Lisp Programming Lecture Notes

AI-TR-85-06

Gordon S. Novak Jr.
Associate Professor
Director, Artificial Intelligence Laboratory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

(512) 471-4353

Arpanet: Novak at UTEXAS-20

Preface

This tutorial is intended as an introduction to Lisp programming for persons who already have experience programming in some language, e.g. FORTRAN.

Most programming languages require that the programmer learn the specific syntax of many different kinds of statements. Lisp syntax is relatively simple and uniform. Much of the work in learning Lisp involves learning the names and effects of the many system functions used in writing application programs. This course presents a set of basic system functions that are frequently used and are present in virtually every Lisp implementation.

The material presented in this course follows the conventions of Common Lisp, which is being standardized with Department of Defense sponsorship. Most dialects of Lisp offer features similar to those described here, though sometimes with differences in function names or order of arguments. This course presents basic (and usually transportable) versions of functions; additional features are available for some functions in some implementations.

How to Learn Lisp

Learning Lisp is like learning other foreign languages: the best method is total immersion in the new language. Fortunately, Lisp is interactive: individual statements can be typed in and evaluated by the computer.

The reader is encouraged to get access to a Lisp implementation and try the examples in these notes; most of the examples given here can be typed in as shown. Try some variations on the examples. Exercises for additional practice are provided at the end of these notes.

A Brief History of Lisp

Lisp is the second-oldest programming language that is still in widespread use (the oldest is Fortran). Lisp was developed by John McCarthy in the late 1950's as a tool for proofs of program correctness; see McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine", *Communications of the ACM*, April 1960.

For many years, Lisp was considered an esoteric language that was mainly used by academics. In the early 1970's, work on high-powered personal workstations specialized for use with Lisp was begun at M.I.T. and at Xerox Palo Alto Research Center; this research resulted in the Lisp machines that are commercially available today.

Lisp's simple and uniform syntax, interpreter, and ability to generate new data structures at runtime have encouraged the development of powerful programming environments that are the best available for any language. Some companies are buying Lisp machines and using Lisp primarily for the programming environment, even when their applications do not use AI techniques.

Advantages of Lisp

- Recursion: A program can call itself as a subroutine.
- Garbage Collection: Data storage is automatically recycled.
- Uniform Representation: Programs and data look the same.
  - Programs can examine other programs.
  - Programs can write programs.
  - Programs can modify themselves ("learn").
  - Data structures can contain programs.
- Interaction: User can combine program writing, compilation, testing, debugging, running in a single interactive session.

"If you want to do A.I., and you don't start with Lisp, you will have to reinvent it."

Applications of Lisp

- Artificial Intelligence
- Symbolic Algebraic Manipulation
- Natural Language Understanding
- Machine Translation
- Formal Logical Reasoning
- Expert Systems:
    - Diagnosis
    - Identification
    - Design
- Automatic Programming
- Robotics
- Perception (Vision, Speech Understanding)

Lisp Data

- Symbols (also called *atomic symbols* or *atoms*): Up to 30 characters, beginning with a letter, followed by letters, numbers, and *some* special characters.

  ```
  MASS CS102 THISISAVERYLONGATOM
  ```

  Special characters that should *not* be included in ATOM names include , . : ' ' " .

- Numbers (*numeric atoms*): integers (whole numbers) and floating-point numbers.

- S-Expressions (*symbolic expressions*): these are defined *recursively* as follows:

  - An atom is an S-Expression.

  - If x_1 ... x_n are S-Expressions, then $(x_1 \ldots x_n)$, called a *list* of $x_1 \ldots x_n$, is an S-Expression.

  Examples:

  ```
  ONTOGENY
  (THIS IS A LIST)
  (* PI (EXPT R 2))
  (ALL X (IF (MAN X) (MORTAL X)))
  ()       ((()))       ((()())())
  ```

The empty list, () is equivalent to the special atom **NIL**.

Computation in Lisp

All computation in Lisp is done by means of *function calls*. A function call is specified as a list consisting of the *function name* followed by the *arguments* of the function; this is sometimes referred to as "Cambridge Polish" notation.

```
    (+ 2 2)

    (+ 3 (* 4 5))

    (PRINT '(HI MOM))

    (* 3.1415926 (EXPT R 2))
```

In most cases, the arguments of the function are *evaluated* before the function is called. For example, in evaluating (+ 3 (* 4 5)) the argument (* 4 5) is evaluated to give 20, and the function + is then applied to the arguments 3 and 20.

A function always returns a *value*; some functions also have *side effects*, such as printing.

Uniformity of Syntax

In Lisp, all computation is done by function calls. A function call is written as a list, with the function name first and arguments following.

```
FORTRAN:                     LISP:

SQRT(2.0)                  (SQRT 2.0)

X = 2                      (SETQ X 2)

X = 2 + 2                  (SETQ X (+ 2 2))

PRINT 1, X                 (PRINT X)

IF(X .GT. Y) Y = 3         (COND ((> X Y)
                                  (SETQ Y 3)))

GO TO 10                   (GO 10)
```

As these examples suggest, it is possible to "write Fortran in Lisp syntax." However, Lisp permits more elegant ways of writing many programming constructs.

Quotation

A symbol (atom) in Lisp can have a *value*, also called its *binding*. There must be a way to distinguish the symbol *itself* from its *value*. In English these are not formally distinguished.

```
The President is the chief executive.

The President's wife is Nancy.
```

In Lisp, we denote the symbol *itself* by Quoting it with a single-quote symbol.

```
(GET 'PRESIDENT 'DUTIES)

(GET PRESIDENT 'SPOUSE)
```

Quoting can also be accomplished by the pseudo-function QUOTE:

```
(GET (QUOTE PRESIDENT) (QUOTE DUTIES))
```

Quoting an S-Expression causes everything inside it to be quoted as well.

```
'(THIS QUOTES (ALL (THIS STUFF)))
```

Variable Values in Lisp

We can think of a symbol (atom) as being a data structure that includes a *value cell* containing a *pointer* to the value of the atom. The value of the atom can be set using the function SET:

```
(SET 'PRESIDENT 'JEFFERSON)
```

If we now *evaluate* PRESIDENT, we will get the value JEFFERSON.

Since we almost always want to quote the first argument of SET, there is a special function SETQ that automatically does this.

```
(SETQ PRESIDENT 'JEFFERSON)


(SETQ PI 3.1415926)
(SETQ RADIUS 5.0)

(* PI (EXPT RADIUS 2))
```

Constructing Lists

An important feature of Lisp for AI applications is the ability to construct new symbolic structures of arbitrary size and complexity at runtime.

A new list structure with a fixed number of elements can be made using the function LIST. To make a list of items x_1 ... x_n, use the form:

```
(LIST x₁ ... xₙ)
```

Each argument of LIST is evaluated unless it is quoted.

```
(LIST 'X 'Y 'Z)  =  (X Y Z)

(LIST (+ 2 3) (* 2 3))  =  (5 6)

(SETQ MAN 'ADAM)
(SETQ WOMAN 'EVE)
(LIST MAN 'LOVES WOMAN)
                 =  (ADAM LOVES EVE)

(LIST (LIST 'A 'B) (LIST 'C 'D))
                      =  ((A B) (C D))
```

Extracting Parts of Lists

Lists can be taken apart by the functions CAR and CDR:

- CAR returns the *first element* of a list.
- CDR returns the *rest* of a list after the first element.

```
(CAR '(A B C))               =  A

(CAR '((A) B C))             =  (A)

(CAR (CAR '((A) B C)))       =  A

(CAR 'A)                     =  Error: A is
                                not a list.


(CDR '(A B C))               =  (B C)

(CDR '((A) B C))             =  (B C)

(CDR (CDR '(A B C)))         =  (C)

(CDR (CAR '((A) B C)))       =  ()  =  NIL

(CDR 'A)                     =  Error: A is
                                not a list.
```

Combinations of CAR and CDR

Since combinations of CAR and CDR are frequently used, all combinations up to four uses of CAR and CDR are defined as functions of the form CxxxR:

- (CAAR X) = (CAR (CAR X))
- (CADR X) = (CAR (CDR X))
- (CADDR X) = (CAR (CDR (CDR X)))

It's worth memorizing some common combinations:

- CAR = First element of a list.
- CADR = Second element.
- CADDR = Third element.
- CADDDR = Fourth element.

