CS 4480
LISP

September 10, 2010

Based on slides by

Istvan Jonyer

Book by

MacLennan

Chapters 9, 10, 11

Fifth Generation

- Skip 4th generation: ADA
  - Data abstraction
  - Concurrent programming
- Paradigms
  - Functional: ML, Lisp
  - Logic: Prolog
  - Object Oriented: C++, Java

Chapter 9: List Processing: LISP

- History of LISP
  - McCarthy at MIT was looking to adapt high-level languages (Fortran) to AI - 1956
  - AI needs to represent relationships among data entities
    - Linked lists and other linked structures are common
  - Solution: Develop list processing library for Fortran
  - Other advances were also made
    - IF function: X = IF(N .EQ. 0, ICAR(Y), ICDR(Y))

What do we need?

- Recursive list processing functions
- Conditional expression

- First implementation
  - IBM 704
  - Demo in 1960
- Common Lisp standardized

Example LISP Program

```
(defun make-table (text table)
   (if (null text)
       table
       (make-table (cdr text)
                   (update-entry table (car text))
       )
   )
)
```

- S-expression is used (for Symbolic language)
  - Other languages use M-expression (for Meta)

Central Idea: Function Application

- There are 2 types of languages
  - Imperative
    - Like Fortran, Algol, Pascal, C, etc.
    - Routing execution from one assignment statement to another
  - Applicative
    - LISP
    - Applying a function to arguments
      - $(f\ a_1\ a_2\ \ldots\ a_n)$
    - No need for control structures

Prefix Notation

- Prefix notation is used in LISP
  - Sometimes called Polish notation (Jan Lukasiewicz)
    - Operator comes before arguments
    - (plus 1 2) same as 1 + 2 in infix
    - (plus 5 4 7 6 8 9)
- Functions cannot be mixed because of the list structure
    - (As in Algol: 1 + 2 – 3)
    - LISP is fully parenthesized
    - No need for precedence rules

`cond` Function

```
(cond
  ((null x) 0)
  ((eq x y) (f x))
  (t (g y)) )
```

- Equivalent to

if null(x) **then** 0
elsif x = y **then** f(x)
else g(y)

Function Definition

```
(defun make-table (text table)
   (if (null text)
        table
        (make-table (cdr text)
                         (update-entry table (car text))
        )
   )
)
```

- Function definition is achieved by calling a function(!) called defun, with arguments
  - Name (*make-table*)
  - Parameters (*text table*)
  - Body (*if* …)

The List is the Data Structure

- Lists contain symbolic data

  (set ‘text ‘(to be or not to be))

  – Lists like (to be or not to be) can be manipulated like numbers in other languages (compared, concatenated, split, passed to functions,…)
- Atoms
  – The list (to be or not to be) has 4 atoms
    - to, be, or, not

  – Functions are provided for manipulation of atoms
- Lists of lists

  ((to be or not to be) (that is the question))

Programs Are Lists

- Programs are also represented as lists
  - (make-table text nil)
    - Can be a list
      - with atoms make-table, text, and nil
    - Can be a function
      - ‘make-table’ with 2 arguments
- How do we tell apart the program from a data list?
  - Quoted lists are not interpreted:
    - (set ‘text ‘(to be or not to be))
  - Unquoted ones are interpreted
    - (set ‘text (to be or not to be))

function: *to*

LISP Is Interpreted

- Most LISP systems provide interactive interpreters
  - One can enter commands into the interpreter, and the system will respond

```
> (plus 2 3)
5
> (eq (plus 2 3) (difference 9 4))
t                          means ‘true’
```

Pure vs Pseudo-Functions

- Pure functions
  - plus, eq, …
  - Only effect is the computation of a value
- Pseudo-functions
  - Has *side-effect*; more like a procedure
  - set
    - (set ‘text ‘(to be or not to be))
    - Side effect:
      - Sets the value of *text* to (to be or not to be)
    - Return value:
      - (to be or not to be)

Data Structures

- Primitives
  - Numbers
    - Operations: plus, minus, times, eq, etc.
  - Non-numeric atoms
    - Strings of characters used as symbols
      - Much like enumerated types in Pascal
      - Not used as strings
    - Operations: eq
    - Special atoms
      - t: true
      - nil: false; non-existent atom; empty list

Data Constructor

- The data constructor is the list
- Lists can have 0, 1 or more elements
  - Empty list: ‘() or nil
- All lists are non-atomic (except empty list)

  > (atom ‘()) or (atom nil) or (atom 5)

  t

  > (atom ‘(to be)) or (atom ‘(()))

  nil

Car and Cdr

- Accessing parts of a list
  - Car
    - Accesses first element of the list

    >(car ‘(to be or not to be))

    to

    >(car ‘((to be) or (not to be)))

    (to be)
    - Returns an element
  - cdr
    - Accesses rest of the list (list without first element)

    >(cdr ‘(to be or not to be))

    (be or not to be)
    - Returns a list

Combining *car* and *cdr*

- How do we select the second element?

