Symbolic Programming

- Symbols: +, -, 1, 2 etc.
- Symbolic expressions: (+ 1 2), (+ (* 3 4) 2)
- Symbolic programs are programs that manipulate symbolic expressions.
- Symbolic manipulation: you do it all the time; now you'll write programs to do it.

Example

Logical rules can be represented as symbolic expressions of the form, ‘antecedent’ implies ‘consequent’.

The following rule states that the relation of ‘being inside something is transitive, ‘x is inside y and y is inside z’ implies ‘x is inside z’.

In lisp notation, we would represent this rule as

```
(SETF rule '(implies (and (inside x y)
                          (inside y z))
                     (inside x z)))
```

Manipulating symbolic expressions

(first rule) ;; The expression is a rule.
;; (FIRST RULE) → IMPLIES
(second rule) ;; The antecedent of the rule.
;; (SECOND RULE)→ (AND (INSIDE X Y) (INSIDE Y Z))
(third rule) ;; The consequent of the rule.
;; (THIRD RULE) → (INSIDE X Z)
(first (second rule)) ;; The antecedent is a conjunction.
;; (FIRST (SECOND RULE)) → AND
(second (second rule)) ;; The first conjunct of the antecedent.
;; (SECOND (SECOND RULE)) → (INSIDE X Y)

Lisp and Common Lisp

What are you looking for in a programming language?

- primitive data types: strings, numbers, arrays
- operators: $+$, -, $+$, $/$, etc.
- flow of control: if, or, and, while, for
- input/output: file handling, loading, compiling programs

Lisp has all of this and more.

- developed by John McCarthy at MIT
- second oldest programming language still in use (FORTRAN is oldest)
- specifically designed for symbolic programming

We use a dialect of Lisp called Common Lisp.

- Common Lisp is the standard dialect

We use a subset of Common Lisp.

Basic Data Types and Syntax

Numbers: 1, 2, 2.72, 3.14, 2/3 (integers, floating point, rationals)

Strings: "abc", "FOO", "this is also a string"

Characters:
- characters: a,b,...,z,A,B,...,Z,0,1,2,...,9,_,-,+,*
- alpha characters: a,b,...,z,A,B,...,Z
- numeric characters: 0,1,2,...,9

Symbols:

sequence of characters including one or more alpha characters
its actually more complicated than this but this will suffice

Examples

foo, my-foo, your_foo, 1foo, foo2, FOO, FOO2

Lisp programs and data are represented by expressions.

Expressions - (inductive definition)

- any instance of a primitive data type: numbers, strings, symbols
- a list of zero or more expressions

List

- open paren "("
- zero or more Lisp objects separated by white space
- close paren ")"

Examples

1, "foo", BAR	(primitive data types)
(), (1), (1 "foo" BAR)	(flat list structures)
(1 (1 "foo" BAR) biz 3.14)	(nested list structure)

Comment character ‘;’
Lisp ‘ignores’ anything to the right of a comment character.

Lisp is case insensitive with regard to symbols.
FOO, Foo, foo, fOO designate the same symbol, but
"FOO", "Foo", "foo", "fOO" designate different strings.

Interacting with the Lisp Interpreter

Instead of

1. Write program
2. Compile program
3. Execute program

you can simply type expressions to the Lisp 'interpreter.'

EVAL compiles, and executes symbolic expressions interpreted as programs.

The READ-EVAL-PRINT loop reads, EVALs, and prints the result of executing symbolic expressions.

Instances of simple data types EVALuate to themselves.

"string"
3.14
t
nil

sym ;; Would cause an error if typed to the interpreter.
;; Interpreter errors look like
;; > sym
;; Error: The symbol SYM has no global value

EVAL expression

— expression is a string
return the expression
— expression is a number
return the expression
— expression is a symbol
look up the value of the expression

You will learn how symbols get values later

Invoking Functions (Procedures) in Lisp

Function names are symbols: +, -, ***, *sort*, *merge*, *concatenate*

Lisp uses prefix notation *(function* $argument_1$ $\ldots$ $argument_n$*)*

(+ 1 2) ;; + is the function, 1 and 2 are the arguments

(+ 1 2 3) ;; + takes any number of arguments

What happens when EVAL encounters a list?

Lists (with the exception of special forms) are assumed to signal the invocation of a function.

APPLY handles function invocation

EVAL expression

— expression is a string or a number
return the expression
— expression is a symbol
look up the value of the expression
— it is of the form *(function_name expression*$_1$ ... *expression*$_n$*)*
APPLY function_name to expression$_1$ ... expression$_n$

APPLY function_name to $\text{expression}_1 \ldots \text{expression}_n$

— EVAL $\text{expression}_1 \rightarrow \text{result}_1$

...

