Assignment 3
Construction of DFA from regular expressions

Student Name: Your name, your.email@ntut.edu.tw
Student ID: Your student ID number,
Instructor: Jyun-Ao Lin, jalin@ntut.edu.tw

In this HW, we study a method for the direct construction of a deterministic finite automaton from a regular expression. This is an efficient algorithm, used in particular in `ocamllex`.

The basic idea is as follows: if a finite automaton recognizes the language corresponding to the regular expression `r` , then any letter of a recognized word can be matched with an occurrence of a letter appearing in `r`. To distinguish the different occurrences of the same letter in `r`, we will index them by integers. For example, consider the regular expression `(ab)*` a (a—b)—, which defines words in the alphabet {`a`, `b`} whose penultimate letter is an `a`. If we index the characters, we obtain

$$(\texttt{a1}|\texttt{b1}) * \texttt{a2}(\texttt{a3}|\texttt{b2})$$

If we consider the word `aabaab` , then it matches the regular expression in the following way

`a1a1b1a1a2b2`

The idea is then to build an automaton whose states are sets of indexed letters, corresponding to the occurrences that can be read at a time. Thus the initial state contains the first possible letters of the recognized words. In our example, this is `a1 , b1 , a2`. To build the transitions, it is enough to calculate, for each occurrence of a letter, the set of occurrences that can follow it. In our example, if we have just read `a1`, then the following possible characters are `a1 , b1 , a2`; if we have just read `a2` , then these are `a3 , b2`.

Exercise 1: Nullity of a regular expression

We consider the following Caml type for regular expressions whose letters are indexed by integers (`ichar` type).

```
type ichar = char * int

type regexp =
  | Epsilon
  | Character of ichar
  | Union of regexp * regexp
  | Concat of regexp * regexp
  | Star of regexp
```

Write a function

```
val null : regexp -> bool
```

which determines whether `epsilon` (the empty word) belongs to the language recognized by a regular expression.

Test with

```
let () =
  let a = Character ('a', 0) in
  assert (not (null a));
  assert (null (Star a));
  assert (null (Concat (Epsilon, Star Epsilon)));
  assert (null (Union (Epsilon, a)));
  assert (not (null (Concat (a, Star a))))
```

Exercise 2: The first and the last

To represent the sets of indexed letters, we give ourselves the following Caml type:

```
module Cset = Set.Make(struct type t = ichar let compare = Stdlib.compare end)
```

Write a function

```
val first : regexp -> Cset.t
```

which calculates the set of first letters of words recognized by a regular expression. (You must use `null`.)

Similarly, writing a function

```
val last : regexp -> Cset.t
```

which calculates the set of last letters of recognized words.

Test with

```
let () =
  let ca = ('a', 0) and cb = ('b', 0) in
  let a = Character ca and b = Character cb in
  let ab = Concat (a, b) in
  let eq = Cset.equal in
  assert (eq (first a) (Cset.singleton ca));
  assert (eq (first ab) (Cset.singleton ca));
  assert (eq (first (Star ab)) (Cset.singleton ca));
  assert (eq (last b) (Cset.singleton cb));
  assert (eq (last ab) (Cset.singleton cb));
  assert (Cset.cardinal (first (Union (a, b))) = 2);
  assert (Cset.cardinal (first (Concat (Star a, b))) = 2);
  assert (Cset.cardinal (last (Concat (a, Star b))) = 2)
```

Exercise 3: The follow

Using the `first` and `last` functions , write a function

```
val follow : ichar -> regexp -> Cset.t
```

which calculates the set of letters that can follow a given letter in the set of recognized words.

Note that the letter `d` belongs to the set `follow c r` if and only if

- either there exists a subexpression of `r` of the form `r1 r2` with `d` element of `first r2` and `c` element of `last r1` ;

- either there exists a subexpression of `r` of the form `r1*` with `d` element of `first r1` and `c` element of `last r1`.

Test with

```
let () =
  let ca = ('a', 0) and cb = ('b', 0) in
  let a = Character ca and b = Character cb in
  let ab = Concat (a, b) in
  assert (Cset.equal (follow ca ab) (Cset.singleton cb));
  assert (Cset.is_empty (follow cb ab));
  let r = Star (Union (a, b)) in
  assert (Cset.cardinal (follow ca r) = 2);
  assert (Cset.cardinal (follow cb r) = 2);
  let r2 = Star (Concat (a, Star b)) in
  assert (Cset.cardinal (follow cb r2) = 2);
  let r3 = Concat (Star a, b) in
  assert (Cset.cardinal (follow ca r3) = 2)
```

Exercise 4: Construction of the automaton

To construct the deterministic finite state automaton corresponding to a regular expression `r`, we proceed as follows:

1. we add a new character `#` at the end of `r`;
2. the starting state is the set `first r`;
3. we have a transition from state `q` to state `q'` when reading the character `c` (this is a non-indexed letter) if `q'` is the union of all the `follow ci r` for all the elements `ci` of `q` such that `fst ci = c`;
4. accepting states are those that contain the `#` character .

