CS 4100
LISP

From *Principles of Programming Languages: Design, Evaluation, and Implementation (Third Edition),* by Bruce J. MacLennan, Chapters 9, 10, 11, and based on slides by Istvan Jonyer

1

Fifth Generation

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

2

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))
    - List processing and conditional statement combined

3

What do we need?

- Recursive list processing functions
- Conditional expression
- First implementation
  - IBM 704
  - Demo in 1960
- Common Lisp standardized

4

Example LISP Program

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

- Called S-expressions (Symbolic)

5

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

6

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

7

`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)
```

8

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* …)

9

Everything Is a List

- Why is everything a list in LISP?
  - *Simplicity Principle*
    - A language should be as simple as possible. There should be a minimum number of concepts, with simple rules for their combination.
    - If there is only one basic mechanism in the language, the language is easier to learn, understand, and implement.

10

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))

11

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*)

12

Implications?

- If programs are lists
  - and data is also list
  - then we can generate a list that can be interpreted as a program
- In other words
  - We can write a program to write and execute another program
  - Useful in artificial intelligence
- Reductive aspects?

13

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')
```

14

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)

15

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

16

Data Constructor

- The data constructor is the list
- Lists can have 0, 1 or more elements
  - Observes the Zero-One-Infinity principle
  - 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
```

17

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

18

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 DS))))))
4
```
- This can be simplified:
```
>(cadadddr DS)
4
```

19

Defining Functions

```
(set 'DS '( (Don Smith) 45 30000 (Aug 4 80)))
```

- 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
```
(day (hire-date DS))
```
- This is more readable and more maintainable

20

Property Lists

- List like this are hard to maintain and read:
  ((Don Smith) 45 30000 (Aug 4 80))
  - We don't know what elements mean
  - Hard to change the structure of the list
- A better way is to use property lists:
  (name (Don Smith) age 45 salary 30000 hire-date (Aug 4 80))
  - This way we can search for property name we want (age) and return value (45)
  - Order of properties becomes immaterial
  - General form (p_1 v_1 p_2 v_2 … p_n v_n)

21

Accessing Property Lists

(name (Don Smith) age 45 salary 30000 hire-date (Aug 4 80))

- How do we find the property?
  - If property we want is the first one, return second element of list
  - else skip first 2 elements, and start over
- In LISP (get property *p* of list *l*)
```
(defun getprop (p l)
    (if     (eq (car l) p)
            (cadr l)
            (getprop p (cddr l)) ))
```

22

Association Lists

- What if the property does not have a value? (e.g. "retired")
- What is the property has more than a single value?
  - Of course, these can be solved using the property list, if we understand the properties of each property…
  - A better, more foolproof way is to use association-lists:

```
( (name (Don Smith))
  (age 45)
  (salary 30000)
  (hire-date (Aug 4 80)) )
```

23

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

24

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)) ))
```

25

```
[3]> (defun mappend (L M) (if (null L) M (cons
  (car L) (mappend (cdr L) M))))
MAPPEND

[4]> (trace mappend)
;; Tracing function MAPPEND.
(MAPPEND)

[5]> (mappend '(to be) '(or not to be))
1. Trace: (MAPPEND '(TO BE) '(OR NOT TO BE))
2. Trace: (MAPPEND '(BE) '(OR NOT TO BE))
3. Trace: (MAPPEND 'NIL '(OR NOT TO BE))
3. Trace: MAPPEND ==> (OR NOT TO BE)
2. Trace: MAPPEND ==> (BE OR NOT TO BE)
1. Trace: MAPPEND ==> (TO BE OR NOT TO BE)
(TO BE OR NOT TO BE)
```

26

Atoms

- LISP was written for AI
  - to represent complex relationships among objects
  - Objects can have many properties in real life; Atoms allow for modeling this
- Each atom comes with its own property list, and some built-in properties
  - pname (print name); mandatory
  - apval (applied value); to store data
    - If atom is bound to a value
  - expr (expression); to store program
    - If atom is bound to a program

27

Adding Properties to Atoms

- Other, arbitrary properties may also be added to an atom using *putprop* *(not in our clisp: setf)*

```
(putprop atom propValue propName)
(putprop 'France 'Paris 'capital)
```

  - Paris, in this case, is also an atom
- Find out the value of a property using get

```
>(get 'France 'capital)
Paris
>(get 'France 'pname)
"France"
```

28

Special Property: apval

- Assigning a value to an atom

```
(set 'Europe '(England France …))
```

  - is the same as

```
(putprop 'Europe '(England France …)
  'apval)
```

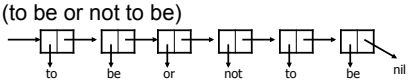
  - 'Applied value' points to the list the atom is bound to

29

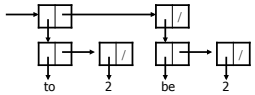
List Representation

- Lists are represented as linked lists

(to be or not to be)



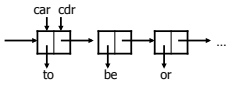
((to 2) (be 2))



30

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"

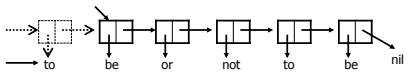


31

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

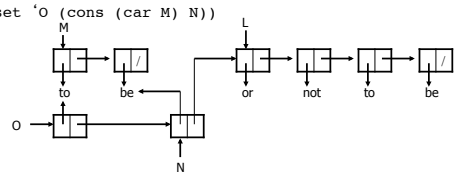
(cons 'to '(be or not to be))



32

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))
```



33

```
[10]> (set 'L '(or not to be))
(OR NOT TO BE)
[11]> (set 'M '(to be))
(TO BE)
[12]> (set 'N (cons (cadr M) L))
(BE OR NOT TO BE)
[13]> (set 'O (cons (car M) N))
(TO BE OR NOT TO BE)
```

34

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)

35

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

36

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)

37

Storage Reclamation

- What happens to *cons*'d pointers that are no longer in use?
- Explicit reclamation is the obvious / traditional way
  - C: malloc, calloc, realloc, free
  - C++: new, delete
  - Pascal: new, dispose
- Issues
  - Complicates programming
    - Requires the programmer to keep track of pointers
  - Violates security of the environment
    - Memory freed, but still referenced (dangling pointers)

38

Automatic Storage Reclamation

- It would be nice for the system to automatically 'reclaim' storage no longer used
- System can keep track of number of references to storage
  - When references decrease to 0, storage is returned to 'free-list'
- Advantage:
  - Storage reclaimed immediately as last reference is destroyed
- Disadvantage:
  - Cyclic structures (points to itself) cannot be reclaimed

39

Garbage Collection

- A different approach is garbage collection
  - Do not keep track of references to location
  - When last reference is destroyed, we still do not do anything, and leave the memory as garbage (unused, non-reusable storage, littering the memory)
  - Collect garbage if system runs out of storage
    - Mark all areas unused
    - Then examine all visible pointers and mark storage they point to as 'used'
    - Leftover is garbage, and can be put on free-list
  - This is called the *mark-and-sweep* method

40

Garbage Collection

- Advantages
  - Fast until runs out of memory
  - No additional memory is needed for tracking references
- Disadvantages
  - Garbage collection itself can be slow
    - If memory is large, and have many references
    - Must halt entire system, since all dynamic memory must be marked as unused first
- Java uses this approach

41