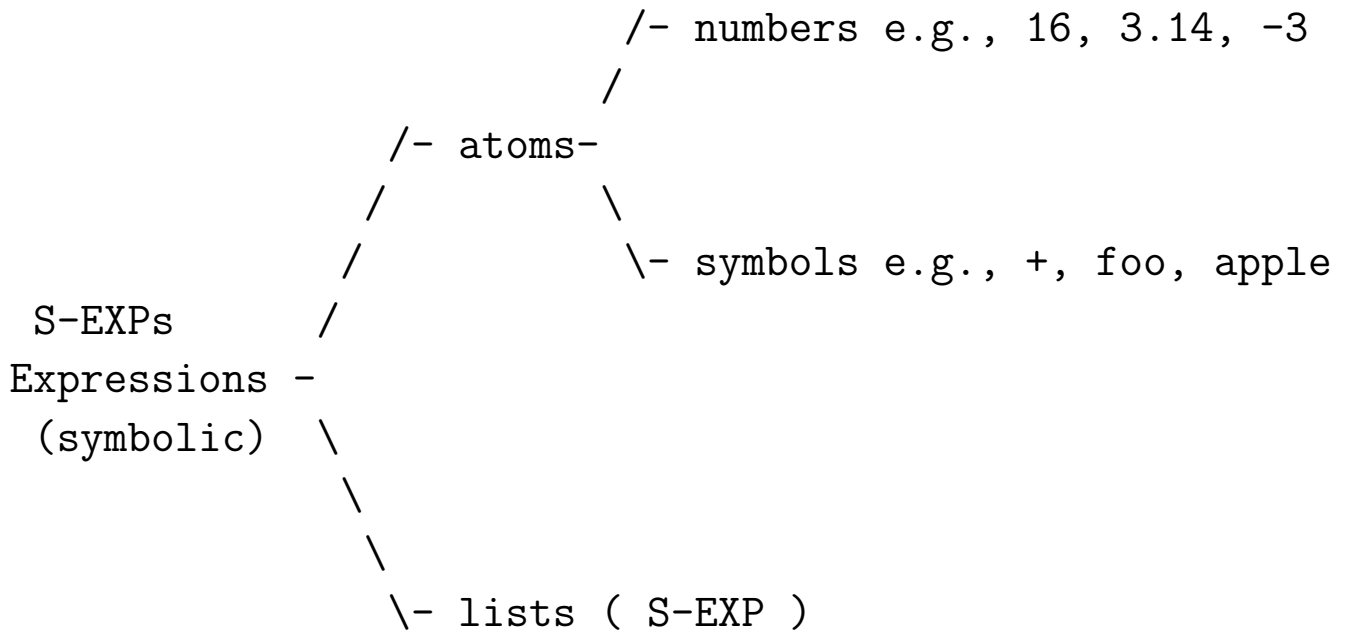

LISP Introduction

Lisp Data Structures



ATOMS

3

a

"hi there"

LISTS

(3)

(a b c)

(hi (there 3) a (b (c)))

Symbols and Lists

When a symbol is introduced in Lisp, a structure is created in the Lisp symbol table. Every Lisp symbol has five parts:

- print name
- value
- function definition
- property list
- package

Lisp functions can be used to set, update, or extract these parts of a symbol.

Symbols

If you type a symbol into the interpreter, it will return the value of the symbol (it returns the value of any expression you type in).

If that symbol does not have a defined value, you will get an error message.

```
> a
```

```
Fatal error in function SYSTEM::INTERPRET (signaled with ER-  
ROR). Symbol has no value: A
```

There are two special symbols, T and NIL, that have predefined values. All other symbols have no predefined value. t has the value T, nil has the value NIL (Lisp is not case-sensitive).

```
> t
```

```
T
```

```
> nil
```

```
NIL
```

```
> T
```

```
T
```

```
> 1
```

```
1
```

```
> (1 2 3)
```

```
ERROR
```

Expressions

A LISP interpreter interprets each “expression” that is entered.

If you type a symbol into the interpreter, it will return the value of the symbol.

