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.
```

Principles of Programming Languages, 2024.06.06

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)

Define a new construct, called let-cond, which works like a conditional let. The basic syntax is like:

`(let-cond ((condition bindings then-part) …) else-part)`, where the semantics is the following:

1) The *then-part* corresponding to the first *condition* that is true is executed. No other code is executed in the construct.

2) If all conditions are false, the *else-part* is executed.

For example:

```
(let-cond
  [((> 5 13)
    [(a 10) (b 20)]
    (+ a b))              ; then-body 1
   ((= 5 5)
    [(c 3) (d 4)]
    (+ c d))]             ; then-body 2
  "all conditions false")) ; else-body
```

should return 7.

Exercise 2, Haskell (12 pts)

Consider the following datatype definition.

```
data W x y = W ([x] -> [y])
```

Make *W* an instance of Functor, Applicative, and Monad.

Exercise 3, Erlang (11 pts)

Pino wants to create a Erlang program which takes two lists of data *[x1, x2, ..]*, *[y1, y2, ...]* and a list of binary functions *[f1, f2, ...]*, and evaluates these functions in parallel, passing them the respective parameters, to obtain *[f1(x1, y1), f2(x2, y2), ...]*.
To this end, Pino tries to use ChatGPT, obtaining the result shown in the next page.

```
-module(parallel_apply).
-export([parallel_apply/3, worker/2, collector/2]).

% Entry function to start the parallel processing
parallel_apply(List1, List2, FunList) ->
    CollectorPid = spawn(fun() -> collector([], length(FunList)) end),
    spawn_workers(List1, List2, FunList, CollectorPid),
    receive
        {results, Results} -> Results
    end.

% Spawn worker processes for each element pair and function
spawn_workers([H1|T1], [H2|T2], [F|Fs], CollectorPid) ->
    spawn(fun() -> worker({F, H1, H2}, CollectorPid) end),
    spawn_workers(T1, T2, Fs, CollectorPid);
spawn_workers([], [], [], _) -> done.

% Worker process to apply function to pair of elements
worker({F, A, B}, CollectorPid) ->
    Result = F(A, B),
    CollectorPid ! {result, Result}.

% Collector process to gather all results
collector(Results, 0) ->
    % Send the final results back to the parent process
    ParentPid = self(),
    ParentPid ! {results, lists:reverse(Results)};
collector(Results, N) ->
    receive
        {result, Result} ->
            collector([Result | Results], N - 1)
    end.
```

1) Is the previous code correct? If not, explain why.
2) Implement you own version of the exercise to make Pino happy.

Solutions

Ex 1

```
(define-syntax let-cond
  (syntax-rules ()
    ((_ ((condition bindings then-body) ...)
        else-body)
     (cond
       (condition
        (let bindings
          then-body))
       ...
       (else
        else-body)))))
```

Ex 2

```
instance Functor (W x) where
  fmap f (W g) = W ((fmap f) . g)

instance Applicative (W x) where
  pure t = W (\x -> fmap (\_ -> t) x)
  (W g) <*> (W h) = W (\x -> let f' = g x
                                h' = h x
                            in f' <*> h')

instance Monad (W x) where
  (W g) >>= f = W (\xs ->
                     concatMap (\y -> let W h = f y
                                      in h xs)
                     (g xs))
```

Ex 3

1) there are many errors: e.g. the collector get its PID instead of the one of its parent; the collector also assumes that the results from the worker will arrive in the correct order, so it doesn't keep PIDs for the workers.

2)

```
parallel_apply(List1, List2, FunList) ->
    W = lists:map(fun(X) -> spawn(?MODULE, worker, [X, self()]) end, zip3(List1, List2, FunList)),
    lists:map(fun (P) ->
                      receive
                          {P, V} -> V
                      end
              end, W).

worker({A, B, F}, CollectorPid) ->
    CollectorPid ! {self(), F(A,B)}.