EVAL $\text{expression}_n \rightarrow \text{result}_n$

— function_name better have a definition; look it up!

the definition for function_name should look something like

function_name $\text{formal\_parameter}_1 \ldots \text{formal\_parameter}_n$

expression involving $\text{formal\_parameter}_1 \ldots \text{formal\_parameter}_n$

— substitute result_i for $\text{formal\_parameter}_i$ in the expression

— EVAL the resulting expression

Example

function name:	WEIRD
formal parameters:	X1 X2 X3
definition:	X2

$>$ (WEIRD 1 "one" 1.0)
EVAL (WEIRD 1 "one" 1.0)
APPLY WEIRD to 1 "one" 1.0
EVAL 1 $\rightarrow$ 1
EVAL "one" $\rightarrow$ "one"
EVAL 1.0 $\rightarrow$ 1.0
substitute 1 for X1
substitute "one" for X2
substitute 1.0 for X3
EVAL "one"

```
> (WEIRD "1st arg 1st call"
            (weird "1st arg 2nd call"
                   "2nd arg 2nd call"
                   (weird "1st arg 3rd call"
                          "2nd arg 3rd call"
                          "3rd arg 3rd call"))
            "3rd arg 1st call")
"2nd arg 2nd call"
```

(+ (* (+ 1 2) 3) (/ 12 2))

```
(+ (* (+ 1              ;; Indenting helps to make nested
         2)             ;; function invocation clearer.
      3)
   (/ 12                ;; What is the order of evaluation
      2))               ;; in this nested list expression?
```

```
           (+ (* (+ 1 2) 3) (/ 12 2)) [9]
                /                  \
       (* (+ 1 2) 3) [5]        (/ 12 2) [8]
           /       \               /   \
    (+ 1 2) [3]     3 [4]      12 [6]   2 [7]
        /  \
   1 [1]    2 [2]
```

Defining Functions (Procedures) in Lisp

(defun function_name list_of_formal_parameters function_definition)

The function_name is a symbol.
The formal_parameters are symbols.
The function_definition is one or more expressions.

Examples

(defun weird (x y z) y)
↑ function name
↑ list of three formal parameters
↑ function definition consisting of one expression

Examples of Functions

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

(defun hypotenuse (a b)
   (sqrt (+ (square a)
            (square b))))
```

How would these functions appear in a text file?

Functions Documented

```
;; HYPOTENUSE takes two arguments corresponding
;; to the length of the two legs of a right triangle and returns
;; length of the hypotenuse of the triangle.

(defun hypotenuse (a b)
   ;; SQRT is a built-in function that
   ;; computes the square root of a number.
   (sqrt (+ (square a)
            (square b))))

;; SQUARE computes the square of its single argument.
(defun square (x)
   (* x x))
```

Boolean Functions and Predicates

In Lisp, NIL is boolean false and any expression that evaluates to anything other than NIL is interpreted as boolean true.

nil

T is the default boolean true. T evaluates to itself.

t

Boolean predicates return T or NIL.

(oddp 3)

(evenp 3)

(< 2 3)

(= 1 2)

Boolean Functions and Predicates

Boolean functions return non-NIL or NIL

An OR expression evaluates to non-NIL if at least one of its arguments must evaluate to non-NIL.

(or t nil)

Degenerate case of no arguments: *(or)*

An AND expression evaluates to non-NIL if All of its arguments must evaluate to non-NIL.

(and t nil)

Degenerate case of no arguments: *(and)*

A NOT expression evaluates to non-NIL if its only argument evaluates to NIL.

(not t)

Any expression can be interpreted as a boolean value.

(and 3.14 "this string is interpreted as boolean true")

Conditional Statements and Flow of Control

Any expression can be used as a test in a conditional statement.

Simple conditional statements

(if test_expression consequent_expression alternative_expression)

Formatted differently using automatic indentation.

(if test_expression
consequent_expression
alternative_expression)

Examples

(if t "consequent" "alternative")

(if nil "consequent" "alternative")

You do not need to include the alternative.

(if t "consequent")

The 'default' alternative is NIL.

(if nil "consequent")

General CONDitional statement

(cond conditional_clause$_1$ ... conditional_clause$_n$)

Conditional clause

(test_expression expression$_1$ ... expression$_m$)

Examples

```
(cond ((and t nil) 1)
      ((or (not t)) 2)
      ((and) 3))
```

```
(cond (t nil))
```

```
(if t nil)
```

```
(defun classify (x)
  (cond ((= x 0) "zero")
        ((evenp x) "even")
        ((oddp x) "odd")))
```

Examples (cont.)

