Principles of Programming Languages, 2024.02.02

Important notes
- Total available time: 2h (*multichance* students do not need to solve Exercise 1).
- 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 following data structure, written in Haskell:

```
data Expr a = Var a | Val Int | Op (Expr a) (Expr a)
instance Functor Expr where
  fmap _ (Val x) = Val 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
  _ <*> Val x = Val x
  Val x <*> _ = Val 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
  Val x >>= _ = Val x
  Var x >>= f = f x
  Op a b >>= f = Op (a >>= f) (b >>= f)
```

Define an analogous in Scheme, with all the previous operations, where the data structures are encoded as lists – e.g. `Op (Val 0) (Var 1)` is represented in Scheme as `'(Op (Val 0) (Var 1))`.

Exercise 2, Haskell (11 pts)

Consider the following datatype definition.

```
data F b a = F (b -> b) a | Null
```

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

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

Exercise 3, Erlang (11 pts)

Consider a list *L* of tasks, where each task is encoded as a function having only one parameter: a PID *P*. When a task is called, it runs some operations and then sends back the results to *P,* in this form: *{result, <Task_PID>, <Result_value>}*; a task could also fail for some errors.
Define a server which takes *L* and runs in parallel all the tasks in it, returning the list of the results (the order is not important). Note: in case of failure, every task should be restarted only once; if it fails twice, its result should be represented with the atom *bug*.
Utilities: you can use from the standard libraries the following functions: *maps:from_list(<List of {Key, Value}>)*, which takes a list of pairs and builds the corresponding map, and *maps:values(<Map>)*, which is its inverse.

Solutions

Ex 1

```
;; Constructors
(define (var x) (list 'Var x))
(define (val x) (list 'Val x))
(define (op x y) (list 'Op x y))

;; predicates
(define (var? x) (eq? 'Var (car x)))
(define (val? x) (eq? 'Val (car x)))
(define (op? x) (eq? 'Op (car x)))

;; Functor
(define (fmap f e)
  (cond
    ((val? e) e)
    ((var? e)
          (let ((x (cadr e)))
            (var (f x))))
    ((op? e)  (let ((x (cadr e))
                 (y (caddr e)))
            (op (fmap f x) (fmap f y))))))

;; Applicative
(define pure var)
(define (<*> x y)
  (cond
    ((val? y) y)
    ((val? x) x)
    ((var? x)
          (let ((f (cadr x)))
            (if (var? y)
              (var (f (cadr y)))
              (let ((a (cadr y))
                    (b (caddr y)))
                (op (fmap f a)(fmap f b))))))
    ((op? x) (let ((a (cadr x))
                (b (caddr x)))
           (op (<*> a y)(<*> b y))))))

;; Monad
(define (>>= x y)
  (cond
    ((val? x) x)
    ((var? x) (y (cadr x)))
    ((op? x) (let ((a (cadr x))
                (b (caddr x)))
           (op (>>= a y)(>>= b y))))))
```

Ex 2

Note: This data structure is basically a combination of State and Maybe, where the State pair is brought out of the function.

```
instance Functor (F x) where
  fmap f (F g t) = F g (f t)
  fmap _ Null = Null

instance Applicative (F x) where
  pure = F id
  Null <*> _ = Null
  _ <*> Null = Null
  (F f x) <*> (F g y) = F (f . g) (x y)

instance Monad (F x) where
  Null >>= _ = Null
  F f x >>= g = case g x of
                Null -> Null
                F f' x' -> F (f . f') x'

-- Example
runit (F f x) s = (f s, x)
ex = F (\x -> 2*x) 5 >>= \x -> pure (x+1)
runit ex 1  -- result: (2,6)
```

Ex 3

```
getres(Fs) ->
    Self = self(),
    process_flag(trap_exit, true),
    Pids = maps:from_list([{spawn_link(fun() -> F(Self) end), {F, wait}} || F <- Fs]),
    getres_loop(Pids, length(Fs)).

getres_loop(Pids, 0) ->
    [R || {done, R} <- maps:values(Pids)];
getres_loop(Pids, Waiting) ->
    receive
        {result, Pid, R} ->
            getres_loop(Pids#{Pid := {done, R}}, Waiting - 1);
        {'EXIT', Pid, Reason} when Reason /= normal ->
            #{Pid := {F, Status}} = Pids,
            Self = self(),
            case Status of
                restart ->
                    getres_loop(Pids#{Pid := {done, bug}}, Waiting - 1);
                _ ->
                    NewPid = spawn_link(fun() -> F(Self) end),
                    getres_loop(Pids#{NewPid => {F, restart}}, Waiting)
            end
    end.
```