An Introduction to LISP

- Introduction
- LISP: A Brief Overview
  - Expressions, the Syntactic Basis of LISP
  - Control of LISP Evaluation: **quote** and **eval**
  - Programming in LISP: Creating New Functions
  - Program Control in LISP: Conditionals and Predicates
  - Functions, Lists, and Symbolic Computing
  - Lists as Recursive Structures
  - Nested Lists, Structure, and **car/cdr** Recursion
  - Binding Variables Using **set**
  - Defining Local Variables Using **let**
  - Data Types in Common LISP

Symbolic

- **Why do we care about symbols?**
  - **U**nderstand human cognition with a language like symbolic processing

Physical Symbol System Hypothesis

"A physical symbol system has the necessary and sufficient means for general intelligent action." (Newell & Simon 1976)

- Physical symbol system
  - Set of entities called symbols - physical patterns that can occur as components
  - Expressions (or symbol structures) built of symbols

Characteristics of LISP

1. **Language for artificial intelligence programming**
   a. Originally designed for symbolic computing
2. **Imperative language**
   a. Describe *how* to perform an algorithm
   b. Contrasts with *declarative* languages such as PROLOG
3. **Functional programming**
   a. Syntax and semantics are derived from the mathematical theory of recursive functions.
   b. Combined with a rich set of high-level tools for building symbolic data structures such as predicates, frames, networks, and objects
4. **Popularity in the AI community**
   a. Widely used as a language for implementing AI tools and models
   b. High-level functionality and rich development environment make it an ideal language for building and testing prototype systems.

LISP: A Brief Overview

- Syntactic elements of LISP
  - Symbolic expressions : S-expressions
    - Atom : basic syntactic units
    - List
  - Both programs and data are represented as s-expressions
- Atom
  - Letters (upper, lower cases)
  - Numbers
  - Characters : * + - / @ $ % ^ & _ < > ~ .
  - Example
    - `3.1416`
    - `Hyphenated-name`
    - `*some-global*`
    - `nil`
    - `X`

SPECIAL FORMS:

1. Forms not evaluated according to the evaluation rule.

2. Special forms:

   defun, defparameter, setf, let, case, if, function, quote.

OTHER DATA TYPES

1. Strings: `(length "abc")` --> `3`
2. Many number types.

LISP EVALUATION RULE

1. Every expression is either a list or an atom.

2. Every list to be evaluated is either a special form or a function application.

3. A special form expression is a list whose first element is a special form operator and is evaluated according to the special rule of that operator.

4. A function application is evaluated by first evaluating the arguments (the rest of the list) and then finding the function named by the first element of the list and applying it to the list of evaluated arguments.

5. Every atom is either a symbol or a non-symbol.

6. A symbol evaluates to the most recent value assigned to the variable.

7. A non-symbol atom evaluates to itself.

Read-Eval-Print

- Interactive Environment
  - User enters s-expressions
  - LISP interpreter prints a prompt
- If you enter
  - Atom: LISP evaluates itself (error if nothing is bound to the atom)
  - List: LISP evaluates as an evaluation of function, i.e. that the first item in the list needs to be a function definition (error if no function definition is bound for the first atom), and remaining elements to be its arguments.

    -
      ```
      > (* 7 9)
      63
      ```

Control of LISP Evaluation: quote & eval

- **quote : ‘**
  - prevent the evaluation of arguments
    - `(quote a) ==> a`
    - `(quote (+ 1 3)) ==> (+ 1 3)`
    - `` `a ==> a ``
    - `` `(+ 4 6) ==> (+ 4 6) ``
- **eval**
  - allows programmers to evaluate s-expressions at will
    - `(eval (quote (+ 2 3))) ==> 5`
    - `` (list `* 2 5) ==> (* 2 5) ``
    - `` (eval (list `* 2 5)) ==> 10 ``

WHAT MAKES LISP DIFFERENT

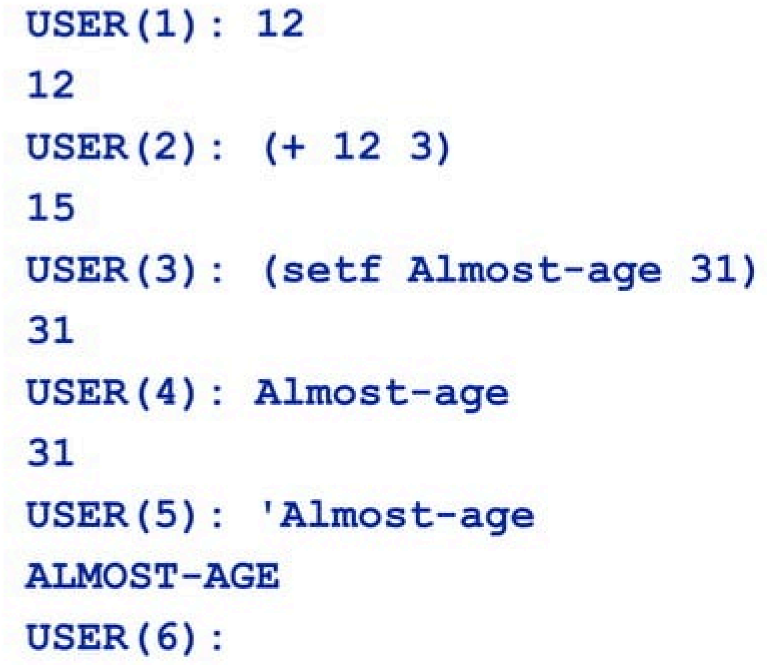
1. Built-in support for Lists.
2. Automatic memory management.
3. Dynamic typing.
4. First-class functions.
5. Uniform syntax.
6. Extensibility

Why LISP?

- Especially designed for symbol manipulation.
- Provides built-in support for lists ("everything is a list..")
- Automatic storage management (no need to keep track of memory allocation).
- Interactive environment, which allows programs to be developed step by step. That is, if a change is to be introduced, only changed functions need to be recompiled.