```
(defun classify_again (x)
  (cond ((= x 0) "zero")
        ((evenp x) "even")
        (t "odd")))          ;; Good programming practice!

(classify_again 0)

(defun classify_once_more (x)
  (cond ((= x 0) "waste of time" "zero")
        ((evenp x) "who cares" "even")
        (t (+ x x) "what a waste" "odd")))

(classify_once_more 2)
```

Examples (cont.)

```
(defun classify_for_the_last_time (x)
   (cond ((= x 0) (princ "so far our lisp is pure") "zero")
         ((evenp x) (princ "side effects simplify coding") "even")
         (t (+ x x) (princ "side effects complicate understanding")
                    "odd")))

(classify_for_the_last_time 3)
```

Recursive functions

Recursion works by reducing problems to simpler problems and then combining the results.

```
(defun raise (x n)
  (if (= n 0)          ;; We can handle this case since x^0 is just 1.
    1
    (* x               ;; Reduce the problem using x^n = x * x^(n-1).
      (raise x (- n 1)))))
```

Example

	Order	Level
call RAISE 3 2	first time	1
call RAISE 3 1	second time	2
call RAISE 3 0	third time	3
return 1 from RAISE	from the third call	
return 3 from RAISE	from the second call	
return 9 from RAISE	from the first call	

APPLY and EVAL work together recursively

	Order	Level
call EVAL (+ (* 2 3) 4)	first time	1
call APPLY + with (* 2 3) and 4		
call EVAL (* 2 3)	second time	2
call APPLY * with 2 and 3		
call EVAL 2	third time	3
return 2 from EVAL	by the third call	
call EVAL 3	fourth time	3
return 3 from EVAL	by the fourth call	
return 6 from APPLY		
...	...	

Evaluating Functions in Files

$>$ *(load "one.lisp")*
or just
$>$ *(load "one")*
if the extension is ".lisp".

Symbols can be assigned Values

The global environment is just a big table that EVAL uses to look up or change the value of symbols.

There are other environments beside the global environment.

SETF changes values of symbols in environments.

Changing the value of a symbol is one example of a side effect.

Assign the symbol FOO the value 1 in the global environment

```
(SETF foo 1)
(SETF bar 2)
(SETF baz (+ foo bar))
```

Symbols assigned values in the global environment are global variables

(SETF sym 3)

(defun double (x) (+ x x))

(double sym)

We use the terms 'variable' and 'symbol' interchangeably.

Global variables can be referenced inside function definitions.

(SETF factor 3)

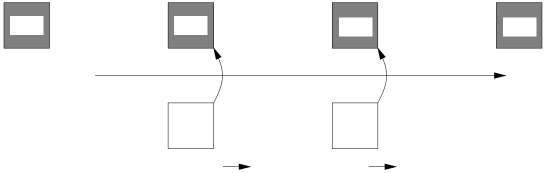
(defun scale (x) (x factor))*

(scale sym)

From a structured programming perspective, global variables are discouraged.

Function Invocation Revisited

New environments are created during function invocation.



```
(defun local (x)
   (SETF x (+ x 1))
   (* x x))
```

In the following example, the symbol X is assigned 2 in the global environment prior to invoking LOCAL on (+ X 1).

