Principles of Programming Languages

2014.02.12

Notes

- Total available time: 2h.
- You may use any written material you need.
- You cannot use computers or phones during the exam.

1 Scheme

Consider a procedure `string-from-strings` that receives as input a list of objects, keeps all the objects that are strings, discarding all the others, and returns the ordered concatenation of all such strings.
E.g. `(string-from-strings '(1 "hello" ", " 2 "world"))` is `"hello, world"`

1.1 Recursive (3 pts)

Define a functional (non tail) recursive version of `string-from-strings` (without using map, filter, fold).

1.2 Tail recursive (4 pts)

Define a tail recursive version of `string-from-strings` (without using map, filter, fold).

1.3 Functional higher-order (3 pts)

Give an implementation of `string-from-strings` using the classical functional higher order functions, i.e. map, filter, fold...

2 Haskell

2.1 Tree (2 pts)

Define a `Tree` data structure, where each node contains a value and can have *any number of children*.

2.2 Visit (4 pts)

Define a visit function, that returns a list of all the elements that are contained in the tree data structure defined before (you can choose any order you like).

2.3 Equality (2 pts)

Two trees are considered equal iff they contain the same elements and those are in the order defined by the `vist` function defined before (so they could be structurally different). Define == for `Tree`.

2.4 zipToList (4 pts)

Define the `zipToList :: [(a,a)] -> [a]` function, that, given a list of pairs, returns a flat list containing all the elements found in the pairs. E.g. `zipToList [(1,2),(3,4)]` is `[1,2,3,4]`.

2.5 Free monoid (4 pts)

Define an infinite list containing all the elements of the free monoid $\{a, b\}^*$ (i.e. all the strings defined on the alphabet $\{a, b\}$, empty string included).

3 Prolog

3.1 Product of the elements (3 pts)

Define the predicate `prod_list` that returns the product of all the element in a list, optimizing it with cut, and without using any library functions.

3.2 Free monoid (4 pts)

Define the predicate `freeM(X, A)` which succeeds if, and only if, X is an element of the free monoid on A, i.e. X is a list made of elements taken from A.

Note: it must be possible to use such predicate also to obtain all the possible lists made of elements of the given list A (e.g. if called as `freeM(X, [0,1])`).

Solutions

Scheme

```
(define (string-from-strings lst)
  (if (null? lst)
    ""
    (let ((head (car lst)))
     (string-append (if (string? head) head "")
                    (string-from-strings (cdr lst))))))

(define (string-from-strings-tr lst)
  (define (sfs-helper lst str)
    (if (null? lst)
      str
      (let ((head (car lst)))
       (sfs-helper (cdr lst)
                   (string-append str
                                  (if (string? head) head ""))))))
  (sfs-helper lst ""))

(define (string-from-strings-ho lst)
  (foldr string-append "" (filter string? lst)))
```

Haskell

```
data Tree a = Leaf a | Branch a [Tree a] deriving (Show)

visit :: Tree a -> [a]
visit (Leaf x) = [x]
visit (Branch v branches) = v : foldl (++) [] (map visit branches)

instance (Eq a) => Eq (Tree a) where
  t1 == t2 = (visit t1) == (visit t2)

zipToList :: [(a, a)] -> [a]
zipToList [] = []
zipToList ((x1,x2):xs) = x1 : x2 : zipToList xs

fm = "" : zipToList [(x ++ "a", x ++ "b") | x <- fm]
```

Prolog

```
prod_list([X], X) :- !.
prod_list([X|Xs], V) :- !, prod_list(Xs, V1), V is V1*X.

freeM([], _).
freeM([X|Xs], A) :- freeM(Xs, A), member(X, A).
```