y-or-n-p, yes-or-no-p PREDICATES

Format:
```
(y-or-n-p)
(y-or-n-p <string>)
(yes-or-no-p)
(yes-or-no-p <string>)
```

Required arguments: none

Optional arguments: 1
Optional argument must evaluate to a string

The user is prompted with the string text (if provided). y-or-n-p returns t if y or Y is entered or nil if n or N is entered, but reprompts the user if given any other response. yes-or-no-p behaves similarly but requires a yes or no respsonse (upper or lower case mixed is ok).

Examples:

```
>(y-or-n-p "Well? ")
Well?   (Y or N) ok
Well?   (Y or N) no
Well?   (Y or N) y
T

>(yes-or-no-p "Answer in full:")
Answer in full:  (Yes or No) y
Answer in full:  (Yes or No) n
Answer in full:  (Yes or No) nO
NIL
```

symbolp — PREDICATE

Format: `(symbolp <exp>)`

Required arguments: 1
`<exp>`: any lisp expression

Returns T if the argument evaluates to a symbol, nil otherwise.

Examples:

```
>(symbolp 'this)
T

>(symbolp "this")
NIL

>(symbolp 3)
NIL

>(symbolp '3)
NIL

>(symbolp '(this is a list))
NIL

>(symbolp 'pi)
T

>(symbolp pi)
NIL
```

```
>(setf (get 'picard 'rank) 'captain)
CAPTAIN

>(get 'picard 'rank)
CAPTAIN

>(defstruct starship crew captain)
STARSHIP

>(setf enterprise (make-starship))
#S(STARSHIP CREW NIL CAPTAIN NIL)

>(setf (starship-crew enterprise)    (rest crew)
       (starship-captain enterprise) (second (first crew)))
PICARD

>enterprise
#S(STARSHIP CREW
      ((COMMANDER RIKER) (LIEUTENANT WORF) (ENSIGN CRUSHER))
      CAPTAIN PICARD)
```

setf ——— MACRO

Format: (setf <place1> <val1>
<place2> <val2>
.
.
<placeN> <valN>)

Required arguments: none

Optional arguments: any even number of arguments
<place>: either (i) the name of a variable, or (ii) an expression referring to part of a larger structure (e.g. a list, property list, structure, or array).
<val>: any LISP expression.

setf assigns the result of evaluating <val> to the location specified in the immediately preceding <place>. It returns the result of evaluating the last <val>. If no <place>-<val> pairs are specified, setf returns nil. setf is used, among other things, to assign values to variables, change parts of list structures, and to manage property lists and structures. Examples of all these uses are given in the chapters of this book. Other uses, too numerous to document here, can be found in Steele.

Examples: (see all chapters for further examples)

```
>(setf crew '(picard riker worf crusher))
(PICARD RIKER WORF CRUSHER)

>(setf (first crew)  (list 'captain (first crew))
       (second crew) (list 'commander (second crew))
       (third crew)  (list 'lieutenant (third crew))
       (fourth crew) (list 'ensign (fourth crew)))
(ENSIGN CRUSHER)

>crew
((CAPTAIN PICARD) (COMMANDER RIKER) (LIEUTENANT WORF)
 (ENSIGN CRUSHER))
```

second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth FUNCTIONS

Format: `(second <list>)`
`(third <list>)`
etc.

Required arguments: 1
The argument must evaluate to a list

These functions return the obvious element from the given list, or nil if the list is shorter than the selected element would require.

Examples:

```
>(second '(1 2 3 4))
2

>(fourth '(1 2 3 4))
4

>(ninth '(1 2 3 4))
NIL
```

reverse FUNCTION

Format: `(reverse <list>)`

Required arguments: 1

`<list>`: An expression which returns a list.

Reverse returns a list that contains all the elements of `<list>` in reversed order.

Examples:

```
>(reverse '(picard riker worf crusher))
(CRUSHER WORF RIKER PICARD)

>(reverse (reverse '(picard riker worf crusher)))
(PICARD RIKER WORF CRUSHER)

>(reverse '((this list) (of words)))
((OF WORDS) (THIS LIST))
```

rest FUNCTION

Format: `(rest <expr>)`

Required arguments: 1

`<expr>`: any LISP expression which returns a list

The argument expression must evaluate to a list. rest returns the list without its first element. If the list is empty, i.e. is NIL, rest returns NIL.

Examples:

```
> (rest '(1 2 3))
(2 3)

> (rest '((a (b (c)) d) e (f)))
(E (F))

> (rest ())
NIL

> (rest 'a)
Error: A is not of type LIST
```

read — FUNCTION

Format: `(read <instream>`
`<eof-error>`
`<eof-value>`
`<recursive>)`

Required arguments: none

Optional arguments: 4
`<instream>`: an expression which returns an input stream
`<eof-error>`: any LISP expression
`<eof-value>`: any LISP expression
`<recursive>`: any LISP expression

Called with no arguments, read waits for input from the standard input (usually the keyboard) and returns a LISP object. If `<instream>` is specified, input is taken from the stream rather than standard input. If `<eof-error>` is specified it controls what happens if an end of file is encountered in the middle of a "read." If `<eof-error>` is nil, no error results, and the result of `<eof-value>` is returned by "read." If `<eof-error>` is not NIL, then encountering the end of a file during a read will cause an error to occur. `<recursive>` controls the kind of error that is signalled when an end of file is encountered. If `<recursive>` is specified and is not NIL, then the end of file is reported to have occurred in the middle of reading in an object. If it is NIL, the the end of file is reported as occurring between objects.

Examples: See chapter 5.

or FUNCTION

Format: (or <exp1> <exp2> ...<expn>)

Required arguments: None

This function evaluates its arguments in order until it reaches a non-nil value, in which case it returns that value, or it returns nil. Evaluation of intermediate expressions may produce side-effects. In the special case where and is given no arguments, it always returns nil.

Examples:

```
>(or 3 (+ 4 5))
3

>(or nil (print 'hello))

HELLO
HELLO

>(or nil '(print hello) 3)
(PRINT HELLO)

>(or)
NIL
```

numberp — PREDICATE

Format: `(numberp <exp>)`

Required arguments: 1
`<exp>`: any LISP expression

The predicate numberp returns T if `<exp>` evaluates to a number (i.e. an object of type integer, ratio, float, or complex); numberp returns NIL otherwise.

Examples:

```
>(numberp 1/2)
T

>(numberp 1235439)
T

>(numberp (/ 5 1.23))
T

>(numberp #C(1.2 -0.9))
T

>(numberp '(+ 1 2 3))
NIL
```

null — PREDICATE

Format: `(null <exp>)`

Required arguments: 1
`<exp>`: any LISP expression

The predicate null returns T if `<expr>` evaluates to the empty list; NIL otherwise. null is just the same as not, but is preferred when its purpose of use is to test whether a list is empty.

Examples:

```
>(null '(picard riker))
NIL

>(null (rest '(picard)))
T
```

nthcdr FUNCTION

Format: `(nthcdr <index> <list>)`

Required arguments: 2

`<index>`: any expression which returns a positive integer (fixnum).

`<list>`: any expression which returns a list.

The function nth returns the `<list>` with the first n elements removed. `<index>` must be a non-negative integer. An index past the end of the list will cause nthcdr to return nil.

Examples:

```
>(setf ds9 '(Sisko Kira Dax Odo Bashir OBrien))
(SISKO KIRA DAX ODO BASHIR OBRIEN)

>(nthcdr 0 ds9)
(SISKO KIRA DAX ODO BASHIR OBRIEN)

>(nthcdr 1 ds9)
(KIRA DAX ODO BASHIR OBRIEN)

>(nthcdr 3 ds9)
(ODO BASHIR OBRIEN)

>(nthcdr 2345 ds9)
NIL
```

nth FUNCTION

Format: `(nth <index> <list>)`

Required arguments: 2

`<index>`: any expression which returns a positive integer (fixnum).

`<list>`: any expression which returns a list.

The function nth returns the indexed element of `<list>`. `<index>` must be a non-negative integer. 0 indicates the first element of `<list>`, 1 the second, etc. An index past the end of the list will cause nth to return nil.

Examples:

```
>(nth 0 '(picard riker work crusher))
PICARD

>(nth 2 '((captain picard)
          (commander riker)
          (lieutenant worf)
          (ensign crusher)))
(LIEUTENANT WORF)
```

not — PREDICATE

Format: `(not <exp>)`

Required arguments: 1
`<exp>`: any LISP expression.

See entry for null. Not is identical to null; its use is preferred for when `<exp>` is not a list.

```
>(member '(lieutenant worf)
         '((captain picard)
           (commander riker)
           (lieutenant worf)
           (ensign crusher))
         :test #'equal)
((LIEUTENANT WORF) (ENSIGN CRUSHER))

>(member 'picard '(picard riker worf crusher) :test-not #'eq)
(RIKER WORF CRUSHER)

>(member 'worf
         '((captain picard)
           (commander riker)
           (lieutenant worf)
           (ensign crusher))
         :key #'second)
((LIEUTENANT WORF) (ENSIGN CRUSHER))
```

member ——— FUNCTION

Format: `(member <item> <list>`
`:test <test>`
`:test-not <test-not>`
`:key <key>)`

Required arguments: 2
`<item>`: Any LISP expression
`<list>`: A expression which returns a list

Keyword arguments: 3
`<test>/<test-not>`: A function or lambda expression that can be applied to compare `<item>` with elements of `<list>`.
`<key>`: A function or lambda expression that can be applied to elements of `<list>`.

The elements of `<list>` are compared with the `<item>`. If `<test>` is not specified, eq is used; otherwise `<test>` is used. If `<item>` is found to match an element of `<list>`, a list containing all the elements from `<item>` to the end of `<list>` is returned. Otherwise NIL is returned. If `<test-not>` is specified, member returns a list beginning with the first UNmatched element of `<list>`. Specifying a `<key>` causes member to compare `<item>` with the result of applying `<key>` to each element of `<list>`, rather than to the element itself.

Examples:

```
>(member 'riker '(picard riker worf crusher))
(RIKER WORF CRUSHER)

>(member '(lieutenant worf)
         '((captain picard)
           (commander riker)
           (lieutenant worf)
           (ensign crusher)))
NIL
```

max, min — FUNCTIONS

Format: `(max <num1> ...<numN>)`
`(min <num1> ...<numN>)`

Required arguments: 1
`<num1>` ... `<numN>` must all evaluate to numbers

Returns the numerical maximum (minimum) of the arguments given.

Examples:

```
>(max 1 4 3 15 (* 9 2))
18

>(min 3 4 (- 7 19) 5 6.0)
-12

>(max 3)
3

>(min 4)
4
```

mapcar ——— FUNCTION

Format: `(mapcar <func> <lis1> . . . <lisN>)`

Required arguments: 2
First argument names a function (usually quoted).
Subsequent arguments must evaluate to lists.

Mapcar applies the named function successively to the first, second, third, etc. elements of the subsequent arguments and returns a list of the results, up to the length of the shortest list provided.

Examples:

```
>(mapcar '+ '(1 2 3))
(1 2 3)

>(mapcar '+ '(1 2 3) '(4 5 6))
(5 7 9)

>(mapcar '+ '(1 2 3) '(4 5 6) '(7 8 9))
(12 15 18)

>(mapcar '+ '(1 2) '(3 4 5))
(4 6)

>(mapcar '< '(1 2 3) '(4 5 0))
(T T NIL)

>(mapcar '< '(1 2 3) '(4 5))
(T T)
```

listp — PREDICATE

Format: `(listp <exp>)`

Required arguments: 1

`<exp>`: any LISP expression.

Returns T if `<exp>` is of the data type list; NIL otherwise.

Examples:

```
>(listp '(a s d f))
T

>(listp 3)
NIL

>(listp (cons '1 '(2 3 4)))
T
```

list — FUNCTION

Format: `(list <exp1> <exp2>...<expN>)`

Required arguments: none

Optional arguments: arbitrary

`<exp1>...<expN>`: any sequence of zero or more LISP expressions.

All the `<arg>`'s are evaluated and the resulting values are returned as elements of a list.

Examples:

```
>(list 'picard 'riker 'worf 'crusher)
(PICARD RIKER WORF CRUSHER)

>(list 'picard '(riker worf crusher))
(PICARD (RIKER WORF CRUSHER))

>(list 1 (+ 1 1) (+ 1 1 1) (+ 1 1 1 1))
(1 2 3 4)
```

let — SPECIAL FORM

Format:
```
(let ((<var1> <init1>)
      (<var2> <init2>)
      .
      .
      (<varN> <initN>))
   <body>)
```

Required arguments: 1

((<var> <init>)...): a list of zero or more lists having the form (<var>) or (<var> <init>). <var> must be a symbol appropriate as the name of a variable. <init> may be any LISP expression.

Optional arguments: arbitrary

<body>: any sequence of zero or more LISP expressions.

The <var>'s are established as local variables for the expressions in <body>. If a <var> is not accompanied by an <init> expression, it is initially bound to NIL. Otherwise, <init> is evaluated and assigned as the value of <var>. let evaluates all the expressions in <body> and returns the value of the last, or NIL if there are none.

Examples:

```
>(let ((a)
       (b))
   (and (not a) (not b)))
T

>(let ((a 3)
       (b 4))
   (setf a (+ a b))
   (setf b (+ a b))
   (+ a b))
18
```

length — FUNCTION

Format: `(length <exp>)`

Required arguments: 1

`<exp>` must evaluate to a sequence (e.g. list, array, vector, string)

Returns the length of the given sequence.

Examples:

```
>(length '(1 2 3 4 5))
5

>(length "1 2 3 4 5")
9

>(length (make-array 3))
3

>(length nil)
0
```

if — SPECIAL FORM

Format: `(if <test>`
`<then>`
`<else>)`

Required arguments: 2
`<test>`: any LISP expression
`<then>`: any LISP expression

Optional arguments: 1
`<else>`: any LISP expression

The `<test>` expression is evaluated; if it returns a value other than NIL, if returns the result of evaluating the `<then>` expression. If `<test>` evaluates to NIL, and no `<else>` expression is provided, NIL is returned. Otherwise the result of evaluating `<else>`is returned.

Examples:

```
>(if (> 4 3) 4 3)
4

>(if (< 4 3)
     (- 4 3)
     (- 3 4))
1

>(if (= 4 3) t)
NIL
```

first — FUNCTION

Format: `(first <exp>)`

Required arguments: 1

`<exp>`: any LISP expression which returns a list

The argument expression must evaluate to a list; first returns the first element of this list. If the list is empty, i.e. is nil, first returns nil.

Examples:

```
> (first '(1 2 3))
1

> (first '((a (b (c)) d) e (f)))
(A (B (C)) D)

> (first ())
NIL

> (first 'a)
Error: A is not of type LIST
```

evenp, oddp — PREDICATES

Format: `(evenp <num>)`
`(oddp <num>)`

Required arguments: 1
`<num>`: any LISP expression that evaluates to an integer.

Returns T if the value of `<num>` is even (odd); NIL otherwise.

Examples:

```
>(oddp 3)
T

>(oddp 68)
NIL

>(evenp 4.0)
Error: 4.0 is not of type INTEGER.
```

eval FUNCTION

Format: `(eval <exp>)`

Required arguments: 1
`<exp>`: any LISP expression.

eval provides direct access to the LISP expression evaluator. It causes the evaluation of whatever `<exp>` returns. Thus, if `<exp>` is an evaluable LISP expression and itself returns and evaluable LISP expression, then eval returns the value of the second evaluation.

Examples:

```
>(eval '(+ 1 2))
3

>(eval (cons '+ '(1 2)))
3

>(eval 3)
3

>(eval (+ 3 4))
7

>(eval (cons 'a '(s d f)))
Error: The function A is undefined.
```

equal — PREDICATE

Format: `(equal <exp1> <exp2>)`

Required arguments: 2
`<exp1>`: any LISP expression
`<exp2>`: any LISP expression

Both argument expressions are evaluated. If the values returned are copies of one another (or even are physically the same by occupying the same memory), equal returns T. In contrast, for two lists to be eql they must represent the same object in memory.