```
(SETF x 2)
```

Function Invocation Revisited (cont.)

In APPLYing LOCAL to (+ X 1), the single argument expression (+ X 1) EVALuates to 3.

Before EVALuating the definition of LOCAL, APPLY creates a new environment that points to the global environment. In this new environment, the formal parameter X is assigned the value 3 of the argument expression.

In looking up the value of a symbol while EVALuating the definition of LOCAL, EVAL looks first in the new environment and, if it can't find the symbol listed there, then it looks in the global environment.

(local (+ x 1))

In general, function invocation builds a sequence of environments that EVAL searches through.

Reconsider the roles of EVAL and APPLY

EVAL expression in ENV

— expression is a string
  return the expression
— expression is a number
  return the expression
— expression is a symbol
  look up the value of expression in ENV
— expression is a special form
  do something special!
— expression has form

$$(function\_name\ arg\_expression_1\ \ldots arg\_expression_n)$$

  APPLY function_name to arg_expression$_1$ … arg_expression$_n$
  in ENV

Reconsider the roles of EVAL and APPLY (cont.)

Note that now that we have side effects, order of evaluation is important. Now it makes sense for COND clauses to have more than one expression besides the test and for function definitions to consist of more than one expression.

(SETF x 1) (SETF x (+ x x)) (SETF x (x x))* ;; $x \rightarrow 4$

(SETF x 1) (SETF x (x x)) (SETF x (+ x x))* ;; $x \rightarrow 2$

Reconsider the roles of EVAL and APPLY (cont.)

APPLY function_name to
arg_expression$_1$... arg_expression$_n$ in ENV

— evaluate the arguments in left to right order
 — EVAL arg_expression$_1$ in ENV $\rightarrow$ arg_result$_1$ ...
 — EVAL arg_expression$_n$ in ENV $\rightarrow$ arg_result$_n$
— look up the definition of function_name:
 $function\_name\ formal\_parameter_1\ \ldots formal\_parameter_n$
 $definition = def\_expression_1\ \ldots def\_expression_m$
— create a new environment ENV' in which for each i
 formal_parameter$_i$ is assigned the value arg_result$_i$
— evaluate the expressions in the definition in left to right order
 — EVAL def_expression$_1$ in ENV' $\rightarrow$ def_result$_1$ ...
 — EVAL def_expression$_m$ in ENV' $\rightarrow$ def_result$_m$
 — return def_result$_m$

Question: How is ENV' constructed?

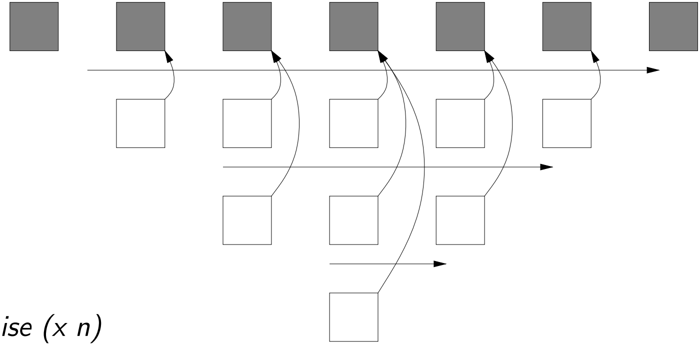
Environments can be described as sequences (linked lists) of tables where each table assigns symbols to values.

Each function is associated with (maintains a pointer to) the environment that was in place at the time the function was defined. In many cases, this is just the global environment, but there are exceptions as you will soon see.

ENV' is constructed by creating a new table in which the symbols corresponding to formal parameters are assigned the values of the arguments. This table then points to the environment associated with the function.

APPLY creates a new environment from the environment associated with the function being applied.

Environments created during recursive function invocation.



```
(defun raise (x n)
   (if (= n 0)
      1
      (* x (raise x (- n 1)))))

> (raise 3 2)
```

Local Variables

As noted above, you can change the value of symbols in environments

In the following function definition, the formal parameters X and Y are treated as variables whose values are determined locally with respect to the function in which the formal parameters are introduced.

```
(defun sqrt-sum-squares (x y)
  (SETF x (* x x))
  (SETF y (* y y))
  (sqrt (+ x y)))
```

You can also introduce additional local variables using LET

LET is a special form meaning it handled specially by EVAL.

(let ((var$_1$ var_expression$_1$) ... (var$_n$ expression$_n$)) body_expression$_1$ ... body_expression$_m$)

```
(let ((x (+ 1 2)))
   (sqrt x))
```

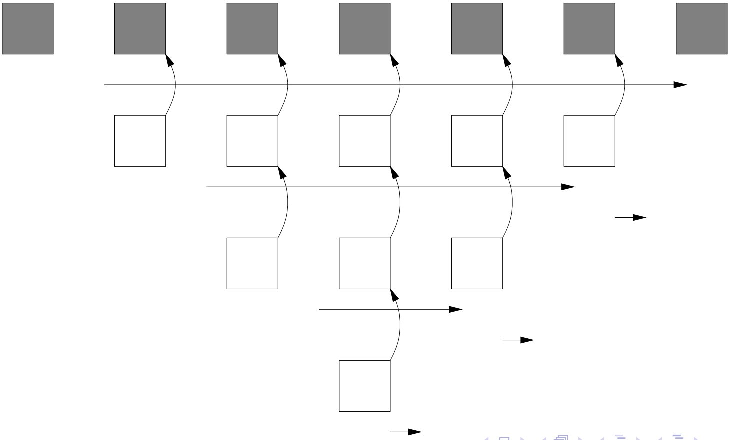
(let (var$_1$ ... var$_n$)
body_expression$_1$ ... body_expression$_m$)

```
(let (x)
   (SETF x (+ 1 2))
   (sqrt x))
(defun sqrt-sum-squares-let (x y)
   (let ((p (* x x)) (q (* y y)))
    (sqrt (+ p q))))
```

LET statements improve readability

LET statements are not strictly necessary but they can dramatically improve the readability of code.

Environments created using nested LET statements.



Example of nested LET statements

```
(let ((x 1))
   (let ((x 2))
     (let ((x 3))
       (princ x))
     (princ x))
   (princ x))
```

Note that the scope of local variables is determined lexically by the text and specifically by the nesting of expressions.

Functions with Local State

Environments persist over time. State information (local memory) for a specific function or set of functions.

