Week 04 - Lecture 3 Slides

Dr. Yeganeh Bahoo

Created: 2023-10-01 Sun 21:41

Table of Contents

Lecture 3: Programmer-defined Linked Lists

By the end of this lecture students should be able to

- implement the abstract data type list as a linked list using the node and reference (aka "cons cell") pattern

What if we were to create our own list data type

- This may be quite common if the built-in list data type does not suit your problem
- Structure

- a **node** tuple stores information about a data item in the list and the next item in the list. It consists of two slots:
    - **data**: a reference to the data item
    - **next**: a reference to the next node
- the sequence of items is a chain of such nodes
- a list object will be represented as a tuple with two slots:
    - **head**: a pointer to the node at the head of the list
    - **size**: the length of the list (**CAVEAT LECTOR**: keeping track of size increases storage requirement and complicates list update operations)

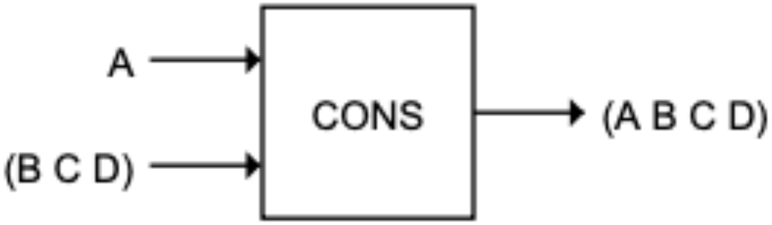
```
(defstruct node
  data next)

(defstruct my-list
  (head nil :type (or node null)) ; HEAD can be a node or nil
  (size 0 :type (integer 0))) ; SIZE is a positive integer
```

