L28: Advanced functional programming

Exercise 3

Due on 24th April 2018

This exercise uses the BER MetaOCaml compiler and several packages. You can install both the compiler and the packages using OPAM:

```
opam remote add advanced-fp https://github.com/ocamllabs/advanced-fp-repo.git
opam switch 4.04.0+BER
eval $(opam config env)
opam install letrec re core_bench
```

The file `README.md` in the accompanying `zip` archive contains further instructions for building and running the code.

Submission instructions

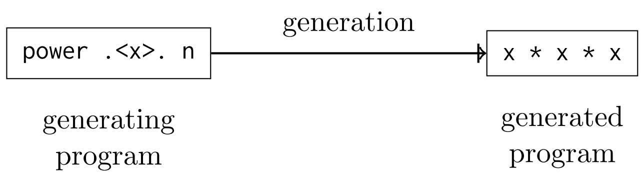
Your solutions for this exericse should be handed in to the Graduate Education Office by 4pm on the due date. Additionally, please email your completed code as a file `exercise3.zip` to `jeremy.yallop@cl.cam.ac.uk`.

Changelog

14:00 Thu 29th Mar	Corrected example in section 2 `(ab|ac)d` previously read `(ab|ac)c*d`.
17:00 Mon 9th Apr	Corrected benchmark `Re_staged_tests.test_a_star_a` previously read `Re_staged_tests.test_ab_star_c`
01:00 Wed 18th Apr	Added missing parentheses to example code in Q1(b)

1 Tests of character: static analysis

A multi-stage program runs in two (or more) stages. The first stage, a *generating program* produces as output a *generated program* that runs in the second stage. For example, the staged power function examined in the lectures is a generating program; it generates a program that performs a sequence of multiplications.



The values in a multi-stage program are divided into two classes: **static** values (such as `n` above) are available to the generating program, while **dynamic** values (such as `x`) are known only to the generated program.

However, it is often possible to determine various facts about dynamic values during code generation. For example, consider the following fragment (which might have been generated by combining smaller fragments):

```
.< if b < 20
   then if b >= 30 || b < 10 then w
        else x
   else if b >= 15 then y
        else z >.
```

Although `b` is a dynamic variable whose value is unknown, it is possible to deduce the outcome of some of the tests involving `b`. In particular, the following code is only executed if `b < 20`:

```
        if b >= 30 || b < 10 then w
        else x
```

and so the condition `b >= 30` can never succeed.

Similarly, the following code is only executed if `b >= 20`:

```
        if b >= 15 then y
        else z
```

and so the condition `b >= 15` can never fail. Combining these insights, we can simplify the code to eliminate redundant checks:

```
.< if b < 20
   then if b < 10 then w
        else x
   else y >.
```

This question involves building structures that keep track of the possible values of dynamic variables in order to eliminate branches and simplify tests.

(Add your answers to this question to `char_test.ml`)

(*a*) The function `mem` checks whether a character is a member of a set:

```
val mem : char -> CharSet.t -> bool
```

Define a staged version of `mem` with a dynamic character and a static set argument:

```
val mem_static : char code -> CharSet.t -> bool code
```

For example, the following program

```
.< fun c -> .~(mem_static .<c>.
                   CharSet.of_list ['a'; 'b'; 'c'; '1'; '2'; '3'])>.
```

might generate this output:

```
.< fun c -> c = 'a' || c = 'b' || c = 'c'
         || c = '1' || c = '2' || c = '3' >.
```

(*b*) Comparing the character to each member of a set individually is inefficient. Build a second staged implementation of `mem`, `mem_static_interval`, that generates a set of interval tests instead. For example, the following program

```
.<fun c -> .~(mem_static_interval .<c>.
                (CharSet.of_list ['a'; 'b'; 'c'; '1'; '2'; '3']))>.
```

might generate this output:

```
.< fun c -> ('a' <= c && c <= 'c')
         || ('1' <= c && c <= '3') >.
```