Constructing List Structure

The basic function that constructs new list structure is the function CONS.

If Y is a list, then we can think of (CONS X Y) as adding the new element X to the *front* of the list Y.

```
(CONS 'A '(B))          =   (A B)

(CONS 'A NIL)           =   (A)

(CONS 'A '())           =   (A)

(CONS '(A) '(B))        =   ((A) B)

(CONS 'A 'B)            =   (A . B)
```

The following axioms always hold:

1. (CAR (CONS x y)) $=$ x
2. (CDR (CONS x y)) $=$ y

List Manipulation Functions

APPEND makes a new list consisting of the members of its argument lists appended together. APPEND takes any number of arguments.

```
(APPEND '(A) '(B))             =   (A B)

(APPEND '(A B) '(C D))         =   (A B C D)

(APPEND '(A) '(B) '(C))        =   (A B C)
```

REVERSE makes a new list that is the reverse of the list given as its argument.

```
(REVERSE '(A B))               =   (B A)

(REVERSE '((A B)(C D)))        =   ((C D)(A B))
```

LENGTH returns an integer that is the length of the list given as its argument.

```
(LENGTH '(A))         =   1

(LENGTH '(A B))       =   2

(LENGTH '((A B)))     =   1
```

Substitution

The function SUBST makes a new S-expression (not just a list) with a specified substitution.

(SUBST x y z) can be read "Substitute x for y in z".

```
(SUBST 'JONES 'NAME '(DEAR MR NAME))

             =  (DEAR MR JONES)


(SUBST 5.0 'RADIUS
           '(* 3.14159 (EXPT RADIUS 2)))

             =  (* 3.14159 (EXPT 5.0 2))


(SUBST 'SOCRATES 'X
             '(IF (HUMAN X) (MORTAL X)))

             =  (IF (HUMAN SOCRATES)
                    (MORTAL SOCRATES))
```

Evaluation

Evaluation is the process by which Lisp determines the *value* of an expression. Expressions are evaluated using the following recursive algorithm:

1. If the expression to be evaluated is a Number, T, or NIL, its value is the expression itself.
2. If the expression is (QUOTE x), its value is x.
3. If the expression is a symbol (atom), its value is the value of the symbol (the symbol's *binding*).
4. Otherwise, the expression must be a function call:
    a. Evaluate each argument of the function call, in order.
    b. Call the function with the resulting values of its arguments.
    c. The value returned by the function is the value of the expression.

Requesting Evaluation

Evaluation of an s-expression can be explicitly requested using the function EVAL. (EVAL x) gives the result of evaluating the value of x in the current execution context. That is, EVAL performs an *extra* level of evaluation. The argument of EVAL is evaluated once, and should return as its value some Lisp code; then EVAL causes that Lisp code to be evaluated.

```
(EVAL 'X)                 =  value of X

(EVAL '(LIST 'A))         =  (A)

(EVAL (SUBST 10.0 'RADIUS

             '(* 3.14159 (EXPT RADIUS 2))))

                          =  314.159
```

EVAL and its relatives allow Lisp code to be part of a data structure; the code can be retrieved from the data structure and executed by calling EVAL.

Building Lists Incrementally

Often one wants to build up a list of things incrementally; this is usually done using CONS. First, note that for any x, (CONS x NIL) = (LIST x).

```
    (CONS 'A NIL)      =   (A)

    (CONS '(A) NIL)    =   ((A))
```

Second, if LST is a list, then

(CONS x LST)

will make a new list with x as its first element, followed by the other elements of LST.

```
    (CONS 'A '(B C))       =   (A B C)

    (CONS '(A) '(B C))     =   ((A) B C)
```

Building Lists Incrementally ...

Therefore, a list LST can be built up as follows:

1. Initially, set LST to NIL or '() (a list of no elements).
2. For each new element x, set LST to (CONS x LST).

The following example illustrates "consing up" a list. Note that the element added *last* will be at the *front* of the list; REVERSE can be used to reverse the order if desired.

```
                                       LST:

   (SETQ LST '())                      NIL = ()

   (SETQ LST (CONS 'A LST))            (A)

   (SETQ LST (CONS 'B LST))            (B A)

   (SETQ LST (CONS 'C LST))            (C B A)

   (SETQ LST (REVERSE LST))            (A B C)
```

Stepping Through A List

CAR (first element of a list) and CDR (remainder of the list) can be used to step through a list one element at a time:

1. Initially, set a variable REST to the list to be processed.
2. If REST is NIL, quit. Otherwise,
   a. (SETQ TOP (CAR REST))
   b. (SETQ REST (CDR REST))
3. Process the element TOP; go to step 2.

Such a loop is frequently used in Lisp, analogous to the DO loop of Fortran. Complete code for such loops is given later.

Defining New Lisp Functions

New Lisp functions can be defined using the system function DEFUN ("DEfine FUNction"). DEFUN associates the function *definition* with the symbol that is the name of the function. It is possible to redefine system functions using DEFUN, as well as user functions; however, normally one should *avoid* redefining system functions.

DEFUN does *not* evaluate its arguments, so none of them need to be quoted. The format of DEFUN is:

```
(DEFUN  <function name>
        (<arg1 > ... <argn>)
        <code>)
```

For example, a function equivalent to CADR but named SECOND could be defined by:

```
(DEFUN SECOND (X)  (CADR X))
```

Execution of Functions

When Lisp encounters a function call, it does the following:

1. Each *actual argument* (i.e., argument specified in the call to the function) is *evaluated* to produce a single value.
2. Each *formal argument* (i.e., variable specified in the argument list of the function definition) of the function is *bound* to the value that was computed for the corresponding actual argument.
3. The code of the function is executed; this code will produce a value, which is returned as the value of the function.

Example of Function Execution

Suppose we have defined a function TRIOPLUS:

```
(DEFUN TRIOPLUS (X Y Z)
        (+ X (+ Y Z)))
```

Now, suppose we enter:

```
(TRIOPLUS 3 (+ 2 2) (* 4 4))
```

Lisp first evaluates the actual arguments to produce three values, 3, 4, and 16. Next, it binds X to 3, Y to 4, and Z to 16. Next, it starts to evaluate the code of TRIOPLUS. Within the outer +, it evaluates X to get the value 3, then evaluates the inner +. It evaluates Y to 4 and Z to 16, then executes + on 4 and 16, giving 20. Finally, it evaluates the outer + on 3 and 20, giving 23; 23 is returned as the value of TRIOPLUS.

Bindings are Stacked

When the value of an atom is changed with SETQ, the existing value is smashed. However, when Lisp binds a variable upon entry to a function, the existing value of that variable (if any) is not smashed. Instead, a new "layer" is put on top of the variable, and the new value is set in this new layer. When the function is exited, this layer is peeled off, leaving the previous value intact.

```
(SETQ X 5)

(TRIOPLUS 2 X X)

X
```

TRIOPLUS returns 12; evaluation of X returns 5, even though X was bound to 2 inside TRIOPLUS.

Predicates

A *predicate* in Lisp is a function that returns either T (True) or NIL (False). Note: Lisp functions that test predicate values consider NIL to be "False" and *anything other than NIL* to be "True". Predicate names often end in "P".

Some commonly used predicates are:

`(ATOM x)`	`True iff x is an ATOM.`
`(NULL x)`	`True iff x is NIL.`
`(NUMBERP x)`	`True iff x is a number.`
`(ZEROP x)`	`True if x is zero; error if x not a number.`
`(MINUSP x)`	`True if x is negative; error if x not a number.`
`(EQUAL x y)`	`True if x equals y.`
`(< x y)`	`True if x < y.`
`(>= x y)`	`True if x >= y.`

Logical Operators

Predicates can be combined by the logical operators AND, OR, and NOT.

(NOT x) returns T iff x is NIL; otherwise, it returns NIL. NOT is therefore the same as NULL.

(AND x_1 ... x_n) evaluates each of x_1 ... x_n in order. As soon as any x_i returns NIL, AND returns NIL without evaluating any remaining **x**'s. If every x_i returns a non-NIL value, the value of AND is the value of x_n.