% also available in lists
zip3([A|As], [B|Bs], [F|Fs]) -> [{A, B, F} | zip3(As, Bs, Fs)];
zip3([], _, _) -> [];
zip3(_, [], _) -> [];
zip3(_, _, []) -> [].
```

Principles of Programming Languages, 2021.08.31

Important notes
- Total available time: 1h 30'.
- You may use any written material you need, and write in Italian, if you prefer.
- 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 (13 pts)

1) Define a procedure which takes a natural number *n* and a default value, and creates a *n* by *n* matrix filled with the default value, implemented through vectors (i.e. a vector of vectors).

2) Let S = {0, 1, ..., n-1} x {0, 1, ..., n-1} for a natural number *n*. Consider a *n* by *n* matrix M, stored in a vector of vectors, containing pairs (x,y) ∈ S, as a function from S to S (e.g. f(2,3) = (1,0) is represented by M[2][3] = (1,0)). Define a procedure to check if M defines a **bijection** (i.e. a function that is both injective and surjective).

Exercise 2, Haskell (11 pts)

Consider a *Slist* data structure for lists that store their **length.** Define the *Slist* data structure, and make it an instance of Foldable, Functor, Applicative and Monad.

Exercise 3, Erlang (8 pts)

Define a function which takes two list of PIDs $[x_1, x_2, ...]$, $[y_1, y_2, ...]$, having the same length, and a function *f*, and creates a different "broker" process for managing the interaction between each pair of processes x_i and y_i.
At start, the broker process *i* must send its PID to x_i and y_i with a message {*broker, PID}*. Then, the broker *i* will receive messages *{from, PID, data, D}* from x_i or y_i, and it must send to the other one an analogous message, but with the broker PID and data D modified by applying *f* to it.
A special *stop* message can be sent to a broker *i*, that will end its activity sending the same message to x_i and y_i.

Solutions

Es 1

```
(define (create-matrix size default)
  (define vec (make-vector size #f))
  (let loop ((i 0))
    (if (= i size)
        vec
        (begin
          (vector-set! vec i (make-vector size default))
          (loop (+ 1 i))))))

(define (bijection? m)
  (define size (vector-length m))
  (define seen? (create-matrix size #f))
  (call/cc (lambda (exit)
             (let loop ((i 0))
               (when (< i size)
                 (let loop1 ((j 0))
                   (when (< j size)
                     (let ((datum (vector-ref (vector-ref m i) j)))
                       (if (vector-ref (vector-ref seen? (car datum)) (cdr datum))
                           (exit #f)
                           (vector-set! (vector-ref seen? (car datum)) (cdr datum) #t)))
                     (loop1 (+ 1 j))))
                 (loop (+ 1 i))))
             #t)))
```

Es 2

```
data Slist a = Slist Int [a] deriving (Show, Eq)

makeSlist v = Slist (length v) v

instance Foldable Slist where
  foldr f i (Slist n xs) = foldr f i xs

instance Functor Slist where
  fmap f (Slist n xs) = Slist n (fmap f xs)

instance Applicative Slist where
  pure v = Slist 1 (pure v)
  (Slist x fs) <*> (Slist y xs) = Slist (x*y) (fs <*> xs)

instance Monad Slist where
  fail _ = Slist 0 []
  (Slist n xs) >>= f = makeSlist (xs >>= (\x -> let Slist n xs = f x
                                                in xs))
```

Es 3

```
broker(X, Y, F) ->
    X ! {broker, self()},
    Y ! {broker, self()},
    receive
        {from, X, data, D} ->
            Y ! {from, self(), data, F(D)},
            broker(X, Y, F);
        {from, Y, data, D} ->
            X ! {from, self(), data, F(D)},
            broker(X, Y, F);
        stop ->
            X ! stop,
            Y ! stop,
            ok
    end.