(*c*) Define a module `PSChar` with the following signature:

```
module PSChar : sig
 type ps
 (* The type of partially-static characters *)

 val inj : char code -> ps
 (* Make a partially-static character from a dynamic character *)

 val in_range : ps -> char*char ->
   (ps -> 'a code) -> (ps -> 'a code) -> 'a code
 (* Test whether a character lies within a range *)
end
```

where `in_range` is a staged version of the following code:

```
let in_range c (l,h) k1 k2 =
  if l <= c && c <= h then k1 c else k2 c
```

with the following behaviour:

- Each call to `in_range` extends the partially-static representation with knowledge about possible values. The argument passed to `k1` incorporates the knowledge that `c` lies in the range `l`...`h`; the argument passed to `k2` incorporates the knowledge that `c` does not lie in the range `l`...`h`.

- Branches are omitted where possible: if it can be determined that the character `c` does not lie in the range `l`...`h` then `k1` is not called; if it can be determined that the character `c` certainly lies in the range `l`...`h` then `k2` is not called.

- Interval tests should be reduced or eliminated where possible. For example, if `c` is already known to lie in the range `'e'..'j'` then

  ```
  in_range c ('a','g') k1 k2
  ```

  might generate code with the following condition

  ```
  if c <= 'g' then ...
  ```

  since `c >= 'a'` must always be true.

One reasonable representation of `PSChar.ps` is a pair of a dynamic variable and a set of possible values for the variable:

```
type ps = char code * CharSet.t
```

However, you may use any representation you choose.

(*d*) Define a final staged version of `mem mem_ps` based on `PSChar` that (like `mem_static_interval`) generates interval tests, and that eliminates or simplifies tests where possible:

```
val mem_ps : PSChar.ps -> CharSet.t ->
  (PSChar.ps -> 'a code) -> (PSChar.ps -> 'a code) -> 'a code
```

For example, the following program

```
.< fun c -> .~(mem_ps (PSChar.inj .<c>.)
                 (CharSet.of_list ['a';'b';'c';'d';'e';'f'])
                 (fun p -> mem_ps p
                     (CharSet.of_list ['b';'d';'e';'f'])
                     (fun _ -> .<"w">.)
                     (fun _ -> .<"x">.))
                 (fun p -> mem_ps p
                     (CharSet.of_list ['c';'d';'e'])
                     (fun _ -> .<"y">.)
                     (fun _ -> .<"z">.))) >.
```

which is intended to behave similarly to the following pseudo-code

```
fun c -> if c ∈ {'a','b','c','d','e','f'}
         then if c ∈ {'b','d','e','f'} then "w" else "x"
         else if c ∈ {'c','d','e'} then "y" else "z"
```

might generate code like this:

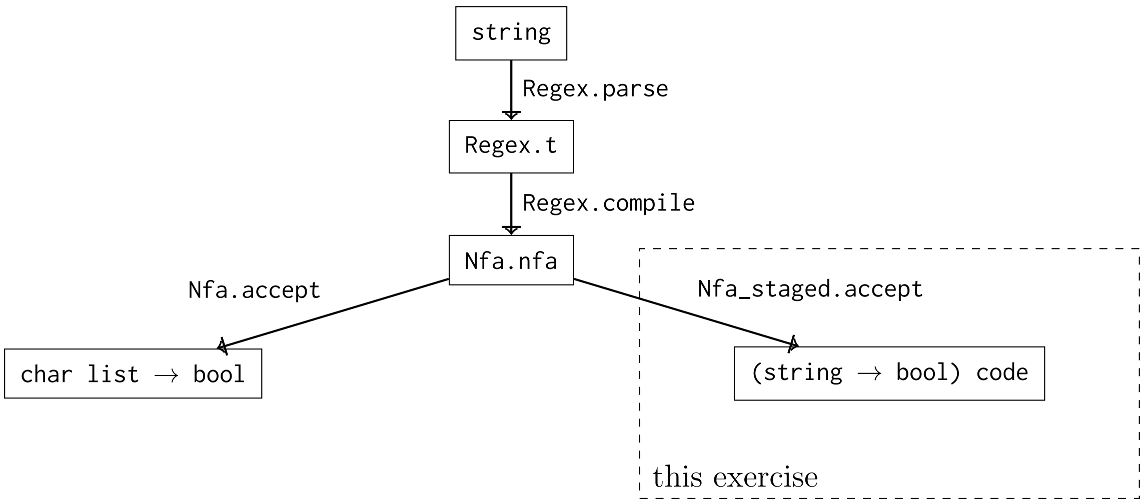
```
.< fun c  ->
    if 'a' <= c && c <= 'f'
    then if c = 'b' then "w"
         else if 'd' <= c then "w" else "x"
    else "z">.
```

