersect *)
            add_rec (min a a1) (max b b1) r
  in
  if a > b then l else add_rec a b l


let complement l =
  let rec comp_rec a l =
    match l with
    | [] →
        if a < 256 then [(a, 255)] else []
    | (a1, b1) :: r →
        if a < a1 then (a, a1 − 1) :: (comp_rec (b1 + 1) r) else comp_rec (b1 + 1) r
  in
  comp_rec 0 l


let interv a b = char_interv (Char.chr a) (Char.chr b)


let rec make_charset l =
  match l with
  | [] → empty
  | (a, b) :: r → alt (interv a b) (make_charset r)


}
```

```
rule token = parse
  | '\\' _
      { CHAR (lexeme_char lexbuf 1) }
  | '.'
      { CHARSET(interv 0 255) }
  | '*'
      { STAR }
  | '+'
      { PLUS }
  | '?'
      { QUESTION }
  | '|'
      { ALT }
  | '('
      { OPENPAR }
  | ')'
      { CLOSEPAR }
  | "[^"
      { CHARSET(make_charset (complement (charset lexbuf))) }
  | '['
      { CHARSET(make_charset (charset lexbuf)) }
  | _
      { CHAR (lexeme_char lexbuf 0) }
  | eof
      { EOF }

and charset = parse
  | ']'
      { [] }
  | '\\' _
      { let c = Char.code(lexeme_char lexbuf 1) in
        add_inter c c (charset lexbuf) }
  | [^ '\\'] '-' _
      { let c1 = Char.code (lexeme_char lexbuf 0)
        and c2 = Char.code (lexeme_char lexbuf 2)
        in
        add_inter c1 c2 (charset lexbuf) }
  | _
      { let c = Char.code(lexeme_char lexbuf 0) in
        add_inter c c (charset lexbuf) }
```

## 3.4 Module *Regexp_syntax*

This module offers a way of building regular expressions syntactically, following more or less the GNU regexp syntax as in egrep. This is summarized in the table of figure 3.1.

Moreover, the postfix operators `*`, `+` and `?` have priority over the concatenation, itself having priority over alternation with `|`

*from_string str* builds a regexp by interpreting syntactically the string *str*. The syntax must follow the table above. It raises exception *Invalid_argument* `"from_string"` if the syntax is not correct.

```
val from_string : string → Regular_expr.regexp
```

| | |
|---|---|
| *char* | denotes the character *char* for all non-special chars |
| `\`*char* | denotes the character *char* for special characters `.`, `\`, `*`, `+`, `?`, `\|`, `[`, `]`, `(` and `)` |
| `.` | denotes any single-character word |
| `[`*set*`]` | denotes any single-character word belonging to *set*. Intervals may be given as in `[a-z]`. |
| `[^`*set*`]` | denotes any single-character word not belonging to *set*. |
| *regexp*`*` | denotes the Kleene star of *regexp* |
| *regexp*`+` | denotes any concatenation of one or more words of *regexp* |
| *regexp*`?` | denotes the empty word or any word denoted by *regexp* |
| $regexp_1$ `\|` $regexp_2$ | denotes any words in $regexp_1$ or in $regexp_2$ |
| $regexp_1\, regexp_2$ | denotes any contecatenation of a word of $regexp_1$ and a word of $regexp_2$ |
| `(`*regexp*`)` | parentheses, denotes the same words as *regexp*. |

Figure 3.1: Syntax of regular expressions

## 3.5 Implementation of module *Regexp_syntax*

```
let from_string s =
  try
    let b = Lexing.from_string s in
    Regexp_parser.regexp_start Regexp_lexer.token b
  with
      Parsing.Parse_error → invalid_arg "from_string"
```

## 3.6 Module *Automata*

This module provides compilation of a regexp into an automaton. It then provides some functions for matching and searching.

### 3.6.1 Automata, compilation

Automata considered here are deterministic. The type of automata is abstract. *compile r* returns an automaton that recognizes the language of *r*.

```
type automaton
```

```
val compile : Regular_expr.regexp → automaton
```

### 3.6.2 Execution of automata

*exec_automaton auto str pos* executes the automaton *auto* on string *str* starting at position *pos*. Returns the maximal position *p* such that the substring of *str* from positions *pos* (included) to *p* (excluded) is acceptable by the automaton, -1 if no such position exists.

Unpredictable results may occur if $pos < 0$.

