# *Principles of Programming Languages, 2023.01.25*

**Important notes**
- Total available time: 2h.
- You may use any written material you need, and write in English or in Italian.
- You cannot use electronic devices during the exam: every phone must be turned off and kept on your table.
- You cannot use library functions not covered in class in your code.

## Exercise 1, Scheme (10 pts)

We want to implement a *for-each/cc* procedure which takes a condition, a list and a body and performs a for-each. The main difference is that, when the condition holds for the current value, the continuation of the body is stored in a global queue of continuations. We also need an auxiliary procedure, called *use-cc*, which extracts and call the oldest stored continuation in the global queue, discarding it.

E.g. if we run:

```
(for-each/cc odd?
             '(1 2 3 4)
             (lambda (x) (displayln x)))
```

two continuations corresponding to the values 1 and 3 will be stored in the global queue.

Then, if we run: `(use-scc),` we will get on screen:

```
2
3
4
```

## Exercise 2, Haskell (11 pts)

We want to define a data structure for the tape of a Turing machine: *Tape* is a parametric data structure with respect to the tape content, and must be made of three components:

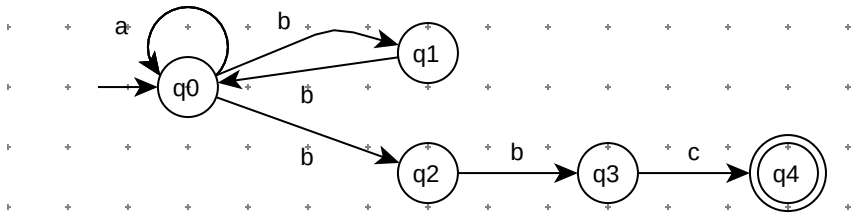
1. the portion of the tape that is on the left of the head;
2. the symbol on which the head is positioned;
3. the portion of the tape that is on the right of the head.

Also, consider that the machine has a concept of "blank" symbols, so you need to add another component in the data definition to store the symbol used to represent the blank in the parameter type.

1. Define Tape.
2. Make Tape an instance of Show and Eq, considering that two tapes contain the same values if their stored values are the same and in the same order, regardless of the position of their heads.
3. Define the two functions *left* and *right*, to move the position of the head on the left and on the right.
4. Make Tape an instance of Functor and Applicative.

## Exercise 3, Erlang (11 pts)

Consider the following non-deterministic finite state automaton (FSA):




Write a concurrent Erlang program that simulates the previous FSA, where each state is implemented as a process.

# Solutions

**Ex 1**

```
(define *scc* '())

(define (use-cc)
  (when (cons? *scc*)
    (let ((c (car *scc*)))
      (set! *scc* (cdr *scc*))
      (c))))

(define (for-each/cc cnd L body)
  (when (cons? L)
    (let ((x (car L)))
      (call/cc (lambda (c)
                 (when (cnd x)
                   (set! *scc* (append *scc* (list c))))
                 (body x)))
      (for-each/cc cnd (cdr L) body))))
```

**Ex 2**

```
data Tape a = Tape [a] a [a] a -- the last one stands for 'blank'

instance Show a => Show (Tape a) where
    show (Tape x c y b) = show (reverse x) ++ (show c) ++ show y

instance Eq a => Eq (Tape a) where
    (Tape x c y _) == (Tape x' c' y' _) = (x ++ [c] ++ y) == (x' ++ [c'] ++ y')

left :: Tape a -> Tape a
left (Tape [] c y b) = Tape [] b (c:y) b
left (Tape (x:xs) c y b) = Tape xs x (c:y) b

right :: Tape a -> Tape a
right (Tape x c [] b) = Tape (c:x) b [] b
right (Tape x c (y:ys) b) = Tape (c:x) y ys b

instance Functor Tape where
    fmap f (Tape x c y b) = Tape (fmap f x) (f c) (fmap f y) (f b)

instance Applicative Tape where
    pure x = Tape [] x [] x
    -- zipwise apply
    (Tape fx fc fy fb) <*> (Tape x c y b) = Tape (zipApp fx x) (fc c) (zipApp fy y) (fb b)
        where zipApp x y = [f x | (f,x) <- zip x y]
```

**Ex 3**

```
q0() ->
    receive
        {S, [b|Xs]} -> q1 ! {S, Xs}, q2 ! {S, Xs};
        {S, [a|Xs]} -> q0 ! {S, Xs}
    end, q0().

q1() ->
    receive
        {S, [b|Xs]} -> q0 ! {S, Xs}
    end, q1().

q2() ->
    receive
        {S, [b|Xs]} -> q3 ! {S, Xs}
    end, q2().

q3() ->
    receive
        {S, [c|Xs]} -> q4 ! {S, Xs}
    end, q3().

q4() ->
    receive
        {S, []} -> io:format("~w accepted~n", [S])
    end, q4().

start() ->
    register(q0, spawn(fun() -> q0() end)), % to avoid exporting qs
    register(q1, spawn(fun() -> q1() end)),
    register(q2, spawn(fun() -> q2() end)),
    register(q3, spawn(fun() -> q3() end)),
    register(q4, spawn(fun() -> q4() end)).

read_string(S) ->
    q0 ! {S, S}, ok.
```