Week 04 - Lecture 2 Slides

Dr. Yeganeh Bahoo

Created: 2023-10-01 Sun 22:03

Table of Contents

Lecture 2: Lists

By the end of this lecture students should be able to

- write programs that use lists

Lists

- Lists are a very useful and basic abstract data structure found in many modern languages
- Lists are a central data structure in lisp
- They are much more flexible than arrays, and allow the programmer to represent any abstract type: sets, tables, graphs, and even english sentences.

What do lists look like?

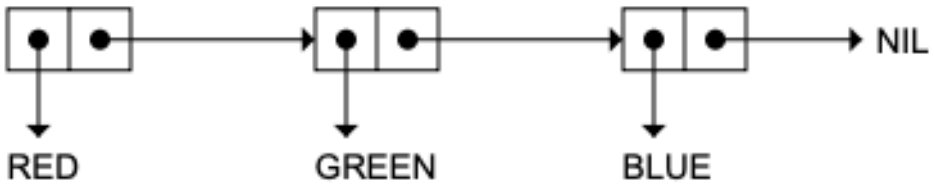
Examples of lists in lisp:

```
(43 12 45 3 -1)

(RED GREEN BLUE)

(3 FRENCH HENS 2 TURTLE DOVES 1 PARTRIDGE 1 PEAR TREE)

(defun sum-n1 (n)      ; A lisp program, as the "LisP" suggests, is a list
  (do ((i 1 (1+ i))
       (sum 0 (+ i sum)))
      ((> i n) sum)))
```

Internal representation (aka "cons cell" structure)

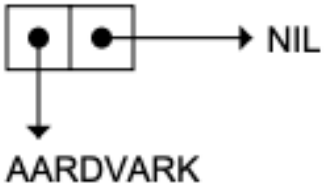
The computer's internal representation of lists

- `(RED GREEN BLUE)`

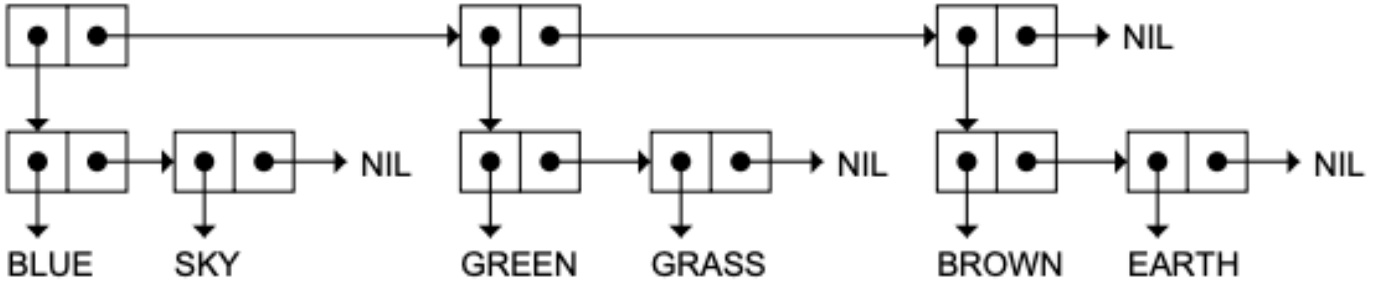


  Notice the cons cell chain ends in NIL. This is a convention in Lisp.

- `(AARDVARK)`



- `((BLUE SKY) (GREEN GRASS) (BROWN EARTH))`



Quiz

https://bit.ly/467YlhR

NIL: the empty list

- A list of zero elements can be written as either `()` or `NIL`
  - E.g.: `(A NIL B)` can also be written `(A () B)`
- `(length NIL) => 0`
- `(length ()) => 0`

List accessors

- Lisp's primitive functions for extracting elements from a list
  - `(first '(a b c d)) => a`
  - `(car '(a b c d)) => a`

    - `(second '(a b c d)) => b`
    - `(third '(a b c d)) => c`
- REST and CDR return a list containing everything **but** the first element.
    - `(rest '(a b c d)) => (b c d)`
    - `(cdr '(b c d)) => (c d)`
    - `(rest (cdr '(a b c d))) => (c d)`

Let's use CAR and CDR to define FORTH

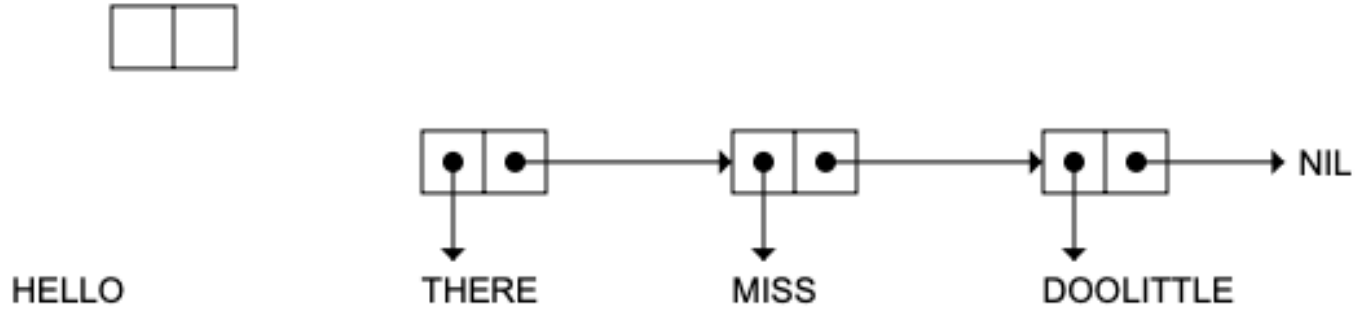
```
(defun forth (a)
  (car (cdr (cdr (cdr a)))))
```