(OR x_1 ... x_n) evaluates each of x_1 ... x_n in order. As soon as any x_i returns a non-NIL value, OR returns that value without evaluating any remaining **x**'s. If every x_i returns NIL, the value of OR is NIL.

CONDition Testing

The way to test conditions in Lisp, analogous to the IF statement in other languages, is the COND statement:

```
(COND  ( <test_1> ... <result_1> )
       ( <test_2> ... <result_2> )
       ...
       ( <test_n> ... <result_n> ) )
```

When the COND is evaluated, it will first evaluate <test$_1$>; if its value is true (*non-NIL*), then the remainder of the first *clause* through <result$_1$> is evaluated, and <result$_1$> is the value of the COND. Otherwise, <test$_2$> is evaluated, and so on. If no <test> is true, the value of the COND is NIL.

Often, it is good practice to make sure that the last test in a COND is T to guarantee that all cases are covered.

IF Statement

Common Lisp and some other dialects provide an IF statement:

```
(IF <test> <then-form>)

(IF <test> <then-form> <else-form>)
```

The <test> is evaluated first. If it returns a non-NIL value, the <then-form> is evaluated and its value is the value of the IF; otherwise, the <else-form> is evaluated and its value is the value of the IF. IF is equivalent to

```
(COND (<test> <then-form>)
      (T <else-form>))

(SETQ Y (IF (< X 0.0) (* X X) (SQRT X)))
```

EQ Predicate

The predicate **EQ** tests for equality of *pointer values* inside the machine. It is faster than EQUAL, but less general in its applicability.

EQ always works for comparisons where at least one comparand is a symbol, since symbols are always unique structures in memory. Comparisons against constant symbols are usually done with EQ:

```
    (COND ((EQ (CAR FORM) '+) ... ))
```

In general, EQ does *not* work for numbers or for non-atomic s-expressions.

```
    (EQ '(A) '(A))          =    NIL

    (EQUAL '(A) '(A))       =    T

    (EQ (+ 2 2) 4)          =    ?  (dialect
                                    dependent)

    (EQUAL (+ 2 2) 4)       =    T
```

Membership Testing

A list of items can be used as a representation of a set. The function (MEMBER x l) tests whether the item x is a member of the list l. If x is a member of l, the value of MEMBER is the tail of the list l beginning with the place where x was found; otherwise, the value of MEMBER is NIL. MEMBER may be used as a predicate to test membership.

```
    (SETQ CLUB '(TOM DICK HARRY))

    (MEMBER 'DICK CLUB)  =  (DICK HARRY)
```

Association Lists

A simple “database” facility is provided by the *association list.* This is a list of sublists, in which the first element of a sublist is the “key” value and the remainder of the sublist is the associated data. Association lists are a simple way to implement small databases.

(ASSOC X L) searches L, a list of lists, for an element that begins with the atom X. The result is the element that was found, or NIL if no matching element is found.

```
    (ASSOC 'TWO '((ONE 1)(TWO 2)(THREE 3)))

                      =   (TWO 2)
```

Note that an additional operation (in this case, CADR) is usually required on the result of ASSOC to get the desired data.

Use of Recursion in Lisp

A *recursive* program is one that calls itself as a subroutine. Use of recursion in Lisp can result in programs that are powerful, yet simple and elegant. Often, a large data structure can be handled by a small program which:

1. Tests for a *terminal case* and computes the value of a terminal case directly.
2. Otherwise, does *part* of the job and calls itself recursively to do the rest of the job.

A good way to learn recursion, and to gain an appreciation of the beauty of Lisp, is to study definitions of basic Lisp functions written recursively in Lisp. Reference to these definitions can also be used to answer questions about how the functions work in particular cases.

Examples of Recursion in Lisp

```
(DEFUN LENGTH (L)
  (COND ((NULL L) 0)
        (T (1+ (LENGTH (CDR L)))) ) )


(DEFUN LAST (L)
  (COND ((NULL L) NIL)
        ((NULL (CDR L)) L)
        (T (LAST (CDR L))) ) )


(DEFUN MEMBER (X L)
  (COND ((NULL L) NIL)
        ((EQUAL X (CAR L)) L)
        (T (MEMBER X (CDR L))) ) )
```

More Examples of Recursion

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

```
(DEFUN REVERSE (L)
  (REVERSE1 NIL L))
```

```
(DEFUN REVERSE1 (ANSWER L)
 (COND ((NULL L) ANSWER)
       (T (REVERSE1 (CONS (CAR L) ANSWER)
                    (CDR L))) ) )
```

```
(DEFUN ASSOC (X L)
  (COND ((NULL L) NIL)
        ((EQUAL X (CAAR L)) (CAR L))
        (T (ASSOC X (CDR L))) ) )
```

Binary Tree Recursion

The recursions we have seen so far have involved linear structures, i.e., lists in which only the top level was involved. A second class of recursive programs operates on both halves of a dotted pair; in general, such functions call themselves *twice*.

```
(DEFUN COPY (X)
  (COND ((ATOM X) X)
        (T (CONS (COPY (CAR X))
                 (COPY (CDR X)))) ) )


(DEFUN EQUAL (X Y)
  (COND ((EQ X Y) T)
        ((OR (ATOM X) (ATOM Y)) NIL)
        ((EQUAL (CAR X) (CAR Y))
             (EQUAL (CDR X) (CDR Y)) ))
)
```

SUBST is a COPY with Substitution

SUBST makes a substitution throughout a structure; (SUBST X Y Z) can be read "substitute X for Y in Z".

```
(DEFUN SUBST (X Y Z)
   (COND ((EQ Y Z) X)
         ((ATOM Z) Z)
         (T (CONS (SUBST X Y (CAR Z))
                  (SUBST X Y (CDR Z))) ))
)
```

Executing Multiple Statements

Often, one wishes to execute multiple function calls in order.

(PROGN <statement$_1$> ... <statement$_n$>)
will execute <statement$_1$> through <statement$_n$> in order; the value of PROGN is the value of <statement$_n$>.

Multiple statements are automatically executed at the "top level" of a function or within a COND clause; this is sometimes referred to as an *implicit PROGN.*

(PROG1 <statement$_1$> ... <statement$_n$>)
will execute <statement$_1$> through <statement$_n$> in order; the value of PROG1 is the value of <statement$_1$>.

The PROG Construct

The PROG construct allows the programmer to do some things that are commonly done in other programming languages:

1. Declare local variables for temporary use.
2. Explicitly control the execution sequence of multiple statements.

The format of a PROG is:

```
(PROG  ( <variables> )

          <statement_1>
          . . .
          <statement_n>  )
```

The <variables> are initially bound to NIL when the PROG is first entered; their values may be changed using SETQ, as usual. *All* local variables used in the PROG should be in the <variables> list; otherwise, they will be treated as *free* (global) variables, and may cause hard-to-find errors.

Flow of Control in a PROG

The statements in a PROG are executed in sequence, one at a time. The value of each statement is discarded. If a value is to be returned from a PROG, it must be done using a RETURN statement:

(RETURN <value>)

The RETURN statement does two things: it causes the PROG to be exited (immediately), and it returns a value. If it is desired just to exit, (RETURN NIL) may be used. If the end of a PROG is reached without a RETURN being executed, the PROG will be exited with the value NIL. A RETURN statement may be placed anywhere a function call could normally go.

GO Statement

If an atomic symbol occurs within the list of <statements> in a PROG, it is ignored (not even evaluated). However, such a symbol may be used as a label for a transfer specified by a GO statement:

(GO <label>)

The GO statement causes control to be transferred to the specified <label>. There are several restrictions on this transfer of control:

1. A GO may only specify a label that is defined within the *same* PROG.
2. Labels may appear only at the "statement level" of a PROG, not within any other code.

Some people consider use of the GO statement to be bad style; iteration constructs provided by many Lisp dialects make the GO statement less important than it once was.

Iteration Using PROG

The PROG construct is often used for iterative code, as illustrated by the following version of LENGTH:

```
(DEFUN LENGTH (L)
  (PROG (COUNT)
        (SETQ COUNT 0)
  LOOP  (COND ((NULL L) (RETURN COUNT)))
        (SETQ COUNT (1+ COUNT))
        (SETQ L (CDR L))
        (GO LOOP)   ))
```

This version of LENGTH uses iteration rather than recursion. Iterative forms of Lisp programs may be more comfortable than recursive forms for those who are familiar with traditional programming languages. Iterative programs may run faster than equivalent recursive programs on some Lisp implementations.

