Computer Science 4150
Artificial Intelligence
Winter, 2005

Instructor:
Nick Cercone - CSBldg 105
902-494-2832 - nick@cs.dal.ca

Quickie Introduction to Lisp

LISP

- due to John McCarthy - "Recursive functions of symbolic expressions and their computation by machine", Communications of the ACM, April, 1960.

- "To iterate is human, to recurse divine"

- Whenever the problem domain is typically characterised by problems with ill-defined data requirements, think of LISP.

1.0 Basic Building Blocks

atoms - either a number of a string of alphanumeric characters containing no delimiters, for example, 105, SPIRO, XYZ, THISISAVERYLONGATOM.

lists - the basic data structure, an ordered set of elements enclosed in a set of parentheses, for example, (PUT (THE BLOCK) (IN THE BOX))

() = NIL is a special atom/list representing the logical value FALSE

t - is a special atom representing the logical value TRUE

2.0 Evaluation

The way LISP programs operate by interpreting lists. The first element of a list is interpreted as the name of a function to be applied to the set of arguments given by the rest of the list, for example,

(+ 3 4)
(* (PLUS 3 4) 2 2)

3.0 Symbolic Expressions

Lisp deals with objects called symbolic expressions (S-EXPRS). The set of symbolic expressions is defined inductively over a base set named <atom>. The elements of <atom> are called atoms, for example,

<atom> ? <literal atom> | <numeral>

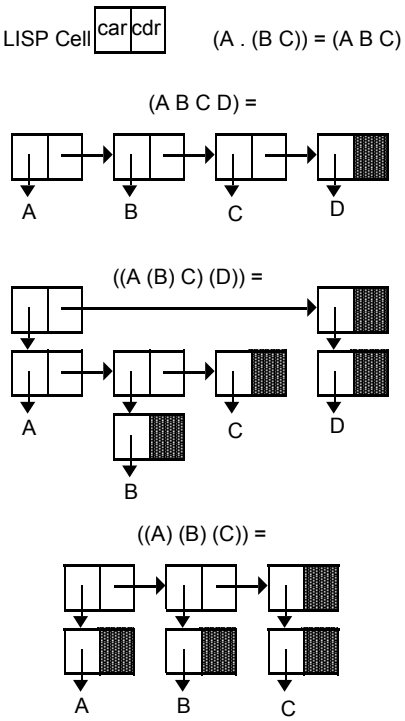
<literal atom> ? <atom letter> | <literal atom> <atom letter> | <literal atom> <digit>

<numeral> ? <digit> | <numeral> <digit>

<atom letter> ? A | B | C | ... | Z

<digit> ? 0 | 1 | 2 | ... | 9

4.0 LISP Storage Structures

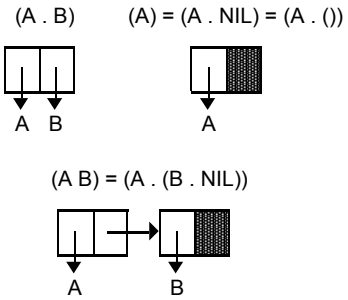


The domain of S-EXPR's is defined inductively over the domain <atom>:

1. *Any element of <atom> is an element of <s-expr>*
2. *If A1 and A2 are elements of <s-expr>, then <A1 . A2> is an <s-expr> and is normally referred to as a 'dotted pair'.*

<s-expr> ? <atom> | (<s-expr> . <s-expr>)

Examples:



5.0 Lists and Dot Notation

IDENTITY-1 A list of one atom is a dotted pair of the atom and NIL, with NIL always the right part of the dotted pair, that is, (atom) = (atom . NIL)

LIST NOTATION TO DOT NOTATION

The first (leftmost) list element, when transformed to dot notation, is always the left part of a dotted pair. If the first element is also the last element of the list, by Identity-1, it is dotted with NIL. If the first element is not the last element of the list then the right part of the dotted pair is the list formed by removing the first element. Then apply this rule to the right part of the dotted pair.

