Principles of Programming Languages, 2024.01.11

Important notes
- Total available time: 2h (*multichance* students do not need to solve Exercise 3).
- 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 (11 pts)

Consider the *for* with *break* as seen in class and reported here for your convenience:

```
(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 (k)
                    (set! *exit-store* (cons k *exit-store*))
                    (let loop ((var min1))
                      body ...
                      (unless (= var max1)
                        (loop (inc var 1))))))))
         (set! *exit-store* (cdr *exit-store*))
         v)))))
```

Define an extension of this construct, to be able to use in it also a *continue* command, with the same semantics as in C and Java, clearly explaining your idea.

Exercise 2, Haskell (11 pts)

Consider the following type of expressions, containing variables of some type *a*, constants that are integers, and a some kind of binary operator called *Op*.

```
data Expr a = Var a | Const Int | Op (Expr a) (Expr a)
```

1) Make it an instance of Functor, Applicative, and Monad.

2) Using an example, show what the >>= operator does in your implementation.

Exercise 3, Erlang (11 pts)

Define a *condition-var-manager* process that receives a list of initial values and a list of the corresponding conditions (unary predicates), both of the same length, and spawns for each ordered pair of value and condition a process that updates its value only when the new value satisfies the condition.
The manager can receive the following messages:
- *{update, F}* where F is a unary function that must be applied to all the saved values to obtain the new values (only if the corresponding condition holds);
- *print* which makes each process print its current value;
- *stop* which stops the manager and all its spawned processes.

Solutions

Ex 1

The main idea is to use also a continuation inside the loop: the outer continuation is used for the break, while the inner covers the continue. Of course, we need to add another store for the inner continuations.

```
(define *cont-store* '())
(define (continue)
  ((car *cont-store*)))
(define (break v)
  (set! *cont-store* (cdr *cont-store*))
  ((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 (k)
                    (set! *exit-store*
                      (cons k *exit-store*))
                    (let loop ((var min1))
                      (call/cc
                        (lambda (cont)
                          (set! *cont-store*
                            (cons cont *cont-store*))
                          body ...))
                      (set! *cont-store* (cdr *cont-store*))
                      (unless (= var max1)
                        (loop (inc var 1))))))))
         (set! *exit-store* (cdr *exit-store*))
         v)))))
```

Ex 2

```
instance Functor Expr where
  fmap _ (Const x) = Const x
  fmap g (Var x) = Var (g x)
  fmap g (Op a b) = Op (fmap g a) (fmap g b)

instance Applicative Expr where
  pure = Var
  _ <*> Const x = Const x
  Const x <*> _ = Const x
  Var f <*> Var x = Var (f x)
  Var f <*> Op x y = Op (fmap f x) (fmap f y)
  Op f g <*> x = Op (f <*> x) (g <*> x)

instance Monad Expr where
  Const x >>= _ = Const x
  Var x >>= f = f x
  Op a b >>= f = Op (a >>= f) (b >>= f)
```

Example:

```
 Var 3 >>= \x -> Var (x*2)
 result: Var 6

 Op (Var 1) (Var 2) >>= \x -> Var (x + 1)
 result: Op (Var 2) (Var 3)
```

Ex 3

```
condvar_mgr(Vs, Conds) ->
    Pids = [
        spawn_link(fun() ->
            condvar(V, C)
        end)
     || {V, C} <- lists:zip(Vs, Conds)
    ],
    condvar_mgr_loop(Pids).

condvar_mgr_loop(Pids) ->
    receive
        stop ->
            exit(ok);
        Msg ->
            [Pid ! Msg || Pid <- Pids],
            condvar_mgr_loop(Pids)
    end.

condvar(V, C) ->
```

```
receive
    {update, Fn} ->
        V1 = Fn(V),
        case C(V1) of
            true -> condvar(V1, C);
            false -> condvar(V, C)
        end;
    print ->
        io:format("~p~n", [V]),
        condvar(V, C)
end.
```