General Form of PROG Iteration

Iterative functions using PROG typically have the following basic form:

```
(DEFUN <function> (L)

  (PROG (<variables>)
        . . .
        Initialize variables
        . . .
  LOOP  (COND ((NULL L)
               (RETURN <result>)))
        . . .
        Process (CAR L)
        . . .
        (SETQ L (CDR L))
        (GO LOOP)   ))
```

Several points worth noting:

1. Make sure the COND that causes the RETURN is inside the loop. Usually it goes at the top in case argument L might be NIL.
2. It may be convenient to set a variable to the current element, (CAR L).
3. Don't forget the (SETQ L (CDR L)), or you will have an infinite loop!

Common Lisp DO Construct

Since loops are frequently used, several Lisp dialects provide a DO construct to allow them to be written conveniently. The format of a DO statement in Common Lisp is:

```
(DO   ( <variable-specs> )
      ( <end-test> <result-expr1>
                ... <result-exprn> )
      <statement1>
      ...
      <statementn>    )
```

Each <variable-spec> is a list of the form:

```
(<var>  <init>  <step-expression>)
```

The inside of the DO body is treated like a PROG, i.e., it may contain labels and GO and RETURN statements.

Example of DO Statement

The function LENGTH can be defined using DO as:

```
(DEFUN LENGTH (X)
   (DO  ( (L X (CDR L))          variables
          (N 0 (1+ N)) )
        ( (NULL L) N )           end test,
                                   value
     ) )                         no ‘‘body’’
```

This definition is equivalent to the following:

```
(DEFUN LENGTH (X)
   (PROG (L N)
      (SETQ L X)
      (SETQ N 0)
  LOOP (COND ((NULL L) (RETURN N)))
      (SETQ L (CDR L))
      (SETQ N (1+ N))
      (GO LOOP)))
```

Problem Reduction Search

Problem reduction search is a basic problem-solving technique of Artificial Intelligence. It involves reducing a hard problem into a set of easier problems whose solutions, if found, can be combined into a solution to the hard problem. Such a search is easily written as a recursive program in Lisp:

1. If the given problem is a *primitive subproblem* (one that can be solved by a known technique), return the solution to it.
2. Otherwise, try breaking the given problem into sets of simpler *subproblems*, and call the program recursively to try to solve the subproblems.
3. If a set of subproblems is found such that all the necessary subproblems can be solved, *combine* the subproblem solutions in an appropriate way to form the solution to the current problem.

Symbolic Differentiation

Symbolic differentiation is a classic problem that illustrates the basic steps of problem reduction search.

Primitive subproblems:

```
    d/dx(x) = 1

    d/dx(c) = 0
```

Subproblem reductions:

```
    d/dx(u + v)  =  d/dx(u) + d/dx(v)

    d/dx(u * v)  =  u * d/dx(v) + v * d/dx(u)
```

In the case of d/dx(u + v), the simpler subproblems are the two smaller derivatives d/dx(u) and d/dx(v). The method of combining subproblem solutions is to add together these two results (*algebraically*, not numerically).

Simple Symbolic Differentiation

```
; Derivative of FORM with respect to VAR
(DEFUN DERIV (FORM VAR)
  (COND ((ATOM FORM)
          (COND ((EQ FORM VAR) 1)
                (T 0)))
        ((EQ (CAR FORM) '+)
          (DERIVPLUS FORM VAR))
        ((EQ (CAR FORM) '*)
          (DERIVTIMES FORM VAR))
        (T (ERROR))) )

; Derivative of a sum
(DEFUN DERIVPLUS (FORM VAR)
  (SPLUS (DERIV (CADR FORM) VAR)
         (DERIV (CADDR FORM) VAR)) )

; Derivative of a product
(DEFUN DERIVTIMES (FORM VAR)
  (SPLUS
    (STIMES (CADR FORM)
            (DERIV (CADDR FORM) VAR))
    (STIMES (CADDR FORM)
            (DERIV (CADR FORM) VAR))) )

; Symbolic Plus
(DEFUN SPLUS (X Y) (LIST '+ X Y))

; Symbolic Times
(DEFUN STIMES (X Y) (LIST '* X Y))
```

Testing and Incremental Development

Having just polished off a whole semester of college calculus in a single page of code, it is time to test our wondrous creation:

```
    (deriv '(+ (* m x) b) 'x)

    (+ (+ (* M 1) (* X 0)) 0)
```

This result is mathematically correct; however, it clearly leaves something to be desired. We would like to get just **M** as the result.

Fortunately, Lisp supports the notion of incremental program development: write an initial version of a program, try it, and fix it if it doesn't quite work right. What we need to do now is to replace our functions SPLUS and STIMES with versions that perform *symbolic simplification.*

Symbolic Simplification: Rewrite Rules

Another commonly used AI programming technique is the use of *productions* or *rewrite rules* to restructure symbolic expressions. The rules we need are some basic identities from algebra:

```
X + 0     =>   X

0 + X     =>   X
```

N_1 + N_2 => N_3 (add constants)

```
X * 0     =>   0

0 * X     =>   0

X * 1     =>   X

1 * X     =>   X
```

N_1 * N_2 => N_3 (multiply constants)

We will implement these transformations directly as programs.

Symbolic Simplification Programs

```
;    Symbolic Plus with Simplification
(DEFUN SPLUS (X Y)
  (COND ((NUMBERP X)
           (COND
             ((NUMBERP Y) (+ X Y))
             ((ZEROP X) Y)
             (T (LIST '+ X Y))))
        ((AND (NUMBERP Y)
              (ZEROP Y)) X)
        (T (LIST '+ X Y)) ))
```

```
;    Symbolic Times with Simplification
(DEFUN STIMES (X Y)
  (COND ((NUMBERP X)
           (COND
             ((NUMBERP Y) (* X Y))
             ((ZEROP X) 0)
             ((EQUAL X 1) Y)
             (T (LIST '* X Y)) ))
        ((NUMBERP Y)
           (COND
             ((ZEROP Y) 0)
             ((EQUAL Y 1) X)
             (T (LIST '* X Y)) ))
        (T (LIST '* X Y)) ))
```

Incremental Testing

Another advantage of Lisp is that subprograms can be tested *directly* by giving them test data from the keyboard. There is no need to write (compile, link, load, ...) special programs to drive subprograms for testing.

```
(splus 0 'x)
X
(splus 'x 0)
X
(splus 2 3)
5
(splus 'x 'y)
(+ X Y)
```

After similarly testing STIMES, we can try our example again:

```
(deriv '(+ (* m x) b) 'x)

M
```

Success! It is now left to the reader to extend the programs given here into an approximation of Macsyma (see exercises).

Invoking Evaluation Explicitly

In addition to the usual ability to call a named function, Lisp makes it possible to *compute* the name of a function and explicitly cause that function to be called. There are several ways to do this. All do essentially the same thing; they differ in the form in which the arguments are presented.

(EVAL <x>) causes the expression <x> to be evaluated *twice*: once to find the value to be evaluated, and then to do the evaluation. That is, <x> is a computation that produces Lisp code as its output, and EVAL causes that Lisp code to be executed (evaluated).

```
    (EVAL (SUBST 5 'X '(* X X)))

          =   (EVAL '(* 5 5))

          =   25
```

The ability to compute new pieces of program at runtime and execute them is a unique and powerful feature of Lisp; it makes Lisp an ideal substrate for implementing *embedded languages* such as Expert System tools.

APPLY and FUNCALL

APPLY and FUNCALL are alternative ways to call a function whose name is computed; their arguments are supplied in different ways, so that explicit construction of Lisp code is not required, as it is for EVAL.