Examples:

```
> (equal 'hey 'hello)
NIL
> (equal -81 -81)
T
> (equal '(1 (2 3)) '(1 (2 3)))
T

> (setq a '(1 2 3))
(1 2 3)
> (setq b '(1 2 3))
(1 2 3)
> (equal a b)
T

> (setq c a)
(1 2 3)
> (equal a c)
T
```

eql — PREDICATE

Format: `(eql <exp1> <exp2>)`

Required arguments: 2
`<exp1>`: any LISP expression
`<exp2>`: any LISP expression

Both argument expressions are evaluated. If they both return atoms, eql returns T if they are the same. If they return lists, eql returns T only if the lists are represented by the same object in memory. In contrast, two values can be "equal" if they are copies of one another (perhaps existing in different memory locations).

Examples:

```
> (eql 'hello 'hello)
T
> (eql -19 -19)
T
> (eql 2 3)
NIL
> (eql '(1 2 3) '(1 2 3))
NIL

> (setq a '(1 2 3))
(1 2 3)
> (setq b '(1 2 3))
(1 2 3)
> (eql a b)
NIL

> (setq c a)
(1 2 3)
> (eql a c)
T
> (eql b c)
NIL
```

documentation FUNCTION

Format: `(documentation <func> <type>)`

Required arguments: 2

`<func>` must evaluate to the name of a function
`<type>` must be function, variable, type, structure, or setf (see below)

This function is useful from the command line for a quick reminder of the format and usage of certain LISP primitives. Both arguments are usually quoted symbols. See examples below.

Examples:

```
>(documentation 'documentation 'function)
"
Args: (symbol doc-type)
Returns the doc-string of DOC-TYPE for SYMBOL; NIL if none exists.
Possible doc-types are:
        FUNCTION  (special forms, macros, and functions)
        VARIABLE  (dynamic variables, including constants)
        TYPE      (types defined by DEFTYPE)
        STRUCTURE (structures defined by DEFSTRUCT)
        SETF      (SETF methods defined by DEFSETF, DEFINE-SETF-METHOD, and
                   DEFINE-MODIFY-MACRO)
All built-in special forms, macros, functions, and variables have their
doc-strings."

>(documentation 'cdr 'function)
"
Args: (list)
Returns the cdr of LIST.  Returns NIL if LIST is NIL."

>(documentation 'pi 'variable)
"
The floating-point number that is appropriately equal to the ratio of the
circumference of the circle to the diameter."
```

Examples:

```
> (do ((lst
        '(3 5 2 9 6)
        (rest lst))        ; adds the elements of lst
       (sum 0))
      ((null lst) sum)
      (setq sum (+ sum (first lst))))
25

> (do ((lst
        '(a (b (c d)) e (f))
        (rest lst))
       (len 0 (+ 1 len)))  ; determines length of lst
      ((null lst) len))    ; "do," here, has no body
4

> (defun my-exp (m n)      ; raise m to power of n
     (do ((result 1)
          (exp    n))
         ((= exp 0) result)
         (setq result (* result m))
         (setq exp    (- exp 1))))
MY-EXP

> (my-exp 5 3)
125

> (defun my-exp2 (m n)     ; simpler version of my-exp
     (do ((result 1 (* result m))
          (exp    n (- exp 1)))
         ((= exp 0) result)))
```

do — SPECIAL FORM

Format:
```
(do ((<var1> <init1> <update1>)
     (<var2> <init2> <update2>)
     .
     .
     (<varN> <initN> <updateN>))
    (<test> <body1>)
    <body2>)
```

Required arguments: 2

`((<var> <init> <update>)...)`: a list of zero or more variable clauses; `<var>` is a symbol appropriate as a variable name; `<init>`, which is optional, is any LISP expression; `<update>`, which is optional, is any LISP expression.

`(<test> <body1>)`: `<test>` is any LISP expression; `<body1>` is a sequence of zero or more LISP expressions.

Optional arguments: arbitrary

`<body2>`: a sequence of zero or more LISP expressions

The special form do allows the programmer to specify iteration. The first part of do is a list of variables; if `<init>` is provided, the `<var>`'s are initialised to the result of evaluating the corresponding `<init>`. If no `<init>` is provided, `<var>` is initialised to NIL. The optional `<update>` expression may specify how the variable is to be modified after every iteration. After every iteration, `<var>` is set to result of evaluating `<update>`. Initialisation and updating for all variables is performed in parallel; thus a `<var>` in one clause may not be used in the `<init>` or `<update>` of another clause.

The second part of "do," the `<test>`, checks for termination. The `<test>` is evaluated before each pass; if it returns a non-NIL value, the sequence of expressions in `<body1>` are evaluated one by one; do returns the value of the last expression in `<body1>`. If `<body1>` contains no expressions, do returns NIL.

The third part of the do is the body of the iteration, `<body2>`. At each pass, the sequence of expressions in `<body2>` are evaluated one by one. If `<body2>` contains an expression of the form `(return <expr>)`, where `<expr>` is any LISP expression, do terminates immediately and returns the result of evaluating `<expr>`.

```
T

> (equal-length '() (rest '(a b)))
NIL

> (defun side-effect (x y)
     (setq x (* x x x))
     (+ x y))
SIDE-EFFECT

> (side-effect 2 3)
11

> (side-effect -4 14)
50
```

defun MACRO

Format: `(defun <name> (<par1> <par2> ...<parN>)`
`<body>)`

Required arguments: 2

`<name>`: a symbol which is appropriate as the name of funtion;

`(<par1> <par2> ...)`: a list of zero or more symbols which are appropriate as parameter names

Optional arguments: 1

`<body>`: a sequence of zero or more LISP expressions

The arguments to defun are not evaluated—they are used to establish a procedure definition. The first argument is a symbol which specifies the name of the function. This name can later be used to execute the `<body>` of the function. The parameter-list follows the name. This list specifies the number and order of arguments in a function call. Each `<par>` is symbol which may appear in the `<body>`. The value of each parameter, `<par>`, is determined by the value of corresponding argument in the function call. defun returns the name of the function.

Examples:

```
> (defun square (x)
     (* x x))
SQUARE

> (square 4)
16

> (square (+ 7 -2))
25

> (defun equal-length (lst1 lst2)
     (cond ((= (length lst1) (length lst2)) t)
           (t nil)))
EQUAL-LENGTH

> (equal-length '(a b c) '((d e) f g))
```

CONS ——— FUNTION

Format: `(cons <exp1> <exp2>)`

Required arguments: 2
`<exp1>`: any LISP expression
`<exp2>`: any LISP expression

Both argument expressions are evaluated. In common usage, the second expression usually returns a list. In this case, cons creates a new copy of the list returned by `<expr2>`, and makes the value returned by `<expr1>` the new first element in this list. However, if `<expr2>` returns an atom, cons returns the *dotted pair* of the values of the two expressions.

Examples:

```
> (cons 'a '(1 2 3))
(A 1 2 3)

> (cons '(a) '(1 2 3))
((A) 1 2 3)

> (setq a 3)
3
> (cons a (list i j k))
(3 I J K)

> (cons 'a a)
(A . 3)                          ;;a dotted pair

> (cons '(1 2) 'q)
((1 2) . Q)                      ;;another dotted pair
```

```
         (setq b (* b b)))
        ((equal action 'square-a)
         (setq a (* a a)))
        ((equal action 'square-b)
         (setq b (* b b))))
16

> (cond ((= a 5) 'found)
        ((= b 5) 'found))
NIL

> (cond ((+ a 1)))    ; strange use of cond
10                    ; before this, a was 9 due to previous
                      ; cond clauses
```

cond —— MACRO

Format: `(cond (<test1> <body1>)`
`(<test2> <body2>)`
`.`
`.`
`(<testN> <bodyN>))`

Required arguments: none

Optional arguments: zero or more `(<test> <body>)` clauses
Here, `<test>` is any LISP expression. The `<body>` is a sequence of zero or more LISP expressions.

Each `<test>` expression is evaluated in turn, until one is found whose value is non-NIL. For this clause, the sequence of `<body>` expressions are evaluated and the value of the last expression is returned as the result of cond. If no successful `<test>` expression is found, cond returns NIL. If a successful `<test>` expression is found, but its clause contains no `<body>` expressions, the result of the `<test>` expression is returned as the result of cond.

Examples:

```
> (setq a '3)
3
> (setq b '4)
4

> (cond ((= a b) 'equal)
        ((< a b) 'b-is-bigger)
        (t 'a-is-bigger))
B-IS-BIGGER

> (setq action 'square-both)
SQUARE-BOTH

> (cond ((equal action 'clear-both)
         (setq a 0)
         (setq b 0))
        ((equal action 'square-both)
         (setq a (* a a))
```

cdr — Function

See the entry for `rest`: `cdr` is the older name for `rest`. Historically its name is derived from Contents of Decrement Register. It's pronunciation rhymes with udder.

caar, cadr, cdar, cddr, etc. ——————FUNCTIONS

Format: `(c-r <list>)`

Required arguments: 1
Argument must evaluate to a list (or cons pair).

The function *cxyr* is a composition of the function *cxr* with *cyr*. So, for example, `(cadr foo)` is equivalent to `(car (cdr foo))`, etc. Up to 3 letters, a or d, may appear between the c and the r.

Examples:

```
>(setf ds9 '(Sisko Kira Dax Odo Bashir OBrien))
(SISKO KIRA DAX ODO BASHIR OBRIEN)

>(cadr ds9)
KIRA

>(cddr ds9)
(DAX ODO BASHIR OBRIEN)

>(cdddr ds9)
(ODO BASHIR OBRIEN)

>(caddr ds9)
DAX

>(cdar ds9)
Error: SISKO is not of type LIST.
```

See the entry for `first`: `car` is the older name for `first`. Historically its name is derived from Contents of Address Register.

butlast FUNCTION

Format: `(butlast <list>)`
`(butlast <list> <int>)`

Required arguments: 1
First argument must evaluate to a list
Optional arguments: 1
`<int>` must evaluate to an integer

If butlast is used with a single argument then it is equivalent to `(reverse (rest (reverse <list>)))`. I.e. it will return the list argument with the last element removed.

If butlast is given an integer second argument, it is equivalent to `(reverse (nthcdr <num> (reverse <list>)))`. I.e. it will remove the number of elements specified from the end of the list.

Examples:

```
>(butlast '(a s d f))
(A S D)

>(butlast '(a s d f) 2)
(A S)

>(butlast '(a s d f) 0)
(A S D F)

>(reverse (nthcdr 2 (reverse '(a s d f))))
(A S)
```

atom — PREDICATE

Format: `(atom <exp>)`

Required arguments: 1
`<exp>` can be any LISP expression

The argument expression is evaluated. If the value returned by this evaluation represents a LISP atom, `atom` returns T, else it returns NIL. Symbols, numbers and strings are all considered LISP atoms.

Examples:

```
> (atom 'hello)
T

> (atom "hello")
T

> (atom 4.6434)
T

> (atom '(hello "hello" 4.6434))
NIL
```

apply ——— FUNCTION

Format: `(apply <func> <list>)`
`(apply <func> <exp1> ... <expn> <list>)`

Required arguments: 2
`<func>` must name a function or predicate (usually quoted); the last argument must be a list.

Optional arguments: arbitrary
Intermediate argument expressions must evaluate to items of the correct type to which `<func>` can be applied

With just two arguments, the function is applied to the elements in the list just as if the expression `(eval (cons <func> <list>))` were evaluated. Intermediate arguments are treated as additional arguments to the function.

Examples:

```
>(apply '+ '(1 2 3 4))
10

>(apply '+ 1 2 '(3 4))
10
```

append ——— FUNCTION

Format: `(append <list1> <list2>...<listN> <exp>)`

Required arguments: none

Optional arguments: zero or more LISP expressions
`<list>`: any LISP expression which returns a list; except the last argument may be *any* LISP expression.

Each of the arguments is evaluated; all except the last must return a list. If all the arguments evaluate to lists, append creates a new list which has as its elements the elements of all the argument lists. If the last argument is not a list, append returns a dotted pair object.

Examples:

```
> (append '(a b) '(c d))
(A B C D)

> (append '(1 (2 (3))) (i (j) k))
(1 (2 (3)) I (J) K)

> (setq tmp '(fee fi fo))
(FEE FI FO)

> (append tmp (list 'fum))
(FEE FI FO FUM)

> (append '(fo) 'fum)
(FO . FUM)                          ;; this is a dotted pair

> (append 'a '(1 2))
Error: A is not of type LIST
```

and FUNCTION

Format: `(and <exp1> <exp2> ...<expN>)`

Required arguments: None

This function evaluates its arguments in order until it reaches a nil value, in which case it returns NIL, or it returns the value of `<expN>`. Evaluation of intermediate expressions may produce side-effects. In the special case where and is given no arguments, it always returns T.

Examples:

```
>(and 3 (+ 4 5))
9

>(and nil (print 'hello))
NIL

>(and t (print 'hello) 3)

HELLO
3

>(and)
T
```

<, >, <=, >= PREDICATES

Format: `(< <num1> <num2> ...)`
`(> <num1> <num2> ...)`
`(<= <num1> <num2> ...)`
`(>= <num1> <num2> ...)`

Required arguments: 1

`<num1> <num2> ...` must evaluate to numbers.

Returns T if the sequence of arguments is ordered (<: ascending, >: descending) or partially ordered (<=: less than or equal, >=: greater than or equal). Otherwise returns nil.

Examples:

```
>(< 2 67)
T

>(< 67 2)
NIL

>(> 67 2)
T

>(< 3 6 9)
T

>(< 4)
T

>(< 2 2 6)
NIL

>(<= 2 2 6)
T

>(>= 2 2 6)
NIL

>(< 6/2 3.0)
NIL
```

= PREDICATE

Format: `(= <num1> <num2> ...)`

Required arguments: 1
`<num1>` must evaluate to a number
Optional arguments: arbitrary
`<num2>` ... must all evaluate to numbers

= returns T if all the arguments are numerically equal to each other; it returns NIL otherwise.

Examples:

```
>(= 2)
T

>(= 2 3)
NIL

>(= 2 3 4)
NIL

>(= 2 2 2 2 )
T

>(= 2 2 3 2)
NIL

>(= 2 'a)
Error: A is not of type NUMBER.
```

1+ and 1- FUNCTIONS

Format: `(1+ <num>)`
`(1- <num>)`

Required arguments: 1
`<num>`: any LISP expression that evaluates to a number.

Simple ways to add or subtract one. Note that `(1- x)` is equal to *x-1* not *1-x*.

Examples:

```
>(1+ 3)
4

>(1- pi)
2.1415926535897931

>(1+ (1- 3))
3
```

— **FUNCTION**

Format: `(- <exp1> <exp2> ...)`

Required arguments: 1
Argument expressions must evaluate to numbers.

Argument expressions are evaluated and their cumulative difference is computed.

Examples:

```
>(- 1)
-1

>(- 1 2)
-1

>(- 1 2 3)
-4

>(- 1 2 3 4)
-8
```

+ FUNCTION

Format: `(+ <exp1> <exp2> ...)`

Required arguments: None.
Argument expressions must evaluate to numbers.

Argument expressions are evaluated and their sum is computed. See below for special cases of zero or one arguments.

Examples:

```
>(+ 4 5 6)
15

>(+ 123.5 12/3)
127.5

>(+ (* 2 3) 8)
14

>(+ 'a 1)
Error: A is not of type NUMBER.
```

Special Cases:

```
>(+ 3)
3

>(+)
0
```

* ——— FUNCTION

Format: `(* <num1> <num2> ...)`

Required arguments: None.
Argument expressions must evaluate to numbers.

Argument expressions are evaluated and their product is computed. See below for special cases of zero or one arguments.

Examples:

```
>(* 4 5 6)
120

>(* 123.5 12/3)
494.0

>(* (* 2 3) 8)
48

>(* 'a 1)
Error: A is not of type NUMBER.
```

Special cases:

```
>(* 3)
3

>(*)
1
```

Appendix A

Selected LISP primitives

Entries are described as FUNCTIONS, PREDICATES, MACROS, or SPECIAL FORMS.