twins([],_,_) ->
     ok;
twins([X|Xs],[Y|Ys],F) ->
    spawn(?MODULE, broker, [X, Y, F]),
    twins(Xs, Ys, F).
```

Principles of Programming Languages, 2021.07.14

Important notes
- Total available time: 1h 45'.
- You may use any written material you need, and write in Italian, if you prefer.
- 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)

Define a *defun* construct like in Common Lisp, where (defun f (*x1 x2 ...*) *body*) is used for defining a function f with parameters *x1 x2 ....*

Every function defined in this way should also be able to return a value *x* by calling *(ret x)*.

Exercise 2, Haskell (11 pts)

1) Define a "generalized" zip function which takes a finite list of possibly infinite lists, and returns a possibly infinite list containing a list of all the first elements, followed by a list of all the second elements, and so on.

E.g. gzip [[1,2,3],[4,5,6],[7,8,9,10]] ==> [[1,4,7],[2,5,8],[3,6,9]]

2) Given an input like in 1), define a function which returns the possibly infinite list of the sum of the two greatest elements in the same positions of the lists.

E.g. sum_two_greatest [[1,8,3],[4,5,6],[7,8,9],[10,2,3]] ==> [17,16,15]

Exercise 3, Erlang (10 pts)

Consider a main process which takes two lists: one of function names, and one of lists of parameters (the first element of with contains the parameters for the first function, and so forth). For each function, the main process must spawn a worker process, passing to it the corresponding argument list. If one of the workers fails for some reason, the main process must create another worker running the same function. The main process ends when all the workers are done.

Solutions

Es 1

```
(define ret-store '())

(define (ret v)
  ((car ret-store) v))

(define-syntax defun
  (syntax-rules ()
    ((_ fname (var ...) body ...)
     (define (fname var ...)
       (let ((out (call/cc (lambda (c)
                             (set! ret-store (cons c ret-store))
                             body ...))))
         (set! ret-store (cdr ret-store))
         out)))))
```

Es 2

```
gzip xs = if null (filter null xs) then (map head xs) : gzip (map tail xs) else []

store_two_greatest v (x,y) | v > x = (v,y)
store_two_greatest v (x,y) | x >= v && v > y = (x,v)
store_two_greatest v (x,y) = (x,y)


two_greatest (x:y:xs) = foldr store_two_greatest (if x > y then (x,y) else (y,x)) xs

sum_two_greatest xs = [ let (x,y) = two_greatest v
                        in x+y | v <- gzip xs]
```

Es 3

```
listmlink([], [], Pids) -> Pids;
listmlink([F|Fs], [D|Ds], Pids) ->
    Pid = spawn_link(?MODULE, F, D),
    listmlink(Fs, Ds, Pids#{Pid => {F,D}}).

master(Functions, Arguments) ->
    process_flag(trap_exit, true),
    Workers = listmlink(Functions, Arguments, #{}),
    master_loop(Workers, length(Functions)).

master_loop(Workers, Count) ->
    receive
        {'EXIT', _, normal} ->
            if
                Count =:= 1 -> ok;
                true -> master_loop(Workers, Count-1)
            end;
        {'EXIT', Child, _} ->
            #{Child := {F,D}} = Workers,
            Pid = spawn_link(?MODULE, F, D),
            master_loop(Workers#{Pid => {F,D}}, Count)
    end.
```

Principles of Programming Languages, 2021.06.22

Important notes
- Total available time: 1h 40'.
- You may use any written material you need, and write in Italian, if you prefer.
- 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)

Define a function *mix* which takes a variable number of arguments $x_0\ x_1\ x_2\ ...\ x_n$, the first one a function, and returns the list $(x_1\ (x_2\ ...\ (x_0(x_1)\ x_0(x_2)\ ...\ x_0(x_n))\ x_n)\ x_{n-1})\ ...\ x_1)$.

E.g.