DOT NOTATION TO LIST NOTATION

Only those dotted pairs in which the only atom adjacent to a right parenthesis is NIL can be represented in list notation.

(1) If the right part of the dotted pair is an atom and not NIL, conversion is impossible.

(2) If the right part of the dotted pair is non-atomic (or NIL) then
- delete the last right parenthesis of the dotted pair.
- delete the dot.
- delete the first left parenthesis of the right part.
- repeat this procedure on the remaining dotted pairs.

examples: (A . (B . NIL)) = (A . (B)) = (A B)
(A . ((B . C) . (D . NIL))) =
(A . ((B . C) . (D))) =
(A . ((B . C) D)) = (A . (B . C) D)

6.0 Basic List Manipulating Functions

```
% cl
Allegro CL 3.1.20 [NeXT] (3/7/91)
Copyright (C) 1985-1990, Franz Inc., Berkeley, CA, USA
<cl> (car '(a b c))                    ; car examples
a
<cl> (car '((a b c) d e f))
(a b c)
<cl> (car '(((a) b) c (d)))
((a) b)
<cl> (cdr '(a b c))                    ; cdr examples
(b c)
<cl> (cdr '((a b c) d e f))
(d e f)
<cl> (cdr '(((a) b) c (d)))
(c (d))
<cl> (cons 'a '(b c))                  ; cons examples
(a b c)
<cl> (cons '(a b c) '(d e f))
((a b c) d e f)
<cl> (cons '((a) b) '(c (d)))
(((a) b) c (d))
<cl> (cadr '(1 2 3 4 5 6 7 8 9))       ; shorthand
2
<cl> (cadddr '(1 2 3 4 5 6 7 8 9))
4
<cl> (cddddr '(1 2 3 4 5 6 7 8 9))
(5 6 7 8 9)
<cl> (cdddr '(1 2 3 4 5 6 7 8 9))
(4 5 6 7 8 9)
<cl> (cddr '(1 2 3 4 5 6 7 8 9))
(3 4 5 6 7 8 9)
<cl> (setq x '(((a) (b (c) d)) ((((e)))) f (g) (h (i (j))) k))
(((a) (b (c) d)) ((((e)))) f (g) (h (i (j))) k)
<cl> (car x)
((a) (b (c) d))
<cl> (cadr x)
((((e))))
<cl> (caaadr x)
((e))
<cl> (cddar x)
nil
<cl> (cdar x)
((b (c) d))
<cl> (exit)
; Exiting Lisp
%
```

7.0 Functions

- more widely used in LISP than in any other programming language
- built-in and user-defined functions
- example (from MACLISP)

$f(x,y) = x^2 + 7xy + 5$

```
(defun f (x y)
    (plus (square x) (times 7 x y) 5))
```

8.0 Variables

- any non-numeric atom name can be used as a variable name
- values assigned to variables in one of two ways:
  1. *when function definitions are applied to arguments*
  2. *use SETQ or SET*

9.0 Lambda Notation

given y^x, evaluate the expression for 3 and 4:

problem ? is x=3 and y=4 or is y=3 and x=4?

solution ? Church's lambda notation

y^x is called a 'form', written in LISP as (exp y x)

f=lambda(x,y) y^x is a 'function' since it provides

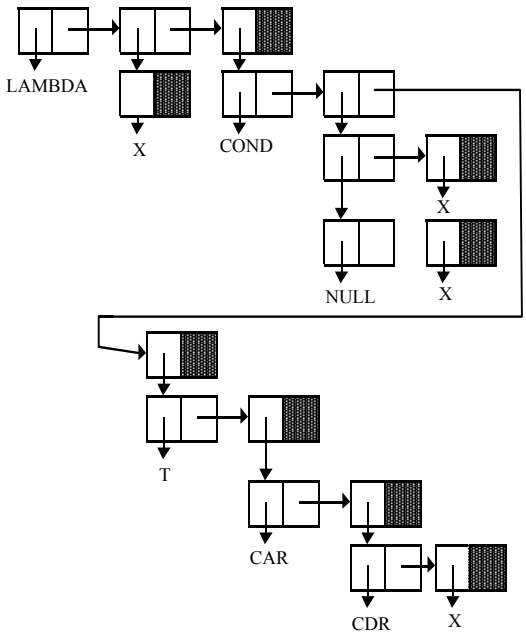
1. a form to be evaluated
2. correspondance between variables of form and arguments of function

thus f(3,4) = 43 = 64. In LISP

```
(lambda (x y)  (exp y x) ) (3 4)
    list of vars - form
```