```
val exec_automaton : automaton → string → int → int
```

### 3.6.3 Matching and searching functions

*search_forward a str pos* search in the string *str*, starting at position *pos* a word that is in the language of automaton *a*. Returns a pair $(b, e)$ where $b$ is position of the first char matched, and $e$ is the position following the position of the last char matched.

Raises *Not_found* of no matching word is found.

Notice: even if the automaton accepts the empty word, this function will never return $(b, e)$ with $e = b$. In other words, this function always search for non-empty words in the language of automaton $a$.

Unpredictable results may occur if $pos < 0$.

val *search_forward* : *automaton* → *string* → *int* → *int* × *int*

*split_strings* $a$ $s$ extract from string $s$ the subwords (of maximal size) that are in the language of $a$. For example *split_strings* (*compile* (*from_string* `"[0-9]+"`)) `"12+3*45"` returns [`"12";"3";"45"`].

*split_delim* $a$ $s$ splits string $s$ into pieces delimited by $a$. For example *split_strings* (*compile* (*char* `':'`)) `"marche:G6H3a656h6g56:534:180:Claude␣Marche:/home/marche:/bin/ba` returns [ `"marche"` ; `"G6H3a656h6g56"` ; `"534"` ; `"180"` ; `"Claude`
`␣␣Marche"` ; `"/home/marche"` ; `"/bin/bash"`].

val *split_strings* : *automaton* → *string* → *string list*

val *split_delim* : *automaton* → *string* → *string list*

*to_dot* $a$ $f$ exports the automaton $a$ to the file $f$ in DOT format.

val *to_dot* : *automaton* → *string* → *unit*

## 3.7 Implementation of module *Automata*

Compilation of regexp into an automaton

open *Regular_expr*

### 3.7.1 The type of automata

Automata considered here are deterministic.

The states of these automata are always represented as natural numbers, the initial state always being 0.

An automaton is then made of a transition table, giving for each state $n$ a sparse array that maps characters codes to states ; and a table of accepting states.

```
type automaton =
  {
    auto_trans : (int × int array) array;
    auto_accept : bool array;
  }
```

### 3.7.2 Execution of automata

*exec_automaton auto str pos* executes the automaton *auto* on string *str* starting at position *pos*. Returns the maximal position *p* such that the substring of *str* from positions *pos* (included) to *p* (excluded) is acceptable by the automaton, -1 if no such position exists.

```
let exec_automaton auto s pos =
    let state = ref 0
    and last_accept_pos =
      ref (if auto.auto_accept.(0) then pos else -1)
    and i = ref pos
    and l = String.length s
    in
    try
      while !i < l do
        let (offset, m) = auto.auto_trans.(!state) in
        let index = Char.code(String.get s !i) − offset in
        if index < 0 ∨ index ≥ Array.length m then raise Exit;
        state := m.(index);
        if !state = − 1 then raise Exit;
        incr i;
        if auto.auto_accept.(!state) then last_accept_pos := !i;
      done;
      !last_accept_pos;
    with
        Exit → !last_accept_pos;
```

### 3.7.3 Searching functions

*search_forward a str pos* search in the string *str*, starting at position *pos* a word that is in the language of automaton $a$. Returns a pair $(b, e)$ where $b$ is position of the first char matched, and $e$ is the position following the position of the last char matched.

Raises *Not_found* of no matching word is found.

Notice: even if the automaton accepts the empty word, this function will never return $(b, e)$ with $e = b$. In other words, this function always search for non-empty words in the language of automaton *a*.

Unpredictable results may occur if $pos < 0$.

```
let rec search_forward auto s pos =
  if pos ≥ String.length s
  then raise Not_found
  else
    let n = exec_automaton auto s pos in
    if n > pos then pos, n else search_forward auto s (succ pos)
```

*split_strings a s* extract from string $s$ the subwords (of maximal size) that are in the language of $a$

```
let split_strings auto line =
  let rec loop pos =
    try
      let b, e = search_forward auto line pos in
      let id = String.sub line b (e − b) in
      id :: (loop e)
    with Not_found → []
  in
  loop 0
```

```
let split_delim auto line =
  let rec loop pos =
    try
      let b, e = search_forward auto line pos in
      let id = String.sub line pos (b - pos) in
      id :: (loop e)
    with Not_found ->
      [String.sub line pos (String.length line - pos)]
  in
  loop 0
```

### 3.7.4 Compilation of a regexp

*compile r* returns an automaton that recognizes the language of *r*.

```
module IntMap =
  Inttagmap.Make(struct type t = int let tag x = x end)

module IntSet =
  Inttagset.Make(struct type t = int let tag x = x end)

let rec compute_max b l =
  match l with
    | [] -> b
    | (_, x, _) :: r -> compute_max x l

module HashRegexp =
  Hashtbl.Make(struct
                 type t = Regular_expr.regexp
                 let equal = (≡)
                 let hash = Regular_expr.uniq_tag
               end)

let compile r =
```

we have a hash table to avoid several compilation of the same regexp

```
  let hashtable = HashRegexp.create 257
```

*transtable* is the transition table to fill, and *acceptable* is the table of accepting states. *transtable* maps any state into a *CharMap.t*, which itself maps characters to states.