```
(mix (lambda (x) (* x x)) 1 2 3 4 5)
```

returns: '(1 (2 (3 (4 (5 (1 4 9 16 25) 5) 4) 3) 2) 1)

Exercise 2, Haskell (11 pts)

Define a data-type called BTT which implements trees that can be binary or ternary, and where every node contains a value, but the empty tree (Nil). Note: there must not be unary nodes, like leaves.

1) Make *BTT* an instance of Functor and Foldable.

2) Define a concatenation for *BTT*, with the following constraints:

- If one of the operands is a binary node, such node must become ternary, and the other operand will become the added subtree (e.g. if the binary node is the left operand, the rightmost node of the new ternary node will be the right operand).
- If both the operands are ternary nodes, the right operand must be appened on the right of the left operand, by recursively calling concatenation.

3) Make *BTT* an instance of Applicative.

Exercise 3, Erlang (11 pts)

Create a function *node_wait* that implements nodes in a tree-like topology. Each node, which is a separate agent, keeps track of its parent and children (which can be zero or more), and contains a value. An integer weight is associated to each edge between parent and child.
A node waits for two kind of messages:

- *{register_child, ...}*, which adds a new child to the node (replace the dots with appropriate values),
- *{get_distance, Value}*, which causes the recipient to search for *Value* among its children, by interacting with them through appropriate messages. When the value is found, the recipient answers with a message containing the minimum distance between it and the node containing "Value", considering the sum of the weights of the edges to be traversed. If the value is not found, the recipient answers with an appropriate message. While a node is searching for a value among its children, it may not accept any new children registrations. E.g., if we send *{get_distance, a}* to the root process, it answers with the minimum distance between the root and the closest node containing the atom *a* (which is 0 if *a* is in the root).

Solutions

Es 1

```
(define (f g . L)
  (foldr (lambda (x y)
           (list x y x))
         (map g L)
         L))
```

Es 2

```
data BTT a = Nil | B a (BTT a)(BTT a) | T a (BTT a)(BTT a)(BTT a) deriving (Eq, Show)

instance Functor BTT where
  fmap f Nil = Nil
  fmap f (B a l r) = B (f a)(fmap f l)(fmap f r)
  fmap f (T a l c r) = T (f a)(fmap f l)(fmap f c)(fmap f r)

instance Foldable BTT where
  foldr f i Nil = i
  foldr f i (B x l r) = f x $ foldr f (foldr f i r) l
  foldr f i (T x l c r) = f x $ foldr f (foldr f (foldr f i r) c) l

x <++> Nil = x
Nil <++> x = x
(B x l r) <++> y = (T x l r y)
y <++> (B x l r) = (T x y l r)
(T x l c r) <++> v@(T x' l' c' r') = (T x l c (r <++> v))


ltconcat t = foldr (<++>) Nil t
ltconcmap f t = ltconcat $ fmap f t

instance Applicative BTT where
  pure x = (B x Nil Nil)
  x <*> y = ltconcmap (\f -> fmap f y) x
```

Es 3

```
node_wait(Parent, Elem, Children) ->
    receive
        {register_child, Child, Weight} ->
            node_wait(Parent, Elem, [{Child, Weight} | Children]);
        {get_distance, Value} -> if
                                     Value == Elem ->
                                         Parent ! {distance, Value, self(), 0},
                                         node_wait(Parent, Elem, Children);
                                     true ->
                                         node_comp_dist(Parent, Elem, Children, Value)
                                 end
    end.

node_comp_dist(Parent, Elem, Children, Value) ->
    [Child ! {get_distance, Value} || {Child, _} <- Children],
    Dists = [receive
                 {distance, Value, Child, D} ->
                     D + Weight;
                 {not_found, Value, Child} ->
                     not_found
             end || {Child, Weight} <- Children],
    FoundDists = lists:filter(fun erlang:is_integer/1, Dists),
    case FoundDists of
        [] ->
            Parent ! {not_found, Value, self()};
        _ ->
            Parent ! {distance, Value, self(), lists:min(FoundDists)}
    end,
    node_wait(Parent, Elem, Children).
```

Principles of Programming Languages, 2020.02.07

Important notes
- Total available time: 2h.
- You may use any written material you need, and write in Italian, if you prefer.
- 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)

Implement this new construct: (**each-until** *var* **in** *list* **until** *pred* **:** *body*), where keywords are written in boldface. It works like a for-each with variable *var*, but it can end before finishing all the elements of *list* when the predicate *pred* on *var* becomes true.

E.g.

