Principles of Programming Languages, 2023.02.15

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)

Consider the following *For* construct, as defined in class:

```
(define-syntax For
  (syntax-rules (from to break: do)
    ((_ var from min to max break: break-sym
        do body ...)
     (let* ((min1 min)
            (max1 max)
            (inc (if (< min1 max1) + -)))
       (call/cc (lambda (break-sym)
                  (let loop ((var min1))
                    body ...
                    (unless (= var max1)
                      (loop (inc var 1)))))))))
```

Define a fix to the above definition, to avoid to introduce in the macro definition the special break symbol *break-sym*, by providing a construct called *break.* E.g.

```
(For i from 1 to 10
      do
      (displayln i)
      (when (= i 5)
        (break #t)))
```

will return #t after displaying the numbers from 1 to 5.

Exercise 2, Haskell (11 pts)

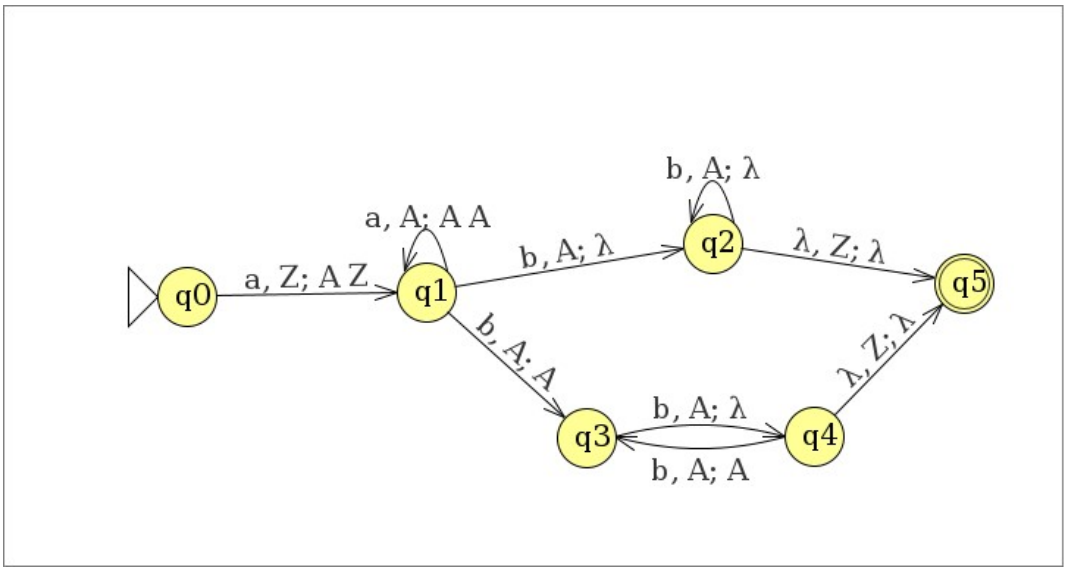
We want to define a data structure for binary trees, called *BBtree*, where in each node are stored two values of the same type. Write the following:

1. The *BBtree* data definition.
2. A function *bb2list* which takes a *BBtree* and returns a list with the contents of the tree.
3. Make *BBtree* an instance of Functor and Foldable.
4. Make *BBtree* an instance of Applicative, using a “zip-like” approach, i.e. every function in the first argument of <*> will be applied only once to the corresponding element in the second argument of <*>.
5. Define a function *bbmax*, together with its signature, which returns the maximum element stored in the *BBtree*, if present, or *Nothing* if the data structure is empty.

Exercise 3, Erlang (11 pts)

Consider the following non-deterministic pushdown automaton (PDA), where *Z* is the initial stack symbol and λ

represents the empty string:



Write a concurrent Erlang program that simulates only the given PDA, and each state of the PDA is implemented as an independent parallel process.

Note: multichance students do not need to solve the last exercise.

Solutions

Ex 1

```
(define exit-store '())
(define (break v)
  ((car exit-store) v))

(define-syntax For+
  (syntax-rules (from to do)
    ((_ var from min to max
        do body ...)
     (let* ((min1 min)
            (max1 max)
            (inc (if (< min1 max1) + -)))
       (let ((v (call/cc (lambda (break)
                           (set! exit-store (cons break exit-store))
                           (let loop ((var min1))
                             body ...
                             (unless (= var max1)
                               (loop (inc var 1))))))))
         (set! exit-store (cdr exit-store))
         v)))))
```

Ex 2

```
data BBtree a = BBnil | BBtree (BBtree a) a a (BBtree a) deriving (Eq, Show)

bbleaf x y = (BBtree BBnil x y BBnil)

bb2list BBnil = []
bb2list (BBtree t1 x y t2) = (bb2list t1) ++ [x,y] ++ (bb2list t2)

instance Functor BBtree where
  fmap f BBnil = BBnil
  fmap f (BBtree t1 x y t2) = BBtree (fmap f t1) (f x) (f y) (fmap f t2)

instance Foldable BBtree where
  foldr f i BBnil = i
  foldr f i (BBtree t1 x y t2) = foldr f (f x (f y (foldr f i t2))) t1

instance Applicative BBtree where
  pure x = bbleaf x x
  BBnil <*> y = BBnil
  x <*> BBnil = BBnil
  (BBtree t1 x y t2) <*> (BBtree t1' x' y' t2') =
    (BBtree (t1 <*> t1') (x x') (y y') (t2 <*> t2'))

bbmax :: (Ord a) => BBtree a -> Maybe a
bbmax BBnil = Nothing
bbmax t@(BBtree t1 x y t2) = Just $ foldr max x t

-- Note: it can be done more easily with
-- maximum :: (Foldable t, Ord a) => t a -> a
-- which is already provided by foldable:
bbmaxf BBnil = Nothing
bbmaxf t = Just $ maximum t
```

Ex 3

```
q0() ->
    receive
        {S, [a|Xs], [z|T]} -> q1 ! {S, Xs, [a,z] ++ T}
    end,
    q0().

q1() ->
    receive
        {S, [a|Xs], [a|T]} -> q1 ! {S, Xs, [a,a] ++ T};
        {S, [b|Xs], [a|T]} -> q2 ! {S, Xs, T}, q3 ! {S, Xs, [a|T]}
    end,
    q1().

q2() ->
    receive
        {S, [b|Xs], [a|T]} -> q2 ! {S, Xs, T};
        {S, Xs, [z|T]} -> q5 ! {S, Xs, T}
    end,
    q2().

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

q4() ->
```

```
    receive
        {S, [b|Xs], [a|T]} -> q3 ! {S, Xs, [a|T]};
        {S, Xs, [z|T]} -> q5 ! {S, Xs, T}
    end,
    q4().

q5() ->
    receive
        {S, [], _} -> io:format("~w accepted~n", [S])
    end,
    q5().

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)),
    register(q5, spawn(fun() -> q5() end)).

stop() ->
   unregister(q0),
   unregister(q1),
   unregister(q2),
   unregister(q3),
   unregister(q4),
   unregister(q5).

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