Principles of Programming Languages

2016.02.10

Notes

- NAME: _____
- Did you present a small project? YES / NO
- Total available time: 2h.
- You may use any written material you need.
- You cannot use computers or phones during the exam.

Introduction

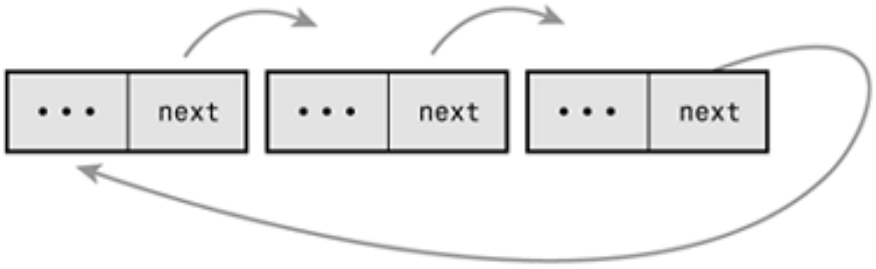

Figure 1: Simply-linked circular list (Clist)

A simply-linked circular list (called *Clist* from now on) is a list in which the last node points to the first node (see figure). It is sometimes useful to have a *sentinel* last node, i.e. a node that does not contain data. The sentinel is used e.g. to check if we have traversed the whole list. An empty list contains only the sentinel node, that points to itself.

1 Scheme

1.1 Data structure definition and constructors (7 points)

Define a data structure for Clists (hint: use struct), together with a constructor for an empty Clist, and a variant of the *cons* operation for Clists, which adds a new element as the head of the previous Clist.

1.2 Map (4 points)

Define *cmap*, a map operation for Clists.

2 Haskell

2.1 Type definition and Eq (5 points)

Define a data structure for Clists with a *data* declaration. Make Clist an instance of Eq – beware: equality test must always terminate.

2.2 Conversions from/to ordinary lists (6 points)

Define two functions *list2clist* and *clist2list*, that are used to convert an ordinary list to a Clist, and vice versa. Write their types.

2.3 Map (6 points)

Define *cmap*, a map operation for Clists. Write its type.

3 Prolog (5 points)

Define a predicate with one argument to check if a given string is a palindrome.
E.g. `palindrome("sator arepo tenet opera rotas")` should return true.

Solutions

Scheme

```
(struct cnode (value next) #:mutable)
(define *end* '---end---)

(define (cend) ; builds a sentinel node
  (let ((node (cnode *end* #f)))
    (set-cnode-next! node node)
    node))

(define (cend? clist)
  (and (cnode? clist)
       (eq? (cnode-value clist) *end*)))

(define (get-end clist)
  (if (cend? clist)
    clist
    (get-end (cnode-next clist))))

(define (ccons x node)
  (if (cend? node)
    (let ((out (cnode x #f)))
      (set-cnode-next! out node)
      (set-cnode-next! node out)
      out)
    (let ((the-end (get-end node))
          (out     (cnode x node)))
      (set-cnode-next! the-end out)
      out)))

(define (cmap f v)
  (if (cend? v)
    (cend)
    (ccons (f (cnode-value v))
           (cmap f (cnode-next v)))))
```

Haskell

```
data Clist a = Node a (Clist a) | End (Clist a)

instance (Eq a) => Eq (Clist a) where
  End _ == End _ = True
  (Node x next) == (Node x' next') = (x == x') && next == next'
  _ == _ = False

clist2list :: Clist a -> [a]
clist2list (End v)       = clist2list v
clist2list (Node x next) = x : clist2list next

list2clist :: [a] -> Clist a
list2clist []     = let new = End new
                    in new
list2clist (x:xs) = let first = Node x $ list2clist' xs first
                    in  first
list2clist' [] first     = End first
list2clist' (x:xs) first = Node x $ list2clist' xs first

cmap :: (t -> a) -> Clist t -> Clist a
cmap f (Node x next) = let first = Node (f x) $ cmap' f next first
                       in  first
cmap' f (End x) first       = (End first)
cmap' f (Node x next) first = Node (f x) $ cmap' f next first
```

Prolog

```
pal(X) :- reverse(X,Y), X == Y.
```