LISP 101

Why do you have to learn it?

- Breadth of CS education
- I use LISP to demonstrate AI programs
- Lingua Franca of AI
- So you know what's missing in other languages
  - But other languages now have garbage collection, first-class objects, funargs, etc.
  - WELL, YOU CAN DO YOUR HOMEWORK IN PYTHON IF YOU REALLY HAVE TO....

History of Lisp

- John Mccarthy implemented list processing language based on Church's lambda calculus
  - lambda calculus is universal (equiv to Turing machine in power)
- Lisp 1.5 (invention of the upgrade)
- Maclisp (MIT lab version)
- Interlisp (Xerox/BBN)
- Franz Liszt (Berkeley version)
- Lisp Machine Incorporated (Special graphic workstations; Xerox Interlisp-D)

The difference between Scheme and Lisp

- Scheme is a Pedagogical Language with minimal elements
  - used for MIT's old intro to computer science, and our CS 121b
- Lisp is a production language with every abstraction known to mankind built-in

Lisp Top View

- Interpreter rather than compiler
  - BUT INCLUDES A COMPILER!
- Interactive
- Bottom-up design
- First class symbols and functions
- Flexible List structures
- Lots of Parentheses

The cool part

- In most languages, your SOURCE code is an editable text file, and after compiling away all the symbolic human-readable parts, the object is a binary file.
- In lisp, the programs are lists made of symbols, so you can write programs which manipulate other programs.

Basic Ideas

- Memory is not a vector, but a bag of "Cons Cells" or "pairs"
- Symbols are unique and internalized in a "symbol table" pointing to functions and variable data structures.
- When symbol values are changed, a "garbage collector" finds disconnected cells and makes them ready for reuse.

Symbols are not "Strings"

- "john" and "john" might be character-by-character equal, but do not reside in same memory location.
- JOHN in lisp is hashed and has a unique place in a "symbol table" indicating its value and/or associated function.
- We will focus mainly on symbols and numbers in lists, but there is more!

Symbols are used as variables

- (setf JOHN 27)
- JOHN -> 27
- (setf JOHN 'ARCHITECT)
- JOHN -> ARCHITECT
- (SETF LST '(1 2 3 4 5)
- LST -> (1 2 3 4 5)

Built-in Type Specifiers

- array, atom, bignum, bit, bit-vector, character, common, compiled-function, complex, cons, double-float, fixnum, float, function, ***hash-table,*** integer, keyword, list, long-float, nil, null, number, package, pathname, random-state, ratio, rational, readtable, sequence, short-float, simple-array, simple-bit-vector, simple-string, simple-vector, single-float, standard-char, stream, string, string-char, symbol, t, vector
  - ***Plus, you can define your own datatypes easily with defstruc!***

Read-Eval-Print Loop (REPL)

- loop
  - read in an expression from the console;
  - evaluate the expression;
  - print the result of evaluation to the console;
- end loop.

Hierarchical Evaluation

- i. Strings/Numbers: Themselves
- ii. Symbols: Value of variable or function
  - T and NIL shouldn't be redefined!
- iii. Lists: ***First entry is a function name***, to be applied to evaluated arguments
  - (function arg arg)
  - (+ 3 (* 5 4))
  - (* (+ a b)(+ c d))

RECURSIVE Evaluation

- in list evaluation, first element function is applied to EVALUATED arguments:
- (* (+ a b)(+ c d))
- * is applied to its arguments
  - (+ a b) and (+ c d)
  - to evaluate (+ a b), + is applied to value of a and value of b
  - to evaluate (+ c d), + is applied to value of c and value of d
- then * is applied

Numbers eval to themselves

- Lisp number are just symbols which evaluate to themselves
- Lisp introduced "bignums" for doing number theory etc.
- 298745379862873098743982984786432
- Other radixes
  - #2r1001 == 9
  - #16r1a = 26
- Rationals (both num and denom kept)
- Complex Numbers

Math Functions

- predicates: zerop plusp minusp oddp evenp
- Comparisons: = /= < > <= >=
- arithmetic: + - * / mod rem floor
- Bitwise: Logand logior logxor etc
- Trig: sin cos tan etc.

List Calculation is PREFIX

- PREFIX notation for calculation

- (1 * 3 + 2 * 4) NOT!
- (+ (* 1 3)(* 2 4)) YUP!

- The evaluator reads the expression sees that + is not a special form
  - evals the arguments (recursively if necessary)
  - First is (* 1 3)
    - Now eval the arguments for *
  - second is (* 2 4)
- Finally (+ 3 8) -> 11

Regular vs. Special Forms

- for 95% of functions, evaluator first evaluates arguments then passes values to functions
- 5% of functions are "special forms" where the arguments are not evaluated first.