10.0 Executable Form (list structure) of a LISP Program

```
(lambda (x)
        (cond ((null x) x)
              (t  (car (cdr x)))))
```

11.0 Translation and Execution of Programs

1. LISP function definitions take the form of list structures

2. Function definition is then READ (as any other list structure)

3. Resulting structure may then be interpreted
   (a) as data
   (b) as executable function by the primitive APPLY
      (1) APPLY accepts as arguments
         - function definitions (LAMBDA-expr's)
         - list of actual parameters
         - a list to be used in resolving non-local references during execution
      (2) APPLY operates
         - evaluating actual parameters
         - pairing the values with the formal parameters of the function

12.0 Some Function Definition Examples

First some Franz Lisp examples:

```
% lisp
Franz Lisp, Opus 38.79
-> (load 'membership)
[load membership]
t
-> (pp membership)
(def membership
  (lambda (a l)
    (cond ((null l) nil) ((equal a (car l)) t)
          (t (membership a (cdr l))))))
t
-> (membership 'b '(a b c))
t
-> (membership 'd '(a b c))
nil
-> (membership '(a) '(a b (a) c d))
t
-> (membership '(a ((b))) '(x y ((((z)))) (a ((b)))))
t

-> (load 'tally)
[load tally]
t
-> (pp tally)
(def tally
  (nlambda (l)
    (prog (ans)
      (setq ans 0)
    x (cond ((null l) (return ans))
            (t (setq ans (add1 ans)) (setq l (cdr l)) (go x))))))
t
-> (tally 1 2 3 4 5 6 7 8 9)
9
-> (tally a b c d e)
5
-> (tally x)
1
-> (tally)
0
-> (tally a b c d e f g h i j k l m n o p
     q r s t u v w x y z)
26

-> (load 'listatoms)
[load listatoms]
t
-> (pp listatoms)
; PROGram  variable initialised to NIL, RETURN
; explicily exits at point of invocation,
; ill-advised use of a GO loop
;
(def listatoms
  (lambda (x)
    (prog (res)
loop (cond ((null x) (return res))
           ((atom (car x)) (setq res (cons (car x) res)))
       (t (setq res (append (listatoms (car x))
                            res))))
     (setq x (cdr x))
     (go loop))))
t
-> (listatoms '(a b c))
(c b a)
-> (listatoms '(a (b) c))
(c b a)
-> (listatoms '(((((a)))))
(a)
-> (reverse (listatoms '(a ((b) c))))
(a b c)
-> (trace listatoms)
[autoload /usr/lib/lisp/trace]
[fasl /usr/lib/lisp/trace.o]
(listatoms)

-> (listatoms '(a (b (c) d)))
1 <Enter> listatoms ((a (b (c) d)))
|2 <Enter> listatoms ((b (c) d))
| 3 <Enter> listatoms ((c))
| 3 <EXIT>  listatoms  (c)
|2 <EXIT>  listatoms  (d c b)
1 <EXIT>  listatoms  (d c b a)
(d c b a)

-> (load 'flat)
[load flat]
t
-> (pp flatten)
(def flatten
  (lambda (l)
    (cond ((null l) nil)
          ((atom l) (list l))
          (t (append (flatten (car l))
                     (flatten (cdr l)))))))
t
```

```
-> (pp flat)
(def flat
  (lambda (x)
    (cond ((atom x) (list x)) (t (mapcan 'flat x)))))
t
-> (flatten '(a (b) c))
(a b c)
-> (flat '(a (b) c))
(a b c)
-> (flatten '((a) (b) (((((c))) d) e)))
(a b c d e)
-> (flat '((a) (b) (((((c))) d) e)))
(a b c d e)

-> (load 'perm)
[load perm]
t
-> (pp perm)
(def perm
  (lambda (l r)
    (cond ((null l))
          ((equal (length l) 1) (setq r l))
          (t (do i 1 (add1 i)
                 (greaterp i (length l))
                 (setq r (append r
                                 (mapcar '(lambda (x)
                                           (append (list (car l))
                                                   (cond ((atom x)(list x))
                                                         (t x))))
                                         (perm (cdr l) nil))))
                 (setq l (append (cdr l)
                                 (list (car l)))))))
  r))
t
-> (perm () ())
nil
-> (perm '(a) ())
(a)
-> (perm '(a b) ())
((a b) (b a))
-> (perm '(a b c) ())
((a b c) (a c b) (b c a) (b a c) (c a b) (c b a))
-> (perm '(a b c d) ())
((a b c d) (a b d c) (a c d b) (a c b d) (a d b c) (a d c b)
 (b c d a) (b c a d) (b d a c) (b d c a) (b a c d) (b a d c)
 (c d a b) (c d b a) (c a b d) (c a d b) (c b d a) (c b a d)
 (d a b c) (d a c b) (d b c a) (d b a c) (d c a b) (d c b a)
)
-> (exit)
%
```