  >(car (cdr ‘(to be or not to be)))
  be
- Third?

  >(car (cdr (cdr ‘(to be or not to be))))
  or
- How about this?

  (set ‘DS ‘((Don Smith) 45 30000 (Aug 4 80)))
  - Select day of hire

    >(car (cdr (car (cdr (cdr (cdr DS))))))
    4
- This can be simplified:

  >(cadadddr DS)
  4

Defining Functions

- Define functions to replace cadadddr

  (defun hire-date (r) (cadddr r))

  (defun day (d) (cadr d))

  – Now we can select the day of the hire date as

  (year (hire-date DS))

- This is more readable and more maintainable

Constructing Lists

- Need inverse of car and cdr
  - car: get first of list
  - cdr: get rest of list
- Inverse:
  - cons: append first of list to rest of list

    >(cons ‘to ‘(be or not to be))
    (to be or not to be)
    >(cons ‘(to be) ‘(or not to be))
    ((to be) or not to be)

  - Returns a list

Appending Lists

>(cons ‘(to be) ‘(or not to be))

((to be) or not to be)

- But we’d like (to be or not to be)

>(append ‘(to be) ‘(or not to be))

(to be or not to be)

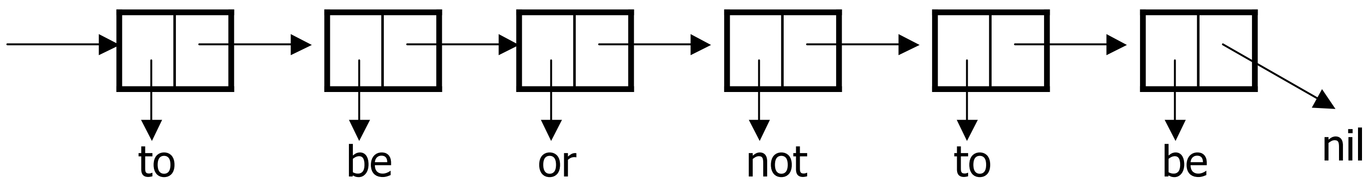
- How would we implement *append* ?
  - We need to extract and cons the last element of the first list successively

```
(defun append (L M)
  (if (null L)
    M
    (cons (car L) (append (cdr L) M)) ))
```

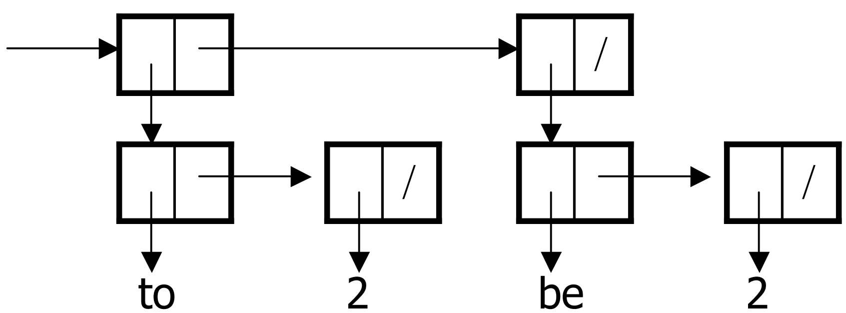
List Representation

- Lists are represented as linked lists

(to be or not to be)

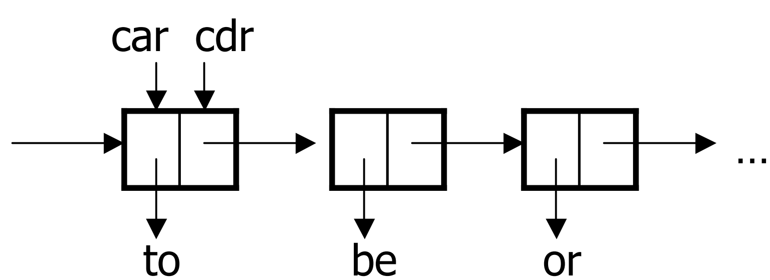


((to 2) (be 2))



Origins of car and cdr

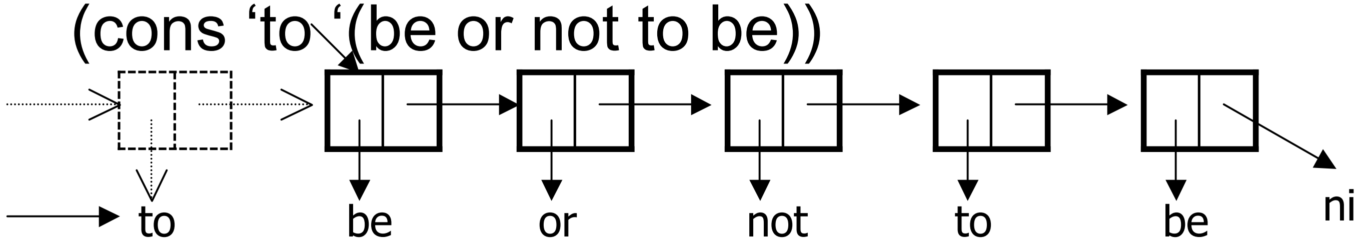
- First LISP was designed for the IBM 704
  - 1 word had 2 fields
    - Address field
    - Decrement field
  - car: “Content of Address part of Register”
  - cdr: “Content of Decrement part of Register”