```
(let ((sum 0))
  (defun give (x)
    (SETF sum (- sum x)))
  (defun take (x)
    (SETF sum (+ sum x)))
  (defun report ()
    sum))
```

```
> (take 5)          > (take 10)
> (report)
> (give 7)          > (report)
```

Functions as Arguments

FUNCTION tells EVAL to interpret its single argument as a function. FUNCTION does not evaluate its single argument.

(SETF foo (function oddp))

FUNCALL takes a FUNCTION and zero or more arguments for that function and applies the function to the arguments.

(funcall foo 1)

(funcall (function +) 1 2)

There is a convenient abbreviation for FUNCTION.

(funcall #'+ 1 2)

What is FUNCALL good for?

Generic functions add flexibility.

```
(defun generic-function (x y test)
  (if (funcall test x y)
    "do something positive"
   "do something negative" ))
```

```
(generic-function 1 2 #'<)
```

There are lots of generic functions built into Common Lisp.

```
(sort '(1 3 2 5 4) #'<)
```

Ignore the '(1 3 2 5 4) for the time being.

```
(sort '(1 3 2 5 4) #'>)
```

Lambda Functions (or what's in a name?)

LAMBDA specifies a function without giving it a name.

```
(SETF foo #'(lambda (x) (* x x)))
(funcall foo 3)
(funcall #'(lambda (x y) (+ x y)) 2 3)
```

LAMBDA functions are convenient for specifying arguments to GENERIC functions.

```
(sort '(1 3 2 5 4)
      #'(lambda (x y) (< (mod x 3) (mod y 3))))
```

Lambda can have memory

Lambda functions can have local memory just like named functions.

```
(defun spawn (x)
  #'(lambda (request)
     (cond ((= 1 request) (SETF x (+ x 1)))
           ((= 2 request) (SETF x (- x 1)))
           (t x))))
(SETF spawn1 (spawn 10) spawn2 (spawn 0))
(funcall spawn1 1)
(funcall spawn1 1)
(funcall spawn2 2)
(funcall spawn2 2)
(funcall spawn1 0)
(funcall spawn2 0)
```

Referring to Symbols Instead of their Values

QUOTE causes EVAL to suspend evaluation.

(quote foo)

Quote works for arbitrary expressions not just symbols.

(quote (foo bar))

There is a convenient abbreviation for QUOTE.

'foo

'(foo bar)

Building Lists and Referring to List Elements

Build a list with LIST.

(SETF sym (list 1 2 3 4))

Refer to its components with FIRST, SECOND, REST, etc.

(first sym)

(second sym)

(rest sym)

LIST, FIRST, REST, etc provide a convenient abstraction for pointers. In fact, you don't have to know much at all about pointers to do list manipulation in Lisp.

Put together a list with pieces of other lists

(SETF new (list 7 sym))

What if you want a list that looks just like SYM but has 7 for its first element and you want to use the REST of SYM?

Use CONS.

(cons 7 (rest sym))

If you want a list L such that (FIRST L) = X and (REST L) = Y then (CONS X Y) does the trick.

(cons 7 ())

Example

Here is a simple (but not particularly useful) recursive function.

```
(defun sanitize-list (x)
   (if (null x)
      x
      (cons 'element (sanitize-list (rest x)))))
```

```
(sanitize-list '(1 2 3))
```

(CONS 1 2) is perfectly legal, but it isn't a list. For the time being assume that we only use CONS to construct lists.

What about nested lists?

Nested lists correspond to trees

```
   (1 (2 3 (4)) (5 6 7))
    /      |         \
   /       |          \
  /        |           \
 /         |            \
1      (2 3 (4))      (5 6 7)
       /  |   \       /  |  \
      /   |    \     /   |   \
     2    3    (4)  5    6    7
                 \
                  4
```

Each nonterminal node in the tree is a list.

The children of a node corresponding to a list are the elements of the list.

How would you SANITIZE a tree?