If you type in an expression that begins with a “(”, Lisp

- evaluates the non-leading arguments (no guaranteed order),
 - looks up the function definition of the first argument (symbol), and
 - applies the function to the arguments
-

Evaluating Expressions in LISP

PREFIX notation (Lisp): (operator arg1 arg2 ... argn)

INFIX notation (C, sometimes): arg1 operator arg2 operator ... op
argn

C is inconsistent. For built-in operators such as +, -, *, /, %, it uses infix notation. For function calls, it uses prefix notation quicksort(array, size).

LISP always uses infix notation. Furthermore, the syntax is

(operator arg1 arg2 ... argn)

and NOT

operator(arg1, arg2, ..., argn)

Evaluating Expressions in LISP

Whenever the interpreter sees "(" , it expects the next symbol to represent a function name, and the following symbols to represent arguments up until the corresponding ")". The function name and all arguments are separated by white space (blank, new line).

```
(+ 1 2 3)    -> 6
(* 5 6)      -> 30
(- 6 5)      -> 1
(- 12 5 3)   -> 4
```

Function calls can be embedded inside other function call

```
(+ 1 (* 2 3)) -> 7
```

LISP evaluates each argument (the other is implementation and then applies the function to the evaluated arguments.

```
(+ 5 (- 7 4) (* 2 1)) -> 10
```

Whenever you enter an expression to the Lisp interpreter, interpreter EVALUATES the input and returns the value of

Examples

Let "<" represent our Lisp interpreter prompt.

Assume that the symbol 'a has the value 3, and a function definition that returns the sum of its arguments.

```
>a
```

```
3
```

```
>(a 1 2 3)
```

```
6
```

```
>(a a a 2)
```

```
8
```

Quote

Sometimes we want to PREVENT evaluation of an expression. To do this, we can use the built-in function QUOTE. For example, we may want to build a list and not have the interpreter treat the list like a function call.

```
(QUOTE x)
```

This returns the print name x.

```
> (QUOTE a)
```

```
A
```

```
> (QUOTE t)
```

```
T
```

```
> (QUOTE hi)
```

```
HI
```

```
> (QUOTE (1 2 3 a b c))
```

```
(1 2 3 A B C)
```

```
> (1 2 3 a b c)
```

```
ERROR
```

We will use QUOTE very often. Therefore, we will use a short-hand representation for QUOTE, that does the same thing. This is to just use the quote symbol '. We don't need parentheses around the short-hand version.

Examples

```
>(a a a 2)
```

```
8
```

```
>'(a a a 2)
```

```
(A A A 2)
```

Building and Manipulating Lists

- One way to build a list: use QUOTE
- Another way to build a list: use the LIST function
(list $a_1 a_2 \cdots a_n$) returns a list of the arguments Remember: the arguments are evaluated first!

```
> (list 1 2 3)
(1 2 3)
```

```
> (list a b c)
ERROR
```

```
> (list 1 'a 2 'b)
(1 A 2 B)
```

```
> (list 1 (list 2 (list 3)))
(1 (2 (3)))
```

Building and Manipulating Lists

- We can extract parts of a list using FIRST (or CAR) and REST (or CDR)
FIRST returns the first element of a list (it may be an atom or a list).
REST returns everything BUT the first element of the list. Almost as though we just cut out the first element.

```
> (first '(a b c))  
A
```

```
> (first (((a))) ((b)) (c))  
ERROR
```

```
> (first '(((a))) ((b)) (c))  
(((A)))
```

REST returns everything BUT the first element of
Almost as though we just cut out the first element

```
> (rest '(a b c))  
(B C)
```

```
> (rest '(((a))) ((b)) (c))  
(((b)) (c))
```

Building and Manipulating Lists

- How do our special symbols, T and NIL, fit in?

T is an atom and not a list.

NIL is both an atom AND a list. NIL is equivalent to the empty list.

They can both be used as elements of a list.

Car and cdr

Older, equivalent terms to first and rest
(developed for IBM 709, which had 2 registers)
car (like first, Contents of Address Register)
cdr (like rest, Contents of Decrement Register)

```
+----+----+  
| * | * |  
+-|---|---+  
  |   |  
  car cdr
```

Examples

```
>(first '(a a a 2))
```

```
A
```

```
>(list a 'a a 2)
```

```
(3 A 3 2)
```

```
>(first (rest (list 'a 'a 'a 2)))
```

```
A ; This looks like a "second" function!
```

Conditionals

- The most basic type of conditional is the IF function

```
(IF condition
```

```
then-action
else-action)
```

The condition is evaluated. If the value is non-NIL (anything but NIL), then the then expression is evaluated and returned as the value of the IF function.

