Principles of Programming Languages, 2019.07.24

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.

Exercise 1, Scheme (8 pts)

Write a functional, tail recursive implementation of a procedure that takes a list of numbers *L* and two values *x* and *y*, and returns three lists: one containing all the elements that are less than both *x* and *y*, the second one containing all the elements in the range *[x,y]*, the third one with all the elements bigger than both *x* and *y*. It is not possible to use the *named let* construct in the implementation.

Exercise 2, Haskell (12 pts)

Consider a non-deterministic finite state automaton (NFSA) and assume that its states are values of a type *State* defined in some way. An NFSA is encoded in Haskell through three functions:

i) *transition :: Char → State → [State]*, i.e. the transition function.

ii) *end :: State → Bool*, i.e. a functions stating if a state is an accepting state (True) or not.

ii) *start :: [State]*, which contains the list of starting states.

1) Define a data type suitable to encode the configuration of an NSFA.

2) Define the necessary functions (providing also all their types) that, given an automaton *A* (through *transition*, *end*, and *start*) and a string *s*, can be used to check if *A* accepts *s* or not.

Exercise 3, Erlang (12 pts)

Define a master process which takes a list of nullary (or 0-arity) functions, and starts a worker process for each of them. The master must monitor all the workers and, if one fails for some reason, must re-start it to run the same code as before. The master ends when all the workers are done.

Note: for simplicity, you can use the library function *spawn_link/1*, which takes a lambda function, and spawns and links a process running it.

Solutions

Es 1

```
(define (3-part L v1 v2)
  (define (3-p L v1 v2 r1 r2 r3)
    (if (null? L)
        (list r1 r2 r3)
        (let ((x  (car L))
              (xs (cdr L)))
          (cond
            ((and (< x v1)(< x v2))
             (3-p xs v1 v2 (cons x r1) r2 r3))
            ((and (>= x v1)(<= x v2))
             (3-p xs v1 v2 r1 (cons x r2) r3))
            ((and (> x v1)(> x v2))
             (3-p xs v1 v2 r1 r2 (cons x r3)))))))

  (3-p L v1 v2 '() '() '()))
```

Es 2

```
data Config = Config String [State] deriving (Show, Eq)

steps :: (Char -> State -> [State]) -> Config -> Bool
steps trans (Config "" confs) = not . null $ filter end confs
steps trans (Config (a:as) confs) = steps trans $ Config as (concatMap (trans a) confs)
```

Es 3

```
listlink([], Pids) -> Pids;
listlink([F|Fs], Pids) ->
    Pid = spawn_link(F),
    listlink(Fs, Pids#{Pid => F}).

master(Functions) ->
    process_flag(trap_exit, true),
    Workers = listlink(Functions, #{}),
    master_loop(Workers, length(Functions)).

master_loop(Workers, Count) ->
    receive
        {'EXIT', Child, normal} ->
            if
                Count =:= 1 -> ok;
                true -> master_loop(Workers, Count-1)
            end;
        {'EXIT', Child, _} ->
            #{Child := Fun} = Workers,
            Pid = spawn_link(Fun),
            master_loop(Workers#{Pid => Fun}, Count)
    end.
```