```
(defun sanitize-tree (x)
   (cond ((null x) x)
           ;; No more children.
           ((not (listp x)) 'element)
           ;; Terminal node.
           (t (cons (sanitize-tree (first x))
                    ;; Break the problem down into two subproblems.
                    (sanitize-tree (rest x))))))

(sanitize-tree '(1 2 (3 4) ((3))))
```

EQUAL determines structural equality

(equal (list 1 2) (cons 1 (cons 2 nil)))

Is there another kind of equality?

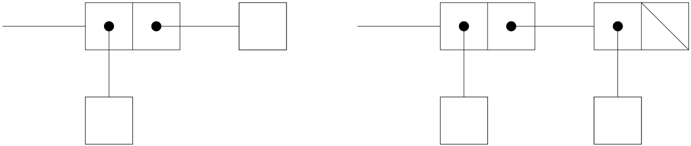
Given *(SETF X (LIST 1 2))* and *(SETF Y (LIST 1 2))* what is the difference between X and Y?

What's the different between *(list 1 2)* and *'(1 2)* ?

It could be that you don't need to know!

List Structures in Memory

CONS creates dotted pairs also called cons cells.



(SETF pair (cons 1 2))

CONSP checks for dotted pairs.

(consp pair)

CAR and CDR are the archaic names for FIRST and REST

(car pair)
(cdr pair)

EQ compares memory locations

EQ checks to see if two pointers (memory locations) are equivalent.

(eq (list 1 2) (list 1 2))
(SETF point pair)
(eq point pair)

Integers point to unique locations in memory.

(eq 1 1)
(SETF one 1)
(eq 1 one)

Floating numbers are not uniquely represented.

(eq 1.0 1.0)

Destructive Modification of List Structures

SETF allows us to modify structures in memory.

The first argument to SETF must reference a location in memory.

Change the first element of the pair from 1 to 3.

(setf (first pair) 3)

Create a nested list structure.

(SETF nest (list 1 2 (list 3 4) 5))

Change the first element of the embedded list from 3 to 7.

(setf (first (third nest)) 7)

Why would we destructively modify memory?

To maintain consistency in data structures that share memory.

```
(SETF mom '(person (name Lynn) (status employed)))
(SETF dad '(person (name Fred) (status employed)))
(SETF relation (list 'married mom dad))
(setf (second (third dad)) 'retired)
dad
relation
```

Difference between list and quote can be critical

```
(defun mem1 ()
   (let ((x (list 1 2))) x))

(defun mem2 ()
   (let ((x '(1 2))) x))

(SETF xmem1 (mem1))

(setf (first xmem1) 3)

(SETF xmem2 (mem2))

(setf (first xmem2) 3)

(mem1)

(mem2)
```

Predicates and Builtin Lisp Processing Functions

```
(SETF x (list 1 2 3))

(SETF y (list 4 5))
```

LAST returns the last CONS cell in an expression.

```
(last x)                ;; (3)


(defun alt_last (x)
  (if (consp (rest x))
    (alt_last (rest x)) x))


(alt_last x)
```

APPEND two or more lists together

APPEND uses new CONS cells.

```
(append x y)

x

(defun alt_append (x y)
   (if (null x)
      y
      (cons (first x)
            (alt_append (rest x) y))))

(alt_append x y)
```

NCONC (destructive append)

NCONC is like APPEND except that it modifies structures in memory.

NCONC does not use any new cons cells.

```
(nconc x y)

x

(defun alt_nconc (x y)
  (setf (rest (last x)) y) x)

(alt_nconc x y)
```

MEMBER

If X is an element of Y then MEMBER returns the list corresponding to that sublist of Y starting with X, otherwise MEMBER returns NIL.

```
(member 1 '(2 3 1 4))

(defun alt_member (x y)
  (cond ((null y) nil)
        ((eq x (first y)) y)
        (t (alt_member x (rest y)))))

(alt_member 1 '(2 3 1 4))
```

Check if X is EQ to Y or any subpart of Y

```
(defun subexpressionp (x y)
  (cond ((eq x y) t)
        ((not (consp y)) nil)
        ((subexpressionp x (first y)) t)
        ((subexpressionp x (rest y)) t)
        (t nil)))
```

```
(SETF z (list 1 2 (list 3 4 (list 5)) 6))

(subexpressionp 5 z)                ;; T
```

This is a typical function.

An alternate definition of SUBEXPRESSIONP

```
(defun alt_subexpressionp (x y)
   (or (eq x y)
       (and (consp y)
            (or (alt_subexpressionp x (first y))
                (alt_subexpressionp x (rest y))))))
```

```
(alt_subexpressionp 4 z)
```

Functions with Optional Arguments

```
(member '(3 4) '((1 2) (3 4) (5 6)))

(member '(3 4) '((1 2) (3 4) (5 6)) :test #'equal)

(member '(3 4) '((1 2) (3 4) (5 6))
        :test #'(lambda (x y)
                  (= (+ (first x) (second x))
                     (+ (first y) (second y)))))

(member '(3 4) '((1 2) (3 4) (5 6))
        :test #'(lambda (x y)
                  (= (apply #'+ x) (apply #'+ y))))
```

Data Abstraction

A data abstraction for input/output pairs.

Constructor for PAIRs.