FUNCTIONS evaluate all their arguments and return a value.

PREDICATES are functions that always return either t or nil.

MACROS and SPECIAL FORMS do not always evaluate all their arguments.

Currently approximately 40 LISP primitives are documented here. Eventually just over 100 primitives will be documented.

We adopt the convention of using angle brackets < > to indicate where appropriate text should be inserted when using these primitives; the examples should make this clear.

In many of the descriptions below, the notion of list is used when either a list or a LISP data type known as the dotted pair would be acceptable. Beginners generally need not use dotted pairs, so they are ignored for the sake of simplicity here.

Chapter 7

Functions, Lambda Expressions, and Macros

This chapter is not yet written, but here are the major sections.

7.1 Eval

7.2 Lambda Expressions

7.3 Funcall

7.4 Apply

7.5 Mapcar

7.6 Backquote and Commas

7.7 Defmacro

```
Error: A comma has appeared out of a backquote.
Error signalled by READ.
Broken at READ.  Type :H for Help.
```

If punctuation is likely to appear in input, then it is necessary to use read-char, which reads one character at a time. It is then possible to inspect each character and process it appropriately if it is problematic. Exactly how to do this will not be covered here as it makes a nice exercise to develop your understanding of LISP input processing.

With-open-file will produce an error if the file foo does not already exist, unless its behavior is controlled using the :if-does-not-exist :create or :if-does-not-exist nil options. The first of these options creates a file with the specified name, the second causes the body of the with-open-file to be ignored, and the value nil is returned.

Format can be used with the name of a stream as its first argument.

```
>(with-open-file (outfile "foo" :direction :output)
   (format outfile "~%This is text.~%"))
NIL
```

This overwrites foo so that it contains three lines – a blank line, a line with

```
This is text.
```

and another blank line.

Print, prin1, princ, terpri, format, read and read-line all can be used with stream specifications to do file input and output.

6.5 Converting Strings to Lists

Sometimes it is useful to convert text strings to lists. For example, to get substantial input, read-line is most convenient, but it returns a string. If the objective is to parse the input, it is much more convenient to have a list of words than a string.

Here is an example of code to convert a string to a list:

```
(defun string-to-list (str)
   (do* ((stringstream (make-string-input-stream str))
         (result nil (cons next result))
         (next (read stringstream nil 'eos)
               (read stringstream nil 'eos)))
        ((equal next 'eos) (reverse result))))

>(string-to-list "this is a string of text")
(THIS IS A STRING OF TEXT)
```

Because of its reliance on read, this function will not work with certain kinds of punctuation. For example:

```
>(string-to-list "Commas cause problems, see")
```

As soon as the variables are initialized, the test (equal next 'eof) is performed. If it is true then (reverse result) is evaluated and returned. If not, then all expressions in the body of the do are evaluated (in this case there are none).

After the body has been evaluated, all the variables are updated with the expressions indicated. In other words, result gets next cons-ed into it, and next gets updated with a new read from instream.

The second and third argments to read control its behavior when it reaches the end of the file. The second argument, nil in this case, indicates that reaching the end of the file should not generate an error. The third argument, in this case the symbol eof, indicates what should be returned instead of an error. This enables the do loop to determine when the end of file has been reached and return the appropriate result. Notice that choosing eof as the result to return would cause the do loop to stop if it reads the symbol eof in the middle of a file. This may or may not be desirable behavior.

Writing to a file is very similar to reading.

```
>(with-open-file (outfile "foo" :direction :output)
    (prin1 '(here is an example) outfile))
(HERE IS AN EXAMPLE)
```

And the file now contains one line:

```
(HERE IS AN EXAMPLE)
```

Note that it is necessary to specify the :direction as :output. With-open-file assumes that a file is being opened for input by default, so this must be explicity overridden when doing file output. Files may also be opened as :direction :io, which allows input and output.

Notice also that this example destroyed the previous contents of foo. This behavior can be controlled with the :if-exists option. For example:

```
>(with-open-file (outfile "foo" :direction :output
                                :if-exists :append)
     (print '(here is a second list) outfile))
(HERE IS A SECOND LIST)
```

This will add a second line to the file foo, so that it contains

```
(HERE IS AN EXAMPLE)
(HERE IS A SECOND LIST)
```

In order to receive input from or send output to a file, the file must be attached to an appropriate stream. The easiest way to handle this is with the macro with-open-file. This is the general form for with-open-file:

```
(with-open-file (<stream> <filename>) <body>)
```

Unless specified otherwise, with-open-file assumes that the stream is an input stream, in other words, that you will be reading data from the named file.

Suppose you have a file called "foo", which looks like this:

```
this is an
example of a file
(that has things)
we
might want (to read in)
4
5
67
```

Here's how to get the first LISP expression from foo.

```
>(with-open-file (infile "foo") (read infile))
THIS
```

A slightly more complicated operation is to make a list of all the expressions in foo. The following will work:

```
(with-open-file (infile "foo")
   (do ((result nil (cons next result)))
        (next (read infile nil 'eof) (read infile nil 'eof))
       ((equal next 'eof) (reverse result)))))
```

(See the appendix entry for 'do' if you do not know how it works.)

If you evaluate this code it will return the list:

```
(THIS IS AN EXAMPLE OF A FILE (THAT HAS THINGS) WE MIGHT WANT
      (TO READ IN) 4 5 67)
```

At the beginning of the do, two variables are specified – result and next. The initial value of result is nil. The intial value of next is the result of evaluating (read infile nil 'eof). The effects of the last two arguments in this read are explained below.

```
(defun f-to-c ()
  (format t "~%Please enter Fahrenheit temperature: ")
  (let* ((ftemp (read))
         (ctemp (* (- ftemp 32) 5/9)))
    (format t
            "~%~s degrees Fahrenheit is ~s degrees Celsius~%"
              ftemp
              (float ctemp))           ;; print floated value
    ctemp))                            ;; return ratio value

>(f-to-c)

Please enter Fahrenheit temperature: 56   ;; user enters 56

56 degrees Fahrenheit is 13.333333333333333 degrees Celsius

40/3
```

Read-line always returns a string. Read-line will take in everything until the return key is pressed and return a string containing all the characters typed. Read-char reads and returns a single character.

6.4 Input and Output to Files

All input and output in Common LISP is handled through a special type of object called a stream. When a stream is not specified, Common LISP's default behavior is to send output and receive input from a stream bound to the constant *terminal-io* corresponding to the computer's keyboard and monitor. Evaluating *terminal-io* shows a printed representation of this stream:

```
>*terminal-io*
#<two-way stream 0016c7a8>
```

The designation "two-way" here specifies that it is a stream that is capable of handling both input and output.

In all the examples so far, printing and reading has been done without specifying a stream, hence the effects have been to interact with the keyboard and screen. However, print, format, read and the other functions mentioned in this chapter allow optional specification of a different stream for input or output.

(Float converts a number from integer or ratio to a floating point number, i.e. one with a decimal point.)

```
>(f-to-c 82)

82 degrees Fahrenheit is
27.777777777777777 degrees Celsius

250/9
```

Format simply evaluates the optional arguments in order they appear to determine the values for each ~s. There must be enough optional arguments following the control string to provide values for all the occurrences of ~s in the control string, otherwise an error will occur. If there are too many optional arguments, format evaluates them all (so side effects may occur) but no error is signalled.

Format will also preserve newlines entered directly in the control string. For example, f-to-c would behave just the same if defined as follows:

```
(defun f-to-c (ftemp)
  (let ((ctemp (* (- ftemp 32) 5/9)))
    (format t "
~s degrees Fahrenheit is
~s degrees Celsius
"
              ftemp
              (float ctemp))           ;; print floated value
    ctemp))                            ;; return ratio value
```

In addition to these two control sequences it is useful to know about ~T to produce tab spacing and ~~ to print a tilde. Some other control sequences are documented in the appendix entry for format.

6.3 Reading

Input from the keyboard is controlled using read, read-line, and read-char.

Read expects to receive a well-formed LISP expression, i.e. an atom, list or string. It will not return a value until a complete expression has been entered – in other words all opening parentheses or quotes must be matched.

Here is f-to-c using read:

The full use of a destination will be introduced further below; for most basic uses, the destination should be specified as t or nil. The control string is a string containing characters to be printed, as well as control sequences. Every control sequence begins with a tilde: ~. The control sequences may require extra arguments to be evaluated, which must be provided as optional arguments to format.

With t as the specified destination, and no control sequences in the control-string, format outputs the string in a manner similar to princ, and returns nil.

```
>(format t "this")
this
NIL
```

With nil as destination and no control sequences, format simply returns the string:

```
>(format nil "this")
"this"
```

Inserting ~% in the control string causes a newline to be output:

```
>(format t "~%This shows ~%printing with ~%newlines.~%")

This shows
printing with
newlines.

NIL
```

~s indicates that an argument is to be evaluated and the result inserted at that point. Each ~s in the control string must match up to an optional argument appearing after the control string.

Here is an example of a function that uses this capability:

```
(defun f-to-c (ftemp)
  (let ((ctemp (* (- ftemp 32) 5/9)))
    (format t
            "~%~s degrees Fahrenheit is ~%~s degrees Celsius~%"
            ftemp                    ;; first ~s
            (float ctemp))           ;; second ~s
    ctemp))                          ;; return ratio value
```

But you expected to get (S A) since you expected to get the list of the second of x with the first of y.

You would like to know why foo returned this. One way to help you see what is going on is to insert print statements into the definition. If for example you want to see what is being listed if the test is true, you can modify the definition like this:

```
(defun foo (x y)
  (if (eq x y) (list (print (second x)) (print (first y))))
    (list y (rest x)))
```

Now when you try the same call, the following happens:

```
>(foo '(a s d f) '(a s d f))
((A S D F) (S D F))
```

But this is just the same as before! Neither print function got called, so you deduce that the test must have returned nil. This should help you to recall the difference between eq and equal. Now you modify the definition of foo again:

```
(defun foo (x y)
  (if (equal x y) (list (print (second x)) (print (first y))))
    (list y (rest x)))
```

And try it out:

```
>(foo '(a s d f) '(a s d f))

S
A
(S A)
```

This time you see the effects of the prints, and what values get combined to produce the result. Thus print helps you to debug your code.

6.2 Nicer Output Using Format

Print and its kin are useful basic printing functions, but for sophisticated and easy to read output, format is more useful.

The basic structure of a format call is

```
(format <destination> <control-string> <optional-arguments>)
```

```
1                               ;; first print
2                               ;; second print
3                               ;; returns sum
```

Print always precedes its output with a newline. Prin1 is just like print except that it does not do a new line before printing.

```
>(+ (prin1 1) (prin1 2))
12
3
```

Print is thus equivalent to terpri followed by prin1. This function will behave just like print when passed a single argument.

```
(defun my-print (x)
  (terpri)
  (prin1 x))
```

Princ and prin1 are the same except in the way they print strings. Princ does not print the quote marks around a string:

```
>(prin1 "this string")
"this string"         ;; printed
"this string"         ;; returned

>(princ "this string")
this string           ;; no quotes can be more readable
"this string"         ;; string returned
```

The print family of functions is useful as a debugging tool. Since they return the values of their arguments, they can be inserted into a previously defined function to reveal what is going on. For example, suppose you have a function defined as follows:

```
(defun foo (x y)
  (if (eq x y) (list (second x) (first y)))
    (list y (rest x)))
```

You try the function out and get this:

```
>(foo '(a s d f) '(a s d f))
((A S D F) (S D F))
```

Chapter 6

Input and Output

Terminal input and output is controlled with variants of print and read. More sophisticated output is available using format. Input and output using system files is achieved using the same functions and associating a file with an input or output "stream".

6.1 Basic Printing

Frill-free printing in LISP is achieved with print, prin1, princ and terpri. The simplest uses of print, prin1, and princ involve a single argument. Terpri, which produces a newline, can be called with no arguments.

All these are functions. In addition to causing output, they return values. With print, princ, and prin1, the value returned is always the result of evaluating the first argument. Terpri always returns nil.

Here are examples using print:

```
>(print 'this)

THIS                            ;; printed
THIS                            ;; value returned

>(print (+ 1 2))

3                               ;; printed
3                               ;; returned

>(+ (print 1) (print 2))
```

```
>(typep t1 'trekkie)
T

>(trekkie-p t1)
T

>(employee-p t3)
NIL
```

There are several advanced features of defstruct, including the ability to create structures which incorporate other structures. If you understand the basics laid out here, however, you will have no trouble understanding the description of these features in Steele.

5.5 Exercises.

1. Design a simple data program that allows the user to enter and retrieve personnel data. Your program should store first and last names, addresses, age, marital status, and names of children. You may use any of the techniques introduced in this chapter, but your solution should not place any arbitrary limits, such as a limit on the number of children that can be associated with a person's record. Your program should allow the user to retrieve information about the person or persons whose records match a given key, for example to give the ages of everyone with the last name "Smith."

2. Write a function, num-kids, which uses the data stored by your program for the previous problem to answer the question how many children an individual has.

3. Write functions, num-boys and num-girls that answer how many of an individual's children are boys and how many are girls, using only the information mentioned in problem 1. In other words, your functions should use the children's names to determine their sex. You must at a minimum decide how to handle storage of a list correlating names and sexes, what to do if names not in your list are encountered, and how to handle names such as "Francis" that may be given to either sex.

```
>(setf employee2 (make-employee :age 34
                                :last-name 'farquharson
                                :first-name 'alice
                                :sex 'female))
#S(EMPLOYEE AGE 34 FIRST-NAME ALICE LAST-NAME FARQUHARSON
      SEX FEMALE CHILDREN NIL)

>(employee-first-name employee2)
ALICE
```

As this example shows, it is not necessary to give values to all the slots when the make function is called. Neither is it necessary to specify slot values in the same order they are specified in the original defstruct.

Defstruct also allows you to specify default values for given slots. Here is an example:

```
>(defstruct trekkie
    (sex 'male)
    (intelligence 'high)
    age)
TREKKIE
```

The values enclosed in parentheses with a slot name are the default values for that slot – that is, the values that these slots will have for created instance, unless explicitly overridden. These three examples of instances illustrate the use of defaults:

```
>(setf t1 (make-trekkie))
#S(TREKKIE SEX MALE INTELLIGENCE HIGH AGE NIL)

>(setf t2 (make-trekkie :age 35))
#S(TREKKIE SEX MALE INTELLIGENCE HIGH AGE 35)

>(setf t3 (make-trekkie :age 28 :sex 'female))
#S(TREKKIE SEX FEMALE INTELLIGENCE HIGH AGE 28)
```

Each instance of a structure has a type which can be tested with the predicate typep, or with the particular predicate automatically set up by defstruct. By default, the type of an instance is determined by the structure name:

```
>(typep t1 'employee)
NIL
```

```
>(defstruct employee
   age
   first-name
   last-name
   sex
   children)
EMPLOYEE
```

In this example "employee" is the name of the structure, and "age", etc. are the slots. Defstruct automatically generates a function to make instances of the named structure. In this example the function is called make-employee, and in general the name of the instance constructor function is make-defstructname.

As with the other data types before, it is useful to associate particular instances with a symbol for easy access:

```
>(setf employee1 (make-employee))
#S(EMPLOYEE AGE NIL FIRST-NAME NIL LAST-NAME NIL SEX NIL
     CHILDREN NIL)
```

(Different implementations of LISP will display structures in different ways.)

