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(_, _, []) -> [].
```