(APPLY <fn> <list-of-args>) applies the function <fn> to the argument list <list-of-args>. Both arguments of APPLY are first evaluated.

```
(SETQ MYFUN '+)

(APPLY MYFUN '(2 3))  =  5
```

(FUNCALL <fn> <arg$_1$> ... <arg$_n$>) calls the function <fn> with the arguments <arg$_1$> ... <arg$_n$>. All arguments of FUNCALL are evaluated.

```
(FUNCALL (GET OBJECT 'DRAWING-PROGRAM)
         (GET OBJECT 'POSITION)
         (GET OBJECT 'SIZE))
```

Property Lists

Each symbol (atom) has a *Property List* on which semi-permanent properties of the atom can be stored. Each property has a *property name* or *indicator*, and a *value*. Property list values are retrieved and set by two functions:

(GET <symbol> <propname>) retrieves the *value* of the specified property for the specified atom. If no such property exists, the value of GET is NIL.

(SETF (GET <symbol> <propname>) <value>)
(PUTPROP <symbol> <value> <propname>)
sets the *value* of the specified property for the specified atom, destroying any previous value. Common Lisp uses the SETF form; some other dialects use PUTPROP.

Each atom has a single property list, which appears the same to all parts of a Lisp program and is *totally unaffected* by binding.

Advantages of Property Lists

The property list provides an easy way to create a *database* of relatively permanent facts about objects, e.g., the parts of speech of a word. It is convenient because more kinds of facts can always be added without interfering with other facts, so long as the property names are different.

```
    (SETF (GET 'CAR 'PART-OF-SPEECH) 'NOUN)
```

Note that the symbol CAR can play several roles: it can hold facts about the English word "car" on its property list; it can be used as a variable name; and it is the name of a system function. These uses of CAR are entirely separate and do not conflict.

The property list provides an easy and safe way to create networks of relationships among objects (sometimes called *semantic networks*) while avoiding potential problems due to circular structures and multiple references to the same entities. These problems are avoided because a symbol is always guaranteed to be a single, unique structure in memory.

DEFLIST

DEFLIST is a function that defines values of the same property for a number of different atoms. Its arguments are a list of sublists, each containing an atom name and corresponding value, and the property name. For example,

```
(DEFLIST '((NIXON REPUBLICAN)
           (CARTER DEMOCRAT)
           (REAGAN REPUBLICAN))
         'PARTY)
```

Not all dialects have DEFLIST; it can be defined as follows:

```
(DEFUN DEFLIST (L PROP)
  (PROG ()

     LP   (COND ((NULL L)(RETURN NIL)))

          (SETF (GET (CAAR L) PROP)

                (CADAR L))

          (SETQ L (CDR L))

          (GO LP)  ))
```

Association Lists

An *association list* is an ordinary list that associates property-names and values in a particular format:

```
( (<property-name_1>  <value_1>)
  (<property-name_2>  <value_2>)
  . . .
  (<property-name_n>  <value_n>) )
```

Data for a particular <property-name> value is retrieved by the function ASSOC:

```
(ASSOC <property-name> <a-list>)
```

where <a-list> is an assiciation list in the proper format. The value of ASSOC is the *whole list* of the corresponding entry, or NIL if no such <property-name> is found.

```
(SETQ WORD 'TWO)

(SETQ NUMBER (ASSOC WORD '((ONE 1)
                           (TWO 2)
                           (THREE 3)) ))
     =  (TWO 2)
```

Access Functions

An *access function* is a function that is used to perform accesses to a data structure; it serves to isolate the *meaning* of the substructure from its *implementation.*

Suppose that an "employee" record is implemented as a list of three items, name, title, and salary. Although the "name" field is the CAR of the list, it might be good to write an access function for it:

```
    (DEFUN EMPLOYEE-NAME (L) (CAR L))
```

Advantages of access functions include:

1. Code is clearer and more readable.
2. Data structures can be changed without rewriting the whole program.

Access functions can be made more efficient by defining them as *macros*.

LAMBDA Expressions

A *lambda-expression* can be used to create an "anonymous" function (literally, one without a name); in general, a lambda-expression can be used where a function name is expected. The format of a lambda-expression is:

```
    (LAMBDA (<args>) <code>)
```

A lambda-expression is "quoted" using the special form FUNCTION:

```
    (APPLY (FUNCTION (LAMBDA (X Y) (+ X Y)))
           '(2 3))
```

In some respects, FUNCTION behaves like QUOTE, but FUNCTION should be used rather than QUOTE for lambda-expressions.

Mapping Functions

Often, it is desired to do the same thing to each element of a list. Lisp provides a set of *mapping* functions to allow this to be done conveniently. Each mapping function takes two arguments:

1. The list whose elements are to be processed.
2. The function to be applied to each list element or sublist.

The *function* argument is often supplied to the mapping function as a lambda-expression. For example, the function MAPCAR applies the specified function to successive CARs of a list, and returns a list of the results. To add 2 to each element of a list of numbers, we could use:

```
(MAPCAR (FUNCTION (LAMBDA (X)
                     (+ X 2)))
        '(2 -1 3 9))

  =  (4 1 5 11)
```

Mapping Functions

There are several mapping functions; they differ according to the argument presented to the applied function and the result returned by the mapping function. Each function has the format: **(MAPname <function> <list>)**

Function	Argument	Result
MAPC	Element	--
MAPCAR	Element	List of results
MAPCAN	Element	Conc of results
MAPL	Sublist	--
MAPLIST	Sublist	List of results
MAPCON	Sublist	Conc of results

In many older dialects, the function MAPL is called MAP.

Mapping Function Arguments

Mapping functions MAPL, MAPLIST, and MAPCON supply succesive CDR's of the list to the specified function; MAPC, MAPCAR, and MAPCAN supply successive *elements*. For example, given the list (A B C), the applied function would see the following arguments in succession:

```
MAPC, MAPCAR, MAPCAN:          A

                               B

                               C


MAPL, MAPLIST, MAPCON:         (A B C)

                               (B C)

                               (C)
```

Functions in the MAPC group are the most often used.

Mapping Function Results

The functions MAPL and MAPC do not return an interesting result; they are used primarily for side-effects (e.g., printing).

MAPLIST and MAPCAR return lists of the results of the mapping.

MAPCON and MAPCAN return the *concatenation* of the results of the applied function; this means:

1. A NIL value returned by the applied function will not appear in the result.
2. The applied function must return a *list* of the result for the non-NIL case.

These functions are useful for implementing *filters*, i.e., functions that return the subset of members of a given list satisfying a specified criterion.

Examples of Mapping Functions

```
(MAPC (FUNCTION (LAMBDA (X)
            (COND ((NUMBERP X)(PRINT X)))))
       '(A 1 B 2 3))

   1
   2
   3


(MAPCAR (FUNCTION NUMBERP)
           '(A 1 B 2 3))
    =  (NIL T NIL T T)

(MAPCAN (FUNCTION (LAMBDA (X)
              (COND ((NUMBERP X)(LIST X))
                    (T NIL))))
           '(A 1 B 2 3))
    =  (1 2 3)
```

Functions Related to Mapping Functions

Some Lisp dialects provide additional functions that are similar in form to mapping functions:

1. **(SOME <predicate> <list>)**
   returns the value of <predicate> the first time <predicate> is true (non-NIL).

   ```
   (SOME (FUNCTION NUMBERP) '(A B 1 C))

         =  T
   ```

2. **(EVERY <predicate> <list>)**
   is true (non-NIL) iff every element of <list> satisfies <predicate>. EVERY returns as soon as any member of <list> causes <predicate> to return NIL.

   ```
   (EVERY (FUNCTION NUMBERP) '(3 7 X 1))

         =  NIL
   ```

Basic Printing and Reading

Several functions are provided for printing output:

1. **(PRINT <s>)** will start a new line, then print the s-expression <s>, then print a space.
2. **(PRIN1 <s>)** will print <s> without starting a new line. In general, the output of PRIN1 contains special characters as necessary so that the output can be read back in using READ.
3. **(PRINC <s>)** is like PRIN1, but does not insert any extra characters for unusual atoms, e.g. "blank".
4. **(TERPRI)** ("TERminate PRInt") starts a new line.

The basic input function is (READ), whose value is a single S-expression read from the terminal. READ does not evaluate the expression that is read, so the input does not need to be quoted.

Symbol Manipulation

New atomic symbols can be created by calling the function **(GENSYM)**. GENSYM will return a new atom whose name is of the form "Gnnnn", where "nnnn" is a number.

Symbols with more meaningful names can be generated by supplying a string that is to be the first part of the generated symbol to GENSYM. (Some Lisp dialects have different functions that allow the same thing to be done.)