If the value of the condition is NIL, then the else expression is evaluated and returned as the value of the IF function.

Remember, there is only one expression for then, and one for else.

Conditionals

- Easy condition functions for numbers:
 - (`= x y`) returns T if they are equal, returns NIL otherwise
 - (`< x y`) returns T if $x < y$, returns NIL otherwise
 - (`> x y`) returns T if $x > y$, returns NIL otherwise

```
(if (= 1 (+ 2 3))
    (* 4 2)
    (/ 4 2))
```

Functions

The command "defun" stands for "define function". The format is as follows:

```
(defun function-name (parameters)
  "comment describing this function"
  exp-1
  exp-2
  ...
  exp2-n)
```

When the function is called, the value returned for the function is the value returned by the last expression in the function.

Example

```
(define test (x y)
  "This is a sample function for class"
  (+ x (* 5 y)))
```

```
>(test 3 5)
28
```

Example

```
(define summation (x)
  "This function computes the summation of integers 1 to x"
  (if (= x 1)
      1
      (+ x (summation (- x 1)))))
```

Example

Now write a recursive function that computes $n!$

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

List Manipulation

CONS is a function that adds a new “first” item to a list.

CONS takes two arguments (cons x y)

The first of the new cell will be x, and the rest of the new cell will be y.

```
(cons 3 '(a b))
```

```
(3 A B)
```

The value of this expression is the NEW created list.

```
(first (cons newitem list)) returns newitem
```

Predicates

listp, null, atom, integerp, numberp, oddp, evenp, =, equal, eq

All of these predicates return T or NIL.

T if the condition is true, NIL if the expression is false.

This is useful for conditional expressions.

Example

Write a recursive function called “count-atoms” that counts the number of non-null top-level atoms in an input expression.

```
(count-atoms '(((3 4) a (b c)) d ((foo) fee) fi fum))  
10
```

Example

```
(defun count-atoms (in)  
  (if (atom in)  
      1  
      (+ (count-atoms (first in)) (count-atoms (rest in)))))
```

Will this work?

Careful, NIL will be counted as an atom.

Example

How about

```
(defun count-atoms (in)  
  (if (not (listp in))  
      1  
      (+ (count-atoms (first in)) (count-atoms (rest in)))))
```

When NIL is reached, will recurse indefinitely.

Example

```
(defun count-atoms (in)
  (if (null in)
      0
      (if (atom in)
          1
          (+ (count-atoms (first in)) (count-atoms (rest in)))))
```

Example

We could use `numberp` to sum numbers in an expression.

```
(defun count-numbers (in)
  (if (atom in)
      (if (numberp in)
          in
          0)
      (+ (count-numbers (first in)) (count-numbers (rest in)))))
```

Variables

Define local variables

Variables can be referred **ONLY** within the body in which they are introduced.

Variables are a new concept in Lisp, truly functional languages do not use variables.

This body is called the “scope” of the variable.

Let

“Let” introduces and initializes local variables. The syntax is

```
(let ((var1 init1)
      (var2 init2)
      ...
      (varn initn))
  exp1
  ...
  expx )
```

var1 is initialized to init1, etc.

The variables are defined within the scope of the let statement. At the close of the let statement, they are undefined (unless defined elsewhere).

The value returned by let is the value of the last expression evaluated in let.

Note that var2 cannot use var1, because they are initialized in a random order. (Use let* to initialize in order.)

Examples

```
(defun modulo (x y)
  "This function computes x mod y."
  (let ((val (floor (/ x y)))
        (product nil)
        (remainder nil)))
```

```
(setf product (* val y))
(setf remainder (- x product))
remainder))
```

Examples

```
(defun test1 (var1 var2)
  (test2 var1 var2))
```

```
(defun test2 (x y)
  (let ((var1 (+ x 1)))
    (* var1 y)))
```

What will (test1 0 5) yield?

The function will return 5, because let defines new binding for the variable.

