Principles of Programming Languages, 2022.01.21

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 (10 pts)

Define a new construct called *block-then* which creates two scopes for variables, declared after the scopes, with two different binding. E.g. the evaluation of the following code:

```
(block
  ((displayln (+ x y))
   (displayln (* x y))
   (displayln (* z z)))
  then
  ((displayln (+ x y))
   (displayln (* z x)))
  where (x <- 12 3)(y <- 8 7)(z <- 3 2))
```

should show on the screen:

```
20
96
9
10
6
```

Exercise 2, Haskell (10 pts)

Consider a *Tvtl* (two-values/two-lists) data structure, which can store either two values of a given type, or two lists of the same type.

Define the *Tvtl* data structure, and make it an instance of Functor, Foldable, and Applicative.

Exercise 3, Erlang (12 pts)

Create a distributed *hash table* with *separate chaining*. The hash table will consist of an agent for each bucket, and a master agent that stores the buckets' PIDs and acts as a middleware between them and the user. Actual key/value pairs are stored into the bucket agents.
The middleware agent must be implemented by a function called `hashtable_spawn` that takes as its arguments (1) the hash function and (2) the number of buckets. When executed, `hashtable_spawn` spawns the bucket nodes, and starts listening for queries from the user. Such queries can be of two kinds:

- Insert: `{insert, Key, Value}` inserts a new element into the hash table, or updates it if an element with the same key exists;
- Lookup: `{lookup, Key, RecipientPid}` sends to the agent with PID "`RecipientPid`" a message of the form `{found, Value}`, where `Value` is the value associated with the given key, if any. If no such value exists, it sends the message `not_found`.

The following code:

```
main() ->
    HT = spawn(?MODULE, hashtable_spawn, [fun(Key) -> Key rem 7 end, 7]),
    HT ! {insert, 15, "Apple"},
    HT ! {insert, 8, "Orange"},
    timer:sleep(500),
    HT ! {lookup, 8, self()},
    receive
        {found, A1} -> io:format("~s~n", [A1])
    end,
    HT ! {insert, 8, "Pineapple"},
    timer:sleep(500),
    HT ! {lookup, 8, self()},
    receive
        {found, A2} -> io:format("~s~n", [A2])
    end.
```

should print the following:

```
Orange
Pineapple
```

Solutions

Es 1

```
(define-syntax block
  (syntax-rules (where then <-)
    ((_ (e1 ...)
        then
        (e2 ...)
        where (v <- a b) ...)
     (begin
       (let ((v a) ...)
         e1 ...)
       (let ((v b) ...)
         e2 ...)))))
```

Es 2

```
data Tvtl a = Tv a a | Tl [a] [a] deriving (Show, Eq)

instance Functor Tvtl where
    fmap f (Tv x y) = Tv (f x) (f y)
    fmap f (Tl x y) = Tl (fmap f x) (fmap f y)

instance Foldable Tvtl where
    foldr f i (Tv x y) = f x (f y i)
    foldr f i (Tl x y) = foldr f (foldr f i y) x

(Tv x y) +++ (Tv z w) = Tl [x,y] [y,w]
(Tv x y) +++ (Tl l r) = Tl (x:l) (y:r)
(Tl l r) +++ (Tv x y) = Tl (l++[x]) (r++[y])
(Tl l r) +++ (Tl x y) = Tl (l++x) (r++y)

tvtlconcat t = foldr (+++) (Tl [][]) t
tvtlcmap f t = tvtlconcat $ fmap f t

instance Applicative Tvtl where
    pure x = Tl [x] []
    x <*> y = tvtlcmap (\f -> fmap f y) x
```

Es 3

```
hashtable_spawn(HashFun, NBuckets) ->
    BucketPids = [spawn(?MODULE, bucket, [[]]) || _ <- lists:seq(0, NBuckets)],
    hashtable_loop(HashFun, BucketPids).

hashtable_loop(HashFun, BucketPids) ->
    receive
        {insert, Key, Value} ->
            lists:nth(HashFun(Key) + 1, BucketPids) ! {insert, Key, Value},
            hashtable_loop(HashFun, BucketPids);
        {lookup, Key, AnswerPid} ->
            lists:nth(HashFun(Key) + 1, BucketPids) ! {lookup, Key, AnswerPid},
            hashtable_loop(HashFun, BucketPids)
    end.

bucket(Content) ->
    receive
        {insert, Key, Value} ->
            NewContent = lists:keystore(Key, 1, Content, {Key, Value}),
            bucket(NewContent);
        {lookup, Key, AnswerPid} ->
            case lists:keyfind(Key, 1, Content) of
                false ->
                    AnswerPid ! not_found;
                {_, Value} ->
                    AnswerPid ! {found, Value}
            end,
            bucket(Content)
    end.

%% You may replace calls to lists:keystore/4 and lists:keyfind/3 with calls to the following functions:
keystore_first(Key, [{TupleKey, _} | TupleTail], NewValue) when Key == TupleKey ->
    [{Key, NewValue} | TupleTail];
keystore_first(Key, [Tuple | TupleTail], NewValue) ->
    [Tuple | keystore_first(Key, TupleTail, NewValue)];
keystore_first(Key, [], NewValue) ->
    [{Key, NewValue}].

keyfind_first(Key, [{TupleKey, TupleValue} | _]) when Key == TupleKey ->
    {TupleKey, TupleValue};
keyfind_first(Key, [_ | TupleTail]) ->
    keyfind_first(Key, TupleTail);
keyfind_first(_, []) ->
    false.
```