Common Lisp LISTS

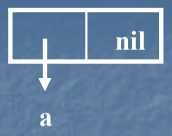
Some material adapted from J.E. Spragg, Mitthögskolan

Lists in Lisp

- Lists are Lisp's fundamental data structures.
- However, it is not the only data structure.
  - There are arrays, characters, strings, etc.
  - Common Lisp has moved on from being merely a LISt Processor.
- However, to understand Lisp you must understand lists
  - common functions on them
  - how to build other useful data structures using them.

UMBC

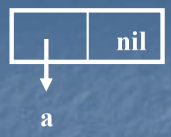
In the beginning was the cons



- What cons really does is combines two objects into a two-part object called a *cons*.
- Conceptually, a cons is a pair of pointers -- the first is the car, and the second is the cdr.
- Conses provide a convenient representation for pairs of any type.
- The two halves of a cons can point to any kind of object, including conses.
- This is the mechanism for building lists.
- (consp '(1 2)) => T

UMBC

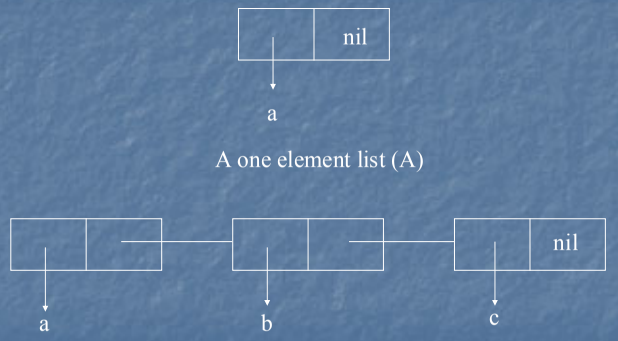
Pairs



- Lists in CL are defined as pairs.
- Any non empty list can be considered as a pair of the first element and the rest of the list.
- We use one half of the cons to point to the first element of the list, and the other to point to the rest of the list (which is either another cons or nil).

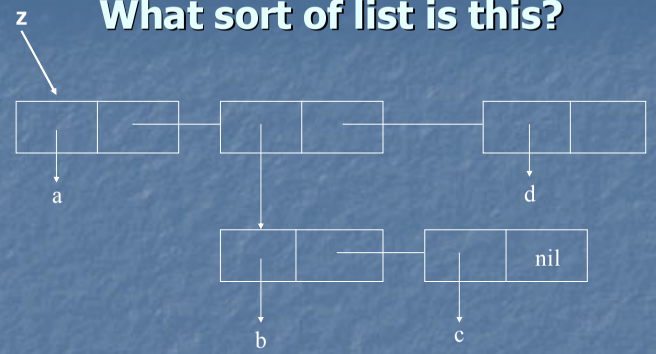
UMBC

Box notation



UMBC

What sort of list is this?



> (setf z (list 'a (list 'b 'c) 'd))
(A (B C) D)

➢(car (cdr z))
➢??

UMBC

Consp

- The function *consp* returns true if its argument is a cons.
- So listp could be defined:

```
(defun myListp (x) (or (null x) (consp x)))
```

- Since everything that is not a cons is an atom, the predicate *atom* could be defined:

```
(defun myAtom (x) (not (consp x)))
```

- Remember, *nil* is both an *atom* and a *list*.

UMBC

Equality

- Each time you call cons, Lisp allocates a new piece of memory with room for two pointers.
- So if we call cons twice with the same arguments, we get back two values that look the same, but are in fact distinct objects:

```
> (setf l1 (cons 'a nil) l2 (cons 'a nil))
(A)
> (eql l1 l2)
NIL
> (equal l1 l2)
T
> (and (eql (car l1)(car l2)) (eql (cdr l1)(cdr l2)))
T
```

UMBC

Equal

- We also need to be able to ask whether two lists have the same elements.
- CL provides an equality predicate for this, *equal*.
- eql returns true only if its arguments are the same object, and
- equal, more or less, returns true if its arguments would print the same.

```
> (equal l1 l2)
T
```

Note: if x and y are eq, they are also equal.

equal

```
defun equal (x y)
   ; this is ~ how equal could be defined
   (cond ((numberp x) (= x y))
         ((atom x) (eq x y))
         ((atom y) nil)
         ((equal (car x) (car y))
          (equal (cdr x) (cdr y)))))
```


Function agrguments

- Defun allows one to define functions which
  - Take optional arguments
  - Take arbitrarily many arguments
  - Take keyword arguments
- This can be very useful

```
(defun f (x & optional y (z 100) &rest therest)
      (list x y z therest))

(f 1) => (1 nil 100 nil)
(f 1 2 3) = (1 2 3 nil)
(f 1 2 3 4 5 6 7) => (1 2 3 (4 5 6 7))
```


Does Lisp have pointers?