```
  and transtable = ref IntMap.empty
  and acceptable = ref IntSet.empty
  and next_state = ref 0
  in
```

*loop r* fills the tables for the regexp *r*, and return the initial state of the resulting automaton.

```
let rec loop r =
  try
    HashRegexp.find hashtable r
  with
      Not_found ->
        (* generate a new state *)
        let init = !next_state
        and next_chars = Regular_expr.firstchars r
        in
        incr next_state;
        (* fill the hash table before recursion *)
        HashRegexp.add hashtable r init;
        (* fill the set of acceptable states *)
        if nullable r then acceptable := IntSet.add init !acceptable;
        (* compute the map from chars to states for the new state *)
        let t = build_sparse_array next_chars in
        (* add it to the transition table *)
        transtable := IntMap.add init t !transtable;
        (* return the new state *)
        init

and build_sparse_array next_chars =
  match next_chars with
    | [] -> (0, [||])
    | (a, b, _) :: r ->
        let mini = a
        and maxi = List.fold_left (fun _ (_, x, _) -> x) b r
        in
        let t = Array.create (maxi - mini + 1) (-1) in
        List.iter
          (fun (a, b, r) ->
             let s = loop r in
             for i = a to b do
               t.(i - mini) <- s
             done)
          next_chars;
        (mini, t)
in
let _ = loop r in
```

we then fill the arrays defining the automaton

```
{
  auto_trans = Array.init !next_state (fun i -> IntMap.find i !transtable);
  auto_accept = Array.init !next_state (fun i -> IntSet.mem i !acceptable);
}
```

### 3.7.5 Output functions

*to_dot a f* exports the automaton *a* to the file *f* in DOT format.

```
open Printf

module CharSet =
  Set.Make(struct type t = char let compare (x :char) (y :char) = compare x y end)

let no_chars = CharSet.empty
```

```
let rec char_interval a b =
  if a > b then no_chars
  else CharSet.add (Char.chr a) (char_interval (succ a) b)

let all_chars = char_interval 0 255

let complement = CharSet.diff all_chars

let intervals s =
  let rec interv = function
    | i, [] → List.rev i
    | [], n :: l → interv ([(n, n)], l)
    | (mi, ma) :: i as is, n :: l →
        if Char.code n = succ (Char.code ma) then
          interv ((mi, n) :: i, l)
        else
          interv ((n, n) :: is, l)
  in
  interv ([], CharSet.elements s)

let output_label cout s =
  let char = function
    | '"' → "\\\""
    | '\\' → "#92"
    | c →
        let n = Char.code c in
        if n > 32 ∧ n < 127 then String.make 1 c else sprintf "#%d" n
  in
  let output_interv (mi, ma) =
    if mi = ma then
      fprintf cout "%s␣" (char mi)
    else if Char.code mi = pred (Char.code ma) then
      fprintf cout "%s␣%s␣" (char mi) (char ma)
    else
      fprintf cout "%s-%s␣" (char mi) (char ma)
  in
  let is = intervals s in
  let ics = intervals (complement s) in
  if List.length is < List.length ics then
    List.iter output_interv is
  else begin
    fprintf cout "[^"; List.iter output_interv ics; fprintf cout "]"
  end

let output_transitions cout i (n, m) =
  let rev_m = ref IntMap.empty in
  for k = 0 to Array.length m − 1 do
    let j = m.(k) in
    let s = try IntMap.find j !rev_m with Not_found → CharSet.empty in
    rev_m := IntMap.add j (CharSet.add (Char.chr(k + n)) s) !rev_m
  done;
  IntMap.iter
    (fun j s →
       fprintf cout "␣␣%d␣->␣%d␣[␣label␣=␣\"" i j;
       output_label cout s;
       fprintf cout "\"␣];\n")
    !rev_m
```

```
let to_dot a f =
  let cout = open_out f in
  fprintf cout "digraph␣finite_state_machine␣{
   ␣␣/*␣rankdir=LR;␣*/
   ␣␣orientation=land;
   ␣␣node␣[shape␣=␣doublecircle];";
  Array.iteri (fun i b → if b then fprintf cout "%d␣" i) a.auto_accept;
  fprintf cout ";\n␣␣node␣[shape␣=␣circle];\n";
  Array.iteri (output_transitions cout) a.auto_trans;
  fprintf cout "}\n";
  close_out cout
```

# Index of Identifiers