Then some Common Lisp examples:

```
% cl
Allegro CL 3.1.20 [NeXT] (3/7/91)
Copyright (C) 1985-1990, Franz Inc., Berkeley, CA.
<cl> (load 'memcl)
; Loading /private/Net/res10w/home/dk5/nick/ursa/lisp/working/memcl.
t
<cl> (pp membership)
(defun membership (a l)
  (cond ((null l)
         nil)
        ((equal a (car l))
         t)
        (t
         (membership a (cdr l)))))
<cl> (membership 'b '(a b c))
t
<cl> (membership 'd '(a b c))
nil
<cl> (membership '(a) '(a b (a) c d))
t
<cl> (membership '(a ((b))) '(x y ((((z)))) (a ((b)))))
t
<cl> (load 'tallycl)
; Loading /private/Net/res10w/home/dk5/nick/ursa/lisp/working/tallycl.
t
<cl> (pp tally)
(defun tally (&rest l)
  (prog (ans)
    (setq ans 0)
 x  (cond ((null l)
           (return ans))
          (t
           (setq ans (1+ ans))
           (setq l (cdr l))
           (go x)))))
<cl> (tally 1 2 3 4 5 6 7 8 9)
9
<cl> (tally 'a 'b 'c 'd 'e)
5
<cl> (tally 1)
1

<cl> (load 'listatomscl)
; Loading /private/Net/res10w/home/dk5/nick/ursa/lisp/working/listatomscl.
t
<cl> (pp listatoms)
(defun listatoms (x)
  (prog (ans)
    loop
    (cond ((null x)
           (return ans))
          ((atom (car x))
           (setq ans (cons (car x) ans)))
          (t
           (setq ans (append (listatoms (car x))
                             ans))))
    (setq x (cdr x))
    (go loop)))
<cl> (listatoms '(a b c))
(c b a)
<cl> (listatoms '(a (b) c))
(c b a)
<cl> (listatoms '((((a)))))
(a)
<cl> (reverse (listatoms '(a ((b) c))))
(a b c)

<cl> (trace listatoms)
; Fast loading /usr/cl/lib/code/trace.fasl.
(listatoms)
<cl> (listatoms '(a (b (c) d)))
 0: (listatoms (a (b (c) d)))
   1: (listatoms (b (c) d))
     2: (listatoms (c))
     2: returned (c)
   1: returned (d c b)
 0: returned (d c b a)
(d c b a)

<cl> (load 'flatcl)
; Loading /private/Net/res10w/home/dk5/nick/ursa/lisp/working/flatcl.
t
<cl> (pp flatten)
(defun flatten (x)
  (cond ((null x)
         nil)
        ((atom x)
         (list x))
        (t
         (append (flatten (car x))
                 (flatten (cdr x))))))
<cl> (pp flat)
(defun flat (x)
  (cond ((atom x)
         (list x))
        (t
         (mapcan 'flat x))))

<cl> (flatten '(a (b) c))
```


```
(a b c)
<cl> (flat '(a (b) c))
(a b c)
<cl> (flatten '((a) (b) (((((c))) d) e)))
(a b c d e)
<cl> (flat '((a) (b) (((((c))) d) e)))
(a b c d e)

<cl> (load 'copycl)
; Loading /private/Net/res10w/home/dk5/nick/ursa/lisp/working/copycl.
t
<cl> (pp copy)
(defun copy (l)
  (cond ((null l)
         nil)
        ((atom l)
         l)
        (t
         (cons (copy (car l)) (copy (cdr l))))))
<cl> '(a b c)
(a b c)
<cl> (copy '(a b c))
(a b c)
<cl> (eq '(a b c) (copy '(a b c)))
nil
<cl> (equal '(a b c) (copy '(a b c)))
t

<cl> (exit)
; Exiting Lisp
```