Variable Assignment

SETF and SETQ

Setf and setq allow you to assign a value to a variable.

```
> (setf x '(john q public))
(john q public)
> x
(john q public)
```

```
> (quote x)
```

```
x
```

```
SETQ vs SETF
```

```
(setq variable value)
```

```
(setf variable value)
```

```
(setf positional-information value)
```

Example

```
> (setf p '((1st element) 2 (element 3) ((4)) 5))
```

```
((1st element) 2 (element 3) ((4)) 5)
```

```
> (first p)
```

```
(1st element)
```

```
> (first (rest (rest p)))
```

```
(element 3)
```

```
>a
```

```
ERROR
```

```
>(setf a 5)
```

```
>a
```

```
5
```

```
>(a a)
```

```
ERROR
```

```
>(defun a (+ a 1))
```

```
>(a a)
```

```
6
```

Note: LISP uses pass by value, so you will need to reset values from the calling function, not from the called function.

More List manipulation

- LENGTH returns the number of expressions in a list
- NTH returns the nth item in a list (numbered from 0)

```
(length '(a)) -> 1
```

```
(length nil) -> 0
```

```
(length '(a b c)) -> 3
```

```
(length '((a (b (c)))))) -> 1
```

"nth" returns the nth item in a list (numbered from 0)

```
(nth 0 '(a b c)) -> a
```

```
(nth 3 '((1 2) (3 4) (5 6) (7 8))) -> (7 8)
```

```
(nth 3 '(a b c)) -> nil
```

```
(setf foo '(a b c)) -> (A B C)
```

```
(setf (nth 0 foo) 'q) -> Q
```

```
foo -> (A B Q)
```

```
(setq (nth 0 foo) 'q) -> ERROR
```

Cond

Consider the function

```
(defun query-user ()
  (let ((answer (read)))
    (if (= answer 0)
        'ok
        (if (= answer 1)
            'almost
            (if (= answer 2)
                'no
                'definitely-not))))))
```

Can we clean up the function (eliminate deep embedding)?

Cond

COND allows multiple cases and the corresponding actions.

```
(cond (cond1 action1)
      (cond2 action2)
      ...
      (condn actionn) )
```

Cond

Cond tests each condition in order. The first one that evaluates to non-NIL, the corresponding action is evaluated and its value is returned.

returned as the value of the cond statement.

```
(defun query-user ()
  (let ((answer (read)))
    (cond ((= answer 0) 'ok)
          ((= answer 1) 'almost)
          ((= answer 2) 'no)
          (t 'definitely-not))))
```

Iteration

DOTIMES iterates exactly n times through its expressions, and DOLIST iterates through a particular list (one time per expression in the list).

(dotimes (index n return-value)	(dolist (one-item l
exp1	exp1
...	...
expn)	expn)

Example

Write a function “seti” that takes a list of phonemes (no embedded lists) and a list of known alien sounds. Seti counts the number of input phonemes that are alien sounds.

```
>(seti '(ug grok boo hah me hello ug attack) '(grok ug))
3
```

```
(defun seti (phonemes asounds)
  (let ((numsounds 0))
    (dolist (x phonemes numsounds)
      (when (asound x asounds) ; WHEN is an if statement with
          ; UNLESS is an if statement without
            (setf numsounds (+ numsounds 1))))))
```

```
(defun asound (sound asounds)
  (if (null asounds)
      nil
      (if (equal sound (first asounds))
          t
          (asound sound (rest asounds)))))
```

Example

Ok, now let's write seti to accept embedded lists of phonemes. Seti will return 'RUN if the number is greater than 10, and 'SAFE otherwise.

```
>(seti '(ug (grok boo hah (me)) hello (ug (attack))) '(grok ug))
RUN
```

```
(defun seti (phonemes asounds)
  (let ((numsounds 0))
    (setf numsounds (count-sounds numsounds phonemes asounds))
    (if (> numsounds 10)
```

```
'run
'safe)))
```

```
(defun count-sounds (numsounds phonemes asounds)
  (dolist (x phonemes numsounds)
    (if (not (listp x))
        (when (asound x asounds) ; when is an if statement with
              (setf numsounds (+ numsounds 1)))
        (setf numsounds (+ numsounds (count-sounds 0 x asound
```

```
(defun asound (sound asounds)
  (if (null asounds)
      nil
      (if (equal sound (first asounds))
          t
          (asound sound (rest asounds))))))
```