2 Regular expressions and finite automata

Functions that match strings against regular expressions (regexes) are a good fit for multi-stage programming for several reasons. First, since the regex is typically available before the string, there is a natural division of the matching function into stages. Second, careful analysis of regexes can significantly improve performance. Third, a typical program reuses the same regex many times with different strings, so performing extra work during code generation in order to produce more efficient code is often worthwhile.

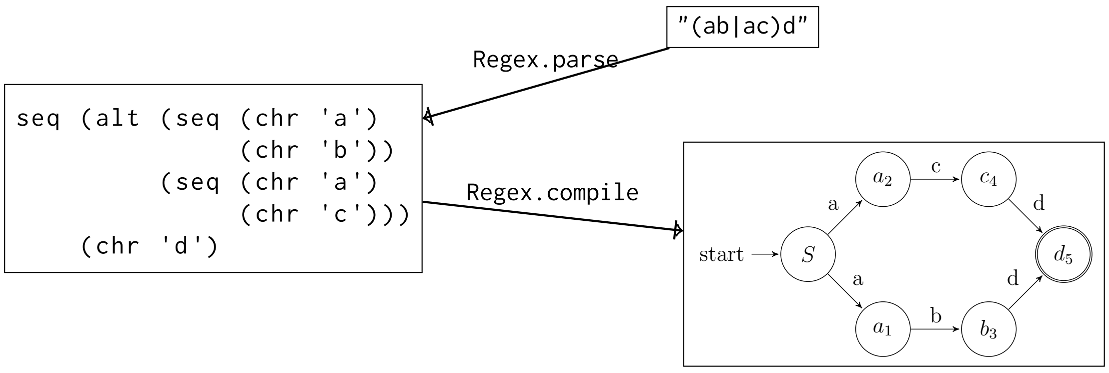
This section of the exercise involves staging a naive regex library to produce code that is competitive with (or even faster than) the widely-used `ocaml-re` library.

The following diagram shows the main components of the library:



The main two functions of interest are `Regex.compile`, which converts a regex to an equivalent Non-deterministic Finite Automaton (NFA), and `Nfa.accept`, which tests whether a sequence of characters is acccepted by an NFA.

The following diagram shows how the regex `"(ab|ac)d"` is first turned into a tree representation by `Regex.parse`, then into an NFA by `Regex.compile`:

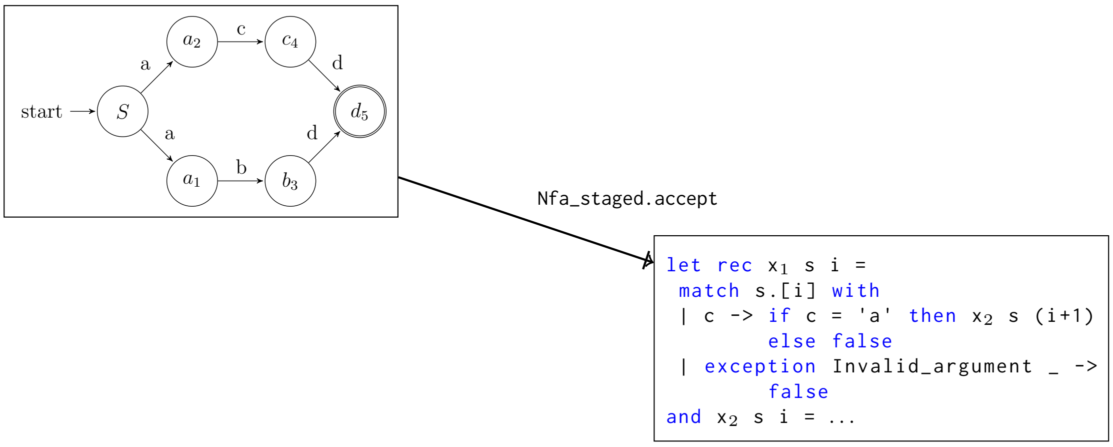


```
seq (alt (seq (chr 'a')
              (chr 'b'))
         (seq (chr 'a')
              (chr 'c')))
    (chr 'd')
```