This is an example which illustrates how to replace arbitrary symbols within lists

```
% cl
Allegro CL 3.1.20 [NeXT] (3/7/91)
Copyright (C) 1985-1990, Franz Inc., Berkeley, CA.
<cl> (load 'deepreplacecl)
; Loading /private/Net/res10w/home/dk5/nick/ursa/lisp/working/deepreplacecl.
t
<cl> (pp replace)
(defun replace (s1 s2 l)
  (cond ((null l)
         nil)
        ((eq (car l) s1)
         (cons s2 (replace s1 s2 (cdr l))))
        (t
         (cons (car l) (replace s1 s2 (cdr l))))))

<cl> (pp replac)
(defun replac (s1 s2 l)
  (cond ((null l)
         nil)
        (t
         (cons (cond ((eq (car l) s1)
                      s2)
                     (t
                      (car l)))
               (replac s1 s2 (cdr l))))))

<cl> (replace 'a 'b '(a b c a))
(b b c b)
<cl> (replac 'a 'b '(a b c a))
(b b c b)
<cl> (replace 'a 'b '(a (a b)))
(b (a b))
<cl> (replac 'a 'b '(a (a b)))
(b (a b))

<cl> (pp deepreplaceatoms)
(defun deepreplaceatoms (at1 at2 l)
  (cond ((null l) nil)
        ((eq l at1) at2)
        ((atom l) l)
        (t
         (cons (deepreplaceatoms at1 at2 (car l))
               (deepreplaceatoms at1 at2
                                 (cdr l))))))

<cl> (deepreplaceatoms 'a 'b '(a (a b)))
(b (b b))


<cl> (pp deepreplace)
(defun deepreplace (el1 el2 l)
  (cond ((null l)
         nil)
        (t
         (cons (deepreplace1 el1 el2 (car l))
               (deepreplace el1 el2 (cdr l))))))

<cl> (pp deepreplace1)
(defun deepreplace1 (el1 el2 l)
  (cond ((equal el1 l)
         el2)
        ((atom l)
         l)
        (t
         (deepreplace el1 el2 l))))

<cl> (deepreplace '(a) 'b '(x (a)))
(x b)

<cl> (trace replace replac deepreplaceatoms
       deepreplace)
; Fast loading /usr/cl/lib/code/trace.fasl.
(deepreplace deepreplaceatoms replac replace)

<cl> (replac 'a 'b '(a b c a))
 0: (replac a b (a b c a))
   1: (replac a b (b c a))
     2: (replac a b (c a))
       3: (replac a b (a))
         4: (replac a b nil)
         4: returned nil
       3: returned (b)
     2: returned (c b)
   1: returned (b c b)
 0: returned (b b c b)
(b b c b)

<cl> (deepreplaceatoms 'a 'b '(a (a b)))
 0: (deepreplaceatoms a b (a (a b)))
   1: (deepreplaceatoms a b a)
   1: returned b
   1: (deepreplaceatoms a b ((a b)))
     2: (deepreplaceatoms a b (a b))
       3: (deepreplaceatoms a b a)
       3: returned b
       3: (deepreplaceatoms a b (b))
         4: (deepreplaceatoms a b b)
         4: returned b
         4: (deepreplaceatoms a b nil)
         4: returned nil
       3: returned (b)
     2: returned (b b)
     2: (deepreplaceatoms a b nil)
     2: returned nil
   1: returned ((b b))
 0: returned (b (b b))
(b (b b))

<cl> (deepreplace '(a) 'b '(x (a)))
 0: (deepreplace (a) b (x (a)))
   1: (deepreplace (a) b ((a)))
     2: (deepreplace (a) b nil)
     2: returned nil
   1: returned (b)
 0: returned (x b)
(x b)

<cl> (load 'hanoicl)
; Loading /private/Net/res10w/home/dk5/nick/ursa/lisp/working/hanoicl.

t
<cl> (pp hanoi)
(defun hanoi (n start temp dest)
  (cond ((< n 1)
         (princ "|Can't have less than one disk!|)
         (terpri))
        ((= n 1)
         (print (list 'move 'disk n 'from start 'to dest)))
        (t
         (hanoi (1- n) start dest temp)
         (print (list 'move 'disk n 'from start 'to dest))
         (hanoi (1- n) temp start dest))))
<cl> (hanoi 0 'peg1 'peg2 'peg3)
```