```
(defun make-PAIR (input output) (list 'PAIR input output))
```

Type tester for pairs.

```
(defun is-PAIR (x) (and (listp x) (eq 'PAIR (first x))))
```

Access for PAIRs.

```
(defun PAIR-input (pair) (second pair))
(defun PAIR-output (pair) (third pair))
```

Modifying PAIRs.

```
(defun set-PAIR-input (pair new) (setf (second pair) new))
(defun set-PAIR-output (pair new) (setf (third pair) new))
```

Using the data abstraction

```
(SETF pairs (list (make-PAIR 3 8)
                  (make-PAIR 2 4)
                  (make-PAIR 3 1)
                  (make-PAIR 4 16)))

(defun monotonic_increasingp (pairs)
   (cond ((null pairs) t)
         ((> (PAIR-input (first pairs))
             (PAIR-output (first pairs))) nil)
         (t (monotonic_increasingp (rest pairs)))))

(monotonic_increasingp pairs)                 ;; NIL
```

Using the data abstraction (cont.)

```
(SETF increasing (list (make-PAIR 1 2)
                       (make-PAIR 2 3)
                       (make-PAIR 3 4)
                       (make-PAIR 4 5)))

(monotonic_increasingp increasing)          ;; T
```

Using the data abstraction (cont.)

```
(defun funapply (input fun)
    (cond ((null fun) nil)
          ((= input (PAIR-input (first fun)))
           (PAIR-output (first fun)))
          (t (funapply input (rest fun)))))

(funapply 2 increasing)                ;; 3

(defun alt_funapply (input fun)
    (let ((pairs (member input fun
                      :test #'(lambda (x y) (= x (PAIR-input y))))))
        (if pairs
            (PAIR-output (first pairs)))))

(alt_funapply 2 increasing)                ;; 3
```

Mapping Functions

MAPCAR

```
(mapcar #'first '((1 2) (3 4) (5 6)))

(mapcar #'(lambda (x y) (list x y))
        '(0 2 4 6 8)
        '(1 3 5 7 9))
```

MAPCAN (splice the results together using NCONC)

```
(mapcan #'(lambda (x) (if (oddp x) (list x) nil))
        '(1 2 3 4 5 6 7 8 9))
```

Mapping Functions (cont.)

EVERY

```
(every #'oddp '(1 3 5 7))
```

SOME

```
(some #'evenp '(1 2 3))
```

APPLY

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

(apply #'+
       (mapcar #'(lambda (x) (if (oddp x) x 0))
               '(1 2 3 4 5 6 7)))
```

Alternative Forms of Iteration

DO for general iteration

The general form is

(do index_variable_specification
(end_test result_expression)
body)

where index_variable_specification is a list specs of the form
(step_variable initial_value step_value)

Alternative Forms of Iteration (cont.)

```
(do ((i 0 (+ i 1)) (nums nil))
    ((= i 10) nums)
    (SETF nums (cons (random 1.0) nums)))
```

We could have done everything in the variable specs.
Notice the body of the DO in this case is empty.

```
(do ((i 0 (+ i 1))
     (nums nil (cons (random 1.0) nums)))
    ((= i 10) nums))
```

Alternative Forms of Iteration (cont.)

DOLIST for iteration over lists

(dolist (x '(1 2 3 4))
(princ x))

DOTIMES for iterating i = 1 to n

(dotimes (i 10)
(princ i))

Which form of iteration should you use?

Which ever one you want, but you should practice using the less familiar methods. In particular, we expect you to be able to understand code written using *recursion* and *mapping functions*.

Tracing and Stepping Functions

```
(defun raise (x n)
   (if (= n 0)
      1
      (* x (raise x (- n 1)))))
```

Tell Lisp to TRACE the function RAISE.

```
(trace raise)
(raise 3 2)
```

Tracing and Stepping Functions (cont.)

Tell Lisp to stop tracing the function RAISE.

(untrace raise)
(raise 3 1)

STEP a function

Use :n to step through evaluation. :h lists options.