Write a function

```
val next_state : regexp -> Cset.t -> char -> Cset.t
```

which calculates the resulting state of a transition.

To represent the finite automaton, we are given the following `autom` type :

```
type state = Cset.t  (* a state is a set of characters *)

module Cmap = Map.Make(Char) (* dictionary whose keys are characters *)
module Smap = Map.Make(Cset) (* dictionary whose keys are states *)

type autom = {
    start : state;
    trans : state Cmap.t Smap.t  (* state dictionary -> (character dictionary ->
    state) *)
}
```

We can choose to represent the # character in this way :

```
let eof = ('#', -1)
```

Write a function

```
val make_dfa : regexp -> autom
```

which constructs the automaton corresponding to a regular expression. The idea is to construct the states by necessity, starting from the initial state. For example, we could adopt the following approach:

```
let make_dfa r =
  let r = Concat (r, Character eof) in
  (* transitions under construction *)
  let trans = ref Smap.empty in
  let rec transitions q =
    (* the transitions function constructs all the transitions of the state q,
      if this is the first time q is visited *)
    ...
  in
  let q0 = first r in
  transitions q0;
  { start = q0; trans = !trans }
```

Note: it is of course possible to ultimately construct an automaton whose states are not sets but, for example, integers, for greater efficiency in the execution of the automaton. This could also be done during construction, or a posteriori. But this is not what interests us here.

Visualization with the `dot` tool

Here is some code to print an automaton in the input format of the tool `dot`:

```
let fprint_state fmt q =
  Cset.iter (fun (c,i) ->
    if c = '#' then Format.fprintf fmt "# " else Format.fprintf fmt "%c%i " c i) q

let fprint_transition fmt q c q' =
  Format.fprintf fmt "\"%a\" -> \"%a\" [label=\"%c\"];@\n"
    fprint_state q
    fprint_state q'
    c

let fprint_autom fmt a =
  Format.fprintf fmt "digraph A {@\n";
  Format.fprintf fmt "  @[\"%a\" [ shape = \"rect\"];@\n" fprint_state a.start;
  Smap.iter
    (fun q t -> Cmap.iter (fun c q' -> fprint_transition fmt q c q') t)
    a.trans;
  Format.fprintf fmt "@]@\n}@."

let save_autom file a =
  let ch = open_out file in
  Format.fprintf (Format.formatter_of_out_channel ch) "%a" fprint_autom a;
  close_out ch
```

To test, we take the example above:

```
(*  (a|b)*a(a|b)  *)
let r = Concat (Star (Union (Character ('a', 1), Character ('b', 1))),
                Concat (Character ('a', 2),
                        Union (Character ('a', 3), Character ('b', 2))))
let a = make_dfa r
let () = save_autom "autom.dot" a
```

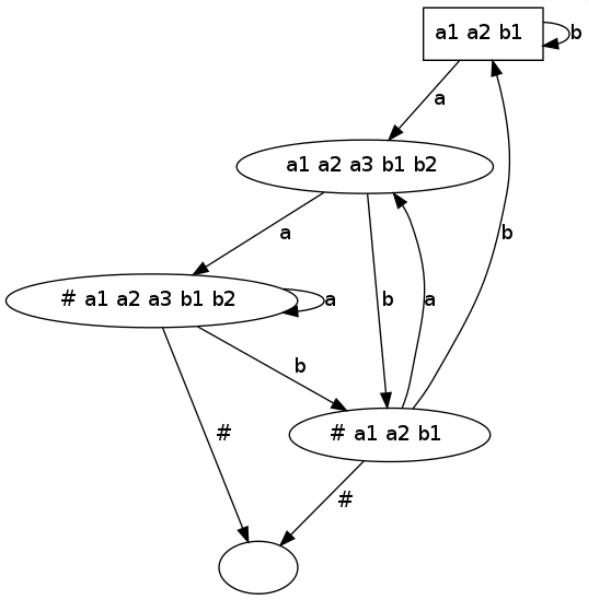
The execution produces an `autom.dot` file which can then be viewed with the Unix command

```
dotty autom.dot
```

or with one of the following two commands:

```
dot -Tps autom.dot | gv -
dot -Tpdf autom.dot > autom.pdf && evince autom.pdf
```

We should get something like the following figure



Exercise 5: Word recognition

Write a function

```
val recognize : autom -> string -> bool
```

which determines whether a word is recognized by an automaton.