Can't have less than one disk!

nil
<cl> (hanoi 1 'peg1 'peg2 'peg3)

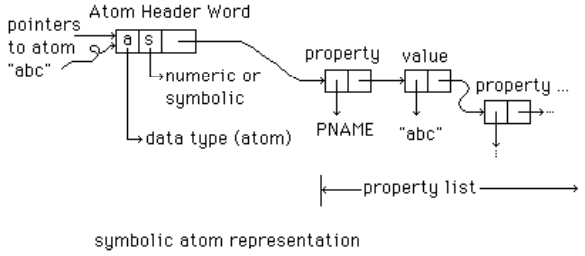
(move disk 1 from peg1 to peg3)
(move disk 1 from peg1 to peg3)
<cl> (hanoi 2 'peg1 'peg2 'peg3)

(move disk 1 from peg1 to peg2)
(move disk 2 from peg1 to peg3)
(move disk 1 from peg2 to peg3)
(move disk 1 from peg2 to peg3)
<cl> (hanoi 3 'peg1 'peg2 'peg3)

(move disk 1 from peg1 to peg3)
(move disk 2 from peg1 to peg2)
(move disk 1 from peg3 to peg2)
(move disk 3 from peg1 to peg3)
(move disk 1 from peg2 to peg1)
(move disk 2 from peg2 to peg3)
(move disk 1 from peg1 to peg3)
(move disk 1 from peg1 to peg3)
<cl> (hanoi 4 'peg1 'peg2 'peg3)

(move disk 1 from peg1 to peg2)
(move disk 2 from peg1 to peg3)
(move disk 1 from peg2 to peg3)
(move disk 3 from peg1 to peg2)
(move disk 1 from peg3 to peg1)
(move disk 2 from peg3 to peg2)
(move disk 1 from peg1 to peg2)
(move disk 4 from peg1 to peg3)
(move disk 1 from peg2 to peg3)
(move disk 2 from peg2 to peg1)
(move disk 1 from peg3 to peg1)
(move disk 3 from peg2 to peg3)
(move disk 1 from peg1 to peg2)
(move disk 2 from peg1 to peg3)
(move disk 1 from peg2 to peg3)
(move disk 1 from peg2 to peg3)
<cl> (exit)
; Exiting Lisp

13.0 Property Lists



symbolic atom representation