```
(each-until x in '(1 2 3 4)
            until (> x 3) :
            (display (* x 3))
            (display " "))
```

shows on the screen: 3 6 9

Exercise 2, Haskell (11 pts)

Consider a data type *PriceList* that represents a list of items, where each item is associated with a price, of type Float:

```
data PriceList a = PriceList [(a, Float)]
```

1) Make *PriceList* an instance of Functor and Foldable.

2) Make *PriceList* an instance of Applicative, with the constraint that each application of a function in the left hand side of a <*> must increment a right hand side value's price by the price associated with the function.

E.g.
```
PriceList [(("nice "++), 0.5), (("good "++), 0.4)]  <*>
PriceList [("pen", 4.5), ("pencil", 2.8), ("rubber", 0.8)]
```

is

```
PriceList [("nice pen",5.0),("nice pencil",3.3),("nice rubber",1.3),("good pen",4.9),
("good pencil",3.2),("good rubber",1.2)]
```

Exercise 3, Erlang (11 pts)

We want to create a simplified implementation of the "Reduce" part of the MapReduce paradigm. To this end, define a process "reduce_manager" that keeps track of a pool of reducers. When it is created, it stores a user-defined associative binary function `ReduceF`. It receives messages of the form `{reduce, Key, Value}`, and forwards them to a different "reducer" process for each key, which is created lazily (i.e. only when needed). Each reducer serves requests for a unique key.
Reducers keep into an accumulator variable the result of the application of `ReduceF` to the values they receive. When they receive a new value, they apply `ReduceF` to the accumulator and the new value, updating the former. When the reduce_manager receives the message `print_results`, it makes all its reducers print their key and incremental result.

(see back)

For example, the following code (where the meaning of *string:split* should be clear from the context):

```
word_count(Text) ->
    RMPid = start_reduce_mgr(fun (X, Y) -> X + Y end),
    lists:foreach(fun (Word) -> RMPid ! {reduce, Word, 1} end, string:split(Text, " ", all)),
    RMPid ! print_results,
    ok.
```

causes the following print:

```
1> mapreduce:word_count("sopra la panca la capra campa sotto la panca la capra crepa").
sopra: 1
la: 4
panca: 2
capra: 2
campa: 1
sotto: 1
crepa: 1
ok
```

Solutions

Es 1

```
(define-syntax each-until
  (syntax-rules (in until :)
    ((_ x in L until pred : body ...)
       (let loop ((xs L))
         (unless (null? xs)
           (let ((x (car xs)))
             (unless pred
               (begin
                 body ...
                 (loop (cdr xs)))))))))))
```

Es 2

```
pmap :: (a -> b) -> Float -> PriceList a -> PriceList b
pmap f v (PriceList prices) = PriceList $ fmap (\x -> let (a, p) = x
                                                      in (f a, p+v)) prices
instance Functor PriceList where
  fmap f prices = pmap f 0.0 prices

instance Foldable PriceList where
  foldr f i (PriceList prices) = foldr (\x y -> let (a, p) = x
                                            in  f a y) i prices

(PriceList x) +.+ (PriceList y) = PriceList $ x ++ y

plconcat x = foldr (+.+) (PriceList []) x

instance Applicative PriceList where
  pure x = PriceList [(x, 0.0)]
  (PriceList fs) <*> xs = plconcat (fmap (\ff -> let (f, v) = ff
                                                 in pmap f v xs) fs)
```

Es 3

```
start_reduce_mgr(ReduceF) ->
    spawn(?MODULE, reduce_mgr, [ReduceF, #{}]).