The final step is to run the NFA against a character sequence using `NFA.accept` to determine whether the character sequence matches the original regex (`"(ab|ac)d"`). The `NFA.accept` function uses the approach described in Wikipedia:

> Keep a set data structure of all states which the NFA might currently be in. On the consumption of an input symbol, unite the results of the transition function applied to all current states to get the set of next states. On the consumption of the last input symbol, if one of the current states is a final state, the machine accepts the string.

The questions that follow focus on this final step. Staging the naively-implemented `NFA.accept` function can bring improvements of several orders of magnitude.



```
let rec x₁ s i =
 match s.[i] with
 | c -> if c = 'a' then x₂ s (i+1)
        else false
 | exception Invalid_argument _ ->
        false
and x₂ s i = ...
```

(Add your answers to this question to `nfa_staged.ml`)

(*a*) *(Staging the naive interpreter)*

The first task is to implement a staged version of the `Nfa.accept` function using staging annotations and the `letrec` function. There are two steps:

(*i*) First, define a function `splitc` that turns a dynamic character (type `char code`) into a static character (type `char`) that is passed to the second argument:

```
val splitc : char code -> (char -> 'a code) -> 'a code
```

(*ii*) Using `splitc` and `letrec`, define the function `Nfa_staged.accept`, starting from the definition of the unstaged `Nfa.accept` function.

The staged `accept` function should use a set of states, `StateSet.t`, as the index type. With a state set as the index type the generated code contains a single function for each of the possible sets of states that an NFA can be in; although the input machine is nondeterministic, the generated code behaves deterministically.

You can test your code in the top-level:

```
# let nfa = (Regex.compile (Regex.parse "(ab)|(ac)"));;
val nfa : Nfa.nfa = {Nfa.start = 0l; finals = <abstr>; next = <fun>}
# let code = Nfa_staged.accept nfa ;;
val code : (char list -> bool) code = .<
let rec x1 = function
  | [] -> false
  | c::cr -> (match c with
              | '\000' -> x3 cr
              | '\001' -> x3 cr
              | ...)
and x2 = ...

# let accept = Runcode.run code;;
val accept : char list -> bool = <fun>
# accept ['a'; 'b'];;
- : bool = true
# accept ['b'; 'a'];;
- : bool = false
```

and using the test suite (type `make test`).

(*b*) *(Improvement: removing redundant branches)*

The size of the code generated by `splitc` can be reduced by using the information available in the NFA representation about the possible transitions from the current states.

(*i*) Write a function `splitc2`:

```
    val splitc2 : Nfa.nfa -> StateSet.t ->
                  char code -> (char -> bool code) -> bool code
```

that generates code that only branches on characters with valid transitions from the current state, returning `false` otherwise.

For example:

```
# let nfa = Regex.compile (Regex.parse "(ab)|c");;
val nfa : Nfa.nfa = {Nfa.start = 0l; finals = <abstr>; next = <fun>}
# let start = Nfa.StateSet.singleton nfa.Nfa.start;;
val start : Nfa.StateSet.t = <abstr>
# .< fun c -> .~(Nfa_staged.splitc2 nfa start .<c>. (fun _ -> .<true>.))>.;;
- : (Nfa_staged.CharSet.elt -> bool) code = .<
fun c  -> if c = 'c' then true else if c = 'a' then true else false>.
```

(*ii*) Using `splitc2`, define a function `Nfa_staged.accept2` that generates more compact code than `Nfa.accept`.

