Principles of Programming Languages, 2022.07.06

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 (10 pts)

Consider the technique “closures as objects” as seen in class, where a closure assumes the role of a class. In this technique, the called procedure (which works like a class in OOP) returns a closure which is essentially the dispatcher of the object.

Define the *define-dispatcher* macro for generating the dispatcher in an automatic way, as illustrated by the following example:

```
(define (make-man)
  (let ((p (make-entity))
        (name "man"))
    (define prefix+name
      (lambda (prefix)
        (string-append prefix name)))
    (define change-name
      (lambda (new-name)
        (set! name new-name)))
    (define-dispatcher methods: (prefix+name change-name) parent: p)))
```

where `p` is the parent of the current instance of class `man`, and `make-entity` is its constructor.

If there is no inheritance (or it is a base class), *define-dispatcher* can be used without the `parent: p` part.

Then, an instance of class `man` can be created and its methods can be called as follows:

```
> (define carlo (make-man))
> (carlo 'change-name "Carlo")
> (carlo 'prefix+name "Mr. ")
"Mr. Carlo"
```

Exercise 2, Haskell (14 pts)

A *deque*, short for *double-ended queue*, is a list-like data structure that supports efficient element insertion and removal from both its head and its tail. Recall that Haskell lists, however, only support O(1) insertion and removal from their head.

Implement a deque data type in Haskell by using two lists: the first one containing elements from the initial part of the list, and the second one containing elements form the final part of the list, reversed.

In this way, elements can be inserted/removed from the first list when pushing to/popping the deque's head, and from the second list when pushing to/popping the deque's tail.

1) Write a data type declaration for `Deque`.

2) Implement the following functions:

- `toList`: takes a Deque and converts it to a list
- `fromList`: takes a list and converts it to a Deque

- `pushFront`: pushes a new element to a Deque's head
- `popFront`: pops the first element of a Deque, returning a tuple with the popped element and the new Deque
- `pushBack`: pushes a new element to the end of a Deque
- `popBack`: pops the last element of a Deque, returning a tuple with the popped element and the new Deque

3) Make Deque an instance of Eq and Show.

4) Make Deque an instance of Functor, Foldable, Applicative and Monad.

You may rely on instances of the above classes for plain lists.

Exercise 3, Erlang (8 pts)

We want to implement an interface to a server protocol, for managing requests that are lists of functions of one argument and lists of data on which the functions are called.
The interface main function is called *`multiple_query`* and gets a list of functions *`FunL`*, and a list of data, *`DataL`*, of course of the same size.
The protocol works as follows (assume that there is a registered server called *`master_server`*):

1. First, we ask the master server for a list of slave processes that will perform the computation. The request has the following form:
   `{slaves_request, {identity, <id_of_the_caller>}, {quantity, <number_of_needed_slaves>}}`
2. The answer has the following form:
   `{slaves_id, <list_of_ids_of_slave_processes>}`
3. Then, the library sends the following requests to the slave processes:
   `{compute_request, {identity, <id_of_the_caller>}, <function>, <data>},`
   where `<function>` is one of the elements of *`FunL`*, and `<data>` is the corresponding element of *`DataL`*.
4. Each process sends the result of its computation with a message:
   `{compute_result, {identity, <slave_id>}, {value, <result_value>}}`
5. *`multiple_query`* ends by returning the list of the computed results, that must be ordered according to *`FunL`* and *`DataL`*.

If you want, you may use *lists:zip/2* and *lists:zip3/3*, that are the standard zip operations on 2 or 3 lists, respectively.

Solutions

Es 1

```
(define (unknown-method ls)
  (error "Unknown method" (car ls)))

(define-syntax define-dispatcher
  (syntax-rules (methods: parent:)
    ((_ methods: (mt ...) parent: p)
     (lambda (message . args)
       (case message
         ((mt) (apply mt args))
         ...
         (else (apply p (cons message args))))))
    ((_ methods: mts)
     (define-dispatcher methods: mts parent: unknown-method))))
```

Es 2

```
data Deque a = Deque [a] [a]

toList :: Deque a -> [a]
toList (Deque front back) = front ++ reverse back

fromList :: [a] -> Deque a
fromList l = let half = length l `div` 2
             in Deque (take half l) (reverse $ drop half l)

instance (Eq a) => Eq (Deque a) where
  d1 == d2 = toList d1 == toList d2

instance (Show a) => Show (Deque a) where
  show d = show $ toList d

pushFront :: a -> Deque a -> Deque a
pushFront x (Deque front back) = Deque (x:front) back

popFront :: Deque a -> (a, Deque a)
popFront (Deque (x:front) back) = (x, Deque front back)
popFront (Deque [] []) = error "Pop on empty Deque!"
popFront (Deque [] [y]) = (y, Deque [] [])
popFront (Deque [] back) = popFront $ fromList back

pushBack :: a -> Deque a -> Deque a
pushBack x (Deque front back) = Deque front (x:back)

popBack :: Deque a -> (a, Deque a)
popBack (Deque front (x:back)) = (x, Deque front back)
popBack (Deque [] []) = error "Pop on empty Deque!"
popBack (Deque [x] []) = (x, Deque [] [])
popBack (Deque front []) = popBack $ fromList front

instance Functor Deque where
  fmap f (Deque front back) = Deque (map f front) (map f back)

instance Foldable Deque where
  foldr f e = foldr f e . toList

instance Applicative Deque where
  pure x = Deque [x] []
  df <*> dx = fromList $ toList df <*> toList dx

instance Monad Deque where
  d >>= f = fromList $ concatMap (toList . f) $ toList d
```

Es 3

```
multiple_query(FunL, DataL) ->
    master_server ! {slaves_request,
                     {identity, self()},
                     {quantity, length(FunL)}},
    receive
        {slaves_id, Slaves} -> ok
    end,
    [ S ! {compute_request, {identity, self()}, F, D} || {S, F, D} <- lists:zip3(Slaves, FunL, DataL)],
    [ receive
         {compute_result, {identity, S}, {value, V}} -> V
      end || S <- Slaves ].
```