Interpreted & interactive

- Interpreted
- Interactive



```
USER(1): 12
12
USER(2): (+ 12 3)
15
USER(3): (setf Almost-age 31)
31
USER(4): Almost-age
31
USER(5): 'Almost-age
ALMOST-AGE
USER(6):
```

LISTS

1. Primitive aggregating type.
2. Primitive functions: first, second, ..., length, last.   append, cons, list.

List and S-expression

- List
  - A sequence of either atoms or other lists separated by blanks and enclosed in parentheses.
    - Example
      - `(1 2 3 4)`
      - `(a (b c) (d (e f)))`
  - Empty list “( )” : nil
    - nil is the only s-expression that is considered to be both an atom and a list.
- S-expression
  - An atom is an s-expression.
  - If s_1, s_2,…, s_n are s-expressions, then so is the list (s_1 s_2 … s_n).

Everything's a List!

- Da` (a b c)`

- Fu
```
(defun plus (x y)
    (+ x y))
```

- Simple syntax:

  (function-name arg1 arg2 …)

List as recursive structures

- Cons cell: data structure to hold a list in LISP
  - car - holds the first element.
  - cdr - holds the rest in the list.
- Accessing a list
  - `(car '(a b c)) ==> a`
  - `(cdr '(a b c)) ==> (b c)`
  - `(first '(a b c)) ==> a`
  - `(second '(a b c)) ==> b`
  - `(nth 1 '(a b c)) ==> b`
- Constructing a list
  - `(cons 'a '(b c)) ==> (a b c)`
  - `(cons 'a nil) ==> (a)`
  - `(cons '(a b) '(c d)) ==> ((a b) c d)`

Nested lists, structure, car/cdr recursion

- More list construction
  - `(append '(a b) '(c d)) ==> (a b c d)`
  - `(cons '(a b) '(c d))  ==> ((a b) c d)`

- Counting the number of elements in the list
  - `(length '((1 2) 3 (1 (4 (5))))) ==> 3`

Dynamic

- Functions are first-class objects
  - Pass functions as arguments to other functions

```
USER(1): (+ 1 2 3)
6
      ↓     ↑
USER(2): (apply #'+ '(1 2 3))
6
```

One Function, and a Side of Fries, To Go

- Create functions on the fly

```
USER(1): (funcall #'(lambda (x y) (+ x y))
            17 14)
31
```

- Lisp contains itself: `eval`

Name Calling

- Lisp remembers function names separately from variable names

```
USER(22): (defun add (x y) (+ x y))
ADD
USER(23): (setf add 9)
9
USER(24): add
9

USER(25): #'add
#<Interpreted Function ADD>
```

Basic terminology

- **Atoms**: word-like indivisible objects which can be <u>numbers</u> or <u>symbols</u>.
- **Lists:** sentence-like objects formed from atoms or other lists, and enclosed in parentheses.
- **S-expressions:** compositions of atoms and lists.
- **Procedures:** step by step specifications how to do something.
  - <u>Primitives</u>: procedures supplied by the LISP itself

    Example: (+ 5 6)
  - <u>User-defined procedures</u>: procedures introduced by the programmer.

    Example: (students 'anna)
- **Program:** a collection of procedures working together.

S-expressions

- An s-expression can have other s-expressions nested in it. Examples:

```
(+    (* 5   7) ( /  2  4))
(This (is a dog) (or a cat))
```

- Expressions can be interpreted both, procedurally and declaratively.
  - If interpreted procedurally, an expression provides a direction for doing something. Such an expression is called a *form*, and its first element is the name of a procedure to be used to produce the value.
  - The process of computing the value of an expression is called *evaluation*.
  - If interpreted declaratively, expressions represent data.

Data and procedures have the same syntax.

Evaluation of atoms

- The value of a number is the number itself.

  Example: 5 ==> 5
- The value of a string is the string itself.

  Example: “Nice day” ==> “Nice day”
- The value of the symbol T is T (true).
- The value of the symbol NIL is NIL (false).
- The symbol NIL and the empty list ( ) are the same thing.
- Variables are names of memory locations. The contents stored in a given memory cell is the value of the variable serving as a name of this location.

  Example: Let x be a variable, and 5 be the contents of the memory cell called x.
  Then, the value of x is 5.

Numbers

- Integers: 179, 45
- Ratio: 5/7, 7/9
- Floating point: 5.2, 7.9
- Examples:

```
* (/  25  5)
5
* (/  46  9)
46/9                    ; do not divide evenly
* (float (/  46  9))
5.111111
* (round (/  46  9))
5                       ; the nearest integer
1/9                     ; the remainder
```

More numeric primitives

```
* (-  6)
-6
* (-  -6)
6
* (max  5  7  2)
7
* (min 5  7   2)
2
* (sqrt (*  (+  1  3) (*  2  2)))
4.0
* (+  (round (/  22  7)) (round (/  7  3)))
5
```

```
*  (+  2  2.5)
4.5
* (expt  3  6)
729
* (sqrt  81)
9.0
*  (sqrt 82)
9.055386
*  (abs 6)
6
*  (abs  -6)
6
```

Conditionals & Predicates

Conditionals & Predicates(1/2)

- Conditions

  -
    ```
    (cond (<condition 1> <action1>)
          (<condition 2> <action 2>)
                   ...
          (<condition n> <action n))
    ```
  - Evaluate the conditions in order until one of the condition returns a non-nil value
  - Evaluate the associated action and returns the result of the action as the value of the **cond** expression
- Predicates
  - Example
    - `(oddp 3)       ; whether its argument is odd or not`
    - `(minusp 6)`
    - `(numberp 17)`

Conditionals & Predicates(2/2)

- Alternative Conditions
  - `(if test action-then action-else)`
    - (defun absolute-value (x)
      (if (< x 0) (- x) x))
    - it returns the result of action-then if test return a non-nil value
    - it return the result of action-else if test returns `nil`
  - `(and action1 ... action-n)` `; conjunction`
    - Evaluate arguments, stopping when any one of arguments evaluates to nil
  - `(or action1 ... action-n)` `; disjunction`
    - Evaluate its arguments only until a non-nil value is encountered

DEFINING NEW FUNCTIONS:

1.
```
(defun <name> (<parameters>)
  "doc"
  <body>)
```

```
(defun last-name (name)
     "Select last name from a name represented
          as a list."
     (first (last name)))

(last-name '(john r vandamme)) --> vandamme
(last-name '(john quirk MD))   --> MD
```

2. Importance of abstraction.