(step (raise 3 2))

Association Lists

An association list associates pairs of expressions.

((name Lynn) (age 29) (profession lawyer) (status employed))

ASSOC

```
(assoc 'b '((a 1) (b 2) (c 3)))

(SETF features '((name Lynn)
                 (profession lawyer)
                 (status employed)))

(assoc 'status features)
```

FIND is more general than ASSOC

```
(find 'b '(a b c))

(find 'b '((a 1) (b 2) (c 3))
      :test #'(lambda (x y) (eq x (car y))))

(SETF mom '(person (name Lynn) (status employed)))
(SETF dad '(person (name Fred) (status employed)))
(SETF parents (list mom dad))

(find 'Fred parents
      :test #'(lambda (x y)
                 (eq x (second (assoc 'name (rest y))))))
```

Writing your own READ-EVAL-PRINT Loop

```
(defun alt_read_eval_print ()
    (format t "~%my prompt > " )
    (format t "~%~A" (alt_eval (read)))
    (alt_read_eval_print))
```

```
let ((history ()) (max 3))
  (defun alt_eval (expression)
     (if (and (listp expression)
              (eq (first expression) 'h)
              (integerp (second expression)))
        (if (and (>= (second expression) 0)
                 (< (second expression) max))
           (eval (nth (second expression) history))
           "No such history expression!")
        (progn (SETF history (cons expression history))
               (if (> (length history) max)
                   (setf (rest (nthcdr (- max 1) history)) nil))
               (eval expression)))))
```

PROGN specifies a sequence of expressions as a block; PROGN returns the value of the last expresion in the sequence.

Search

```
(defun srch (nodes goal next insert)
  (let ((visited nil))
    ;; add the predecessor to each node in nodes
    (SETF nodes (mapcar #'(lambda(x) (list x nil)) nodes))
    (loop
      ;; if there are no more nodes to visit or visitedmax. # of nodes,
      ;; return NIL as failure signal and the number of visited nodes.
      (if (or (null nodes) (>= (length visited) *visited_max*))
          (return (list nil nil (length visited))))
      ;; if goal has been reached, return T as success signal,
      ;; the solution path, and the number of visited nodes.
      (if (funcall goal (first (first nodes)))
          (return (list t (return-path (first nodes) visited)
                          (+ 1 (length visited)))))
      ;; else, add first node to visited, put its children on the list & iterate
      (SETF visited (cons (first nodes) visited))
      (SETF nodes (funcall insert (funcall next (first (first nodes)))
                                  (rest nodes)
                                  visited))
    )))
```

Insertion

;; eql-vis checks if the first elements of the two pairs are equalp

```
(defun eql-vis (x y)
       (equalp (first x) (first y)))
```

; dfs insertion puts new chidren on the front of the list

```
(defun dfs (children nodes visited)
  (append (remove-if #'(lambda (x)
                          (or (member x visited :test #'eql-vis)
                              (member x nodes :test #'eql-vis)))
                     children)
          nodes))
```

next & goal

```
(defun children (graph)
    #'(lambda (node)
              (mapcar #'(lambda(x) (list x node))
                      (second (assoc node graph)))))

(defun find-node (goal-node)
    #'(lambda (node) (equalp node goal-node)))


(defun print-list (l)
    (dolist (elt l t) (format t " ~A~%" elt)))

(SETF graph1 '((a (b e g)) (b (c d f)) (c nil)
               (d (c f)) (e (b f)) (f nil)
               (g (h i)) (h (b d)) (i (b e h))))
```

Using SRCH function

```
(SETF *visited_max* 100)

(SETF *init-pos* 'a)
(SETF *final-pos* 'f)
(SETF result (srch (list *init-pos*) (find-node *final-pos*)
                   (children graph1) #'dfs))
(format t "~%Graph search - dfs: ~A~%" (first result))
(format t "~%Initial position: ~A~%" *init-pos*)
(format t "Final position: ~A~(format t "~%Visited ~A nodes ~%"
                                    (third result))
(format t "Path:~%")
(print-list (second result))
(format t "~%~%")
```

Results

```
CL-USER 66 > (load "srch.lsp")

; Loading text file srch.lsp
#P"/home/u2/zduric/cs580/srch.lsp"

CL-USER 67 > (load "proj.skel")

; Loading text file proj.skel
#P"/home/u2/zduric/cs580/proj.skel"
```

Results (cont.)

```
CL-USER 68 > (proj)

Graph search - dfs: T

Initial position: A
Final position: F

Visited 5 nodes
Path:
    A
    B
    F
```