In this case, employee1 is an instance of the type employee, and all its slots are initially given the value nil. Each slot is provided an automatic access function, by joining the structure name with the slot name:

```
>(employee-age employee1)
NIL

>(employee-sex employee1)
NIL
```

Slot values can be assigned using setf:

```
>(setf (employee-age employee1) 56)
56

>(employee-age employee1)
56
```

It is also possible to assign values to the slots of a particular instance at the time the instance is made, simply by preceding the slot name with a colon, and following it with the value for that slot:

Technically strings are arrays, but it is probably best (at first) to ignore this fact and treat them as a separate data type. Typing a string directly to the interpreter simply causes the interpreter to return the string:

```
>"This is a string"
"This is a string"
```

Notice that the string may contain spaces, and that the distinction between upper and lowercase letters is preserved. A string is completely opaque to the interpreter and may contain punctuation marks and even new lines:

```
>"This is a larger piece of text.
It contains a few, otherwise unmanageable
punctuation marks.  It can even have blank lines:



^Like these!"
"This is a larger piece of text.
It contains a few, otherwise unmanageable
punctuation marks.  It can even have blank lines:



^Like these!"
```

Strings can also be included as elements of lists. For example:

```
>(cons "this" '(here))
("this" HERE)
```

Strings are very useful for giving nicely formatted responses to user commands. This will be explored in the next chapter.

5.4 Defstruct

Defstruct allows you to create your own data structures and automatically produces functions for accessing the data. Structures have names and "slots." The slots are used for storing specific values. Defstruct creates a generic type of which particular "instances" can be made. Here is an example using defstruct to establish a structure type:

```
>(aref my-array 0 1)
NIL

>(setf (aref my-array 0 1) 'hi)
HI

>(setf (aref my-array 1 0) 'bye)
BYE

>my-array
#2A((NIL HI NIL) (BYE NIL NIL))
```

From this example you should be able to work out the indexing scheme.

Make array has a number of additional features we will not cover here. However, one that is particularly useful is the :initial-contents keyword. Here is an example to illustrate the use of :initial-contents.

```
>(make-array '(2 3 4) :initial-contents
             '(((a b c d) (e f g h) (i j k l))
               ((m n o p) (q r s t) (u v w x))))
#3A(((A B C D) (E F G H) (I J K L)) ((M N O P) (Q R S T)
     (U V W X)))
```

Initial contents are specified with a list of elements having the required sublist structure to match the array.

The use of :initial-contents is entirely optional with make-array, as is the use of other keywords not introduced here.

Programmers like to use arrays because they give uniformly fast access to all their elements. In general, however, arrays are less flexible for representing structure than lists. Consider the structure represented in this list:

```
(a (b c) (d e f (g h)))
```

No array can concisely reproduce this structure, since it must have a uniform number of elements for each of its dimensions.

5.3.2 Strings

A string in LISP is represented by characters surrounded by double quotes: ". Strings are very useful for manipulating chunks of text. They are also used by Common LISP to manage input and output.

Object System (CLOS). Arrays, vectors, and strings will be introduced in the rest of this chapter. Customized list structures will not be covered here because they are highly dependent on the specific application. CLOS will not be covered here, either, as it the purpose of this book is to bring you to a point where you can easily understand other presentations of advanced LISP topics.

5.3 Arrays, Vectors, and Strings

5.3.1 Arrays and Vectors

An array is a special type of data object in LISP. Arrays are created using the make-array function. To make an array it is necessary to specify the size and dimensions. The simplest case is an array of one dimension, also called a vector.

Make-array returns an array. Most often one wants to bind this array to a symbol. Here's an example:

```
>(setf my-vector (make-array '(3)))
#(NIL NIL NIL)
```

In this case, the argument to make-array specified that the array should have one dimension of three elements. The array that was returned has three elements, all of them initially set to nil. (The actual printed representation of the array may vary between different LISP implementations.)

These elements can be accessed and changed using aref. These examples illustrate:

```
>(aref my-vector 2)
NIL

>(setf (aref my-vector 0) t)
T

>my-vector
#(T NIL NIL)
```

Indexing of arrays starts with 0. (Just like indexing of lists using nth.) Here's an example of a two-dimensional array and some assignments.

```
>(setf my-array (make-array '(2 3)))
#2A((NIL NIL NIL) (NIL NIL NIL))
```

5.2 Property Lists

An alternative way to attach data to symbols is to use Common LISP's property list feature. For each symbol, the LISP interpreter maintains a list of properties which can be accessed with the function get.

Get expects to be given a symbol and a key. If a value has been set for that key, it is returned, otherwise get returns nil.

```
>(get 'mary 'age)
NIL

>(setf (get 'mary 'age) 45)
45

>(get 'mary 'age)
45
```

As this example shows, the value to be returned by a get expression is set using setf. (This is another place where setf works but setq will not.) The way to think of setf's behavior here is that you tell it exactly what you will type and what should be returned. The example here assigns the value 45 as the value to be returned when you type (get 'mary 'age).

Additional properties can be added in the same way.

```
>(setf (get 'mary 'job) 'banker)
BANKER

>(setf (get 'mary 'sex) 'female)
FEMALE

>(setf (get 'mary 'children) '(bonnie bob))
(BONNIE BOB)
```

If, for some reason, you need to see all the properties a symbol has, you can do so:

```
>(symbol-plist 'mary)
(SEX FEMALE JOB BANKER AGE 45 CHILDREN (BONNIE BOB))
```

Get is a convenient way to manage related data, and in some ways is more flexible than assoc. For many applications, however, it is necessary to build more sophisticated data structures. Options for doing this include customized nested list structures, arrays, and use of the Common Lisp

LISP provides a function, assoc, to retrieve information easily from association lists given a retrieval key.

For example:

```
>(assoc 'age person1)
(AGE 23)

>(assoc 'children person1)
(CHILDREN JANE JIM)
```

Notice that assoc returns the entire key-expression sublist. It does not matter to assoc what order the keys appear in the association list or how many expressions are associated with each key.

Setf can be used to change particular values. For example, here is a function that can be used on a birthday to update a person's age automatically.

```
(defun make-older (person)
  (setf (second (assoc 'age person))
        (1+ (second (assoc 'age person)))))
```

Have your LISP interpreter evaluate this definition, then see that it works:

```
>(make-older person1)
24

>(assoc 'age person1)
(AGE 24)
```

Assoc will return nil if the key is not found.

```
>(assoc 'sex person1)
NIL
```

But it is very easy to add new key-expression sublists, again using setf.

```
>(setf person1 (cons '(sex male) person1))
((SEX MALE) (FIRST-NAME JOHN) (LAST-NAME SMITH) (AGE 24)
 (CHILDREN JANE JIM))
```

Chapter 5

Simple Data Structures in LISP

Many applications of LISP involve storing a variety of values associated with a symbolic name. Association lists and property lists provide two ways of doing this, unique to LISP. Arrays, vectors and strings are also used for data storage and manipulation. Their use is similar to programming languages other than LISP. Defstruct provides a way to create customized data structures.

5.1 Association Lists

An association list is any list of the following form:

```
((<key1> ...<expressions>)
 (<key2> ...<expressions>).....)
```

The keys should be atoms. Following each key, you can put any sequence of LISP expressions.

Use the interpreter to enter this example of an association list:

```
>(setf person1 '((first-name john)
                 (last-name smith)
                 (age 32)
                 (children jane jim)))
((FIRST-NAME JOHN) (LAST-NAME SMITH) (AGE 32)
 (CHILDREN JANE JIM))
```

2. Write a function called "replace," which takes a list and two elements as arguments, and returns the original list with all instances of the first element replaced by the second element.

3. Write a function which counts all of the atoms in a nested list.

4. Write a function called "insert," which takes a nested list and two atoms as arguments, and returns the original list in which the second atom has been inserted to the right of all occurrences of the first atom (if the first atom occurs in the list at all).

5. A new mathematical, binary operator \$ is defined as follows:

$$x\$y = x^2 + y,$$

   where x, y are integers. Extend the definition of "evaluate" presented earlier to include the operators \$ and / (normal division).

6. The Fibonacci series is defined as follows:

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & \text{if } n > 1 \\ 1 & \text{if } n = 0 \text{ or } n = 1 \end{cases}$$

   Implement a recursive function to calculate the nth fibonacci number.

7. Write a function called "merge," which takes two number-lists of equal length. It adds the corresponding members of each list and then returns the product of the resulting numbers.

   For example, `(merge '(1 2 3) '(2 2 2))` should return 60, since $(1+2)*(2+2)*(3+2))$ equals 60.

8. Will the following piece of code always terminate? Be careful to consider all possible cases.

```
(defun mystery (n)
   (cond ((= n 0) 0)
         (t (mystery (- n 1)))))
```

4. When evaluating an expression, do three things:
   a. check for the termination condition;
   b. identify operator;
   c. apply operator to recursive calls on the operands.
5. When building a number value using +, return 0 at the terminating line. When building a number value using *, return 1 at the terminating line.
6. When recurring on a number, do three things:
   a. check for the termination condition;
   b. use the number in some form;
   c. recur with a changed form of the number.
7. To ensure proper termination do two things:
   a. make sure that you are changing at least one argument in your recursive call;
   b. make sure that your test for termination looks at the arguments that change in the recursive call.
8. Two simple cases may occur when changing an argument in a recursive call:
   a. if you are using "rest" to change an argument which is a list, use "null" in the test for termination;
   b. if you are decreasing an argument which is a number, compare it with 0 in the test for termination.
9. Use "let" to reduce the number of function calls.
10. Encapsulate program fragments into new functions to improve clarity.
11. Encapsulate repeated program fragments into new functions to reduce program size.

4.8 Exercises

1. Write a function called "b-remove" (for better remove), which takes a list and an element as arguments, and returns the original list with all occurrences of the element removed.

Note that we are performing almost the same computation in the first three cases of the "cond" clause; specifically, it is to calculate the distance between two points.

Example 13:
Write a function called "distance," which takes two points (represented as two-number lists), and returns the euclidean distance between them. Use "distance" to rewrite "get-side."

The function "distance" can be implemented simply as follows:

```
(defun distance (pt1 pt2)
   (sqrt (+ (exp (- (first  pt1) (first  pt2)) 2)
            (exp (- (second pt1) (second pt2)) 2))))
```

Now we can rewrite "get-side" as follows:

```
(defun get-side (a b c k)
   (cond ((= k 1) (distance a b))
         ((= k 2) (distance b c))
         ((= k 3) (distance c a))
         (t 0)))
```

Thus, using Rule of Thumb 11, we have reduced the size of the program significantly, making it easier to understand.

The concept of abstraction is not unique to LISP. It is also used in other high-level programming languages such as C, Pascal, or FORTRAN.

4.7 Summary of Rules

1. When recurring on a list, do three things:
   a. check for the termination condition;
   b. use the first element of the list;
   c. recur with the "rest" of the list.
2. If a function builds a list using "cons," return () at the terminating line.
3. When recurring on a nested list, do three things:
   a. check for the termination condition;
   b. check if the first element of the list is an atom or a list;
   c. recur with the "first" and the "rest" of the list.

Example 11:

Write a function called "cube," which takes a number and returns its cube. Use "cube" to rewrite "cube-list."

"cube" is defined simply as follows:

```
(defun cube (elt)
   (* elt elt elt))
```

Now we can use "cube" to rewrite "cube-list":

```
(defun cube-list (lst)
   (cond ((null lst) nil)
         (t (cons (cube (first lst))
                  (cube-list (rest lst))))))
```

These last two definitions are much easier to read and understand, and they do not waste any function calls. Furthermore, cube is a useful tool that may be used in other function definitions.

RULE OF THUMB 11:

Encapsulate repeated program fragments into new functions to reduce program size.

Example 12:

Suppose a list of two numbers represents a point in euclidean space. Write a function called "get-side," which takes three points, a, b, and c, and a key, k. The three points represent the vertices of a triangle. The function returns a value as follows:

if k = 1, returns length of side a-b;
if k = 2, returns length of side b-c;
if k = 3, returns length of side c-a;
else, returns 0.

One possible solution is the following:

```
(defun get-side (a b c k)
   (cond ((= k 1)
          (sqrt (+ (exp (- (first  a) (first  b)) 2)
                   (exp (- (second a) (second b)) 2))))
         ((= k 2)
          (sqrt (+ (exp (- (first  b) (first  c)) 2)
                   (exp (- (second b) (second c)) 2))))
         ((= k 3)
          (sqrt (+ (exp (- (first  c) (first  a)) 2)
                   (exp (- (second c) (second a)) 2))))
         (t 0)))
```

4.6 Abstraction

As a program becomes larger, it becomes increasingly difficult to understand. When all of the details of the program are considered at once, they may easily exceed the intellectual grasp of one person. To increase the readability of the program, it is useful to abstract away or "hide" unnecessary details. There are two areas in which abstraction may be used: one may hide the details of the data one is working with by using abstract data structures and associated routines (chapter 5); or one may hide fragments of the program to improve clarity. The latter is explored in this section. Hiding fragments of the program increases clarity and often also results in shorter programs.

RULE OF THUMB 9:

Use "let" to reduce the number of function calls.

Example 10:

Write a function called "cube-list," which takes a list of numbers and returns the same list with each element replaced with its cube. Thus, `(cube-list '(5 3 -15))` should return `(125 9 -3375)`.

By now, with the techniques presented above, the reader should be able to see that the following is a possible solution:

```
(defun cube-list (lst)
   (cond ((null lst) nil)
         (t (cons (* (first lst)
                     (first lst)
                     (first lst))
                  (cube-list (rest lst))))))
```

But note that to compute the cube of each element we must extract it three times from the list using "first." Using Rule of Thumb 9, we can reduce this to only one use of "first" for each element. This may be done as follows:

```
(defun cube-list (lst)
   (cond ((null lst) nil)
         (t (let ((elt (first lst)))
              (cons (* elt elt elt)
                    (cube-list (rest lst)))))))
```

RULE OF THUMB 10:

Encapsulate program fragments into new functions to improve clarity.

```
(length '(a b c))
        = (+ 1 (length '(a b c)))
        = (+ 1 (+ 1 (length '(a b c))))
        = (+ 1 (+ 1 (+ 1 (length '(a b c)))))
        = (+ 1 (+ 1 (+ 1 (+ 1 (length '(a b c))))))
        = (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (length '(a b c)))))))
        = ...
        = ...
```

The list is passed-in unmodified in the recursive call. Such simple mistakes are very common, and their frequency increases with longer and more complicated programs. Compare this definition of "length" with the one given in section 4.2.

RULE OF THUMB 7:

To ensure proper termination do two things:

(1) make sure that you are changing at least one argument in your recursive call;

(2) make sure that your test for termination looks at the arguments that change in the recursive call.

Example 9:

A function, "exp," takes two positive, non-zero integers, x and y, and raises x to the y power. Will the following recursive definition for "exp" terminate properly?

```
(defun exp (x y)
   (cond ((= y 0) 1)
         (t (* x (exp x (- y 1))))))
```

Yes, the above definition is correct and will terminate properly according to Rule of Thumb 7. We fulfill both requirements of the rule:

(1) We are changing at least one argument in the recursive call, namely y, which is decremented by one.

(2) In our test for termination, `(= y 0)`, we are using an argument that we change in the recursive call, namely y.

RULE OF THUMB 8:

Two simple cases may occur when changing an argument in a recursive call:

(1) if you are using "rest" to change an argument which is a list, use "null" in the test for termination;

(2) if you are decreasing an argument which is a number, compare it with 0 in the test for termination.

(2) At each level of recursion we will subtract m from n; this represents our use of n.

(3) We will recur with the value of n changed; specifically, we will recur with n-m and m.

These steps translate into the following function definition:

```
(defun remainder (n m)
   (cond ((< n m) n)
         (t (remainder (- n m) m))))
```

The following notation gives an idea of the execution of "remainder":

```
(remainder 30 7)
        = (remainder 23 7)
        = (remainder 16 7)
        = (remainder 9 7)
        = (remainder 2 7)
        = 2
```

4.5 Ensuring Proper Termination

Often it happens that the LISP programmer unknowingly implements an infinite loop. This could happen in two different ways: an infinite "do" loop, or an improper recursion. In the first case, it may be that the programmer is using the general "do" construct and has specified a test for termination that will never occur. This problem can be avoided by using the more specific constructs "dotimes" or "dolist" (chapter 3). These constructs have a built-in test for termination; "dotimes" iterates a specified number of times, and "dolist" iterates once for each element in a given list. Improper recursion, however, is often more difficult to discover.