```
 (defun first-name (name) (first name))
```

Creating New Functions

- Syntax of Function Definition

  -
    ```
    (defun <function-name> (<formal parameters>)
           <function body>)
    ```
    - defun : define function
- Example

  -
    ```
    (defun square (x)
           (* x x))
    ```

  -
    ```
    (defun hypotenuse (x y)     ;the length of the hypotenuse is
          (sqrt (+ (square x)   ;the square root of the sum of
                   (square y)))) ;the square of the other sides.
    ```

USING FUNCTIONS:

1. (setf names '((john x ford) (peter p rabbit) (fabianna f
   wayne)))

```
 (mapcar #'last-name names) --> (ford rabbit wayne)
```

#' from name of function to function object.

mapcar primitive.

2. (defparameter *titles*
   '(Mr Mrs Miss Madam Ms Sir Dr Admiral Major General))

```
(defun first-name (name)
     "Select first name from a list representing a name."
     (if (member (first name) *titles*)
         (first-name (rest name))
         (first name)))

   (if <test> <then-part> <else-part>)

   (first-name '(Madam Major General Paula Jones)) --> Paula
```

3. Trace functions

HIGHER ORDER FUNCTIONS

1. Functions as first-class objects: can be manipulated, created, modified by running code.
2. Apply: `(apply #'+ '(1 2 3 4))` --> 10
3. Funcall: `(funcall #'+ 1 2)` --> 3

   `(funcall #'+ '(1 2))` -->

   error (1 2) is not a number.
4. Function constructor: lambda.

   `(lambda (parameter ...) body...):`

   non atomic name of a function.

   `((lambda (x) (+ x 2)) 4)` --> 6

   `(funcall #'(lambda (x) (* x 2)) 4)` --> 8

   Can create functions at run time.

Binding variables

Binding variables: set(1/2)

- `(setq <symbol> <form>)`
  - bind <form> to <symbol>
  - <symbol> is NOT evaluated.

- `(set <place> <form>)`
  - replace s-expression at <place> with <form>
  - <place> is evaluated. (it must exists.)

- `(setf <place> <form>)`
  - generalized form of set: when <place> is a symbol, it behaves like setq; otherwise, it behaves like set.

Binding variables: set(2/2)

- Examples of set / setq
  - `(setq x 1) ==> 1  ;;; assigns 1 to x`
  - `(set a 2) ==> ERROR!! ;;; a is NOT defined`
  - `(set ‘a 2) ==> 2  ;;; assigns 2 to a`
  - `(+ a x) ==> 3`
  - `(setq l ‘(x y z)) ==> (x y z)`
  - `(set (car l) g) ==> g`
  - `l ==> (g y z)`
- Examples of setf
  - `(setf x 1) ==> 1`
  - `(setf a 2) ==> 2`
  - `(+ a x) ==> 3`
  - `(setf l ‘(x y z)) ==> (x y z)`
  - `(setf (car l) g) ==> g`
  - `l ==> (g y z)`

Local variables: let(1/2)

- Consider a function to compute roots of a quadratic equation: $ax^2+bx+c=0$

  -
    ```
    (defun quad-roots1 (a b c)
         (setq temp (sqrt (- (* b b) (* 4 a c))))
         (list (/ (+ (- b) temp) (* 2 a))
               (/ (- (- b) temp) (* 2 a))))
    ```
  - `(quad-roots1 1 2 1) ==> (-1.0 -1.0)`
  - `temp ==> 0.0`
- Local variable declaration using `let`

  -
    ```
    (defun quad-roots2 (a b c)
         (let (temp)
              (setq temp (sqrt (- (* b b) (* 4 a c))))
              (list (/ (+ (- b) temp) (* 2 a))
                    (/ (- (- b) temp) (* 2 a)))))
    ```

Local variables: let(2/2)

- Any variables within *let* closure are NOT bound at top level.
- More improvement (binding values in local variables)

```
(defun quad-roots3 (a b c)
(let ((temp (sqrt (- (* b b) (* 4 a c))))
      ((denom (*2 a)))
      (list (/ (+ (- b) temp) denom)
            (/ (- (- b) temp) denom))))
```

Recursion

Looping Constructs -- Recursion

Are there enemies in the list of attendees of King Arthur's party tonight?

(*setf guest-list '(micky snowwhite bishop robocop sirlancelot))*

(*setf enemies '(robocop modred)*

Is there an 'enemies' in 'guest-list'?

How do we find out?

What the function should do

Given a the function and a list of guests, return ‘t’ if there is an enemy in the list.

(find-enemy guest-list enemies)

T

It should return *t* if the program finds an enemy in the list.

Recursive Functions

- Functions that call themselves
- A partial LISP version of Martin's solution

```
(defun find-enemy (guest-list  enemies)
   (cond   ( (member (first guest-list) enemies)    T )     ; Yes, there is an enemy
              ( T  (find-enemy (rest guest-list enemies) ) ) ;  this is the recursion
    ) ; end of cond
)  ; end of defun
```

In English: If the first person on the guest-list is a member of the enemies list, then return true, otherwise call the function again with the rest of the list. But, this isn't very good. What if there are no guests who are enemies – you'll try to keep going until you get an error…..