- One of the secrets to understanding Lisp is to realize that variables have values in the same way that lists have elements.
- As conses have pointers to their elements, variables have pointers to their values.
- What happens, for example, when we set two variables to the same list:

```
> (setf  x  '(a  b  c))
(A B C)
> (setf  y  x)
(A B C)
```

This is just like in Java

Does Lisp have pointers?

- The location in memory associated with the variable x does not contain the list itself, but a pointer to it.
- When we assign the same value to y, Lisp copies the pointer, not the list.
- Therefore, what would the value of

  > (eql x y)

  be, T or NIL?

Building Lists

- The function *copy-list* takes a list and returns a copy of it.
- The new list will have the same elements, but contained in new conses.

```
> (setf x '(a b c)
        y (copy-list x))
(A B C)
```

- Spend a few minutes to draw a box diagram of x and y to show where the pointers point.

Copy-list

Copy-list is a built in function but could be defined as follows

```
(defun copy-list (s)
  (if (atom s)
      s
      (cons (copy-list (car s))
            (copy-list (cdr s)))))
```

There is also a copy-tree that makes a copy of the entire s-expression.

Append

- The Common Lisp function *append* returns the concatenation of any number of lists:

```
> (append '(a b) '(c d) '(e))
(A B C D E)
```

- *Append* copies all the arguments except the last.
  - If it didn't copy all but the last argument, then it would have to modify the lists passed as arguments. Such side effects are very undesirable, especially in functional languages.

append

- The two argument version of append could have been defined like this.

```
(defun append2 (s1 s2)
    (if (null s1)
        s2
        (cons (car s1)
              (append2 (cdr s1) s2))))
```

- Notice how it ends up copying the top level list structure of it's first argument.

List access functions

- To find the element at a given position in a list we call the function *nth*.

```
> (nth  0  '(a  b  c))
A
```

- and to find the *n*th cdr, we call *nthcdr*.

```
> (nthcdr  2  '(a  b  c))
(C)
```

- Both nth and nthcdr are zero indexed.

nth and nthcdr

```
(defun nth (n l)
  (cond ((null l) nil)
        ((= n 0) (car l))
        (t (nth (- n 1) (cdr l)))))

(defun nthcdr (n l)
  (cond ((null l) nil)
        ((= n 0) (cdr l))
        (t (nthcdr (- n 1) (cdr l)))))
```


Accessing lists

- The function *last* returns the last cons in a list.

```
> (defun last (l)
    (cond ((null l) nil)
          ((null (cdr l)) l)
          (t (last (cdr l)))))
last
> (last  '(a  b  c))
(C)
```

- We also have: *first*, *second* … *tenth*, and *CxR*, where *x* is a string of up to four **a**s or **d**s.
  - E.g., cadr, caddr, cddr, cdadr, …

Mapping functions

- Common Lisp provides several mapping functions.
- ***Mapcar*** is the most frequently used.
- It takes a function and one or more lists, and returns the result of applying the function to elements taken from each list, until one of the lists runs out:

```
> (mapcar #'abs '(3 -4 2 -5 -6))
(3 4 2 5 6)
> (mapcar #'cons '(a b c) '(1 2 3))
((a . 1) (b . 2) (c . 3))
 > (mapcar  #'(lambda  (x)  (+  x  10))  '(1  2  3))
(11  12  13)
> (mapcar  #'list  '(a  b  c)  '(1  2  3  4))
((A  1)  (B  2)  (C  3))
```


Maplist

- The related function *maplist* takes the same arguments, but calls the function on successive cdrs of the lists:

```
> (maplist  #'(lambda  (x)  x)  '(a  b  c))
((A  B  C)  (B  C)  (C))
> (maplist #'(lambda (x) (cons 'foo x))  '(a b c d))
((foo a b c d) (foo b c d) (foo c d) (foo d))
> (maplist #'(lambda (x) (if (member (car x) (cdr x)) 0 1)))
  '(a b a c d b c))
(0 0 1 0 1 1 1)
```

- There is also *mapcan, mapc, and mapl.* Use the on-line *Common Lisp the Language* to discover what these mapping functions do.

Member

- *Member* returns true, but instead of simply returning *t*, its returns the part of the list beginning with the object it was looking for.

```
> (member  'b  '(a  b  c))
(B  C)
```

- By default, *member* compares objects using *eql.*
  - You can override this behavior by employing a *keyword* argument.