The symbol returned by GENSYM is not *interned*; that is, it is not entered in the Lisp symbol table and will not be correctly recognized if typed in. If it is desired to make the symbol a "first-class citizen", the function INTERN can be used:
(INTERN (SYMBOL-NAME (GENSYM)))

Special Argument Handling

Ordinary Lisp functions take a fixed number of arguments, all of which are evaluated before the function is entered. Sometimes, however, it is desirable to have a variable number of arguments, or arguments which are unevaluated; the &REST form of arguments allows this.

A function can be defined using the form:

```
   (DEFUN <function-name>
             (<vars> &REST <var>) <code>)
```

When the function is entered, the <vars> are bound to argument values as usual; if any arguments remain in the call to the function, the variable <var> is bound to a *list* of the remaining actual arguments, which are not evaluated. The function may use EVAL to explicitly evaluate arguments if desired.

Macros

A *macro* is a special functional form that translates its call into a Lisp expression; this Lisp expression is returned as the value of the macro, and *it* is then evaluated by the Lisp interpreter or compiled by the Lisp compiler. The macro thereby provides a way of extending the Lisp language.

A macro is defined using the form:

```
    (DEFMACRO <name> (<arguments>) <code>)
```

DEFMACRO is itself a macro that is useful in defining user macros. A macro definition looks much like an ordinary function definition; however, the value returned by the macro is *Lisp code* to do what is wanted.

Example of Macros

Some Lisp dialects provide the function NEQ, which is a negated form of EQ. NEQ can be defined as a macro as follows:

```
   (DEFMACRO NEQ (X Y)
        (LIST 'NOT (LIST 'EQ X Y)) )
```

When NEQ is used, e.g. in (NEQ (CAR L) 'FOO), the Lisp interpreter will first evaluate the macro with X bound to the expression (CAR L) and Y bound to the expression (QUOTE FOO). The macro will return (NOT (EQ (CAR L) (QUOTE FOO))), which will be evaluated by the interpreter or compiler just as if it were the original expression.

It does not matter if the variables X and Y conflict with user variables, since none of the user's code is *evaluated* within the macro.

Uses of Macros

There are several advantages in using macros:

1. Macros can extend the Lisp language by providing "system functions" desired by the user. For example, a CASE statement similar to the CASE statement of Pascal is often provided as a system macro.

2. Macros can reduce the amount of code the user must enter for certain frequently-used sequences.

3. Since a macro expands *in-line* as ordinary Lisp code, it does not involve any extra function calls when used in compiled code. This makes macro use virtually free.

Macros can be relatively slow for interpretive use, since they may cause construction of new code *each time* the macro is executed; for compiled code, there is no penalty.

Backquote

The usual standard in Lisp is that all function arguments are evaluated unless they are quoted. The *Backquote* feature allows that standard to be locally reversed: within a backquote, everything is quoted unless it is prefixed by a comma, in which case it is evaluated. Backquote is especially useful in defining macros. If macros are defined as standard Lisp code, the code of the macro does not "look like" the Lisp code that is desired as output:

```
    (DEFMACRO NEQ (X Y)

        (LIST 'NOT (LIST 'EQ X Y)) )
```

Using the backquote feature, the macro code looks more normal. The variables whose values are to be substituted into the new code are preceded by commas, and the new code itself is preceded by the backquote character:

```
    (DEFMACRO NEQ (X Y) `(NOT (EQ ,X ,Y)))
```

Note that Backquote actually does more than merely avoiding evaluation; it is actually building new list structure, and is in fact just a shorthand way of writing the structure-building code as shown in the first version of

NEQ.

Push and Pop

Since lists are a natural implementation for *stacks*, many Lisp implementations provide PUSH and POP as system functions.

(PUSH <value> <place>)
will push <value> onto the front of the list pointed to by <place>. A simplified version of PUSH could be defined as:

```
(DEFMACRO PUSH (VALUE PLACE)
   `(SETQ ,PLACE (CONS ,VALUE ,PLACE)) )
```

(POP <stack>)
will return the CAR of <stack> and set <stack> to its CDR. A simplified implementation of POP is shown below. (A real implementation would avoid evaluating X more than once at runtime.)

```
(DEFMACRO POP (X)
   `(PROG1 (CAR ,X)
           (SETQ ,X (CDR ,X))) )
```

Internal Representation of List Structure

A *pointer* in Lisp may be thought of as the address of a cell in the computer's memory. List structures are formed from words of memory that are big enough to contain two pointers; such a word is called a *cons cell* or *dotted pair*. The actual physical location of a cons cell is usually not important; what counts is its relation to other cons cells and Symbols, i.e., the *structure* in which it participates.

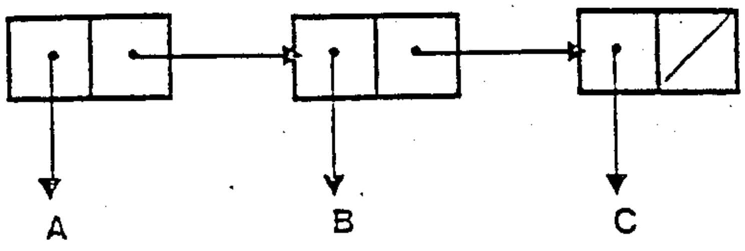
Drawing Box Diagrams of List Structure

A "box structure" diagram of a list structure can be drawn using the following recursive algorithm:

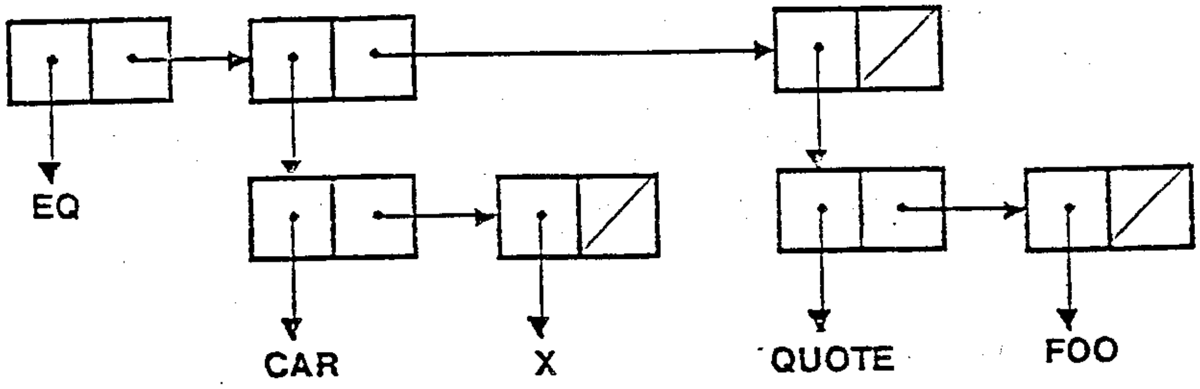
1. If the item to be drawn is a Symbol, write its name.
2. If the item to be drawn is a list of N elements,
   a. Draw a horizontal set of N boxes.
   b. Chain the boxes together by their CDR halves, filling in the last CDR with a diagonal to represent NIL.
   c. Draw a vertical arrow down from the CAR half of each box.
   d. At the end of each arrow, draw the substructure corresponding to that list element.