Fixing the infinite recursion

Need to stop the recursion . In this case we use a null! But there are other ways (test for an atom? Test for something else).

```
(defun find-enemy ( guest-list  enemies)
   (cond  ( (null guest-list)   NIL )                          ; end of the list
           ( (member (first guest-list) enemies)   T ) ; Yes, there is
           ( T  (find-enemy (rest  guest-list )  enemies) )
   ) ; end of cond
)  ; end of defun
```

What does this look like?

- *(find-enemy guest-list enemies)*
  - *guest-list = '(micky snowwhite bishop robocop sirlancelot))*
  - *enemies = (robocop modred) (this is always the same so I won't repeat it)*
- *(find-enemy guest-list enemies)*
  - *guest-list = '(snowwhite bishop robocop sirlancelot)*
- *(find-enemy guest-list enemies)*
  - *guest-list = (bishop robocop sirlancelot)*
- *(find-enemy guest-list enemies)*
  - *guest-list = (robocop sirlancelot) -> this is where it finds robocop and stops*

Next version of program

What are the names of those enemies that are in the attendee list? Right now, it only tells me if there is a single enemy – and then it stops. But there could be more than one, and we would want to know who they are.

So, now I can't just return true – I need to make a separate list of the enemies <u>as I go down the list….</u>

Building lists using recursion

This is the same program, but now we need to do something instead of return 't'.....This code will go where the ? Is

```
(defun identify-enemy ( guest-list  enemies)
   (cond ( (null guest-list)   NIL )    ; the end of the guest list.
           ( (member (first guest-list) enemies)   ?what to do? )
             (t  (identify-enemy (rest guest-list ) enemies) )
   ) ; end of cond
)  ; end of defun
```

We need to add something that will build a list of the enemies.....
How do we build lists?????-> we use *cons*

The answer ---

```
(defun identify-enemy ( guest-list  enemies)
   (cond ( (null guest-list)   NIL )  ; Reached the end of the guest list.
           ( (member (first guest-list) enemies)
                  (cons (first guest-list)
                  (identify-enemy (rest  guest-list) enemies)))   ;
             ( T  (identify-enemy (rest  guest-list ) enemies ) )
   ) ; end of cond
)  ; end of defun
```

What does this look like?

- *(find-enemy guest-list enemies)*
  - *guest-list = '(micky snowwhite bishop robocop sirlancelot))*
  - *enemies = (robocop modred) (this is always the same so I won't repeat it)*
- *(find-enemy guest-list enemies)*
  - *guest-list = '(snowwhite bishop robocop sirlancelot)*
- *(find-enemy guest-list enemies)*
  - *guest-list = (bishop robocop sirlancelot)*
- *(find-enemy guest-list enemies)*
  - *guest-list = (robocop sirlancelot) -> this is where it finds robocop and calls 'cons' --- BUT BEFORE actually do the cons, I make the recursive call with....*

- *(find-enemy guest-list enemies)*
  - *guest-list = (sirlancelot)*
- *(find-enemey guest-list enemies)*
  - *guest-list = () -> The (null guest-list) is executed and nil is returned….but wait, I've got to finish my cons….*
  - *So the () is returned to the (cons (first guest-list) to …..and the (robocop) gets returned from the function….*

Other uses of recursion

It can be used for filtering – example, removing negative numbers from a list

```
( defun filter-negatives (ls)
  (cond ((null ls) '())
          ((<  (first ls) 0) (filter-negatives (rest ls)))
         (t (cons (first ls) (filter-negatives (rest ls))))))
```

(filter-negatives '(-50 0 50 100 150)) ===> (0 50 100 150)

In English: If the *first* of the list is less than 0, call the function with the *rest* of the list (eliminating the negative number), else cons that *first* of the list to the recursive call…

Recursion can be used to count

```
(defun count-atoms (ls)
        (cond ((null ls) 0)
            (t (+ 1 (count-atoms (rest ls))))))
```

This is the same as length.

(count-atoms '(one two three four))

4

English: When the list is empty return 0, else add 1 every time you call the recursive function (it will add as it returns to the calling function.

Recursion Problem

Write a function called *stop* that takes a list and a number and returns a list that does not exceed the number.

For example:

(stop ‘(this is a list for recursion) 4) -> (this is a list)

(stop ‘(this is a list for recursion) 4) -> (this is a list)

- We need to define a function that will take two arguments
- How are we going to stop it?
- We want to return a list….so we will need to create a new list (which means a cons)….

Now, back to our guests…..

Let’s go back to the guests…..What if the guests come in subgroups such as…..

((Micky SnowWhite) Bishop (BlackKnight WhiteKnight) RoboCop ((SirLancelot Modred) Magician))

How do we write a function to find the enemies in that type of list?

Recursion on both head and tail

```
(defun identify-enemy ( guest-list )
  (cond ( (null guest-list)   NIL )  ; Reached the end
         ( (listp (car guest-list))  ; If the first element is a list
           (append (identify-enemy (first guest-list))
                        (identify-enemy (rest guest-list))) )
         ( (member (first guest-list) enemies)
           ; add to the list of enemies identified from  the rest
           ;  of guests
           (cons (first guest-list)

                    (identify-enemy (rest guest-list) ) ))
         ( T  (identify-enemy (rest  guest-list) ) )
  ) ) ; end of  defun
```

Keys to programming recursive functions

- Find out how to solve simple special cases
  - How to handle empty list?
  - Are all possible ways to terminate the recursion considered?
- Break the task down into smaller subtasks (e.g., using recursion with a combination of first/rest)
- Know how to synthesize partial solutions generated by subtasks into an overall solution. (e.g., using CONS, APPEND, LIST, +, etc.)
- Look at the arguments separately, and determine what needs to be passed into the function – which arguments need to change and how should they be changed (e.g., decremented, added, rest, etc.
- If all else fails, make 'two' functions --- one calls the other to do the work.