Example 8:

The following definition wrongly implements the function to determine the length of a list:

```
(defun length (lst)
  (cond ((null lst) 0)
        (t (+ 1 (length lst)))))
```

The following notation illustrates why the above function does not terminate:

The above problem is naturally recursive. In fact, it is very often represented mathematically as the following recurrence relation:

$$factorial(n) = \begin{cases} n * factorial(n-1) & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

This translates easily into the following LISP function definition:

```
(defun factorial (n)
   (cond ((= n 0) 1)
         (t (* n (factorial (- n 1))))))
```

Let us try to identify the three element of Rule of Thumb 6:

(1) We check for termination by testing if n has been reduced to 0.

(2) We use the number, n, in the multiplication.

(3) We do recur on a changed form of the number, i.e. on n decremented by one.

The following notation gives an idea of the execution of "factorial":

```
(factorial 4)
        = (* 4 (factorial 3))
        = (* 4 (* 3 (factorial 2)))
        = (* 4 (* 3 (* 2 (factorial 1))))
        = (* 4 (* 3 (* 2 (* 1 (factorial 0)))))
        = (* 4 (* 3 (* 2 (* 1 1))))
        = (* 4 (* 3 (* 2 1)))
        = (* 4 (* 3 2))
        = (* 4 6)
        = 24
```

Example 7:

Write a function called "remainder," which takes two positive non-zero numbers, n and m, and returns the remainder when n is divided by m.

Our strategy will be to repeatedly subtract m from n till n is less than m; at this point n will be the value of the remainder. Let us proceed in the steps suggested by Rule of Thumb 6:

(1) We know that we can stop when $n < m$. This will be our termination condition and we will return the value of n.

When building a number value using +, return 0 at the terminating line. When building a number value using *, return 1 at the terminating line.

Example 5:

Write a function, "sum-of-list," which takes a list of numbers and returns their sum. Write a similar function, "product-of-list," which takes a list of numbers and returns their product.

Using the ideas presented in previous sections, writing these functions should be simple. Again, we need to do three things: first we will check for the end of the list; second we will use the first element of the list in the addition (or multiplication); lastly the number will be added (or multipled) to the recursive call on the "rest" of the list. According to Rule of Thumb 5, we must remember that at the terminating line we must return 0 for addition and 1 for multiplication.

The following are the proposed solutions:

```
(defun sum-of-list (lst)
   (cond ((null lst) 0)
         (t (+ (first lst)
               (sum-of-list (rest lst))))))

(defun product-of-list (lst)
   (cond ((null lst) 1)
         (t (* (first lst)
               (product-of-list (rest lst))))))
```

In the above examples, although we are building a number, we are still recurring on a list. It is also possible to recur on a number. The structure of recursion on numbers is very similar to that on simple lists.

RULE OF THUMB 6:

When recurring on a number, do three things:

1. **check for the termination condition;**
2. **use the number in some form;**
3. **recur with a changed form of the number.**

Example 6:

The factorial of a non-negative integer, n, is

n * (n-1) * (n-2) * ... * 3 * 2 * 1.

Also, the factorial of 0 is 1. Implement the factorial function in LISP.

```
(defun evaluate (expr)
   (cond ((numberp expr) expr)
         ((equal (first expr) '+)
          (+ (evaluate (second expr))
             (evaluate (third expr))))
         ((equal (first expr) '-)
          (- (evaluate (second expr))
             (evaluate (third expr))))
         (t
          (* (evaluate (second expr))
             (evaluate (third expr))))))
```

Since there are only three possible operators, we can use the default case for *. The following notation gives an idea of the execution of "evaluate":

```
(evaluate '(* (+ 6 3) (- (+ -1 2) 3))
        = (* (evaluate '(+ 6 3)) (evaluate '(- (+ -1 2) 3)))
        = (* (+ (evaluate 6) (evaluate 3))
             (evaluate '(- (+ -1 2) 3)))
        = (* (+ 6 (evaluate 3)) (evaluate '(- (+ -1 2) 3)))
        = (* (+ 6 3) (evaluate '(- (+ -1 2) 3)))
        = (* 9 (evaluate '(- (+ -1 2) 3)))
        = (* 9 (- (evaluate '(+ -1 2)) (evaluate 3)))
        = (* 9 (- (+ (evaluate -1) (evaluate 2))
                  (evaluate 3)))
        = (* 9 (- (+ -1 (evaluate 2)) (evaluate 3)))
        = (* 9 (- (+ -1 2) (evaluate 3)))
        = (* 9 (- 1 (evaluate 3)))
        = (* 9 (- 1 3))
        = (* 9 -2)
        = -18
```

4.4 Recursion on Numbers

There are some functions which recur on a list of elements, but which return a number. The function "length" defined earlier is such a function. It takes an arbitrarily long list of elements and returns a count of its top-level elements. In the case of length, we are building a number by additions; at each level we add a one. One can also imagine a function which builds a number with consecutive multiplications. In both cases one must be careful in choosing which value to return at the terminating line.

RULE OF THUMB 5:

```
(search '(a (1 c) 2 7) 'c)
        = (search '((1 c) 2 7) 'c)
        = (or (search '(1 c) 'c) (search '(2 7) 'c))
        = (or (search '(c) 'c) (search '(2 7) 'c))
        = (or 't (search '(2 7) 'c))
        = 't
```

Note that "or" only needs to evaluate upto the first non-nil argument to return true. Similarly, "and" only needs to evaluate upto the first nil argument to return nil. Another interesting application of recursion is the evaluation of mathematical or logical expressions.

RULE OF THUMB 4:

When evaluating an expression, do three things:

1. **check for the termination condition;**
2. **identify operator;**
3. **apply operator to recursive calls on the operands.**

Example 4:

Write a function, "evaluate," which takes a prefix expression represented as a nested list and returns the numerical value represented by the expression. Only +, -, and * may be used as operators and each operator can have only two operands. Thus, `(evaluate '(* (+ 4 6) 4)` should return 40.

Let us again try to identify the three elements of the previous Rule of Thumb:

(1) Note that we are no longer working with a list of elements. The argument of "evaluate" represents an expression. If this argument is a list we know that it will have three parts: an operator, the first operand sub-expression, the second operand sub-expression. In this case, we will need to further evaluate the operands. If the argument is a number, we can stop. We can use the predicate "numberp" to test for a numerical value.

(2) We can identify the first element of the argument list as the operator. For each operator we will need to apply a different function.

(3) For each possible operator we will recursively call evaluate on the first and second operands.

The above translates into the following simple piece of code:

RULE OF THUMB 3:

When recurring on a nested list, do three things:

1. **check for the termination condition;**
2. **check if the first element of the list is an atom or a list;**
3. **recur with the "first" and the "rest" of the list.**

Example 3:

Write a function, "search," which takes a nested list and an atom, and it returns `'t` if it finds the atom within the nested list and returns `nil` otherwise.

To write this function, let's take the steps recommended by Rule of Thumb 3:

(1) We will move through the list one element at a time and if we reach the end without finding the given atom, we will return (). To check for termination we can use the predicate "null."

(2) At each step we will look at the first element of the list; if it is an atom we will check if it equals the given atom. If they match, we can return `'t` immediately, else we go on with the search in the rest of the list. We can use the predicate "atom" to check if the first element is an atom. We can use "equal" to test for equality.

(3) Lastly, if we discover that the first element is a list, we may need to perform two searches: the first within the nested list represented by the first element; the second within the "rest" of the original list. If the result of either of these searches is 't, the overall result is also true. We can use the logical function "or" to get this effect.

The above translates into the following simple piece of code:

```
(defun search (lst elt)
   (cond ((null lst) nil)
         ((atom (first lst))
          (if (equal (first lst) elt)
              't
              (search (rest lst) elt)))
         (t (or (search (first lst) elt)
                (search (rest lst) elt)))))
```

The following notation gives an idea of the execution of "search":

The solution is:

```
(defun length (lst)
   (cond ((null lst) 0)
         (t (+ 1 (length (rest lst))))))
```

We can identify the three components mentioned in Rule of Thumb 1:

(1) We still use "null" to test for termination, but, since now we want to return a count of top level elements, when we reach the end of the list we return 0 (the length of a null list).

(2) We only use the first element implicitly; we account for its presence by adding a one to the value returned by the recursive call.

(3) We do recur with the "rest" of the given list. Although we do not explicitly use `(first lst)`, the first element of the list is not forgotten; by adding a one to the result of the recursive call, we keep a track of the top level elements.

The following notation gives an idea of the execution of "length":

```
(length '(a (2 q) 64))
        = (+ 1 (length '((2 q) 64)))
        = (+ 1 (+ 1 (length '(64))))
        = (+ 1 (+ 1 (+ 1 (length '()))))
        = (+ 1 (+ 1 (+ 1 0)))
        = (+ 1 (+ 1 1))
        = (+ 1 2)
        = 3
```

4.3 Recursion on Nested Lists and Expressions

So far we have worked only with simple lists. These are lists for which the top-level elements are our only concern. For example, given the list '(1 a (5 g) up), the top-level elements are: 1, a, (5 g), and up. But often we are interested in looking deeper than just the top-level. A nested list consists of atoms and lists; the latter may themselves be nested. Searching, deleting, inserting, replacing atoms in a multi-level list, or evaluating arbitrarily complex mathematical expressions are all naturally recursive problems on nested lists. Such recursion is slightly more complicated than recursion on simple lists, but it again follows a common, general structure.

(3) We must remember that the function "remove" returns the original list with the first occurrence of the given element removed. When we recur with the "rest" of the list, it is important to preserve the elements that do not match the given element. Thus, in the third part, we should use "cons" to save these elements.

Note that we are building a list using "cons," specifically a list of elements excluding the element to be removed. Using Rule of Thumb 2, we know that in such a case we should return () at the terminating line. Thus, if the test for "null" returns true, our function will return ().

The following solution clearly shows the three parts of Rule of Thumb 1 and also illustrates the use of Rule of Thumb 2:

```
(defun remove (lst elt)
   (cond ((null lst) nil)
         ((equal (first lst) elt) (rest lst))
         (t (cons (first lst)
                  (remove (rest lst) elt)))))
```

The following notation gives an idea of the execution of "remove":

```
(remove '(a 1 c 2 c 7) 'c)
        = (cons 'a (remove '(1 c 2 c 7) 'c))
        = (cons 'a (cons '1 (remove '(c 2 c 7) 'c)))
        = (cons 'a (cons '1 '(2 c 7)))
        = (cons 'a '(1 2 c 7))
        = '(a 1 2 c 7)

(remove '(a (1 q) 2) 'q)
        = (cons 'a (remove '((1 q) 2) 'q))
        = (cons 'a (cons '(1 q) (remove '(2) 'q)))
        = (cons 'a (cons '(1 q) (cons 2 (remove '() 'q))))
        = (cons 'a (cons '(1 q) (cons 2 '())))
        = (cons 'a (cons '(1 q) '(2)))
        = (cons 'a '((1 q) 2))
        = '(a (1 q) 2)
```

Note, Rule of Thumb 1 provides a general framework within which to think about recursion. In different examples, the importance and length of the three components may differ. In the following example the second part of the rule (using the first element of the list) comes into play only implicitly.

Example 2:

Write a function, "length," which takes a list and returns a count of all the top level elements in the list.

4.2 Recursion on Simple Lists

Lists are one of the basic data structures in LISP. Very often programmers need to manipulate lists. Think of the broad possibility of operations one may want to perform on lists: counting the number of elements in a list; searching for an element in a list; removing a particular element from a list; replacing an element in a list; these represent only a small number of possibilities. All of these problems are inherently recursive. Also, their solutions all follow the same structure.

RULE OF THUMB 1:

When recurring on a list, do three things:

1. **check for the termination condition;**
2. **use the first element of the list;**
3. **recur with the "rest" of the list.**

RULE OF THUMB 2:

If a function builds a list using "cons," return () at the terminating line.

Example 1:

Write a function, "remove," which takes a list and an element, and returns the original list with the first occurrence of the element removed.

To solve this problem we will have to look at each of the elements of the given list one by one, checking if it is the element to be removed. We know that we can stop searching if we

(1) have reached the end of the list without finding the element to be removed, or

(2) have found the element to be removed (we can stop, since we are only interested in its first occurrence).

Let's try to apply Rule of Thumb 1.

(1) From the rule we know that at each step we want to recur with the "rest" of the list. Thus, at each recursive call the list will get shorter by one element. We must stop when the list becomes (). We can use the predicate "null" to check for this condition.

(2) The second part states that we should try to use the first element of the list. Before each recursive call we want to check if the first element of the list equals the given element. We can use the predicate "equal" to test for equality.

Chapter 4

Programming Techniques

As has been mentioned before, LISP has its origins in lambda calculus and recursive function theory. The basic definition of LISP is found in McCarthy et al [1965]. Almost all implementations of it have modified this definition in various ways. The beauty of programming in LISP is still largely due to its original definition. Good programming style in Common LISP is no different from other versions of LISP. In this chapter we will examine several techniques that have proved to be very useful in effectively programming in LISP.

4.1 A Word about LISP

LISP is a functional programming language. This means it is based on the use of expressions. The readers may be familiar with Pascal, C, or FORTRAN, which are all classified as imperative programming languages. These languages are statement-oriented, the programs consisting of a sequence of statements. LISP programs, as originally defined, were specified entirely as expressions. Current day implementations of LISP, however, have extensions which allow LISP programs to be more statement-oriented. At the heart of LISP is recursion (chapter 3). Due to their regular, recursive structure, LISP programs tend to be short and elegant. But also, to be able to program effectively in LISP, a different kind of thinking is required; one must learn to think recursively. This is very different from statement-oriented thinking required for languages such as Pascal, C, or FORTRAN. A few simple rules presented in the next few sections will help the reader to think recursively and create better LISP programs.

5. A palindrome is something that reads the same backwards as forwards. The following is a definition of PALINDROMEP which checks if a list is a palindrome.

```
(defun palindromep (lst)
  (equal lst (reverse lst)) )
```

So, for example:

```
>(palindromep '(1 2 3 4 5 4 3 2 1))
T
>(palindromep '(a b b a))
T
>(palindromep '(1 2 3))
NIL
```

Write a recursive version of this function called R-PALINDROMEP without using the function reverse.

6. A mathematical expression in LISP may look something like:

```
(+ (* 1 2 pi) 3 (- 4 5))
```

This would be more readable (to most humans) in "infix" notation:

```
((1 * 2 * pi) + 3 + (4 - 5))
```

Write a function INFIX which given LISP-like mathematical expressions, returns the infixed version.

Real time refers to the time it actually took the computer to return the answer to the screen. In this example, there is no difference in real time between the two functions. Run time and real time can differ because computers running multitasking operating systems, such as UNIX(tm), switch cpu time between different processes to give the appearance of running them simultaneously. If there are a lot of other processes running on the computer, the run time for the LISP function is unchanged, but real time increases.

3.8 Exercises

1. Redefine power so that it can handle negative integer exponents (without using LISP's own `expt` function).

2. Write a function called MY-REMOVE which removes all occurrences of an element from a list. MY-REMOVE should use recursion. Here are a some examples of how your function should behave:

```
>(my-remove 'hello '(hello why dont you say hello))
(WHY DONT YOU SAY)
>(my-remove '(oops my) '(1 2 (oops my) 4 5))
(1 2 4 5)
```

3. Write an iterative version of the above and call it REMOVE-I.

4. LISP provides a predefined function MEMBER which behaves as follows:

```
>(member 3 '(1 2 3 4 5) )
(3 4 5)
>(member 'hey '(whats going on here) )
NIL
>(member 'key '(where did (i (put) the (key))) )
NIL
```

   Write a recursive function called MY-MEMBER which checks for an atom inside nested lists. Here are examples:

```
>(my-member 3 '(1 2 3 4 5) )
T
>(my-member 'hey '(whats going on here) )
NIL
>(my-member 'key '(where did (i (put) the (key))) )
T
```

```
                  10> (MYMAX (3412 54))
                  11> (MYMAX (3412))
                  <11 (MYMAX 3412)
                  <10 (MYMAX 3412)
                 <9 (MYMAX 3412)
               <8 (MYMAX 3412)
             <7 (MYMAX 3412)
           <6 (MYMAX 3412)
         <5 (MYMAX 3412)
       <4 (MYMAX 3412)
     <3 (MYMAX 3412)
   <2 (MYMAX 3412)
 <1 (MYMAX 3412)
3412
```

Notice that the answer 3412 is carried all the way from the bottom to the top-level. Mymax is a tail-recursive function.