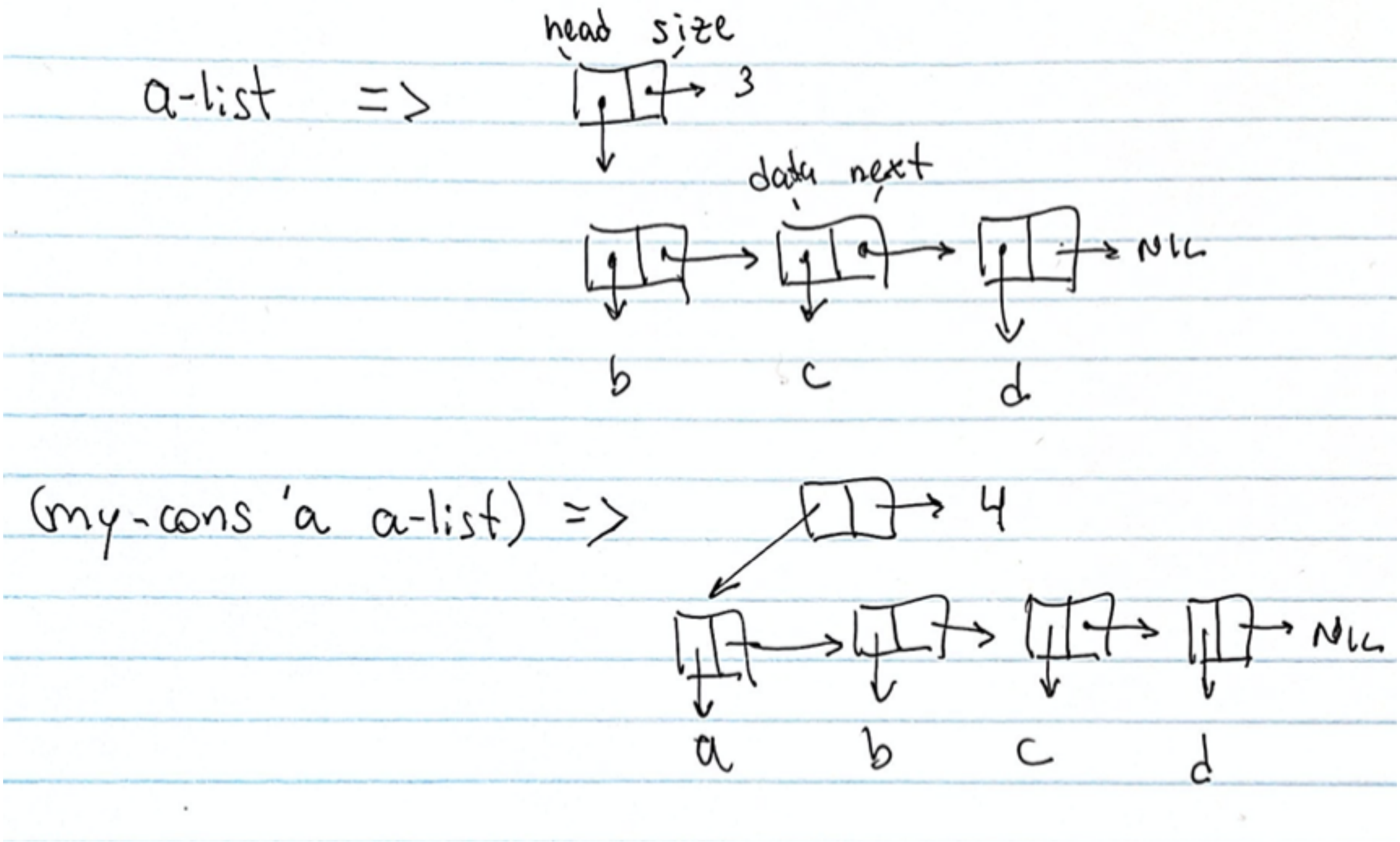
Linked lists: constructors

Let's define a MY-CONS constructor: `(MY-CONS ITEM LIST)`

Works like lisp's CONS



but operates on our own list data type



Constructors: MY-CONS

```
(defun my-cons (data list)
  "Creates new node, initializes it as appropriate linking it to the list,
   creates a new list head tuple, storing in its head slot a reference to the
   new first element of the list and the new list size"
  (let ((new-head (make-node :data data
                             :next (my-list-head list))))
```

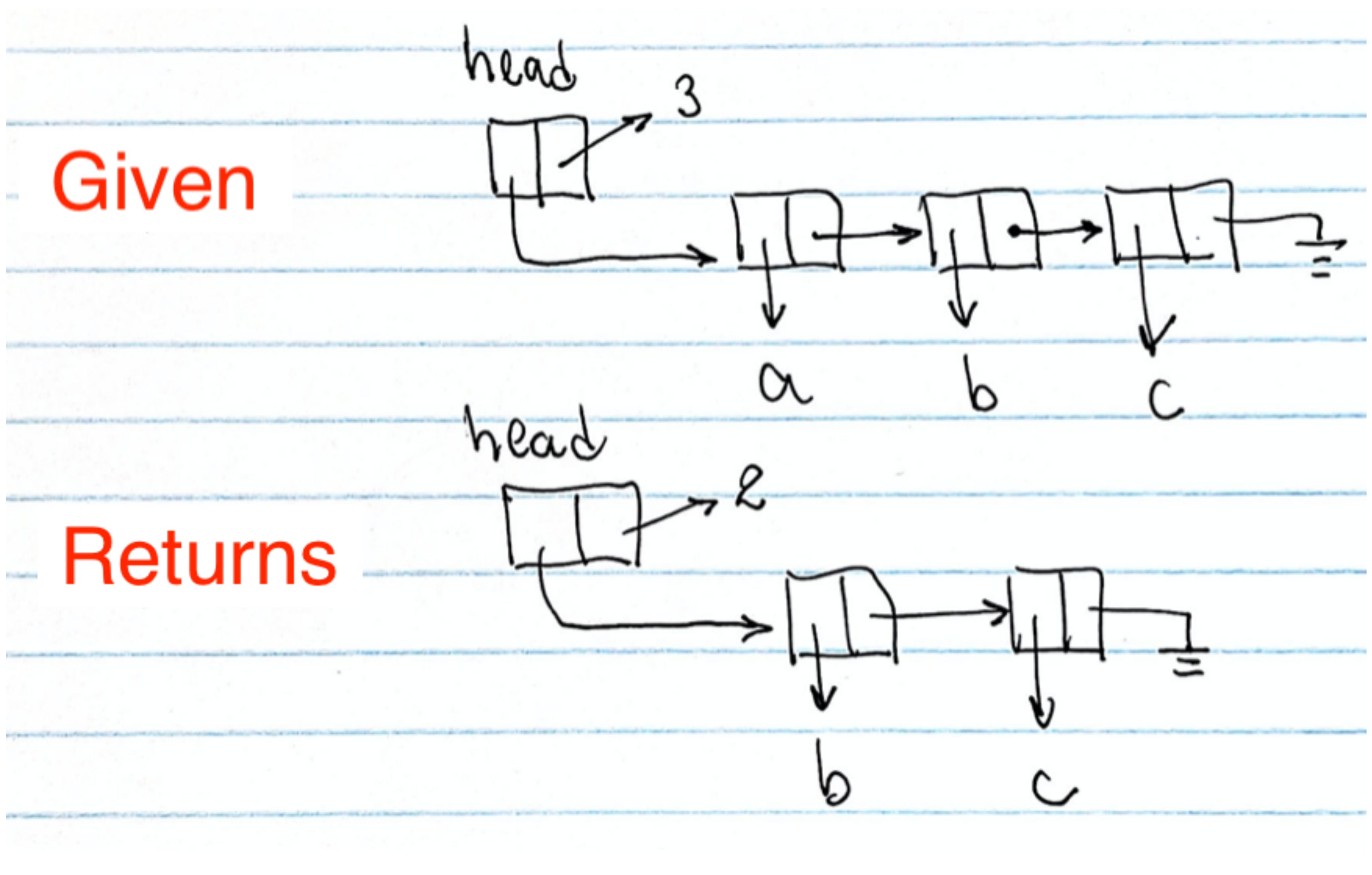
```
    (make-my-list :head new-head
                  :size (1+ (my-list-size list)))))
```

```
CL-USER> (my-cons 'a (make-my-list))
#S(MY-LIST :HEAD #S(NODE :DATA A :NEXT NIL)
           :SIZE 1)               ; In lisp list notation: (A)
CL-USER> (my-cons 'b (my-cons 'a (make-my-list)))
#S(MY-LIST
   :HEAD #S(NODE :DATA B :NEXT #S(NODE :DATA A :NEXT NIL))
   :SIZE 2)                            ; In lisp list notation: (B A)
```