How to declare local variables (we are going to need this for iterative functions)?

```
(LET ( ( <var1> <val1> )
              ...
        ( <vark> <valk> ) )
        <exp1>
          ...
       <expN> )
```

- LET is your way to set up temporary variables. You can initialize each local variable to its value concurrently, then evaluates expressions sequentially. It returns the result of evaluating the last expression.
- The default value of local variables declared by LET is NIL.

Iteration

An Example of LET

Is the situation of the party likely to be safe for you ? That is, are there more ‘friends’ then ‘enemies’ at the party!

```
(defun is-safe-p ( guests )
 (let ( (friendly-guests nil)
        (hostile-guests  nil) )
        (setq friendly-guests (identify-friends guests))
        (setq hostile-guests (identify-enemy guests))
        (> (length friendly-guests)
          (length hostile-guests) )
  ) )
```

Let*

Let* evaluates ALL of the expressions before ANY of the variables are bound. *Let* allows you to evaluate things in the order that they are listed (this is not necessarily true of LET)*

Example:

```
(let* ((sum (+ 8 3 4 2 7))     ; sum needs value
       (mean  (/  sum 5)))
   (* mean mean))
```

****Most of the time it doesn't matter, but check this if you keep getting errors.*

Iteration

- The simplest form of iteration is the – LOOP

```
(loop
   (<test condition> ) (return <optional var/val)
   [body]
  ) ;end loop
```

Example Loop

```
(let ((counter 1))    ; initializing my variable
  (loop
     (If  (= counter 2) (return (+ counter 5))
      (+ counter 1))))
```

6 is returned.

Another Example

```
(defun test ()
   (let ((n 0) (lis nil))   ; initializing
          (loop
           (if (> n 10) (return lis)
           (setq lis (cons n lis))) ; this is end
   of if
           (setq n (+ n 1)))))
```

=> (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

dotimes

Use dotimes for a counted loop

```
 (dotimes (<var--count> <intobj- upper limit> [retobj>])
          <body>)

(defun test ()
  (let ((sum 0))
    (dotimes (x   5   sum) ; x var initialized  to 0; 5 is limit; sum is return
      (print x) (print sum)
      (setq sum (+ sum x)))))
```

Dolist

Use DOLIST to process elements of a list.
This is like cdring down the list – recursion!

(dolist (<var><list> [<retobj>]
<body>)

Evaluates <list> ;which must be a list

The var is then bound to each item in list

Body is executed in order

Example dolist

```
(defun sq-list (lst1)
   (let ((lst2   nil))
         (dolist (ele   lst1   (reverse  lst2))
                         ; assigns first element lst1 to ele – again
             ;remember that were first/resting through            the
    list.
                         ; will go through each element of lst1
                         ; will return the reverse of lst2
                   (setq  lst2  (cons (  *   ele  ele )
                                                 lst2)))))
(sq-list '(1 2 3 4)
⇒ (1 4 9 16)
```

Another example:

```
(defun greet (people)
     (dolist (person people)
        (print (append '(so nice to see you)
                                        (list person)))))

(setq guests '(sally bobo mr-potato-head))
(greet guests) ->
  (so nice to see you sally)
  (so nice to see you bobo)
  (so nice to see you mr-potato-head)
```

The Do iteration

- The DO function

- (do ((<var1> [<init1> [<step1>]])

```
      .
       (<varn [<initn>[stepn>]]))
      (<end-pred><fn>…..<fm> <retobj >)
   body-exp1
   body-exp2
    [<body>]
```

Example of Do

Countdown

```
(defun iter (max)
        (do ((num max (- num 1)))
                ; assign max to num                                        ;
    subtract 1 from num in loop
            ((<= num 0) 'end) ;this is end cond.
          (print num)))
```

Another example

Summing all integers

```
(defun iter-sum (max)
          (do ((num max (- num 1))
               (sum 0 (+ sum num)))
             ((<= num 0) sum)))
```

In English: Assign max to number, and 0 to sum. Each time through the loop, subtract 1 from num and add sum to number. When Number is less than or equal to 0, return sum.

Yet Another

```
(defun reverse-list (ls)
      (if (listp ls)
        (do ((new-ls  () (cons (first old-ls)
    new-ls))
                (old-ls ls (rest old-ls)))
               ((null old-ls) new-ls) ) ) )
```

```
[user]: (reverse-list '(3 1 4 1 5 9))
(9 5 1 4 1 3)
```

* With do function, you probably don’t need a let --- because it’s in the do

Iterative Function

Write an iterative function called *stop* that takes a list and a number and returns a list that does not exceed the number

EVAL

EVAL

- (EVAL <exp>) invokes the evaluation procedure.

```
> (EVAL (LIST '+ 4 5))
9
> (SETQ A 'X)
X
> (SETQ X 100)
100
> (CONS (EVAL A) (CONS A '( A ) ) )
```

Using EVAL to create fn at run-time

Create a function to recognize cups based on the description learned:

```
(defun create-cup-recognizer ( cup-description )
   (eval (list ‘defun ‘cup-recognizer ‘( obj )
              (list ‘subset
                   (list ‘quote cup-description)
                   ‘obj)
            ) ) )
```

subset tests whether the first arg is a subset of the second arg.

```
> (create-cup-recognizer cup-def)
cup-recognizer
```

APPLY

- (APPLY <fn-name> <arg-list>)

```
>(APPLY  'CONS '( A  ( B C ) ) )
(A B C)
> (APPLY 'CAR '( (A B C) ) )
A
> (SETQ OP '+)
+
> (APPLY OP '( 5 3 ) )
8
```