Diagrams of List Structure

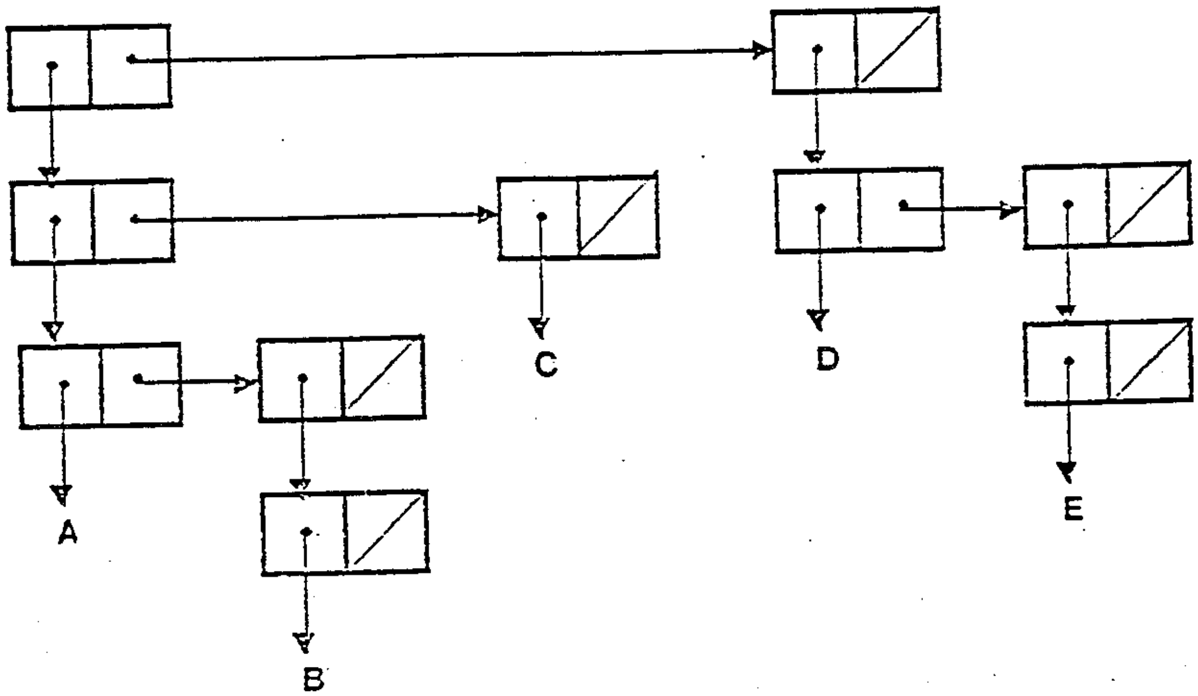
(A B C)



(EQ (CAR X) (QUOTE FOO))



(((A (B)) C) (D (E)))



Destructive Alteration of List Structure

All of the structure-manipulation mechanisms studied so far have been *constructive*, i.e., new structures could be built, but old ones could never be destroyed in the process. There exist functions that modify list structure in a *destructive* fashion, i.e., they smash existing structures; these functions should be used with care.

(RPLACA x y)
replaces the CAR field of the Cons Cell x with the value y. The value of RPLACA is x.

(RPLACD x y)
replaces the CDR field of the Cons Cell x with the value y. The value of RPLACD is x.

Destructive Modification of List Structures

(NCONC x y)

destructively concatenates the two lists x and y. Its value "looks like" the value of APPEND, but the operation is destructive: the last Cons Cell of x is destructively modified to point to y.

```
   (SETQ X '(A B C))
   (SETQ Y '(D E))
   (NCONC X Y)  yields (A B C D E)
   but also X now has the value (A B C D E).
```

Warning: Don't use constant (quoted) data in calls to NCONC, or you will alter your code!

(DELETE x y)

destructively removes occurrences of x from the list y.

In general, use of these functions should be accompanied by an "extra" store operation for safety in special cases:

```
   (SETQ ANIMALS
      (NCONC ANIMALS (LIST NEWANIMAL)))
```

Destructive Reversal of a List

Often, Lisp programs construct lists that are in the reverse of the desired order. NREVERSE reverses such a list destructively, without doing any CONSes.

```
(DEFUN NREVERSE (L)
  (PROG (TEMP RESULT)
    LP  (COND ((NULL L)(RETURN RESULT)))
        (SETQ TEMP (CDR L))
        (RPLACD L RESULT)
        (SETQ RESULT L)
        (SETQ L TEMP)
        (GO LP) ))
```

SETF: Generalized Value Setting

In ordinary Lisp, there are many ways to construct data structures and correspondingly many ways to change parts of those data structures. For example, a variable value is set using the function SETQ, the CAR of a CONS cell is changed using RPLACA, and the CDR is changed using RPLACD.

SETF is a macro that can be thought of as a generalization of SETQ. While SETQ works only for changing variable values, SETF accepts as its first argument a "generalized variable" that specifies how to *access* the value of the desired part of the data structure. This makes access and update code look similar.

```
SETF Code:                  Equivalent Code:

(SETF V VALUE)              (SETQ V VALUE)

(SETF (CAR X) VALUE)        (RPLACA X VALUE)

(SETF (CADR X) VALUE)       (RPLACA (CDR X)
                                    VALUE)
```

Note that SETF can be combined with *other* macros to provide additional power. For example, real implementations of PUSH and POP use SETF rather than SETQ so that they can be used directly on stacks that are part of other data structures; thus, one can write (POP (CAR X)).

Implementation of CONS

Conceptually, CONS is implemented by removing a *Cons Cell* from the *Free List* and replacing its CAR and CDR pointers with the desired values. The following is a simple implementation of CONS:

```
(DEFUN CONS (X Y)
  (PROG (CELL)
    (OR FREELIST
        (GARBAGE-COLLECT)
        (GET-MORE-SPACE)
        (ERROR) )
    (SETQ CELL FREELIST)
    (SETQ FREELIST (CDR FREELIST))
    (RPLACA CELL X)
    (RPLACD CELL Y)
    (RETURN CELL) ))
```

CONS Cell Implementation

On a computer with a large enough word size, each CONS pointer can be put into one half of the word. For example, the DEC-2060 has a 36-bit word, which allows 18 bits for each pointer.

If **n** bits are allowed for a Lisp pointer, the *Address Space* of the Lisp implementation is 2^n of the basic addressable unit of memory. The 18-bit address of the DEC-2060 CONS cell gives an address space of 262,144 words; this is too small for many existing A.I. programs. Modern Lisp machines have address spaces of 24, 28, or 32 bits.

CDR Coding

In theory, both the CAR and CDR pointers of a Cons Cell can point to any location in memory. *In practice,* the CDR pointer usually points to a location close to the location of the Cons Cell that contains it.

CDR coding is a method of encoding the CDR pointer of a Cons Cell that takes advantage of this *locality* to represent the CDR of most cells with less memory. For example, the CAR pointer might be represented by 24 bits (a "full address"), while the CDR pointer might be represented by only 8 bits; this provides a "48-bit" CONS cell using only 32 bits of memory.

Garbage Collection

Garbage collection identifies the cells of memory that are in use; the remaining cells, which are not used for anything, are collected and added to the Free List. This automatic recycling of unused memory is a major advantage of Lisp; it makes it possible for programs to create ("*cons up*") new structures at will without having to worry about explicitly returning unused storage.

Identification of "in use" memory starts from Symbols, which are always "in use". Symbols, in turn, may have several pointers to other data structures:

1. Binding (value) of the Symbol.
2. Function Definition.
3. Property List.

Each of these structures, if present, must also be marked as being "in use".

Mark-And-Sweep Garbage Collection

Mark-and-sweep garbage collection first marks all storage cells that are in use, then sweeps up all unmarked cells. Symbol cells are marked, and all pointers from the symbols are followed using the following recursive algorithm:

1. If the pointer points to a Symbol or to a marked cell, do nothing.
2. Otherwise (pointer points to a Cons Cell),
    a. Mark the cell itself.
    b. Apply the marking algorithm to the CAR of the cell.
    c. Apply the marking algorithm to the CDR of the cell.

Mark-and-Sweep ...

After all cells that are in use have been marked, the Sweep phase is run. All memory cells are examined, in order of increasing address. Those cells that are not marked are pushed onto the Free List.

"Marking" a cell may use an available bit within the cell, or it may use a separate "bit table" that uses one bit to represent each word.

Mark-and-Sweep garbage collection is conceptually simple. However, it requires time that is proportional to the total size of the *address space*, independent of how much garbage is collected. This is a disadvantage for large address spaces.

Another disadvantage of this algorithm is that all computation stops for several seconds while garbage is collected. This is not good for real-time applications, e.g., a robot walking down stairs.

Copying Garbage Collection

Another method of garbage collection is to divide the total address space of the machine into two halves. When storage is exhausted in one half, garbage collection occurs by copying all storage that is in use to the other half. Unused storage, by definition, doesn't get copied.

A copying collector uses time proportional to the amount of storage that is *in use*, rather than proportional to the address space. This is advantageous for programs that generate lots of garbage.

A copying collector has two disadvantages:

1. Half the address space of the machine may be lost to Lisp use, depending on the implementation.
2. There is a long period during which computation stops for garbage collection.