Keyword arguments

- An example of a *keyword* argument is **:test.**
- If we pass some function as the :test argument in a call to member, than that function will be used to test for equality instead of *eql.*

```
> (member  '(a)  '((a)  (z))  :test  #'equal)
((A)  (Z))
```

- *Keyword* arguments are always optional.
- Another of member's *keyword* arguments is **:key,** allowing one to specify a function to be applied to each element before comparison:

```
> (member  'a  '((a  b)  (c d))  :key  #'car)
((A  B)  (C  D))
```

How member could be defined

```
(defun member (s list &key (test #'eql)
                           (key #'(lambda (x) x))
  (cond ((atom list) nil)
        ((funcall test s (funcall key (car list))) list)
        (t (member s (cdr list)))))
```

In general

- a function can be defined to take any number of keyword arguments
- Each can have an associated default value
- The default default is NIL
- Keyword parameters have to come "last" and they key names preceded by a :

UMBC

Member-if

- If we want to find an element satisfying an arbitrary predicate we use the function *member-if*:

```
> (member-if #'oddp '(2 3 4))
(3 4)
```

Which could be defined like:

```
(defun member-if (f l)
   (cond ((null nil) nil)
         ((funcall f (car l)) l)
         (t (member-if f (cdr l)))))
```

UMBC

adjoin

- The function *adjoin* is like a conditional *cons*.
- It takes an object and a list, and conses the object onto the list only if it is not already a member:

```
> (adjoin 'b '(a b c))
(A B C)
> (adjoin 'z '(a b c))
(Z A B C)
```

UMBC

Sets

- CL has the functions, *union*, *intersection*, and *set-difference* for performing set operations on lists.
- These functions expect exactly two lists and also the same *keyword* arguments as *member*.
- Remember, there is no notion of ordering in a set. These functions won't necessarily preserve the order of the two lists.

UMBC

Set Union Example

Can you guess the algorithm used to compute union from this example?

```
> (setf l1 '(d a c b)  l2 '(c 4 2 1 3 d))
(C 4 2 1 3 D)
> (union l1 l2)
(A B C 4 2 1 3 D)
> l1
(D A C B)
> l2
(C 4 2 1 3 D)
```


Sort

- Common Lisp has a built in function called *sort*.
- It takes a sequence and a comparison function of two arguments, and returns a sequence with the same elements, sorted according to the function:

```
> (sort '(0 2 1 3 8) #'>)
(8 3 2 1 0)
```

- ***Sort is destructive***!!
  - What can you do if you don't want your list modified?

Every and Some

- *every* and *some* take a predicate and one or more sequences
- When given just one sequence, they test whether the elements satisfy the predicate:

```
> (every #'oddp '(1 3 5))
T
> (some #'evenp '(1 2 3))
T
```

- If given >1 sequences, the predicate must take as many arguments as there are sequences, and arguments are drawn one at a time from them:

```
> (every #'> '(1 3 5) '(0 2 4))
T
```


Push and Pop

- The representation of lists as conses makes it natural to use them as pushdown stacks.
- This is done so often that CL provides two macros for the purpose, *push*, and *pop*.
- Both are defined in terms of *setf*.

```
(push obj lst)
```

is the same as

```
(setf lst (cons obj lst)
```

Push and pop

```
(defmacro push (s var)
   `(setf ,var (cons ,s ,var)))

(defmacro pop (var)
  `(let  ((temp  (car ,var)))
         (setf  ,var (cdr ,var))
         temp))
```

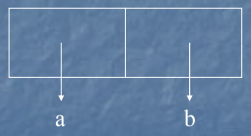

Dotted Lists

- The kind of lists that can be built by calling *list* are more precisely known as *proper lists*.
  A proper list is either *nil*, or a *cons* whose *cdr* is a proper list.
- However, conses are not just for building lists -- whenever you need a structure with two fields you can use a cons.
- You will be able to use *car* to refer to the first field and *cdr* to refer to the second.

```
> (setf  pair  (cons 'a  'b))
(A  .  B)
```

- Because this cons is not a proper list, it is displayed in *dot notation*.
  In dot notation the car and cdr of each cons are shown separated by a period.

- A cons that isn't a proper list is called a dotted list.
- However, remember that a dotted list isn't really a list at all.
- It is a just a two part data structure.



(A . B)

Assoc-lists

- It is natural to use conses to represent mappings.
- A list of conses is called an association list otr *assoc-list* or *alist*.
- Such a list could represent a set of translations, for example:

```
> (setf  languages '((lisp .  easy) (C .
  hard) (Pascal  .  good)  (Ada  .  bad)))
```

Assoc-lists

- Assoc-lists are slow, but convenient when engaged in rapid prototyping.
- Common Lisp has a built in function, *assoc*, for retrieving the pair associated with a given key:

```
> (assoc 'C languages)
(C . HARD)
> (assoc 'Smalltalk languages)
NIL
```

- Like *member*, *assoc* takes *keyword* arguments, including **:test** and **:key.**
- Most uses of alists have been replaced by hashtables

assoc

- Here's how assoc (w/o key word arguments) could be defined:

```
(defun assoc (key alist)
   (cond  ((null alist) nil)
          ((eql key (caar alist)) (car alist))
          (t (assoc key (cdr alist)))))
```