A Better Example of APPLY

Suppose we want to ``do something'' to an enemy identified, but the action is determined at run time (based on their actions, degree of threats, etc.).

```
(defun action (fn enemy)
  (APPLY fn (cons enemy nil) ) )
(defun seize (person) .... )
(defun kill (person) ... )
(defun drunk (person) ... )
```

FUNCALL -- A sibling of APPLY

- (FUNCALL <fn-name> <arg1> <arg2> ...)

```
>(FUNCALL  'CONS 'A  '( B C ) )
(A B C)
> (FUNCALL  'CAR  '(A B C) )
A
> (SETQ OP '+)
+
> (FUNCALL OP  5 3 )
8
> (SETQ OP '-)
-
> (FUNCALL OP  5 3  )
```

What if we want to apply a function to an entire list?

```
(MAPCAR <fn-name> <arg1-list> <arg2-
   list>)
> (MAPCAR ‘+ ‘(1 2 3 4) ‘(100 200 300
   400) )
(101 202 303 404)
> (SETQ action ‘drunk)
DRUNK
> (SETQ hostile-guests (identify-enemy
   guests))
```

LAMBDA -- An anonymous function

- (LAMBDA (<arg1> <arg2> ...) <exp1> ....)
- How to invoke it? It is typically invoked by EVAL, APPLY, FUNCALL, or mapping functions.

```
> (MAPCAR (lambda (x) (+ x bonus)) salary-
  list )
```

- When to use it? It is typically used for defining a function that is needed only in a specific situation.
- Can it also appear in the position of a

Understanding LAMBDA form

- Viewing it as a function with a name you imagined.

(@@@@ .... (lambda ....)
###...)

$$\Big\downarrow$$

(@@@@ ... <imaginary-fn-name> ###
...)

Why Lisp is very important?

From an MIT job advert

…Applicants must also have extensive knowledge of C and UNIX, although they should also have sufficiently good programming taste to not consider this an achievement…

Why should I teach Lisp?

Lisp is the basis for the teaching of programming at MIT. Why?

- Lisp doesn’t force any single style. The same language can be used to teach procedural, data-flow, and object-oriented programming.
- Students don’t have to learn a different syntax for each style of programming.
- Students learn how the different styles work, by implementing them in Lisp as well as programming them.
- Lisp has a closeness to formal language theory that helps understanding of compilers, interpreters, and language semantics.

Lisp’s success as a teaching language isn’t a coincidence. It has a unique simplicity and mathematical elegance that corresponds to the concepts that are taught in programming.

First-course programming language frequencies

140	Pascal
57	Ada
48	Scheme
45	Modula (52 for all dialects)
25	C
8	Fortran
7	C++
6	Modula-2
5	SML
4	Turing
…	…

Lisp: A language surrounded by myths

Many things are said of Lisp that aren't true.

- It is big.
- It is slow.
- It is difficult to learn.
- It is an "artificial intelligence" language.
- It is an academic/research language.

Twenty years ago, these were true, but Lisp has evolved, and the requirements on programming languages have changed.

- Software costs outweigh hardware costs; Lisp's fast development can reduce costs significantly.
- Lisp has changed to meet new requirements; today it has as much Pascal in its ancestry as it does traditional Lisp.

What is Lisp?

- "Lisp" is an acronym for "list processing"
- Derived from Church's lambda calculus, but made efficiently computable
- Originally designed for symbolic processing, eg. differentiation and integration of algebraic expressions
- Data as symbolic expressions; simple, general data structures, for both programs and data.
- Consistent; everything in Lisp is a first-class object, numbers, lists, functions, symbols, objects, etc.
- Lisp has a thirty-odd year long history of success; it has a culture, and has been proved many times over.

Lisp dialects today

There are two dialects in common use today, reflecting different requirements

- Common Lisp;
  - designed as an "industrial strength" language.
  - powerful, expressive, and efficient.
- Scheme;
  - designed as a teaching language, and in common use today.
  - a small, clean "gem-like" language with Pascal in its ancestry.

These languages have many features in common.

- Both run on almost every kind of computer.
- Many public domain implementations exist.
- Both are designed to be compiled efficiently.
- Both use static binding, rather than the dynamic binding common in interpreted implementations in the past.

Data typing in Lisp

Lisp is a "dynamically typed" language; it associates the type with the value rather than the variable.

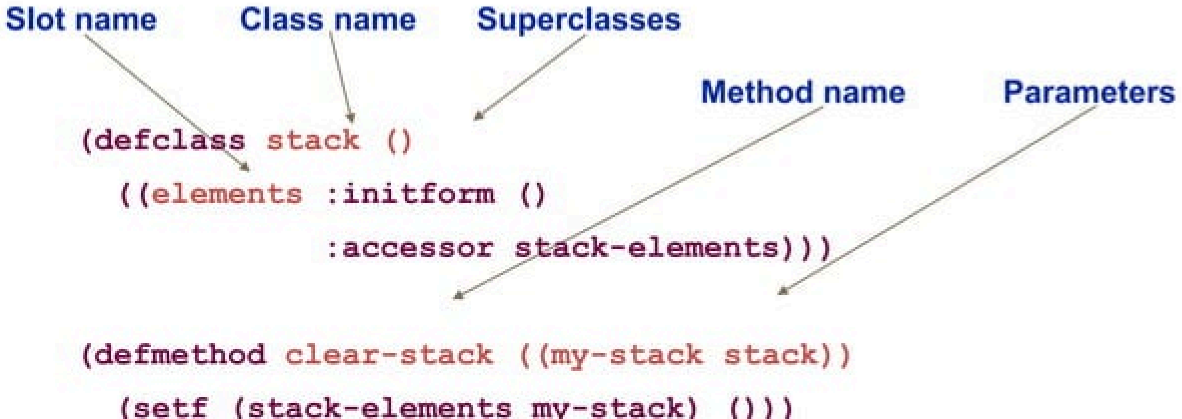
- First class data types include:
  - Symbols; nil, factorial, +
  - Cons cells, lists; (1 2 3 4 5), (foo . bar)
  - Numbers; 1, 1.63552716, 2/3, 66159327278283638
  - Functions; #'factorial, #'+
  - Closures; #<lexical-closure #x62AF40>
  - Objects; #<stack #x62AE20>
  - Arrays; "Hello there", #(1 2 3 4 5)
  - Hash tables; #<hash-table #x61AE90>

Detecting errors at run-time is easy; the debugger will appear whenever there is a type error.