Iteration

DO loop, DO vs DO*

```
do ((var1 init1 update1)
    (var2 init2 update2)
    ...
    (varn initn updaten))

(end-condition return-value)
```

```

exp1
exp2
...
expn)

x = 2;
for (i=0; i<10&&x!=20; i++)
    x = foo(i, x);

(do ((i 0 (+ i 1))
     (x 2))
    ((or (>= i 10) (= x 20)) nil)
    (setf x (foo i x)))

```

Example

```

(defun seti (phonemes asounds)
  "A version of the seti program that uses do*.
  This version of the program expects only atoms in the list.
  (do* ((numsounds 0 numsounds)
        (phlist phonemes (rest phlist))
        (x (first phlist) (first phlist)))
        ((null phlist) numsounds)
        (when (asound x asounds)
              (setf numsounds (+ numsounds 1))))))

```

Append

APPEND concatenates two lists.

```
(append '(a b) '(c d))
```

```
>(setf a '(a b))
```

```
>(append a '(a b))
```

```
(A B C D)
```

Remove the inner “)” and “(” symbols.

I/O

```
(defun welcome-message ()
  (let ((name nil))
    (print 'hi!)           ; "Print" prints out a single s-
    (terpri)               ; Newline
    (format t "Your Name:  ")
    (setf name (read))
    (format t "Well, I am glad you logged in, ~a~%" name)
```

```
> (welcome-message)
```

```
hi!
```

```
Name: Diane
```

```
Well, I am glad you logged in, DIANE
```

```
NIL
```

Other Lisp functions

- Print prints out a single item, terpri prints a newline
- Read reads in a single expression (does not have to be quoted)
- read-char reads in one character
- Format enables exotic printing.
- The t parameters specifies the screen as output, can replaces with other outputs (read about using files as output)
- Directives

```
~a  Insert print name of argument
~%  Newline
```

- Read about format to get more directives.

Final Comment - Comments!

Putting a ; anywhere in a line treats the rest of the line as a comment. In functions, there is a place for function documentation.

```
(defun function-name (parameters)
  "comments about the function"
  exps)
```

Indentation and comments are VERY important to readability

Example Program

```
(defun hangman ()
  "Simple hangman game. The player has to keep track of letters
  he/she has already guessed."
  (let ((maxtries 15)
        (word '(i n t e l l i g e n c e))
        (current '(- - - - - - - - - - -))
        (guess nil))
    (dotimes (i maxtries)
      (print current)
      (terpri)
      (format t "What is your guess? ")
      (setf guess (read))
      (terpri)
      (setf current (update-word word current guess))
      (if (equal word current)
          (progn
              (print current) ; Here
              (terpri)
              (format t "Congratulations! You have won the game.")
              (return nil))
          (if (equal i (- maxtries 1))
              (format t "You lost this time. Try again!")
              nil))))))
```

Example Program

```
(defun update-word (word current guess)
```

```
"Update the current guess.  We can assume that the input is
(let ((found nil))
  (dotimes (i (length word) current) ; Word and Current hav
    (if (equal (nth i word) guess)
        (progn
          (setf found t)
          (setf (nth i current) guess))
        nil))
  (if found
      (format t "Good try, keep guessing")
      (format t "Nope, sorry~%"))
  current))
```

Defstruct

Define complex data structures.

```
(defstruct structure-name
  (parm1 default-value)
  (parm2 default-value)
  ...
  (parmn default-value))
```

Constructor

```
(setf var (make-structure-name :parmi value))
```

Selector

```
(print (structure-name-parmi value))
```

```
(setf (structure-name-parmi value))
```

Files

```
(defun open (filename &key :direction)) ...)
```

The OPEN function opens a filename and returns a STREAM object that points to the opened file. We can now direct functions to the specified stream.

:direction specifies whether the stream should handle input (:input), output (:output), or both (:io). The default is :input. The :direction option (:probe) just determines whether a file already exists (status or NIL).

```
(when (open "output" :direction :probe)
  (open "output" :direction :input)
  ...)
```

Files

We can now supply an optional parameter to READ specifying the stream from which to read.