When compiling LISP code, a "smart" compiler is capable of recognizing tail recursion, and cutting off processing as soon as the lowest level returns. This can greatly speed up the operation of recursive functions.

3.7 Timing Function Calls

The LISP interpreter provides a interesting feature for timing the evaluation of a function call. Any expression can be timed using `(time <exp>)`. For example, here's time applied to the recursive and iterative versions of power as evaluated by the interpreter:

```
>(time (power 2 100))
real time : 0.200 secs            ;; your output will vary
run time  : 0.133 secs
1267650600228229401496703205376

>(time (power-i 2 100))
real time : 0.200 secs
run time  : 0.083 secs
1267650600228229401496703205376
```

To assess the relative speeds of these two functions, run time (i.e. the amount of time required by the computer's central processing unit to perform the calculation) is the key figure. In this example, power-i uses about 2/3 as much cpu time as power.

(1) Iterative functions are typically faster than their recursive counterparts. So, if speed is an issue, you would normally use iteration.

(2) If the stack limit is too constraining then you will prefer iteration over recursion.

(3) Some procedures are very naturally programmed recursively, and all but unmanageable iteratively. Here, then, the choice is clear.

3.6 Tail Recursion

Item (1) above is only a rough rule of thumb. If you get to the stage of compiling your LISP code (a compiler takes code in a programming language and turns it into binary machine language to make it run faster) some compilers are smart enough to distinguish tail-recursive processes from those that are not. None of the above examples of recursive functions are tail recursive. To be tail-recursive, the answer ultimately returned by the top-level call to the function must be identical to the value returned by the very bottom level call. In the trace output from power, above, you can see that the ultimate answer, 81, is not the same as the deepest level returned value which was 1, so power is not a tail-recursive function.

This function has the property of being tail-recursive:

```
(defun mymax (nums)                        ;; finds the largest
  (cond ((= (length nums) 1) (car nums)) ;; termination
        ((> (car nums) (cadr nums))      ;; first > second so
         (mymax (cons (car nums)
                      (cddr nums))))     ;; get rid of second
        (t (mymax (cdr nums)))))         ;; else dump first
```

If you trace this function applied to a list of numbers you will see something like the following:

```
>(MYMAX '(1 3 412 43 1 1 3412 53 43 43 54))
  1> (MYMAX (1 3 412 43 1 1 3412 53 43 43 54))
    2> (MYMAX (3 412 43 1 1 3412 53 43 43 54))
      3> (MYMAX (412 43 1 1 3412 53 43 43 54))
        4> (MYMAX (412 1 1 3412 53 43 43 54))
          5> (MYMAX (412 1 3412 53 43 43 54))
            6> (MYMAX (412 3412 53 43 43 54))
              7> (MYMAX (3412 53 43 43 54))
                8> (MYMAX (3412 43 43 54))
                  9> (MYMAX (3412 43 54))
```

```
            6> (NUM-NUMS (4 (/ 3 7)))
              7> (NUM-NUMS ((/ 3 7)))
                8> (NUM-NUMS (/ 3 7))
                  9> (NUM-NUMS (3 7))
                    10> (NUM-NUMS (7))
                    11> (NUM-NUMS NIL)
                    <11 (NUM-NUMS 0)
                    <10 (NUM-NUMS 1)
                  <9 (NUM-NUMS 2)
                <8 (NUM-NUMS 2)
                8> (NUM-NUMS NIL)
                <8 (NUM-NUMS 0)
              <7 (NUM-NUMS 2)
            <6 (NUM-NUMS 3)
          <5 (NUM-NUMS 4)
        <4 (NUM-NUMS 4)
        4> (NUM-NUMS ((* 15 2)))
          5> (NUM-NUMS (* 15 2))
            6> (NUM-NUMS (15 2))
              7> (NUM-NUMS (2))
                8> (NUM-NUMS NIL)
                <8 (NUM-NUMS 0)
              <7 (NUM-NUMS 1)
            <6 (NUM-NUMS 2)
          <5 (NUM-NUMS 2)
          5> (NUM-NUMS NIL)
          <5 (NUM-NUMS 0)
        <4 (NUM-NUMS 2)
      <3 (NUM-NUMS 6)
    <2 (NUM-NUMS 7)
  <1 (NUM-NUMS 7)
7
```

It would be hard to define num-nums iteratively. (It is not impossible, but requires you know how to use a stack to mimic the recursion.)

Many artificial intelligence tasks involve searching through nested structures. For example, tree representations of the moves in a game are best represented as a nested list. Searching the tree involves recursively tracking through the tree. For this kind of application, recursive function definitions are an essential tool.

When should you use iteration, and when use recursion? There are (at least) these three factors to consider:

3.5 When To Use Recursion/When To Use Iteration

So far, the two examples of operations presented are just as easy to program recursively as iteratively. However, there are many problems for which recursion is natural and iteration is extremely difficult. This typically arises when considering objects with a complex nested list structure. For example, consider this LISP-format mathematical expression:

```
>(setf math-formula '(+ 3 (* (- 5 pi) 4 (/ 3 7)) (* 15 2)))
(+ 3 (* (- 5 PI) 4 (/ 3 7)) (* 15 2))
```

Math-formula contains lists within lists within lists.

Suppose we would like to know how many numbers are buried in the depths of this formula. Here is a recursive function that will find out:

```
(defun num-nums (mf)
  (cond
    ((null mf) 0)                  ;; empty list has none
    ((numberp (first mf))          ;; if first is number
     (1+ (num-nums (rest mf))))    ;; add to number in rest
    ((atom (first mf))             ;; if it's any other atom
     (num-nums (rest mf)))         ;; ignore it, count rest
    (t (+ (num-nums (first mf))    ;; else it's list to count
          (num-nums (rest mf))))))) ;; and add to num in rest
```

Try this function out and watch it using trace. Notice that the depth of recursion fluctuates as sub-lists are processed.

```
>(num-nums math-formula)
  1> (NUM-NUMS (+ 3 (* (- 5 PI) 4 (/ 3 7)) (* 15 2)))
    2> (NUM-NUMS (3 (* (- 5 PI) 4 (/ 3 7)) (* 15 2)))
      3> (NUM-NUMS ((* (- 5 PI) 4 (/ 3 7)) (* 15 2)))
        4> (NUM-NUMS (* (- 5 PI) 4 (/ 3 7)))
          5> (NUM-NUMS ((- 5 PI) 4 (/ 3 7)))
            6> (NUM-NUMS (- 5 PI))
              7> (NUM-NUMS (5 PI))
                8> (NUM-NUMS (PI))
                  9> (NUM-NUMS NIL)
                  <9 (NUM-NUMS 0)
                <8 (NUM-NUMS 0)
              <7 (NUM-NUMS 1)
            <6 (NUM-NUMS 1)
```

Sometimes when setting up a lot of local variables one would like the value of one variable to depend on another. Let, however, does all its assignments in parallel, so that, for instance:

```
(let ((a 3)
      (b a)))
```

produces an error since a is still an unbound variable when the let statement tries to assign the value of b. Let has a sibling let* which does its variable assignments simultaneously. So:

```
(let* ((a 3)
       (b a)))
```

is fine, and intializes both a and b to the value 3.

Let has applications beyond iterative processes. Anytime you need a temporary storage handle for a value, good LISP programming demands that you should think in terms of using a let statement rather than a global variable.

3.4 Iteration Using Dolist

Dolist is very much like dotimes, except that the iteration is controlled by the length of a list, rather than by the value of a count. Here is the general form for dolist:

```
(dolist (<next-element> <target-list> <result>) <body>)
```

In a dolist statment result and body work just as in dotimes. Next-element is the name of a variable which is intially set to the first element of target-list (which must, therefore be a list). The next time through, next-element is set to the second element of target-list and so on until the end of target-list is reached, at which point result-form is evaluated and returned.

Here is num-sublists done as an example:

```
(defun num-sublists-i (lis)
   (let ((result 0))
      (dolist (next lis result)
         (if (listp next)
             (setf result (1+ result))))))
```

The function num-sublists-i works because result is incremented only whenever the current element of the list indicated by next is itself a list. Notice that in the case where the next element of the list is not itself a list there is no need to do anything. Thus, in the if statement of the definition it is possible to omit the else clause entirely.

This time the existing value of result, i.e. 3, is multiplied by 3, so the new value of result is 9. As you can see, the body will be evaluated four times before count eventually is equal to 4 and the value of result is returned. (1 * 3 * 3 * 3 * 3) = 81, so the correct answer to (power 3 4) is returned.

The correct performance of power-i clearly depends on the variable "result" having the right initial value. If for example, result started at 0, 0 multiplied by 3, four times, would still be 0. We could define result as a global variable and set it to 1 before evaluating the dotimes. This approach is messy for two reasons. First, there is the possibility of another function using a global variable called result. If power-i changes that value, this may cause the other function to malfunction, and vice versa. Second, once power-i has returned its answer we no longer need result for anything. However, if result is a global variable it will keep on using up memory.

The proper way to handle the result variable is to treat it as a local variable, accessible only to power-i. A local variable cleanly disappears when power-i is done, so neither of the aforementioned problems arises. The let statement is what enables us to create local variables.

3.3 Local Variables Using Let

The general form of a let statement is:

```
(let ((<vbl1> <expr1>)
       ..............
      (<vbln> <exprn>))
     <body>)
```

The first part of a let statement is a list of pairs of variable names and expressions providing initial values for those variables. If any of the value expressions is omitted, the corresponding variable is set initially to nil. The left parenthesis before let matches the right parenthesis after the body, and together they define the code block within which the local variables are recognized. The body, as usual, is any sequence of LISP expressions. Let returns the value of the last expression evaluated in the body.

Here is an example of a let statement evaluated directly by the interpreter:

```
>(let ((x 3)
       (y (- 67 34)))
   (* x y))
99
```

```
(defun num-sublists (lis)
  (cond
    ((null lis) 0)                    ;; stop if no elements
    ((listp (first lis))              ;; first is a list
     (1+ (num-sublists (rest lis))))  ;; add to number in rest
    (t (num-sublists (rest lis)))))   ;; else count in rest
```

Try this function out on various examples (and don't forget to use trace to see what it is doing). Also, think about what would happen if num-sublists is called with an atom as its argument, and how you might alter the definition to handle this.

3.2 Iteration Using Dotimes

Common LISP provides several means to do iteration, including loop, do, do*, dotimes, and dolist. Comprehensiveness is not our aim here, so you are referred to Steele for information on loop, do, and do*. Here, we will discuss two simple iteration macros, dotimes and dolist.

The specification for dotimes is as follows:

```
(dotimes (<counter> <limit> <result>) <body>)
```

Counter is the name of a local variable that will be initially set to 0,then incremented each time after body is evaluated, until limit is reached; limit must, therefore, evaluate to a positive integer. Result is optional. If it is specified, then when limit is reached, it is evaluated and returned by dotimes. If it is absent, dotimes returns nil. The body of a dotimes statement is just like the body of a defun – it may be any arbitrarily long sequence of LISP expressions.

Here is power defined with dotimes:

```
(defun power-i (x y)
   (let ((result 1))
      (dotimes (count y result)
         (setf result (* x result)))))
```

For the moment, ignore the let statement here, except to note that it establishes a local variable called "result" whose initial value is 1. Concentrate on the dotimes. What happens when the interpreter evaluates (power-i 3 4)? First x and y are set to 3 and 4, as before. Then count is intialized at 0. Since count does not equal y, the body – (setf result (* 3 result)) – is evaluated. Since result is initially 1, its new value is 3. Then count is incremented to 1, which is again not equal to y, so body is evaluated again.

like provide the correct answer! (Or, at least provide an informative error message.)

3.1.2 Using Trace To Watch Recursion

LISP provides a nice way to watch the recursive process using trace. Enter the following:

```
>(trace power)
POWER

>(power 3 4)
  1> (POWER 3 4)          ;; Your actual output
    2> (POWER 3 3)        ;;  may vary in format
      3> (POWER 3 2)
        4> (POWER 3 1)
          5> (POWER 3 0)
          <5 (POWER 1)
        <4 (POWER 3)
      <3 (POWER 9)
    <2 (POWER 27)
  <1 (POWER 81)
81
```

In this output, each number at the beginning of the line represents the depth of the recursion, from the top-level to the deepest call. All calls to power are documented, along with the values returned. This is a useful debugging tool to see whether your recursive function is doing what you think it should be doing.

If you redefine power and want to keep on watching the trace, then the instruction to (trace power) may have to be repeated. If you want to turn trace off, then (untrace power) will turn off trace for that specific function; (untrace) with no arguments will turn off trace for all functions you may have traced.

3.1.3 Another Example

Here's another recursive function. This one counts the number of elements of a list that are themselves lists:

```
>(power 3 4)
81
```

All well and good. Here is what happened. When (power 3 4) is evaluated, the local variables x and y are set to 3 and 4 respectively. Then the if statement is evaluated. Since y is not equal to zero, the expression (* x (power x (1- y))) is evaluated. 1- is a function that decrements its argument by 1. There is a corresponding function 1+.) This is the same, then, as (* 3 (power 3 3)). Obviously, this multiplication cannot be completed without first calculating (power 3 3). So, another call to power is made. This time, x and y are both 3. It is important to realize that these new values of x and y are local only to the second call to power – the previous call knows nothing about them, and neither will any subsequent call. As before, the test clause of the if statement is false, so the expression (* 3 (power 3 2)) needs to be evaluated, which results in a third call to power with x equal to 3 and y equal to 2. The fifth time that power gets called, its arguments are 3 and 0; this time the test clause is true, so (power 3 0) returns 1 and the recursion stops. Now the previous call to power (the fourth one) is able to complete the multiplication (* 3 (power 3 0)) and returns 3 to the next level up, and so on. Eventually the first call to power (the "top-level" call) is able to finish its multiplication, the result is 81.

Each time power is called recursively, a new level of recursion is created. Is there a limit to how deep the recursion can go? The answer is yes, but how deep this limit is, and how easy it is to change it, will depend on the particular version of LISP you are using. If you exceed the limit, you will see an error message saying something like "Function call stack overflow." This limit is a practical limit imposed by hardware limitations and details of the specific implementation of LISP you are using. In principle there is no limit to how deep recursion can go.

Any recursive function will cause a stack overflow if it does not have a proper termination condition. In the case of power, the termination condition (given in the test clause of the definition) is that the second argument is 0. The termination condition is the most important part of a recursive definition. You want to make sure your program will terminate! Were it not for the stack overflow, a bad terminate condition would result in an infinite spiral.

Think too, what would happen if you called this version of power with y as a negative number or a non-integer as the second argument. This shortcoming of power does not make it a "robust" piece of programming, since it is not always safe to assume that whoever uses power will be careful not to make y negative or an integer. A robust power function would do something appropriate with negative or non-integer second arguments,

Chapter 3

Recursion and Iteration

Many, if not most, uses for computers involve repetitive procedures. LISP provides two paradigms for controlling repetition – recursion and iteration – both covered in this chapter. Also in this chapter you will find out how to set up code blocks with local variables using let.

3.1 Recursive Definitions

3.1.1 A Simple Example

The distinctive feature of a recursive function is that, when called, it may result in more calls to itself. For example, look at this definition of a function to calculate x to the power of y (for positive integer values of y):

```
(defun power (x y)
   (if (= y 0) 1
       (* x (power x (- y 1)))))
```

The else clause of the if statement in this definition contains a reference to power. How can a function make use of itself in its own definition? To understand this, let's examine how power works when it is called. (As always, we assume that you have a machine running LISP in front of you, and that you experiment with the examples we give.)