Is Lisp object-oriented?

The Common Lisp object system is perhaps the most advanced object system in the world. It features:

- Multiple inheritance
- Multimethods; dispatching on more than one parameter



```
(defclass stack ()
  ((elements :initform ()
             :accessor stack-elements)))

(defmethod clear-stack ((my-stack stack))
  (setf (stack-elements my-stack) ()))
```

An example using CLOS

It is easy to define a stack class in CLOS:

```
(defclass stack ()
  ((elements :initform ()
             :accessor stack-elements)))

(defmethod push-element ((my-stack stack) element)
  (push element (stack-elements my-stack)))

(defmethod pop-element ((my-stack stack))
  (check-type (stack-elements my-stack) cons)
  (pop (stack-elements my-stack)))
```

`defclass`	defines a class
`defmethod`	defines a method
`setf`	assignment
`push/pop`	Common Lisp stack primitives

Beyond message-passing: multimethods

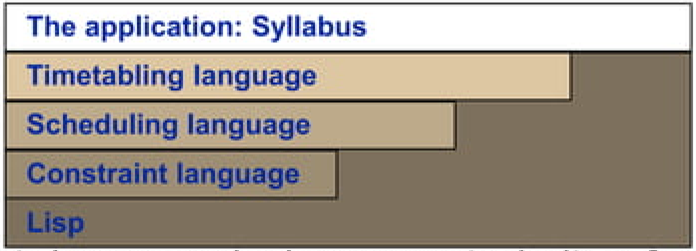
In CLOS, methods can dispatch on more than one parameter; message-passing is replaced by "generic functions".

```
(defmethod append-list ((list1 list) (list2 list))
  (append list1 list2))

(defmethod append-list (list1 (list2 stack))
  (append-list list1 (stack-elements list2)))

(defmethod append-list ((list1 stack) list2)
  (append-list (stack-elements list1) list2))
```

Calling `(append-list stack1 stack2)` calls the third method, which calls the second method, which calls the first method, and then returns the final result.

Cascades of languages

Lisp programs often end up as cascades of languages, starting with general-purpose ones which gradually become more specialised to the task.



This has several advantages, including, for example:

- Reuse
- Reliability; ease of debugging

Languages and Lisp

Lisp makes the treatment of different languages simple; this is the core of a Lisp interpreter for Scheme.

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quoted? exp) (text-of-quotation exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ((lambda? exp) (make-procedure exp env))
        ((conditional? exp) (eval-cond (clauses exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else (error "Unknown expression" exp))))
```

Exploration of the underlying mechanism makes programs easier to understand. Lisp is a natural for this.

New dimensions of Lisp: macros and macro packages

Macros are Lisp programs that transform other Lisp programs, by operating on them as data structures.

- General purpose macro packages can be developed.
  - Series package; data-flow programming.
  - Screamer package: nondeterministic programming.

- Normal definition for the factorial function

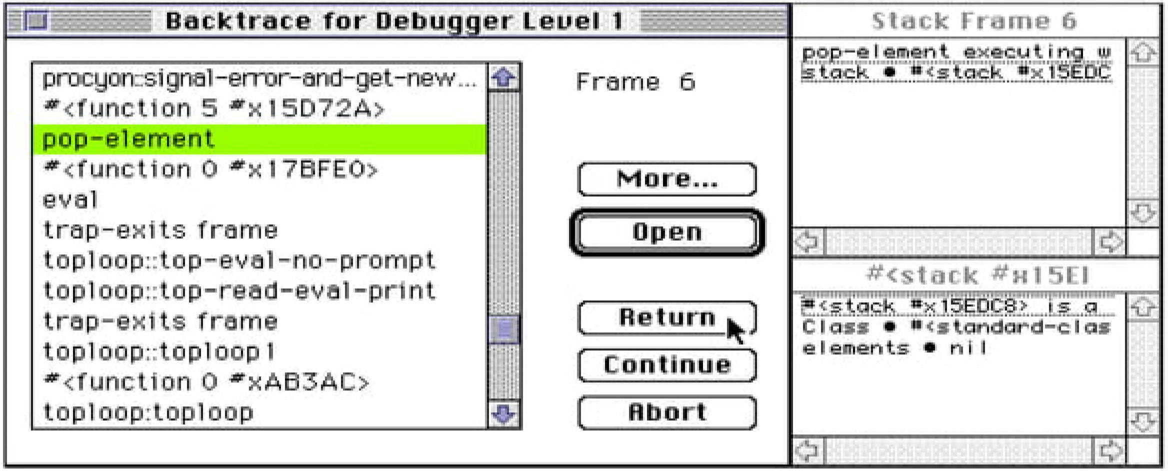
```
(defun fact (n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
```

- Series package definition for the factorial function

```
(defun fact (n)
  (collect-product (scan-range :start 1 :upto n)))
```

Debugging in Lisp

Lisp has an interactive debugger. When an error happens, a window like this appears, and the inspector can be used to help find the problem.



Backtrace for Debugger Level 1

```
procyon:signal-error-and-get-new...
#<function 5 #x15D72A>
pop-element
#<function 0 #x17BFE0>
eval
trap-exits frame
toploop::top-eval-no-prompt
toploop::top-read-eval-print
trap-exits frame
toploop::toploop1
#<function 0 #xAB3AC>
toploop:toploop
```

Frame 6

Stack Frame 6