Consider a file “test” containing data

3

```
(setf infile (open "test" :direction :input))
```

```
(setf temp1 (read infile))
```

```
(setf temp2 (read infile))
```

```
temp1 -> 3
```

```
temp2 -> 5
```

(read nil) is the same as reading from standard input. You can also use (read *standard-input*).

(read t) uses *terminal-io* as the stream.

If you try to read past the end of file, you will get an error.

The actual read function definition is

```
(defun read (&optional input-stream eof-error-p eof-value)) ...).
```

If nil is specified for eof-error-p, then instead of returning an error when you hit the end of the file, the value of eof-value will be returned instead.

```
(close infile)
```

Close the file specified by the stream parameter. The file can be reopened and the pointer will be set back to the beginning of the file.

Output

Now let's store output in a file.

```
(setf outfile (open "output" :direction :output))
```

```
(format outfile (The result of the computation is ~a~%" resul
```

(close outfile)

Normally, we use `(format t "")`, which sends the output to `*standard-output*`. If we use `(format nil "")`, the output is not sent to any file. Instead, the string generated from the function is returned.

Macros

A macro is essentially a function that generates Lisp code - a program that writes programs. This is very useful for AI applications such as natural language and automatic programming.

- Macro definitions look very similar to function definitions. Functions return results, but macros return **expressions** (which could themselves return results if they are evaluated).
- Write a macro called `"nil!"` that sets its argument to NIL, similar to the expression `(setf x nil)`.

```
> (defmacro nil! (var)
      (list 'setf var nil))
NIL!
```

This expression tells Lisp “whenever you see an expression of the form `(nil! var)`, turn it into one of the form `(setf var nil)` before evaluating it”.

Macros

- When the macro is called, it

- Builds the expression specified by the macro definition
- Evaluates that expression in place of the original macro call
- The step of building the new expression is called “macroexpansion”. A macro accepts parameters the same as a function.
- After macroexpansion is “evaluation”, where Lisp evaluates (setf x nil) as if THAT had been typed in in the first place.

The macro called from a function will be expanded when the function is compiled, but it won't be evaluated until the function is called.

Macros can call other macros.

Backquote

BACKQUOTE is a function similar to quote which is often used to create expressions for use by macros.

`'(a b c)` is equal to `'(a b c)`

same when applied by itself to an expression.

Different, however, when combined with ``,`` and ``,`@``.

Examples

- When a ``,`` appears inside the scope of a backquote, the element is evaluated (the quoting is turned off for that

`'(a b c)` is the same as `(list 'a 'b 'c)`

`'(a ,b c ,d)` is the same as `(list 'a b 'c d)`

```

> (setf a 1)
      (setf b 2)
      (setf c 3)

> '(a ,b c)
(A 2 C)

> '(a (,b c))
(A (2 C))

> '(a b ,c (',( + a b c)) (+ a b) 'c '((,a ,b)))
(A B 3 ('6) (+ A B) 'C '((1 2)))

> (defmacro nil! (var)
      '(setf ,var nil))

```

Examples

Compare these two versions of a numeric if (positive, 0,

```

(defmacro nif (expr pos zero neg)
  '(cond ((> ,expr 0) ,pos)
        ((= ,expr 0) ,zero)
        (< ,expr 0) ,neg)))

```

```

(defmacro nif (expr pos zero neg)
  (list 'cond

```

```
(list (list '> expr 0) pos)
(list (list '= expr 0) zero)
(list (list '< expr 0) neg)))
```

Now call it:

```
> (mapcar #'(lambda (x)
             (nif x 'p 'z 'n))
          '(0 2.5 -8))
(Z P N)
```

This is expanded into

```
(cond ((> x 0) 'p)
      ((= x 0) 'z)
      ((< x 0) 'n))
```

- The ",@" combination inside a backquote behaves like ``,`` except that the value is SPLICED into the list (the outer elements of the following expression are removed).

```
> (setf b '(1 2 3))
(1 2 3)
```

```
> `(a ,b c)
(A (1 2 3) C)
```

```
> `(a ,@b c)
(A 1 2 3 C)
```

This can only be used inside of another list. ``,`@b` would

Examples

```
> (defmacro our-when (test body)
  '(if ,test
      (progn
        ,@body)
      nil))
```

```
> (our-when t ((print 'it) (print 'works)))
IT
WORKS
```

Expands out to:

```
(if t
    (progn
      (print 'it)
      (print 'works)))
```

Notice that the argument to "our-when" is not quoted, be evaluated.

