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.
```