```
pop-element executing w
stack ● #<stack #x15EDC
```

#<stack #x15EI

```
#<stack #x15EDC8> is a
Class ● #<standard-clas
elements ● nil
```

Expanding the business: using Lisp in industry

- Many large companies have small groups using Lisp to solve real problems, but the prejudices still live.
- A principal advantage of Lisp is its development speed; other languages need more effort for the same result.
- Even with an incremental C++ environment, Lisp programmers of the same skill are 2–5 times more productive; this makes some projects viable which otherwise wouldn’t be.
- Skill shortage is a problem, but training is straightforward and people quickly become productive.

Conclusions: the “take home” message

- Lisp is the victim of pride and prejudice.
- Lisp is proven; it has adapted and survived through its thirty-year history.
- Lisp can be—and is—used effectively as a teaching resource.
- Lisp encourages a style of cascading languages; ideal for software reuse.
- Lisp can make—and has made—some “niche” applications viable in industry.
- Lisp is still adapting to new circumstances and breaking new ground. It’ll be here for many years yet.

Pride and prejudice: Four decades of Lisp

Stuart Watt
Human Cognition Research Laboratory,
The Open University,
Walton Hall,
Milton Keynes, MK7 6AA.
Email: S.N.K.Watt@open.ac.uk

Presented to:
Workshop on the choice of programming languages
29-30th September 1993

If..then..else

IF <test> <then> <else>

(If (> n 1) (+ n 1) (n n))*

In English: If n is greater than 1 then add 1 to n, else times n by itself.

You can also have just:

(If (> n 1) (+ n 1)) -> but it will return a nil

Try and avoid nested ifs....

Functions as Arguments

- EVAL
- APPLY
- Mapping Functions

Homework Problems

Homework Problems

Problems:

1. Write a function (power 3 2) = 3^2 = 9
2. Write a function that counts the number of atoms in an expression.
   (count-atoms '(a (b) c)) --> 3
3. (count-anywhere 'a '(a ((a) b) a)) --> 3
4. (dot-product '(10 20) '(3 4)) --> 10x3 + 20x4 = 110
5. Write a function (flatten '(a (b) () ((c)))) --> (a b c)
   which removes all levels of parenthesis and returns a flat list of atoms.
6. Write a function (remove-dups '(a 1 1 a b 2 b)) --> (a 1 b 2)
   which removes all duplicate atoms from a flat list.
   (Note: there is a built-in remove-duplicates in Common Lisp, do not use it).

Solutions 1-3

```
(defun power (a b)
"compute a^b  - (power 3 2) ==> 9"
  (if (= b 0) 1
      (* a (power a (- b 1)))))


(defun count-atoms (exp)
"count atoms in expresion - (count-atoms '(a (b) c)) ==> 3"
  (cond ((null exp) 0)
             ((atom exp) 1)
             (t (+ (count-atoms (first exp)) (count-atoms (rest exp))))))

(defun count-anywhere (a exp)
"count performances of a in expresion -
 (count-anywhere 'a '(a ((a) b) (a))) ==> 3"
  (cond ((null exp) 0)
             ((atom exp) (if (eq a exp) 1 0))
             (t (+ (count-anywhere a (first exp))
                   (count-anywhere a (rest exp))))))
```

Solutions

```
(defun flatten (exp)
"removes all levels of paranthesis and returns flat list of
atomsi
 (flatten '(a (b) () ((c)))) ==> (a b c)"
   (cond ((null exp) nil)
              ((atom exp) (list exp))
              (t (append (flatten (first exp))
                         (flatten (rest exp))))))
```

Homework

- VECTORPLUS X Y

  : Write a function (VECTORPLUS X Y) which takes two lists and adds each number at the same position of each list.

- **Example**
  - (VECTORPLUS '(3 6 9 10 4) '(8 5 2))

    → (11 11 11 10 4)

Solution

```
(DEFUN  VECPLUS  (X  Y)
    (COND
            ((NULL  X)  Y)
            ((NULL  Y)  X)
          (T   (CONS  (+  (CAR  X)  (CAR  Y)
                                    (VECPLUS  (CDR  X)   (CDR  Y))))))
```

For Homework

- Representing predicate calculus with LISP
- ∀x likes (x, ice_cream)

  (Forall (VAR X) (likes (VAR X) ice_cream))
- ∃x foo (x, two, (plus two three))

  (Exist (VAR X) (foo (VAR X) two (plus two three)))
- Connectives

$\neg S$	(nott S)
$S_1 \wedge S_2$	(ANDD S_1 S_2)
$S_1 \vee S_2$	(ORR S_1 S_2)
$S_1 \rightarrow S_2$	(imply S_1 S_2)
$S_1 \equiv S_2$	(equiv S_1 S_2)

- Do not use the builtin predicate names.

For Homework

- Examples of predicates
  - **true, false**
  - **likes (george, kate)   likes (x, george)**
    likes (george, susie)   likes (x, x)
    likes (george, sarah, tuesday)
    friends (bill, richard)
    friends (bill, george)
    friends (father_of(david), father_of(andrew))
    helps (bill, george)
    helps (richard, bill)
    equals (plus (two, three), five)

For Homework

```
(DEFUN  atomic_s (Ex)
   (cond ((member Ex '(true  false)) T)
         ((member (car Ex) '(likes …))
            (Test_term (cdr Ex)))
         (T nil)))

(DEFUN Test_term (ex1)
   (cond ((NULL ex1) T)
         ((member (car ex1) '(george …..)
          (test_term (cdr ex1)))
         ((VARP (car ex1)) (test_term (cdr ex1)))
         ((functionp (car ex1)) (test_term (cdr ex1)))
         (T  nil)))
```

Exercises

Evaluate the following:

```
(setq x 'outside)
(let ((x 'inside) (y x))
    (list x y))

(let* ((x 'inside) (y x))
      (list x y))
```