Exercise

Let's implement the MY-CDR accessor. It works analogously to Lisp's REST / CDR. That is, when given a list, MY-CDR returns everything in the given list but the first element.

For example



Exercise (cont.)

Given the definitions below, complete the blanks to define MY-CDR:

```
(defstruct node
    data next)
```

```
(defstruct my-list
  (head nil :type (or node null))
  (size 0 :type (integer 0)))

(defun is-empty (list)
  (equalp list (make-my-list))) ;using EQUALP to compare structures. Can't use EQUAL

(defun my-car (alist)
  (node-data (my-list-head alist)))

(defun my-cdr (alist)
  (if (is-empty alist) alist
      (make-my-list :head ...
                    :size ...
```

```
CL-USER> (my-cdr (my-cons 'b (my-cons 'a (make-my-list))))
#S(MY-LIST :HEAD #S(NODE :DATA A :NEXT NIL) :SIZE 1)
```

Solution

```
(defun my-cdr (alist)
  (if (is-empty alist) alist
      (make-my-list :head (node-next (my-list-head alist))
                    :size (1- (my-list-size alist)))))
```

Accessors (cont.)

Complete the blanks in the MY-ELT function below that works analogously to lisp's ELT function (AREF for lists).

```
(defun my-elt (list index)
  "Returns the element in position 'index' in the list;
   otherwise returns NIL"
  (dotimes (i (my-list-size list))
    (if (= i index) (return ...)
        (setf list (my-cdr list)))))
```

```
CL-USER> (let ((alist (my-cons 'c (my-cons 'b (my-cons 'a (make-my-list))))))
           (values (my-elt alist 0)
                   (my-elt alist 1)
                   (my-elt alist 2)))
C
```

```
B
A
```

Solution

```
(defun my-elt (list index)
  "Returns the element in position 'index' in the list;
otherwise returns NIL"
  (dotimes (i (my-list-size list))
    (if (= i index) (return (my-car list))
        (setf list (my-cdr list))))))
```

Homework Exercise

Function MY-SEARCH uses iteration. It returns the item if the item is in the list; otherwise it returns NIL.

Complete the blanks in function MY-SEARCH-REC that does the same thing but using recursion instead of iteration.

NOTICE: we can't use lisp's built-in DOLIST because it operates on lisp's built-in lists

```
(defun my-search (item alist)
"Returns the item if it is in the linked list; otherwise returns NIL"
  (dotimes (i (my-list-size alist))
    (if (equalp item (my-car alist)) (return item)
        (setf alist (my-cdr alist)))))

(defun my-search-rec (item alist)
"Returns the item if it is in the linked list; otherwise returns NIL"
  (cond ((is-empty alist) ...)
        ((equal ... item) item)
        (T (my-search-rec item ...))))
```

Solution

```
(defun my-search-rec (item alist)
"Returns the item if it is in the linked list; otherwise returns NIL"
  (cond ((is-empty alist) nil)
        ((equal (my-car alist) item) item)
        (T (my-search-rec item (my-cdr alist)))))
```