Here are some positive tests:

```
let () = assert (recognize a "aa")
let () = assert (recognize a "ab")
let () = assert (recognize a "abababaab")
let () = assert (recognize a "babababab")
let () = assert (recognize a (String.make 1000 'b' ^ "ab"))
```

and some negative tests:

```
let () = assert (not (recognize a ""))
```

```
let () = assert (not (recognize a "a"))
let () = assert (not (recognize a "b"))
let () = assert (not (recognize a "ba"))
let () = assert (not (recognize a "aba"))
let () = assert (not (recognize a "abababaaba"))
```

Here is another test with a regular expression characterizing an even number of `b`'s :

```
let r = Star (Union (Star (Character ('a', 1)),
                     Concat (Character ('b', 1),
                             Concat (Star (Character ('a',2)),
                                     Character ('b', 2)))))
let a = make_dfa r
let () = save_autom "autom2.dot" a
```

Some positive tests:

```
let () = assert (recognize a "")
let () = assert (recognize a "bb")
let () = assert (recognize a "aaa")
let () = assert (recognize a "aaabbaaababaaa")
let () = assert (recognize a "bbbbbbbbbbbbbb")
let () = assert (recognize a "bbbbabbbbabbbabbb")
```

and some negative tests:

```
let () = assert (not (recognize a "b"))
let () = assert (not (recognize a "ba"))
let () = assert (not (recognize a "ab"))
let () = assert (not (recognize a "aaabbaaaaabaaa"))
let () = assert (not (recognize a "bbbbbbbbbbbbb"))
let () = assert (not (recognize a "bbbbabbbbabbbabbbb"))
```

Exercise 6: Generating a lexical analyzer

In this last question, we propose an automatic construction, from the automaton corresponding to a regular expression, an OCaml code which performs a lexical analysis, that is, which cuts a string of characters into the longest possible tokens.

More precisely, we will produce a code of the following form:

```
type buffer = { text: string; mutable current: int; mutable last: int }

let next_char b =
  if b.current = String.length b.text then raise End_of_file;
  let c = b.text.[b.current] in
  b.current <- b.current + 1;
  c

let rec state1 b =
  ...
and state2 b =
  ...
and state3 b =
```

```
  ...
```

The type `buffer` contains the string to be analyzed (`text` field), the position of the next character to be examined (`current` field), and the position following the last recognized token (`last` field). The function `next_char` returns the next character to be analyzed and increments the `current` field . If the end of the string is reached, it raises the `End_of_file` exception .

Each state of the automaton corresponds to a function `statei` with an argument `b` of type `buffer` . This function performs the following work:

1. If the state is accepting, then `b.last` is set to the value of `b.current`.
2. Then we call `next_char b` and examine its result. If it is a character for which a transition exists, then we call the corresponding function. Otherwise, we raise an exception (for example, with `failwith "lexical error"`).

Note that the `statei` functions do not return anything. They can only terminate on an exception (either `End_of_file` to mean that the end of the string has been reached, or `Failure` to mean a lexical error).

Write a function

```
val generate: string -> autom -> unit
```

which takes as arguments the name of a file and an automaton and produces in this file the OCaml code corresponding to this automaton, according to the form above. Indications:

- We can start by numbering all the states, for example by constructing a dictionary of type `int Smap.t` which associates a unique number with each state.
- We can draw inspiration from the code of the `save_autom` function given above for what concerns the display in a file.

Note: It will be useful to add a last line of the form to the product code

```
let start = state42
```

corresponding to the initial state of the automaton.

To test, write (in another `lexer.ml` file, this time written by hand) a program that cuts a string into tokens using the automatically produced code (by fixing the name of the file, for example `a.ml`). The principle is a loop that performs the following actions:

1. we set the `last` field to `-1` ;
2. we call the `start` function and we catch the exception `e` that it throws
3. if the `last` field is still `-1`, then no token has been recognized and we end by re-throwing the exception `e` ;
4. otherwise, we display the `token` that was recognized and we assign the value of the `last` field to the `current` field .

Be careful to handle program termination correctly.

We can test for example with the regular expression `a*b` , by executing

```
let r3 = Concat (Star (Character ('a', 1)), Character ('b', 1))
let a = make_dfa r3
let () = generate "a.ml" a
```

then linking the product code with the `lexer.ml` file :

```
% ocamlopt a.ml lexer.ml
```

On the string `abbaaab`, the analysis must produce three tokens and succeed:

```
--> "ab"
--> "b"
--> "aaab"
```

On the string `aba` , the analysis should produce a first token and then fail:

```
--> "ab"
exception End_of_file
```

Finally, on the string `aac`, the analysis should fail on a lexical error:

```
exception Failure("lexical error")
```

We can also test with the regular expression `(`bespilon)(ab)*(a—epsilon)— of words alternating the letters `a` and `b`. On the string `abbac` , we should obtain three tokens:

```
--> "ab"
--> "ba"
--> ""
would now loop
```

The last token being the empty string, the program stops, meaning that we would now obtain this empty token indefinitely.

This algorithm is known as the Berry-Sethi algorithm. It is notably described in Dragon (Aho, Sethi, Ullman, Compilers ...), section 3.9.