Use an editor to enter the definition of power given above and evaluate it in the LISP interpreter. Now enter the following expression at the interpreter's prompt:

```
(defun circulate (lst)
  (append (rest lst)
          (list (first lst))))
```

This function takes a list and constructs a new list by taking the first element of the old list and making it the last element of the new. For example:

```
>(circulate '((whats) happening here))
(happening here (whats))
```

Rewrite the function and call it CIRCULATE-DIR so that it can circulate lists in both directions. Thus it should work as follows:

```
>(circulate-dir '(1 2 3 4) 'left)
(4 1 2 3)

>(circulate-dir '(1 2 3 4) 'right)
(2 3 4 1)
```

6. With the definition of CIRCULATE given above, what happens (and explain why) when we evaluate

   ```
   a. (circulate 'hello)

   b. (circulate nil)
   ```

7. Define a function called MY-AND which acts like the LISP AND function (but only takes 2 arguments) using only IF.

Suppose you are running the LISP interpreter and you enter the following:

```
>(setf a 'oh-boy)
```

Then you do the following:

```
>(life 'gummi a)
```

What are the global and local values of a and b before, during, and after this command?

3. Consider the following function definition:

```
(defun who-knows (lst1 lst2)
  (cond ((= (length lst1) (length lst2))
         (+ (length lst1) (length lst2)))
        ((> (length lst1) (length lst2))  (length lst2))
        (t   (length lst1))))
```

   a. What does this function do? Be precise as what would happen in each case.

   b. How would you make this function crash (return an ERROR)? Be careful in explaining why it will happen.

4. Write a function called BLENGTH (B stands for better) which is more tolerant of bad arguments, and is more informative. It works as follows:

```
>(blength '(a b c d))
4

>(blength 'hello)
(sorry hello is an atom)

>(blength 4)
(sorry 4 is a number)
```

   Thus, if a list is passed in it should return the proper length, else if a number, or another type of atom is passed in, it should identify them as such.

5. Consider the following definition for the function CIRCULATE:

2.7.3 Logical Operators: And and Or

And and or are functions but not predicates since they may return values other than t or nil. Each evaluates its arguments in the order they appear, but only enough arguments to provide a definitive result are evaluated. So, some arguments to and and to or may not be evaluated at all.

And returns nil as soon as it finds an argument which evaluates to nil; otherwise it returns the value of its last argument. For example:

```
>(and 1 2 3 4)
4

>(and 1 (cons 'a '(b)) (rest '(a)) (setf y 'hello))
NIL
```

In the example immediately above, the expression (setf y 'hello) is never evaluated since (rest '(a)) returns nil. You can check this out by evaluating y directly:

```
>y
27
```

Or returns the result of its first non-nil argument, and does not evaluate the rest of its arguments. If all the arguments evaluate to nil, then or returns nil. Examples:

```
>(or nil nil 2 (setf y 'goodbye))
2

>(or (rest '(a)) (equal 3 4))
NIL
```

Once again, you will see that y's value is unchanged by these examples.

2.8 Exercises

1. Use the LISP interpreter to help you learn or refresh your memory about the behavior of these predicates: >, <, >=, <=, =, zerop, numberp, symbolp, atom, constantp, listp, functionp

2. Consider the following definition:

```
(defun life (a b)
  (cond ((null a) b)
        ((null b) a)
        (t 'its-tough)))
```

```
>(eq (first a) (first b))
T or NIL (depending on implementation of Common LISP)

>(eql (first a) (first b))
T
```

In most cases, you will want to use either = or equal, and fortunately these are the easiest to understand. Next most frequently used is eq. Eql is used by advanced programmers.

2.7.2 Checking for NIL

The predicates null and not act identically. Good programming style dictates that you use null when the semantics of the program suggest interest in whether a list is empty, otherwise use not:

```
>(null nil)
T

>(not nil)
T

>(null ())
T

>(not ())               ;;preferable to use null
T

>(null '(a s))
NIL

>(not '(a s))           ;;preferable to use null
NIL

>(not (= 1 (* 1 1)))
NIL

>(null (= 1 (* 1 1)))   ;;preferable to use not
NIL
```

```
>(eql 3 6/2)
T

>(equal 3 3)
T

>(equal 3 3.0)
T
```

Suppose now we have the following variable assignments:

```
>(setf a '(1 2 3 4))
(1 2 3 4)

>(setf b '(1 2 3 4))
(1 2 3 4)

>(setf c b)
(1 2 3 4)
```

Then:

```
>(eq a b)
NIL

>(eq b c)
T

>(equal a b)
T

>(equal b c)
T

>(eql a b)
NIL

>(eql b c)
T

>(= (first a) (first b))
T
```

2.7 More Predicates and Functions

2.7.1 Equality Predicates

Common LISP contains a number of equality predicates. Here are the four most commonly used:

= (= x y) is true if and only x and y are numerically equal.

equal As a rule of thumb, (equal x y) is true if their printed representations are the same (i.e. if they look the same when printed). Strictly, x and y are equal if and only if they are structurally isomorphic, but for present purposes, the rule of thumb is sufficient.

eq (eq x y) is true if and only if they are the same object (in most cases, this means the same object in memory).

eql (eql x y) is true if and only if they are either eq or they are numbers of the same type and value.

Generally = and equal are more widely used than eq and eql.
Here are some examples involving numbers:

```
>(= 3 3.0)
T

>(= 3/1 6/2)
T

>(eq 3 3.0)
NIL

>(eq 3 3)
T or NIL (depending on implementation of Common LISP)

>(eq 3 6/2)
T

>(eq 3.0 6/2)
NIL

>(eql 3.0 3/1)
NIL
```

We will call each of the lines (<test>.....<result>) a "cond clause", or sometimes just "clause" for short. In cond clauses, only the test is required, but most commonly cond clauses contain just two elements: test and result.

Instead of nesting if statements as before, the set of conditions (if A B (if C D E)) can be expressed using cond. It would look like this:

```
(cond (A B)
      (C D)
      (t E))
```

A through E can be any LISP expressions at all. If the evaluation of the test expression at the beginning of a clause returns nil, then the rest of the clause is not evaluated. If the test returns a non-nil value, all the other expressions in that clause are evaluated, and the value of the last one is returned as the value of the entire cond statement. (The intermediate forms are, therefore, only useful for producing side-effects. It is common to put t as the test for the last clause since this means that the last clause always will act as a default if none of the other tests succeed.

Here's a simple example, using cond instead of if, defining absdiff to act as before:

```
(defun absdiff (x y)
  (cond ((> x y) (- x y))
        (t (- y x))))
```

Notice the double left parenthesis immediately following cond in this example. A common beginner's programming error is to omit one of these, but both are required since the test in this cond clause is (> x y). With just one parenthesis, the cond statement would attempt to treat the symbol > as the test, which would result in an unbound variable error. Conversely, there is only one left parenthesis in front of the t in the second clause. Again, the explanation is that the test is to evaluate the constant t.

Cond and if statements are most powerful in defining functions that must repeat some step or steps a number of times. In the next chapter, you will see how cond and if are essential for giving recursive function definitions which exploit the power of LISP to solve repetitive problems. Before we can exploit that potential, it will be necessary to learn some more commonly used predicates and functions, that serve well as test statements.

2.6 Conditional Control

2.6.1 If

An if statement has the form:

```
(if <test> <then> <else>)
```

The test, then, and else expressions can be any evaluable Lisp expressions – e.g., symbols or lists. If the evaluation of the test expression returns anything other than nil, e.g. T, 3, FOO, (A S D F), the interpreter evaluates the then expression and returns its value, otherwise it returns the result of evaluating the else expression.

We can use if to define a function to return the absolute difference between two numbers, by making use of the predicate > (greater than). Here it is:

```
(defun absdiff (x y)
  (if (> x y)
      (- x y)
      (- y x)))
```

If x is greater than y, then the test, i.e. (> x y), returns T, so the then clause is evaluated, in this case (- x y), which gives the positive difference. If x is less than or equal to y, then the expression (- y x) gets evaluated, which will return 0 or a positive difference.

2.6.2 Cond

If is useful, but sometimes one would like to have multiple branching conditions. E.g. if condition A is met then do B, but if condition A is not met but condition C is met, then do D, but if neither condition A nor C is met then do E. This could be coded using if as follows (schematically):

```
(if A B (if C D E))
```

This is not too bad. But things can get a lot worse if you want to have a long chain of test conditions.

LISP provides a very convenient macro called cond for such situations. The general form of a cond statement is as follows:

```
(cond (<testa> <form1a> <form2a> ... <resulta>)
      (<testb> <form1b> <form2b> ... <resultb>)
      ...
      (<testk> <form1k> <form2k> ... <resultk>))
```

2.5 Functions with Extended Bodies

As mentioned before, a function definition may contain an indefinite number of expressions in its body, one after the other. Take the following definition, which has two:

```
>(defun powers-of (x)
   (square x)
   (fourth-power x))
POWERS-OF

>(powers-of 2)
16
```

Notice that this function only returns the value of the last expression in its body. In this case the last expression in the body is (fourth-power x) so only the value 16 gets printed in the example above.

What is the point of having more than one expression in the body of a function if it only ever returns the last? The answer to this question is that we may be interested in side effects of the intermediate evaluations.

Powers-of does not have any side effects as it is, but change the definition as follows:

```
>(defun powers-of (x)
   (setq y (square x))
   (fourth-power x))
POWERS-OF
```

Watch what happens here:

```
>y
27

>(powers-of 7)
2401

>y
49
```

The side effect of powers-of was to set the value of the variable y to 1. Since y did not appear in the parameter list of powers-of, it is treated as a global variable, and the effect of the set lasts beyond the life of your call to powers-of.

When typing code in an editor, two elements of good programming practice should be followed.

1. LISP code should be indented in a way which reveals its structure. Some editors, such as emacs, will automatically indent LISP code in a reasonable way. If your editor will not do this, follow examples in these chapters as a guide to one common indenting practice.

2. All code should be commented to improve readability. In LISP, anything on a line following a semi-colon is treated as a comment and is ignored by the interpreter. Opinions vary on the best use of comments. Too many comments can make code as hard to read as too few. Choosing mnemonic names for functions and variables can cut down on the number of comments needed for readability. You might wonder why you need comments if you are to be the only person to read your own code. Write some code and then come back to it two months later and you will understand why comments are important.

You are not recommended to go any further in this chapter until you know how to use a program editor on your computer.

2.4 Using Your Own Definitions in New Functions

User-defined functions can be included in new function definitions. For example:

```
>(defun fourth-power (x)
   (square (square x)))
FOURTH-POWER

>(fourth-power 2)
16
```

It is worth paying attention to what happens when fourth-power is called. During the computation, the variable "x" gets assigned as many as four times. First, x is bound to 2, which is its local value for the function fourth-power. Fourth-power evaluates (square (square 2)), which means that square is called once with the argument 2, and again with the argument 4. Each time square is called, a local version of x is bound to the appropriate value. Finally, x is restored to whatever global value it previously had.

```
>(y-plus 23)
25

>x
5

>(setq y 27)
27

>(y-plus 43)
70
```

Go through these examples (and try out others) to make sure you understand why they behave as shown.

The distinction between local and global variables is very important, and we will come back to it several times. If LISP is not your first programming language one of the most likely signs of an "accent" from the other language(s) is the overuse of global variables. Good LISP programmers use local variables as much as they can, rather than using global ones. Local variables disappear when the function that uses them is done. Global variables hang around for ever, so they take up more memory.

2.3 Using an Editor

Simple function definitions, like square, are easily typed directly at the interpreter's prompt. However, for longer functions this soon becomes inadequate, because you will want to slightly modify definitions without having to completely retype them, not to mention the likelihood that you will make typographic errors when typing them in.

The solution is to write your functions in a program editor and then transfer them to the interpreter for evaluation. Exactly how you do this will vary from system to system and editor to editor. One of the most popular editors for writing LISP code on UNIX(tm)[1] systems is emacs, since it provides convenient methods for transferring code between editor to interpreter. Additionally, most LISP interpreters allow you to directly call an editor from the interpreter prompt by typing (ed "filename").

If you have lots of function definitions, they will need to be stored in a file and then loaded into the interpreter using the instruction (load "filename"). Once again, the details on how to save files and specify their names for loading will vary from system to system.

[1] UNIX(tm) is a registered trademark of AT&T Bell Laboratories

```
119025

>x
3
```

Setting x to 3 has no effect on the operation of square – neither does the function square have a (lasting) effect on the value of x. This is because the interpreter makes a distinction between local variables and global variables.

Global variables have values that can be accessed by any function. The values of local variables are defined only relative to a certain "block" of code. The body of a function definition implicitly constitutes a code block.

In the definition of square, the variable list (x) tells the interpreter what variables are local to the body of the function, i.e. in this case x is a local variable while the block (* x x) is evaluated.

When you make a call to a square, e.g., (square 345), the interpreter assigns 345 as the value of x that is local to the function square. "Local" means that functions other than square do not know about this value for x. Inside the body of square the local value of x (e.g., 345) is preferred to the global value (e.g., 3) that you gave it at the top level. As soon as (square 345) returns 119025, the local value of x no longer is stored, and the only value of x the interpreter knows about is its global value, 3.

The rule the interpreter follows for evaluating symbols is that inside code blocks local values are always looked for first. If a local value for the variable does not exist, then a global value is sought. If no global value is found then the result is an error. This precedence order can be seen with the following example. First define the following function:

```
>(defun y-plus (x)
   (+ x y))
Y-PLUS
```

If you have not assigned a value to y, then typing (y-plus 2) will give an error (unbound variable y). Now do the following:

```
>(setq y 2)
2

>(y-plus 4)
6

>(setq x 5)
5
```

For present purposes, the parameter list should be a list of symbols. The symbols in this list must be variables, not constants. (Recall: T and nil are constants.) In some versions of LISP, pi is also given as a predefined constant that may not be set. The number of symbols in the parameter list determines the number of arguments that must be passed to the function when it is called, otherwise an error message will occur. The function square must be given exactly one argument. Check this out by typing (square 2 3) or (square).

(More advanced programming allows the use of &rest, &optional, &key in the parameter list to permit variable numbers of arguments. The use of these will not be covered here. You are referred to Steele for details.)

Inside the parameter list, x is used as a variable to stand for whatever argument is provided when square is called. Other symbols would have worked just as well in the definition of square. x is short, so we like it here, but sometimes it makes more sense to use something like nmbr that can play a mnemonic role when you look back at your code. Choosing to use number-to-be-squared for the name of the variable would soon get tiresome after you had mistyped it for the umpteenth time!

The body of a function can be a single LISP instruction, or it can be an indefinitely long set of instructions. Good programming style dictates that you keep the body of a function reasonably short (short enough to read on one screen, for example). Good programming technique also includes building small functions that perform specialized tasks and then using those as building blocks for more complicated tasks. The advantage of this technique is that the building blocks can be written and tested separately. Simple, short functions are much easier to debug than 30-line monsters.

Even though we don't want the interpreter to evaluate the name, parameters, and body components when the function is being defined, notice that none of them requires a quote. Defun, like setq and setf, takes care of this automatically.

2.2 Local and Global Variables

An important question that might occur to you is what would happen if you had set x to have some value, before using the function square? Try it out:

```
>(setf x 3)
3

>(square 345)
```

Chapter 2

Defining LISP functions

2.1 Defining Functions: Defun

Use defun to define your own functions in LISP. Defun requires you to provide three things. The first is the name of the function, the second is a list of parameters for the function, and the third is the body of the function – i.e. LISP instructions that tell the interpreter what to do when the function is called.