The first Special Form: QUOTE

- how can we type in a symbol?
  - john -> 5 we get its value.
- (QUOTE john) -> john the symbol

- quote has a special "character macro"
  - '(1 2 3) expands to (QUOTE (1 2 3))
  - which returns the list (1 2 3)
- What happens if you type (1 2 3)?

Special Forms

- (quote john) also 'john
- (defvar john 25)
- (setf john 9)
- (if T (print 'hello)(FIB 1000))
  - you don't want to eval both branches.
- If and other conditionals have to be special forms because only one of its branches gets evaluated.
- DEFUN is a special form

DEFVAR, DEFPARAMETER, DEFCONST

- (DEFVAR *NAME* (VALUE EXPRESSION))
- DEFVAR DECLARES A "SPECIAL" (GLOBAL) VARIABLE
  - USUALLY *NAME* TO SET IT APART
  - DOES NOT EVAL ASSIGN IF *NAME* ALREADY HAS VALUE
  - This means if you change value and reload file it doesn't reset
- DEFPARAMETER always evaluates the assignment
- DEFCONSTANT tells compiler the variable won't change

IF is a Special Form

- T and NIL are used for True and False
- IF is a special form so both branches aren't evaluated:
- (if T (print 'hello)(FIB 1000))
  - you don't want to eval both branches.
- If and other conditionals have to be special forms because only one of its branches gets evaluated.

Defining a new function

- (DEFUN name (args) body)
  - arguments include &optional &key &aux &rest
- DEFUN is a special form which binds the f-value of a symbol
- (DEFUN square (x)(* x x))
- Lisp is EXTENSIBLE, now square is no different from built-in functions

CONS

- Definition of basic "compound" data structure:
- (setf something (cons a b))
- (car something) -> value of a
- (cdr something) -> value of b
  - CAR and CDR are historic names
    - contents of address register
    - contents of data register

(Anything . Anything)

- A cell is a tree composed of 2 things.
- A list is either NIL or (thing . NIL)
  - a right-branching tree ending in NIL prints flat: (A . (B . NIL)) prints (A B)
  - (1 2 3) is really (1 . (2 . (3 . NIL))), OK?
- Lists of lists:
- ((john . male)(mary . female)) is
- ((john . male) . ((mary . female) . NIL))
- More below...

How is CONS implemented?

- Need to know?
- Basic unit is called a "CONS CELL"
- CELL contains Two pointers to addresses of any other datatype
- Addresses point to other CELLS or to distinctive places in memory holding numbers or symbols.

S-EXPRESSIONS

- In Lisp, everything is an S-expression
- An S-expression is an atom or a list
- You can think of these as using two different kinds of storage locations--one kind for atoms, another kind for the parts of a list
  - This is an oversimplification, but it will do for now

ATOMS

- An atom is a simple thing, and we draw it in a simple way:

- Sometimes we don't bother with the boxes:

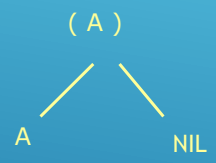
HELLO ABC NIL

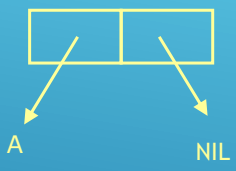
LISTS

- A list has two parts: a CAR and a CDR
- We draw this as a box , called a cons cell, with two compartments, called the car field and the cdr field

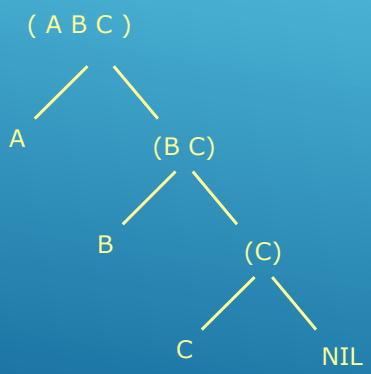
- In each of these compartments we put an arrow pointing to its respective value:

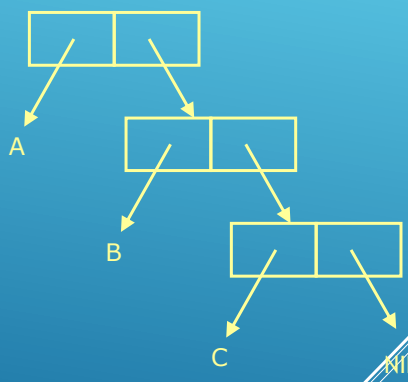
EXAMPLE I





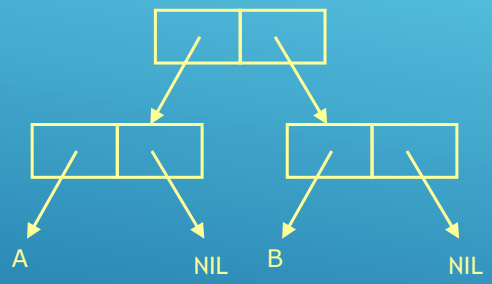
EXAMPLE II





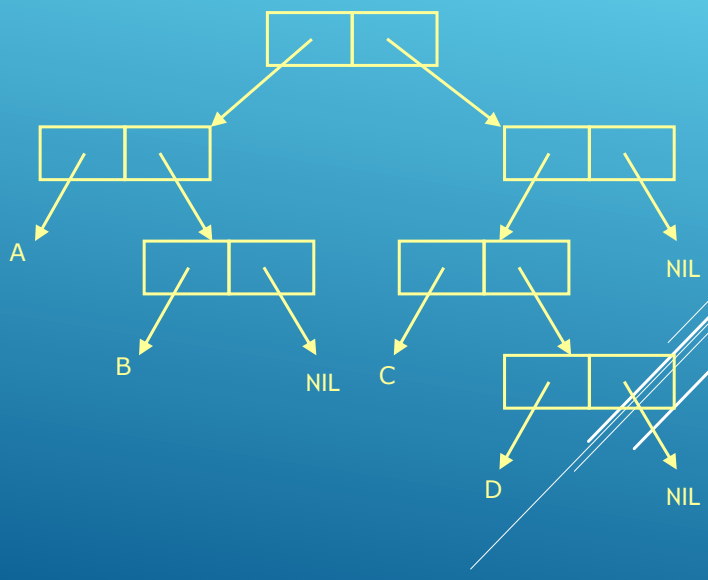
EXAMPLE III

((A) B)



EXAMPLE IV

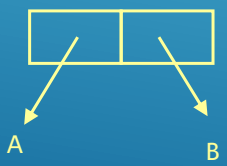
((A B) (C D))



DOTTED PAIRS

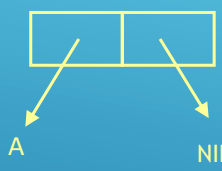
- In a simple list, every right-pointing arrow points to a cons cell or to NIL
- If a right-pointing arrow points to an atom, we have a dotted pair

(A . B)



LISP LISTS ARE IMPLEMENTED WITH DOTTED PAIRS

(A) =

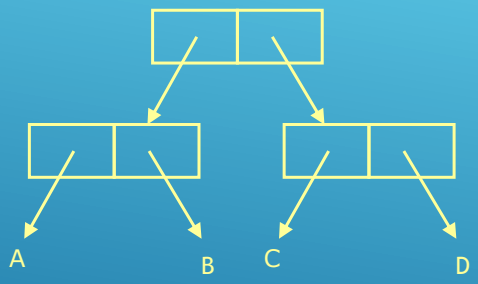


= (A . NIL)

- Therefore, (A) = (A . NIL)
- All structures in Lisp can be created from atoms and dotted pairs

EXAMPLE V

((A . B) . (C . D))



Basic for building lists

- (cons 1 2) -> (1 . 2)
- (cons 1 nil) -> (1)
- (cons 1 (cons 2 nil)) -> (1 2)
- (cons (cons 1 2) (cons 3 nil)) ->
- ((1 . 2) 3)
- (list 1 2 3) ->
- (cons 1 (cons 2 (cons 3 nil))) ->
- (1 2 3)

Cons is powerful gizmo!

- Can construct arbitrarily complex data forms
- CAR and CDR can be used to access them
  - CAAR CADAR CDDR down 4 levels
    - This is not great form anymore
  - Use First, Second, Third, fourth for
  - Car CADR CADDR CADDDR
    - Beware of "Last"

Lists

- A list is a right-branching tree terminating in NIL.
- the LIST function is a super-CONS:
- (list 1) -> (cons 1 nil) -> (1)
- (list 1 2 3 4) --> (1 2 3 4)
- (list 'john (list 'loves 'mary)) ->
- (john (loves Mary))
- (list 1 2 3 'a 'b 'c "foobar")

What can be done with lists?

- (length list)
- (reverse list)
- (append list1 list2)
- (NTH i list)
- (SUBST new old tree)
- (member item list)
- (sort list #'<)

FUNCTIONS ON LISTS

- CAR, CDR, FIRST, REST, SECOND, THIRD, NTH, NTHCDR, LAST, BUTLAST
- UNION, INTERSECTION, SET-DIFFERENCE, SUBSETP, MEMBER
- LENGTH, REVERSE

Flow Control in Lisp

- (IF (Predicate) (then expression))
- (IF (predicate) (then expr)(else expr))
- (AND (exp1)(exp2)(exp3)...) Stops at first NIL
- (OR (exp1)(exp2)(exp3)...) Returns first NON-NIL
- (Null exp) Test for NIL

More control

- (PROGN (exp1)(exp2)(exp3)...) a sequence construct returns val of exp3

- (COND ((pred1)(exp1) (exp2)...)
- ((pred2)(exp2)...))
- (T NIL))

- PROG1 and PROG2 return the first and second result from a sequence