reduce_mgr(ReduceF, Reducers) ->
    receive
         print_results ->
             lists:foreach(fun ({_, RPid}) -> RPid ! print_results end, maps:to_list(Reducers));
         {reduce, Key, Value} ->
             case Reducers of
                  #{Key := RPid} ->
                      RPid ! {Key, Value},
                      reduce_mgr(ReduceF, Reducers);
                  _ ->
                      NewReducer = spawn(?MODULE, reducer, [ReduceF, Key, Value]),
                      reduce_mgr(ReduceF, Reducers#{Key => NewReducer})
             end
    end.

reducer(ReduceF, Key, Result) ->
    receive
         print_results ->
             io:format("~s: ~w~n", [Key, Result]);
         {Key, Value} ->
             reducer(ReduceF, Key, ReduceF(Result, Value))
    end.
```

Principles of Programming Languages, 2020.01.15

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

Exercise 1, Scheme (11 pts)

Consider the *Foldable* and *Applicative* type classes in Haskell. We want to implement something analogous in Scheme for *vectors*. Note: you can use the following library functions in your code: *vector-map, vector-append*.

1) Define *vector-foldl* and *vector-foldr*.

2) Define *vector-pure* and *vector-<*>*.

Exercise 2, Haskell (12 pts)

The following data structure represents a cash register. As it should be clear from the two accessor functions, the first component represents the current item, while the second component is used to store the price (not necessarily of the item: it could be used for the total).

```
data CashRegister a = CashRegister { getReceipt :: (a, Float) } deriving (Show, Eq)

getCurrentItem = fst . getReceipt

getPrice = snd . getReceipt
```

1) Make CashRegister an instance of Functor and Applicative.

2) Make CashRegister an instance of Monad.

Exercise 3, Erlang (9 pts)

We want to implement something like Python's *range* in Erlang, using processes.

E.g.

```
R = range(1,5,1)      % starting value, end value, step

next(R)               % is 1

next(R)               % is 2

…

next(R)               % is 5

next(R)               % is the atom stop_iteration
```

Define *range* and *next*, where *range* creates a process that manages the iteration, and *next* a function that talks with it, asking the current value.

Solutions

Es 1

```
(define (vector-foldr f i v)
  (let loop ((cur (- (vector-length v) 1))
             (out i))
    (if (< cur 0)
        out
        (loop (- cur 1) (f (vector-ref v cur) out)))))

(define (vector-foldl f i v)
  (let loop ((cur 0)
             (out i))
    (if (>= cur (vector-length v))
        out
        (loop (+ cur 1) (f (vector-ref v cur) out)))))

(define (vector-concat-map f v)
  (vector-foldr vector-append #() (vector-map f v)))

(define vector-pure vector)

(define (vector-<*> fs xs)
  (vector-concat-map (lambda (f) (vector-map f xs)) fs))
```

Es 2

```
instance Functor CashRegister where
  fmap f cr = CashRegister (f $ getCurrentItem cr, getPrice cr)

instance Applicative CashRegister where
  pure x = CashRegister (x, 0.0)
  CashRegister (f, pf) <*> CashRegister (x, px) = CashRegister (f x, pf + px)

instance Monad CashRegister where
  CashRegister (oldItem, price) >>= f = let newReceipt = f oldItem
                                        in CashRegister (getCurrentItem newReceipt, price + (getPrice
newReceipt))
```

Es 3

```
ranger(Current, Stop, Inc) ->
    receive
        {Pid, next} ->
            if
                Current =:= Stop -> Pid ! stop_iteration;
                true -> Pid ! Current,
                        ranger(Current + Inc, Stop, Inc)
            end
    end.

range(Start, Stop, Inc) ->
    spawn(?MODULE, ranger, [Start, Stop, Inc]).

next(Pid) ->
    Pid ! {self(), next},
    receive
        V -> V
    end.
```