Schematically then, a function definition looks like this:

```
(defun <name> <parameter-list> <body>)
```

Here is an example of a function to calculate the square of a number. It is short enough to be typed directly at the interpreter's prompt:

```
>(defun square (x)
   (* x x))
SQUARE

>(square 2)
4

>(square 1.4142158)
2.0000063289696399
```

The name of a user-defined function can be any symbol. (Recall: A symbol is any atom that is not a number.) It is even possible to redefine LISP's predefined functions such as first, cons, etc. Avoid doing this!

c. (watergate and no viewer)

d. (bush nixon kennedy)

e. ((bush broccoli) (nixon watergate) (letterman mail))

   b. (rest '((((f))))))
   c. (first '(rest (a b c)))
   d. (first '(rest (rest (a b c))))
   e. (cons '(my life as) '(a dog))
   f. (append '(my life as) '(a dog))
   g. (list '(my life as) '(a dog))
   h. (cons (rest nil) (first nil))
   i. (abs (- (length (rest '(((a b) (c d))))) 5))
   j. (reverse (cons '(rest (reverse '(its gut na mur ta give captin)))))

2. Using first and rest extract the atom "jim" from the following:

   a. (he is dead jim)
   b. (captain (((jim) kirk)))
   c. (((((spock) asked) jim) if) he was all right)
   d. (after (looking at the (lizard man) ((((jim))) asked for warp 9)))

3. What is returned by each of the following expressions (assume they are evaluated in the given order)?

   a. (setf trek '(picard riker laforge worf))
   b. (cons 'data trek)
   c. trek
   d. (length (cons 'troi trek))
   e. (setf trek (cons 'data trek))
   f. (length (cons 'troi trek))

4. Given the following definition:

```
(setf mylist '((bush broccoli) (nixon watergate)
               (letterman (viewer mail))
               (you are no jack kennedy)
               (and please) (scorsese (robert deniro))))
```

   Construct the following with any of the functions you have learned so far.

   a. (no broccoli please)
   b. ((scorsese and deniro) are no robert kennedy)

LISP programs very frequently make use of changes of this sort. But sometimes one would like to change just part of the value of a variable. Suppose you assign a value to a variable as follows:

```
>(setq words '(a list of words))
(A LIST OF WORDS)
```

What if you want to change just part of the list that is the value of words? Well, you could say

```
>(setq words (cons 'this (rest words)))
(THIS LIST OF WORDS)
```

but with lengthy list structures this can get complicated. What you need is a way to change just part of a list; setf is what you need. Look at this sequence to see just some of the ways in which it can be used.

```
>(setf (first words) 'the)
THE
>words
(THE LIST OF WORDS)
>(setf (third words) 'is)
IS
>words
(THE LIST IS WORDS)
>(setf (rest words) '(game is up))
(GAME IS UP)
>words
(THE GAME IS UP)
```

Now you know enough to do the exercises below.

1.7 Exercises

Note: A temptation if you are not used to computers is to sit down and try to work out these exercises in your head and be satisfied when you have reached some answer or other. *DON'T!* Use the LISP interpreter to check your understanding. If you don't understand why the interpreter responds in some way, try to figure it out by playing with some slight variations of the problems.

1. Evaluate the following:

    a. (first '(((a)) (b c d e)))

```
>(length '(1 2 3))
3
>(length a)
5
>(length (append a a))
10
>(length '(append a a))
3
>(length (list a a))
2
```

Predicates are functions that always return either t or nil. Atom is a predicate that determines whether its argument is an atom. Listp returns t if its argument is a list, and nil otherwise.

```
>(atom 'a)
T
>(atom a)
NIL
>(listp 'a)
NIL
>(listp a)
T
```

Find out for yourself how atom and listp work with the empty list, NIL.

Symbolp and numberp are also useful predicates. Experiment with them to find out how they work. Constantp is less frequently used, but might come in handy some time.

Use the appendix entries, together with the LISP interpreter, to figure out how these functions and predicates work:

second, third, fourth,..., last, nthcar, nthcdr, butlast, nbutlast, reverse, caar, cddr, cadr, cdar, constantp, integerp.

1.6 Setf

Setq is useful for changing the values of variables. For example:

```
>(setq my-age (+ my-age 1))
11
>(setq a (cdr a))
(a s d f)
```

```
>(first a)
a
>(rest a)
(s d f)
>(first (rest a))
s
```

You can figure out the rest, like how to get at the third and fourth elements of the list using first and rest.

1.4 Changing Variable Values

What happens to the value of a, after saying (cons 'a a)? Nothing. That is, it looks like this:

```
>(cons 'a a)
(a a s d f)
>a
(a s d f)
```

Obviously, it would be useful to make these changes stick sometimes. To do that you can use setq as follows:

```
>(setq a (cons 'a a))
(a a s d f)
>a
(a a s d f)
```

and henceforth, that is the new value of a.

We'll let you play with the possibilities here, but using setq with just the three functions first, rest, and cons you can do *anything* you want to with lists. These primitives are sufficient. Append and list are strictly superfluous – although they are very convenient. For practice, try to achieve the same effects using just first, rest, and cons as in the examples that used append and list, above.

1.5 More Functions and Predicates

To find out the length of a list, there is a function called, appropriately enough, length. It takes a single argument which should be a list.

append has the effect of taking all the members of the lists that are its arguments and creating a new list from those elements. For example:

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

Beginning LISP programmers (and even some experienced ones) frequently have trouble deciding whether they should use cons or list or append to achieve a particular effect. The best way to get around the difficulty is to play with the different functions and see what works.

1.3.3 Selectors: First and Rest

There are two primitive list selectors. Historically, these were known as car and cdr, but these names were hard to explain since they referred to the contents of various hardware registers in computers running LISP. In Common LISP the functions have been given alternative names, first and rest, respectively. (You can still use the old names in Common LISP. One of us learned LISP in the old days, so occasionally we'll use car or cdr instead of first or rest.)

First takes a list as an argument and returns the first element of that list. It works like this:

```
>(first '(a s d f))
a
>(first '((a s) d f))
(a s)
```

Rest takes a list as an argument and returns the list, minus its first element.

```
>(rest '(a s d f))
(s d f).
>(rest '((a s) d f))
(d f)
>(rest '((a s) (d f))
((d f))
```

You can use setq to save yourself some typing. Do the following:

```
>(setq a '(a s d f))
(a s d f)
```

You can now use a instead of repeating the list (a s d f) every time. So:

Without the quote you would have received an error message. So now you might try (cons 'a (b c d)), but this still won't work since the interpreter tries to evaluate the second argument (b c d), treating b as the name of a function. But (we are assuming) you have not defined b to be a function, so it is an error. Once again you can use the ' to block evaluation of the list (b c d). Thus:

```
>(cons 'a '(b c d))
(a b c d)
```

Notice that you need only one quote on the second argument. The quote blocks evaluation of the whole thing, so you don't need quotes on the internal elements of the list.

Notice, also, that in using setq, the first argument – the variable to be set – does not need a quote; the interpreter does not evaluate it if it is an atom. Originally, LISP had only the function set, whose proper use included things like (set 'a 4) or (set 'a 'b). LISP programmers got tired of putting (or, more likely, forgetting to put) the quote on the first argument, so they wrote setq to take care of it automatically (now you understand the name, right?!) So, (setq a 4) is equivalent to (set 'a 4).

In fact, the ' itself is shorthand for another function. The interpreter expands 'a to the expression (quote a). So (setq a 4) is really short for (set (quote a) 4). Shorthand commands like setq and ' are called "macros" by programmers.

Cons always takes two arguments. (Try it with one or three and see that you get an error message.) Usually, the second argument is a list, but the first can be either an atom or a list. (Cons can be used with an atom as the second argument. What gets returned is a slightly esoteric data type called a "dotted pair." The use of dotted pairs is relatively rare and will be ignored here.)

The list building functions list and append both allow arbitrary numbers of arguments. List takes all its arguments and makes a list with them as elements. Arguments to list will typically be either atoms or lists. For example:

```
>(list 2 3 4)
(2 3 4)
>(list 'a '(a s d f))
(a (a s d f))
```

Make sure you understand why list works as shown in these examples.

Append is normally used with lists as arguments. (Only the last argument may be an atom, resulting in a dotted pair.) When used with lists,

1.3.1 Constructors: Cons, List, and Append

The primitive function "cons" allows you to add a member to the front of a list. Here are two examples:

```
>(cons 1 nil)
(1)
>(cons 1 (cons 2 nil))
(1 2)
```

It is worth looking at what is going on in these examples. In the first, cons is given two arguments. The first is 1 and the second is nil, which, you will remember, is the same as (). So, cons takes the first argument and inserts it as the first element of the second.

To understand the second example, it is useful to anthropomorphize the interpreter. It says, "OK, cons is a function I know and I've got two arguments to that function, 1 and (cons 2 nil). The first argument is an atom, but a special one, and evaluates to 1. The second is a list, so its first element names a function, i.e. cons, and there are two arguments to that function, 2 and nil. 2 evaluates to 2, and nil evaluates to nil (the empty list). So, putting 2 into the empty list we get (2). Now I know what the second argument to the first cons is, we can go ahead and insert the first argument, i.e. 1. Hence we get (1 2)."

Now we discuss what happens when we deal with atoms that are not special. Suppose you wanted to construct the list (a b c d) and you try to do this by saying:

```
>(cons a (b c d))
```

What would happen? Well, the interpreter would say "OK, cons I know, and I've got two arguments to evaluate. First, eval a."

Immediately you get an error because (we are assuming) a is an unbound variable (you haven't set it to anything). How do you fix this problem?

1.3.2 Quote

The answer to the previous question is to use the single quote mark, ', in front of an item that you do not wish the interpreter to evaluate. It works as follows:

```
>'a
a
```

```
>(+ 2 13 45)
60
```

In this case, the interpreter applied the function + to the evaluated arguments and return with the value 60. Since the numbers are predefined, eval finds values for all the arguments, and everyone is happy. You could also enter:

```
>(+ my-age 1)
11
```

This works fine because my-age is evaluated and the value 10 is found (assuming you did just what was described in the section above). However:

```
>(+ your-age 1)
```

will generate an error (unbound variable your-age).

Also, if you attempt to use something that is not a function, you will generate an error message. So, for example, typing

```
>(foo 1 3 4)
```

causes an error (undefined function foo), unless you had previously defined foo to be a function. (More on defining functions in a later chapter).

1.3 Some Primitive Functions

Functions that are built into the LISP language are called "primitive functions." There are (of course) lots of primitive functions in LISP, including all the math functions you would expect. Here's a list of some of the more common math functions:

+, -, *, /, exp, expt, log, sqrt, sin, cos, tan, max, min.

You should look at the appendix entries for these functions and play with them to learn how they work.

More important to the list-processing identity of LISP are the primitive functions that allow selection from lists and construction of lists. The important constructor functions are cons, list, and append. The two principal selector functions are first and rest.

```
>()
NIL
>your-age
Error: The variable YOUR-AGE is unbound.
Error signalled by EVAL.
```

The last item illustrates what happens if you try to evaluate an atom that has not been set to a value. (The exact error message will vary between versions of LISP, but all will say something about an unbound variable).

Most LISP systems throw you into a debugger mode when an error occurs. From the debugger you can find out lots of useful things about the state of the interpreter when the problem occurred. Unfortunately, LISP debuggers are not at all standardized so it is impossible to give a description here. Even in debugger mode, although the prompt usually is different, the LISP interpreter continues to evaluate LISP expressions normally. So we will ignore what is going on when an error occurs and assume that you can just carry on giving expressions to the interpreter for evaluation.

Notice that it is an error to attempt to set a value for special atoms: numbers, t, or nil.

```
>(setq 1 2)
Error: 1 is not a symbol.
Error signalled by SETQ.
>(setq t nil)
Error: Cannot assign to the constant T.
Error signalled by SETQ.
```

From these error messages you can see that the interpreter distinguishes between symbols, numbers, constants. Numbers and symbols are mutually exclusive subcategories of atoms. Constants (such as t and nil) are a subcategory of symbol. Only symbols which are not constants may be assigned a value with setq.

1.2.2 Lists

The second rule of evaluation concerns lists. The interpreter treats any list as containing the name of a function followed by the arguments to the function. Schematically then, a list is read like this:

```
(name-of-function first-argument second-argument ...)
```

For example, try the following:

1.2.1 Atoms

The first rule of evaluation is that for any atom the evaluator, known as "eval," attempts to find a value for that atom.

For most atoms, eval will return an error unless you have previously assigned a value to it. To assign a value to an atom use setq (or you can use its more sophisticated cousin, setf; more on setf in later chapters). So, for instance, to assign the value 9 to the atom "my-age" type the following to the interpreter:

```
>(setq my-age 9)        ; you assign 9 to the atom my-age
9                       ; interpreter responds with value
```

Now, you may test what you have done by giving the atom to the interpreter.

```
>my-age                 ; you tell interpreter to eval my-age
9                       ; it responds with the set value
```

If a birthday has just passed, you can change the value of my-age as follows:

```
>(setq my-age 10)
10
>my-age
10
```

9, 10, 1.234 and all the other numbers are special atoms in LISP – they are pre-defined to evaluate to themselves. So, you may give any number to the interpreter, and it will respond by repeating the number.

In addition to numbers, two other special atoms are predefined, t and nil (think of them as true and false respectively). The interpreter considers nil to be identical to the empty list. Typing () directly to the interpreter will cause it to respond with nil.

Try the following sequence:

```
>9
9
>10
10
>my-age
10
>t
T
>nil
NIL
```

```
>(+ 1 2 3 4)
10
>
```

In this example, the user typed the LISP expression (+ 1 2 3 4) and the interpreter responded with 10 and a new prompt.

The interpreter runs what is known as a *read-eval-print* loop. That is, it *reads* what you type, *evaluates* it, and then *prints* the result, before providing another prompt.

In what follows, it will be assumed that you have a LISP interpreter running in front of you. Exactly how to start LISP up will depend on the computer system you are using. If you need to, check the instructions for your LISP interpreter or ask a local person to find out how to get started. You will also need to know how to get out of the LISP interpreter. This too varies, even between implementations of Common LISP, but (quit), (exit), and (bye) are some common alternatives.

Do not go any further until you know how to start up and exit LISP on your system.

1.2 Basic Data Types

The two most important kinds of objects in LISP for you to know about are atoms and lists. These two kinds are mutually exclusive, with the exception of one special entity, the empty list, known as "()" or "nil," which is both an atom and a list.

Atoms are represented as sequences of characters of reasonable length.

Lists are recursively constructed from atoms. This means that a given list may contain either atoms or other lists as members.

Examples:

ATOMS	LISTS
a	()
john	(a)
34	(a john 34 c3po)
c3po	((john 34) a ((c3po)))

Atoms and lists are both legal LISP expressions for the interpreter to read and evaluate. The rules for evaluating atoms and lists in LISP are very simple (one of the great beauties of the language). They are covered in the next two sections.

Chapter 1

LISt Processing

1.1 Background and Getting Started

LISP is an acronym for **LIS**t **P**rocessor. It was developed by John McCarthy in the late 1950s. LISP found many adherents in the artificial intelligence community, and it is one of the oldest computer languages still in widespread use. The idea for LISP came from a logical system called "lambda calculus" developed by Alonzo Church.

There are many variants (or dialects) of LISP including Scheme, T, etc. In the 1980s there was an attempt to standardize the language. The result is Common LISP (see Guy L. Steele, Jr., *Common LISP: The Language, 2nd Edition*, Digital Press, 1990). Common LISP is now the most popular dialect.

If you are familiar with another programming language, such as C, Pascal, or Fortran, you will be familiar with the concept of a compiler. A compiler is a program that takes a complete program written in one of these languages and turns it into a set of binary instructions that the computer can process. Unlike most languages, LISP is usually used as an interpreted language. This means that, unlike compiled languages, you start an interpreter which can process and respond directly to programs written in LISP. When you start up a LISP interpreter, a prompt will appear, something like this:

```
>
```

The LISP interpreter waits for you to enter a well-formed LISP expression. Once you have entered an expression, it immediately evaluates the expression entered and returns a response. For example:

Contents

LISP Primer

Colin Allen Maneesh Dhagat

1992