Principles of Programming Languages, 2024.07.03

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)

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

```
(let-cond+ ((condition bindings then-part) …) else-part).
```

Semantics:

1) The *then-parts* corresponding to all the *conditions* that are true are executed in sequence.

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

3) The returned value is the one of the last condition which is true, or the evaluation of the *else-part*.

For example, the next code shows “hello” on the screen, and returns 7:

```
(let-cond+
  (((< 5 13)  ; condition
    ((a 10))  ; bindings
    (begin (displayln "hello") a)) ; then-part
   ((= 5 5)   ; condition
    ((b 3)    ; bindings
     (c 4))
    (+ c b))) ; then-part
  "all conditions false") ; else-part
```

Exercise 2, Haskell (11 pts)

Consider the following datatype definition.

```
data T x y z = T (x -> y -> z)
```

Make *T* an instance of Functor, Applicative, and Monad. (Hint: follow the types.)

Exercise 3, Erlang (11 pts)

Define the main function of a broker process for centralized PID-less interaction among processes. The broker must respond to these messages:

- *{new, Pid, Id}* to bind the local broker identifier Id to the PID Pid;
- *{send, Id, Msg}* to send Msg to the process having the local broker identifier Id;
- *{delete_id, Id}* to delete the local broker identifier binding for Id;
- *{delete_pid, Pid}* to delete the local data for PID Pid;
- *{broadcast, Msg}* to send Msg to all the processes known by the broker;
- *stop* to stop the broker.

You can use the following OTP functions, if you need them:
maps:remove(Key, Map), to remove Key from Map
maps:filtermap(F/2, Map), which is a filter, where F/2 takes a pair (Key, Value) and returns a Boolean
maps:foreach(F/2, Map), which runs F/2 on all the pairs (Key, Value) in Map.

Solutions

Ex 1

```
(define-syntax let-cond+
  (syntax-rules ()
    ((_ ((condition bindings then-body) ...)
        else-body)
     (let ((flag #f)
           (result #f))
       (when condition
        (set! result
          (let bindings
            (set! flag #t)
            then-body)))
       ...
       (if flag
         result
         else-body)))))
```

Ex 2

```
instance Functor (T x y) where
    fmap f (T g) = T (\x y -> f (g x y))

instance Applicative (T x y) where
    pure z = T (\_ _ -> z)
    (T f) <*> (T g) = T (\x y -> f x y (g x y))

instance Monad (T x y) where
    (T g) >>= f = T (\x y -> let (T t) = f $ g x y
                             in t x y)
```

Ex 3

```
broker(Map) ->
    receive
        {send, Id, Msg} ->
            #{Id := Pid} = Map,
            Pid ! Msg,
            broker(Map);
        {new, Pid, Id} ->
            broker(Map#{Id => Pid});
        {delete_id, Id} ->
            broker(maps:remove(Id, Map));
        {delete_pid, Pid} ->
            broker(maps:filtermap(fun (_,Y) ->
                                              Y =/= Pid
                                  end, Map));
        {broadcast, Msg} ->
            maps:foreach(fun (_,V) ->
                                V ! Msg
                             end, Map),
            broker(Map);
        stop ->
            ok
    end.
```