The expression `'(a ,@b c)` is equal to `(cons 'a (append b c))`. As you can see, `"'"` and `",@"` make expressions more readable.

You can use backquote outside of defmacros - it has a little more syntax.

```
(defun greet (name)
  `(hello ,name))
```

Grouping Arguments

Instead of passing in a list of expressions to “our-when”, we can use the “&rest” option in our parameter list, which groups a set of parameters together in one list.

```
(defmacro our-when (test &rest body)
  `(if ,test
      (progn
        ,@body)
      nil))
```

```
(our-when t (print 'it) (print 'works))
```

Let’s write a WHILE macro which will take an expression body of code, and loop through the code as long as the remains true. We will use the Lisp DO loop to impleme

```
(defmacro while (test &rest body)
  `(do ()
      ((not ,test))
      ,@body))
```

The built-in function macroexpand takes an expression macroexpansion.

```
> (macroexpand '(our-when t (print 'it) (print 'works))
(IF T (PROGN (PRINT 'IT) (PRINT 'WORKS)) NIL)
```

Parameter List Options

aux, keys, optional

These can be mixed, but be careful aux has to go at the end of the list

```
(defun foo (&optional (a nil) &key (r1 1.0) (r2 2.0) &aux (
  (if a
      (- count r1 r2)
      (+ count r1 r2))))
```

```
(foo) -> 3.0
```

```
(foo t) -> -3.0
```

```
(foo nil :r1 5.0) -> 7.0
```

```
(foo :r1 5.0) -> ERROR
```

Lambda Functions

Functions in Lisp can be called, and they can also be manipulated like other objects. They can be passed as parameters.

Lambda functions provide a method of introducing a function without giving it a name. The syntax is

```
(lambda (parameters...) body...)
```

A lambda function can be used anytime a function expects a function as an argument. Remember, if we want to get at the actual function, rather than its name, we must use `#'`.

```
#'(lambda (x y) (+ x (* x y)))
```

Apply

APPLY is one function that takes two arguments, the first of which is another function. The function definition of the first argument is accessed and applied to the rest of the arguments which are given in a list.

```
(apply #'(lambda (x y) (+ x (* x y))) '(3 5))
```

```
18
```

```
(apply #'+ '(1 2 3 4))
```

```
10
```

The funny `#'` notation maps from the NAME of a function to the FUNCTION itself.

Funcall

FUNCALL is like APPLY except it allows any number of arguments (they do not have to be supplied in a list).

```
(funcall #'(lambda (x y) (+ x (* x y))) 3 5)
```

```
18
```

Mapcar

A method of applying a function to each element of a list (without using `dolist`).

```
(mapcar function list)
```

`mapcar` returns the list created from applying FUNCTION to the input LIST.

```
(mapcar #'sqrt '(4 9 16 25 36))
```

```
(2 3 4 5 6)
```

Note that a lambda function will take one argument, which is replaced one at a time by elements of the list.

```
(mapcar #'(lambda (x) (* x 2)) '(1 2 3 4))  
(2 4 6 8)
```

Member

(member element list)

Predicate

If ELEMENT appears anywhere in LIST, returns portion of LIST starting with ELEMENT (always returns non-NIL)

Returns NIL if ELEMENT does not appear anywhere in LIST

Uses eq

```
(member element list :test #'equal)
```

Can use to keep track of visited states, for example

Debugging

Programs do not always work as expected, so you may need some debugging tools.

Debugging Tools

(trace function)

(untrace function)

CMU Common Lisp

quit, abort
leave debugger
GO
continues with execution (in case user put a break somewhere)
BACKTRACE num
shows flow of last num commands leading to error
ERROR
prints error message
(debug:arg i)
prints out ith argument in current function
#'function
prints out Lisp code if user-defined for current function
breakpoint num
This points a break at line num of the current function
delete-breakpoint [n]
delete breakpoint n or all breakpoints
list-breakpoints