Alternatively

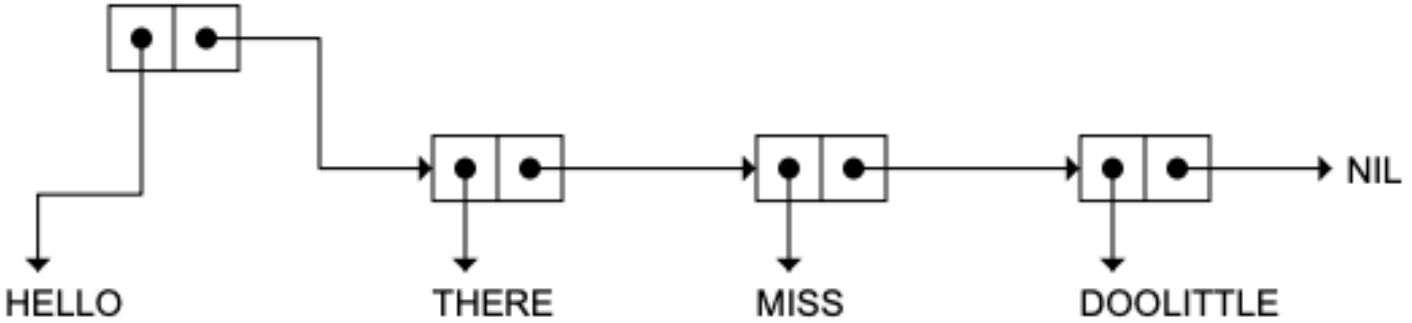
```
(defun forth (a)   ; Ask ChatGPT for the car and cdr combinations available in Lisp
  (cadddr a))
```

The list CONStructor

- CONS creates a cons cell
    - takes two parameters
    - returns a pointer to a new cons cell whose CAR points to the first parameter and whose CDR points to the second
- E.g.:
    - `(cons 'sink '(or swim)) => (sink or swim)`
    - `(cons 'sink ()) => (sink)`
    - `(cons '(or swim) ()) => ((or swim))`
    - `(cons 'hello '(there miss doolittle)) =>` First, CONS creates a cons cell:



Then it fills in the CAR and CDR pointers:



Examples

Creating a list using recursion and using iteration:

```
(defun mymake-list-rec (n element &optional (acc nil))
  (if (= n 0) acc
    (mymake-list-rec (1- n) element (cons element acc))))

(defun mymake-list-it (n elem)
  (let ((acc nil))
    (dotimes (i n acc)
      (setf acc (cons elem acc))))) ; alternatively (push elem acc)
```

```
CL-USER> (mymake-list-rec 3 'a)
 (A A A)
RTL-USER> (mymake-list-it 4 'a)
 (A A A A)
```

Creating a list from a bunch of elements

Lisp provides three list constructors

- QUOTE
- MAKE-LIST
- LIST

```
RTL-USER> '("hello" world 111)  ; Caveat: this creates a literal (constant) list.
("hello" WORLD 111)             ; It's contents should not be changed.
RTL-USER> (make-list 3)
(NIL NIL NIL)
RTL-USER> (make-list 3 :initial-element 'a)
(A A A)
RTL-USER> (make-list 3 :initial-contents '(a b c))
(A B C)
RTL-USER> (list "hello" 'world 111)
("hello" WORLD 111)
```

DOLIST: Looping across the elements of a list

```
(DOLIST (var list-form [result-form])
  body-form*)
```

- First `list-form` is evaluated once to produce a list.
- Then the `body-form*` is evaluated once for each item in the list with the variable `var` holding the value of the item.
- lastly, if `result-form` is provided, it is evaluated and its value is returned; otherwise DOLIST returns NIL

```
  CL-USER> (dolist (x '(1 2 3)) (print x))
1
2
3
NIL
```

Used this way, the DOLIST form as a whole evaluates to NIL.

If you want to break out of a DOLIST loop before the end of the list, you can use RETURN.

```
CL-USER> (dolist (x '(1 2 3))
```

```
          (print x)
          (if (evenp x) (return "DONE!")))
1
2
"DONE!"
```

Homework exercise

The function below returns the list of odd numbers present in its parameter. It uses iteration.

```
(defun get-odds-it (alist)
  (let ((acc nil))
    (dolist (elem alist acc)
      (if (oddp elem) (setf acc (cons elem acc))))))  ; alternatively (push elem acc)
```

Complete the blanks in the function bellow that uses recursion to do the same thing.

```
(defun get-odds (alist &optional (acc ...))
  (cond ((null alist) ...)
        ((oddp ...) (get-odds ... (cons ... acc)))
        (t (get-odds ... acc))))
```

```
RTL-USER> (get-odds '(3123 1 21  2 1  313 412))
(313 1 21 1 3123)
```

Solution

```
(defun get-odds (alist &optional (acc nil))
  (cond ((null alist) acc)
        ((oddp (car alist)) (get-odds (cdr alist) (cons (car alist) acc)))
        (t (get-odds (cdr alist) acc))))
```