Implementation of cons

- car and cdr simply return the respective parts of the register
- cons has the job of constructing a new register using two pointers
  - Allocate new memory location
  - Fill in left and right parts of new location

(cons ‘to ‘(be or not to be))



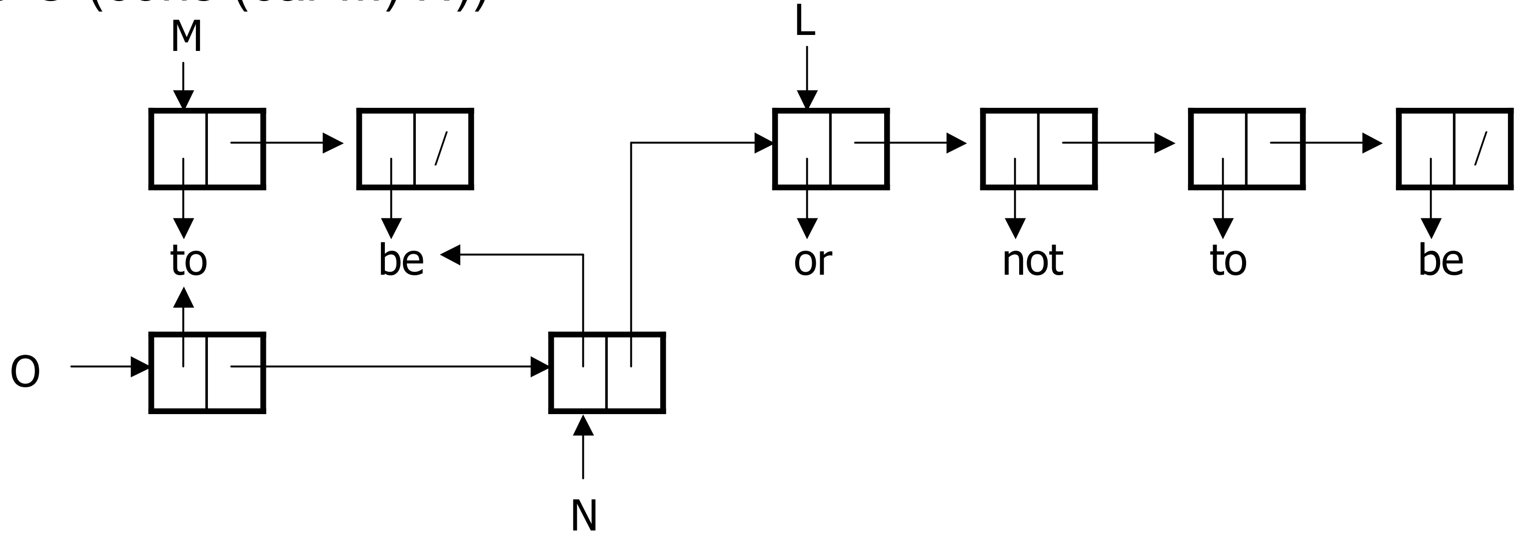
Sublists Can Be Shared

(set ‘L ‘(or not to be))

(set ‘M ‘(to be))

(set ‘N (cons (cadr M) L))

(set ‘O (cons (car M) N))



List Structures Can Be Modified

- Functions discussed so far do not modify lists
- Modifying lists is possible via
  - replaca (replace address part)
  - replacd (replace decrement part)
- It is possible that more than one symbol points to a list
  - which can be modified using replaca and replacd
  - This can cause unexpected problems (like equivalence in Fortran)

Iteration by Recursion

- Iteration is done by recursion
- Iteration is mostly needed to perform an operation on every element of a list
  - This can be done using combination of
    - testing for end of list,
    - operating on first element, and
    - recursing on rest of the list

```
(defun plus-red (a)
   (if (null a)  nil
      (plus (car a) (plus-red (cdr a))) ))
```

  - Notice: No array bounds are needed! Function is very general

Iteration = Recursion

- Theoretically, recursion and iteration have the same power, and are equivalent
- One can be translated to the other (although may not be practical)
  - Recursion → iteration
    - Use iteration and keep track of auxiliary information in an explicit stack
  - Iteration → recursion
    - Need to pass control information (variables)