For example:

```
# Nfa_staged.accept2 (Regex.compile (Regex.parse "a*b"));;
- : (Nfa_staged.CharSet.elt list -> bool) code = .<
let rec x1 = function
 | [] -> false
 | c::cr -> if c = 'b' then x3 cr
            else if c = 'a' then x2 cr else false
and x2 = function
  | [] -> false
  | c::cr -> if c = 'b' then x3 cr
             else if c = 'a' then x2 cr else false
```

```
and x₃ = function
 | [] -> true
 | _::_ -> false
 in x₁>.
```

(*c*) *(Improvement: using strings rather than lists)*

The `accept` functions developed so far operate on lists of characters. However, a function that accepts strings instead provides a more convenient (and perhaps more efficient) interface.

Write a function `accept3` with the following signature:

```
    val accept3 : nfa -> (string -> bool) code
```

using your `accept2` from question (*b*) as a starting point.

For example:

```
# Nfa_staged.accept3 (Regex.compile (Regex.parse "a*b"));;
- : (Nfa_staged.CharSet.elt list -> bool) code = .<
let rec x₁ s i = match s[i] with
 | exception Invalid_argument _ -> false
 | c -> if c = 'b' then x₃ s (i+1)
        else if c = 'a' then x₂ s (i+1) else false
and x₂ s i = match s.[i] with
 | exception Invalid_argument _ -> false
 | c -> if c = 'b' then x₃ s (i+1)
        else if c = 'a' then x₂ s (i+1) else false
and x₃ s i = match s.[i] with
 | exception Invalid_argument _ -> true
 | c -> false
 in fun s -> x₁ s 0>.
```

(*d*) *(Improvement: using more efficient tests)*

(*i*) The `splitc2` function (question (*b*)(*i*)) suffers from various inefficiencies: characters are matched individually, which can lead to a slow linear search where the number of valid transitions is large, and the continuation function is invoked once for each matching character, leading to large code for certain regexes. For example, the following call may (depending on the exact implementation of `accept3`) generate extremely large and slow code:

```
Nfa_staged.accept3 (Regex.compile (Regex.parse ".a"))
```

Use your implementation of `mem_ps` from question 1 to write a function `splitc3` with the following signature:

```
let splitc3 : nfa -> StateSet.t ->
   char code -> (StateSet.t -> bool code) -> bool code
```

that generates smaller faster code without linear searches, and that coalesces calls to the next function where possible. For example, while the following call using `splitc2` inserts `true` three times, once for each of `a`, `b`, and `c`,

```
# let nfa = Regex.compile (Regex.parse "a|b|c");;
val nfa : Nfa.nfa = {Nfa.start = 01; finals = <abstr>; next = <fun>}
# .< fun c -> .~(Nfa_staged.splitc2 nfa
                  (Nfa.StateSet.singleton nfa.Nfa.start)
                  .<c>. (fun _ -> .<true>.)) >.;;
- : (Nfa_staged.CharSet.elt -> bool) code = .<
fun c  ->
  if c = 'c'
  then true
  else if c = 'b' then true else if c = 'a' then true else false>.
```

a similar call to `splitc3` inserts a single instance of `true`:

```
# .< fun c -> .~(Nfa_staged.splitc3 nfa
                  (Nfa.StateSet.singleton nfa.Nfa.start)
                  .<c>. (fun _ -> .<true>.)) >.;;
- : (char -> bool) code = .<
fun c  -> if ('a' <= c) && (c <= 'c') then true else false>.
```

(ii) Write a final staged NFA interpreter, `accept4`, that uses `splitc3` to generate efficient, compact code. For example:

```
# Nfa_staged.accept4 (Regex.compile (Regex.parse ".a"));;
- : (string -> bool) code = .<
let rec x₁ s i = match s.[i] with
  | c -> x₂ s (i + 1)
  | exception Invalid_argument _ -> false
and x₂ s i =
  match s.[i] with
  | c -> if c = 'a' then x₃ s (i + 1) else false
  | exception Invalid_argument _ -> false
and x₃ s i = match s.[i] with
  | c -> false
  | exception Invalid_argument _ -> true
 in fun s  -> x₁ s 0>.
```

You may like to use the benchmarks in `benchmark.ml` to compare the performance of the code generated by `accept4` to the unstaged NFA interpreter and to the `ocaml-re` library. (Type `make bench` to run the benchmarks.)