Principles of Programming Languages

2014.09.23

Notes

- Total available time: 1h 30'.
- You may use any written material you need.
- You cannot use computers or phones during the exam.

1 Scheme

1.1 Duplicates (7 pts)

Define a procedure called `rep` which takes a list L of elements and returns a list of the elements of L that are repeated at least twice. The procedure must have linear time complexity, and it can be imperative and use imperative data structures.

E.g. `(rep '(3 2 "hi" 2 "hello" hello "hi"))` is `(2 "hi")`.

1.2 Duplicates, functional version (6 pts)

Define a purely functional version of `rep` without any limits of time complexity.

2 Haskell

2.1 Duplicates (5 pts)

Define `rep` in Haskell, without any limits of time complexity, and declaring its type.

2.2 List comparison with duplicates (5 pts)

Define a predicate `comprep` for comparing lists, declaring its type. The predicate must accept another predicate (e.g. <=) and use it to compare the lists. The lists are compared counting the number of duplicated elements in them:
e.g. `comprep((<=), [1,2,1,2], [0,0,1,0])` is false.

3 Prolog (8 pts)

Define a Prolog predicate that, given a generic term t, returns a list containing all the atomic elements that appear in t at least twice.

E.g. given `a(1,b(2),1,a(3,b))`, it must return `[a,1,b]`.

Solutions

Scheme

```
;; linear complexity version
;; (under the standard hypotesis that the complexity of hash access is constant)
(define (rep L)
  (let ((h (make-hash))
        (out '()))
   (for-each (lambda (x)
               (hash-set! h x (+ 1 (hash-ref h x 0)))
               )
             L)
   (hash-for-each h
                  (lambda (el n)
                    (when (> n 1)
                      (set! out (cons el out)))))
   out))

;; functional, quadratic version
(define (repff L)
  (define (rep0 lst out)
    (if (null? lst)
      out
      (let ((x  (car lst))
            (xs (cdr lst)))
        (rep0 xs
              (if (and (member x xs)
                       (not (member x out)))
                (cons x out)
                out)))))
  (rep0 L '()))
```

Haskell

```
rep :: Eq a => [a] -> [a]
rep [] = []
rep ls = reph ls [] where
    reph [] out = out
    reph (x:xs) out = reph xs
                           (if (x `elem` xs) && not (x `elem` out)
                           then (x:out) else out)

comprep :: (Eq a, Eq b) => (Int -> Int -> Bool) -> [a] -> [b] -> Bool
comprep pred x y = (length (rep x)) `pred` (length (rep y))
```

Prolog

```
duplicates(Tree,X) :- treeToList(Tree,Y), onlydup(Y,X).

treeToList(Atom, [Atom]) :- atomic(Atom), !.
treeToList(Tree, [X|Xs]) :- Tree =.. [X|Args], maplist(treeToList,Args,Ys), flatten(Ys,Xs).

% defined also in Exercise 3 of the exam of 2013.02.13
onlydup([],[]).
onlydup([Y|Xs],[Y|Ys]) :- member(Y,Xs), onlydup(Xs,Ys).
onlydup([X|Xs],Ys) :- \+ member(X,Xs), onlydup(Xs,Ys).
```