Principles of Programming Languages

2014.07.03

Notes

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

1 Scheme

A program collects data from different nodes of the network and put them in a list containing elements of different types - we call this list “unsorted”. E.g. (3 "bob" #(6 6 1) 4 #(1 2) -2 end 9).

We want to take from the unsorted list all the elements that are numbers or vectors: the numbers are summed, while the vectors are collected in another list (it is not necessary to maintain the order of the original list). To memorize the data, we introduce a structure called demuxed, that has two fields named num and vec. E.g. for the previous case: 5 and (#(6 6 1) #(1 2)), respectively.

1.1 Imperative (5 pts)

Define the demuxed data structure, and a procedure (called demux-imperative) that has two parameters d and l. d is a demuxed data structure, while l is an unsorted list. This procedure must update d with data from l, stopping when the element end is found, if present.

1.2 Tail recursive (6 pts)

Assume that demuxed is immutable. Define a functional, tail-recursive procedure demux-tail-rec that takes an unsorted list and returns a demuxed data structure containing the data, processed as in before. You may use as many additional parameters as you need, but you must specify their initial value.

2 Haskell

Consider a variant of the problem seen in Exercise 1: an unsorted list can contain list of elements of some type, integer numbers, or the special value End.
E.g. (in a pseudo-Haskell syntax) [3, [6, 6, 1], 4, [1, 2], -2, End, 9].

2.1 Data structures (5 pts)

Define the data structure for the unsorted list, and Demuxed, analogous to the structure introduced in Exercise 1 (i.e. with two fields, one containing an integer value, the sum of the integer elements found, and a list of all the found lists).
2.2 Demux (6 pts)

Define the \texttt{demux} function, that takes an unsorted list \( l \) and builds up a \texttt{demuxed} data structure containing the processed data of \( l \) (only that before \texttt{End}, if present). \texttt{demux} must be \texttt{strict} (i.e. non lazy).

3 Prolog

3.1 Demux (5 pts)

Consider a variant of the problem seen in Exercise 1: an unsorted list can contain either atoms or integer numbers. E.g. \([1,3,\text{house},4,\text{of},-7,\text{cards},\text{end},-12]\).

Define a \texttt{demux} predicate, that has an unsorted list as first argument, the sum \( n \) of the numbers present in the unsorted list as second argument, and puts all the symbols in the list \( v \) as third argument. \texttt{demux} must stop if an atom \texttt{end} is found, and must be optimized with \texttt{cut}.

E.g. for the previous example list \( n = 1 \) and \( v = [\text{house}, \text{of}, \text{cards}] \).

3.2 Shuffle (5 pts)

Define a \texttt{shuffle} predicate, that takes three lists \( n, a, m \), where \( n \) is a list of number, \( a \) is a list of atoms, and \( m \) is a list of numbers and atoms containing the elements of \( n \) and \( a \), by maintaining their relative order (e.g. \texttt{shuffle([1,2], [house,next,foot], [house,1,next,foot,2])} must hold). \texttt{shuffle} must be able to \texttt{shuffle} two lists together, e.g. to obtain as output the third list in the example.
Solutions

Scheme

(struct demuxed
  (num
   vec) #:mutable)

(define (demux-imperative dem lst)
  (if (null? lst)
      dem
      (let loop ((cur (car lst))
                  (ls (cdr lst)))
          (cond
            ((number? cur)
             (set-demuxed-num! dem
              (+ cur (demuxed-num dem))))
            ((vector? cur)
             (set-demuxed-vec! dem
              (cons cur (demuxed-vec dem))))
            (if (or (null? ls)
                    (eq? cur 'end))
                dem
                (loop (car ls) (cdr ls)))))))

(define (demux-tail-rec-h lst num vec)
  (if (or (null? lst) (eq? 'end (car lst)))
      (demuxed num vec)
      (let ((cur (car lst)))
        (demux-tail-rec-h (cdr lst)
          (if (number? cur)
              (+ num cur)
              num)
          (if (vector? cur)
              (cons cur vec)
              vec)))))

Haskell

data Muxel a = Ls [a] | Nm Integer | End deriving Show
data Demuxed a = Demuxed Integer [[a]] deriving Show
demux' n ls [] = Demuxed n ls
demux' n ls ((Ls x):xs) = let ls' = (x:ls)
in ls' ‘seq’ demux' n ls' xs
demux' n ls ((Nm x):xs) = let n' = x+n
in n' ‘seq’ demux' (x+n) ls xs
demux' n ls (End:xs) = Demuxed n ls
demux = demux' 0 []
Prolog

demux([],0,[]) :- !.
demux([end|Xs],0,[]) :- !.
demux([X|Xs],V,[Y|Ys]) :- atom(X), !, demux(Xs,V,Ys).
demux([X|Xs],V,Y) :- number(X), !, demux(Xs,V1,Y), V is V1+X.

shuffle([],[],[]).
shuffle(V,[X|Ys],[X|Xs]) :- atom(X), shuffle(V,Ys,Xs).
shuffle([X|Ys],V,[X|Xs]) :- number(X), shuffle(Ys,V,Xs).