Reference Counting

Another method of managing Lisp storage involves *reference counting*. Conceptually, within each Cons Cell there is room for a small counter that counts the number of pointers which point to that cell. Each time another pointer to the cell is constructed, the counter is incremented by one. Each time a pointer is moved from the cell, the counter is decremented by one.

Whenever the reference count of a cell becomes zero, the cell is garbage and may be added to the Free List. In addition, the reference counts of whatever *its* pointers point to must also be decremented, possibly resulting in additional garbage.

Reference Counting...

Advantages:

1. Garbage collection can be *incremental*, rather than being done all at once. Garbage collection can occur in short pauses at frequent time intervals, rather than in one long pause.
2. Time spent in collection is proportional to *amount collected* rather than to *address space*.

Disadvantages:

1. More complexity in system functions (CONS, SETQ, RPLACA, etc.).
2. Requires storage bits within each CONS cell, or other clever ways of representing counts.
3. Cannot garbage-collect circular structures (since reference count never becomes zero).

The Bottom Line: CONS Is Expensive

The ability to create new data structures at runtime using CONS is a uniquely valuable feature of Lisp.

However, the programmer should be aware that, whatever the implementation, CONS is usually computationally expensive. As a rough rule of thumb, one can think of CONS as taking 100 microseconds on a machine whose basic instructions take 1 microsecond.

The moral: avoid unnecessary conses. For example, if a program has just consed up a list that is in the reverse of the desired order, NREVERSE can be used to reverse the list rather than REVERSE. NREVERSE reuses existing storage and does no conses, while REVERSE makes a new list and does as many conses as there are list elements.

Lisp Reader

The *Lisp reader* is a program that reads input in character form and converts it into Lisp's internal form, including:

1. Looking up Symbol names to find the pointers to the corresponding Symbol data structures in memory. Lisp structures within the computer *always use pointers* rather than names. The only times Lisp uses the name of a Symbol are:

    a. When reading in the Symbol, the Lisp reader looks up the Symbol's name in a symbol table to find the corresponding pointer.

    b. When printing a Symbol, Lisp gets the pointer from the Symbol data structure to the character string representing the symbol's *print name* and prints that string as the external representation of the symbol.

2. Building list structure in response to parenthesized expressions.

3. Ignoring comments.

4. Changing '*expr* into (QUOTE *expr*).

5. Performing other transformations of input into more complex internal forms.

Lisp Reader Implementation

The following programs illustrate how the Lisp reader can be implemented in terms of a function READTOKEN that reads an atomic "item" from the input line and looks it up in the symbol table to get a pointer to its Symbol structure.

```
;   Read an S-expression (Atom or List).
(DEFUN READ-SEXPR ()
  (PROG (NEXT)
    (SETQ NEXT (READTOKEN))
    (COND ((EQ NEXT LEFTPAREN)
              (RETURN (READ-LIST)))
          ((EQ NEXT QUOTECHAR)
              (RETURN (LIST 'QUOTE
                            (READ-SEXPR))))
          (T (RETURN NEXT))) ))


;   Read a List of items.
(DEFUN READ-LIST ()
  (PROG (NEXT RESULT)
 LP (SETQ NEXT (READ-SEXPR))
    (COND ((EQ NEXT RIGHTPAREN)
            (RETURN (NREVERSE RESULT)))
          (T (SETQ RESULT
                   (CONS NEXT RESULT))
             (GO LP))) ))
```

Examples Using Lisp Reader

```
;   Simulate readtoken and test.
(DEFUN TESTREADER (L) (READ-SEXPR))

(DEFUN READTOKEN ()
  (COND (L (POP L))
        (T RIGHTPAREN)))

; Simulate parens as L and R, quote as Q
(SETQ LEFTPAREN 'L)
(SETQ RIGHTPAREN 'R)
(SETQ QUOTECHAR 'Q)

(testreader '(l r))
NIL
(testreader '(l a r))
(A)
(testreader '(q a))
(QUOTE A)
(testreader '(l setq x q l hi mom r r))
(SETQ X (QUOTE (HI MOM)))
(testreader '(l l a b r l c d r r))
((A B) (C D))
(testreader '(l l r r))
(NIL)
(testreader '(l eq q l a r q l a r r))
(EQ (QUOTE (A)) (QUOTE (A)))
```

Read Macros

A *read macro* is a macro that is invoked when Lisp code or data is being read in by the Lisp reader. Usually, a read macro is associated with a particular character and is called whenever that character is encountered in the input.

The quote character (') is often implemented as a read macro. When a ' is seen, the macro is called; the macro function reads the next item and returns the list **(QUOTE <item>)**.

Read Macros are often used to provide an abbreviated *surface syntax* for a language whose underlying form is more complex.

Lisp Data Type Coding

Since there are several Lisp data types that can be denoted by a pointer, there must be some way to distinguish them. There are several ways to do this.

1. Each datum can contain a *type* field that tells the type of the datum.
2. Each pointer can contain a *type* field that tells the type of the datum pointed to.
3. Each pointer can implicitly contain the type by keeping only one type of data on a given page of computer memory; the type is then found by table lookup using the high-order bits of the pointer address.

Shallow Binding

In *shallow binding*, the current binding of a symbol is kept as part of the Symbol data structure. This makes it easy to get the current binding of the atom, so evaluating a variable is fast.

When a symbol is *bound* (e.g., upon entering a function), a pointer to the symbol and its previous value are saved on the Lisp stack. The set of information saved on the stack upon entering a new function is called a *stack frame*.

When a symbol is *unbound* (e.g., upon leaving a function), the previous value of the symbol is restored from the value saved in the stack frame.

Deep Binding

In *deep binding*, the value of a symbol is found by looking for the symbol and its value on the Lisp stack itself. The most recent binding will be the one that is found first. This method of finding symbol values takes more time than shallow binding for evaluating a variable.

When a variable is bound (e.g., upon entering a function), a pointer to the symbol and its new value are saved in the Lisp stack frame. When a variable is unbound (e.g., upon leaving a function), the stack frame is simply popped off the stack; thus, leaving a function is less expensive than with shallow binding.

Global variables (bound at top level but not rebound within functions) are declared to the system and treated as if they were shallow-bound to avoid searching to the bottom of the stack for them.

Lisp Compilation

Compilation allows Lisp programs that have been debugged to run much faster (roughly 10 times faster) than interpreted Lisp. There are several factors that account for the speedup:

1. Many basic Lisp functions (e.g., CAR, CDR, NULL, EQ, 1+) can be compiled direclty as in-line machine code. This saves the overhead of function calls.

2. Function calls can be speeded up by eliminating information that is saved on the stack during interpretation. (This may prevent use of debugging aids such as tracing of calls to compiled functions.)

3. Variable binding lookup can be eliminated by keeping local variables in special locations (e.g., registers), avoiding the usual binding mechanisms.

4. Recursive functions can sometimes be converted to iterative functions by the compiler.

Declarations for Compilation

Lisp compilers often require declarations for non-local variables.

Global variables are those that have values at "top level" and are never rebound within functions. Declaring such variables to be Global, in some dialects, allows lookup of their values to bypass the stack and be fast.

Special variables are those that are used freely outside the defining function. A Special declaration causes the compiler to put such variables on the stack (as is done during interpretation). Variables that are not declared Special will appear to be unbound if referenced freely by compiled code. In some Lisp dialects, variables referenced in a LAMBDA-expression within a call to a MAPping function must also be declared special.

Lisp Dialects

Implementations of Lisp vary *widely* in size and capabilities:

- Small Lisps: These run on small machines. Okay for learning, but may be frustrating and lack expansion capabilities. Examples: MuLisp, IQ-Lisp, Golden Common Lisp (runs on large IBM-PC; perhaps deserves a listing outside the "small" category. Has teaching programs built-in.)
- Large Lisps: Run on "mainframes" or Lisp machines. Examples: Maclisp, Interlisp-20, UCI Lisp, Portable Standard Lisp, Lisp/370, Franz Lisp, Common Lisp.
- Giant Lisps: Run on Lisp machines and are classified as "giant" due to the huge amount of programming environment software provided. Examples: Interlisp-D, Zetalisp.

Common Lisp

Common Lisp is an attempt to standardize the various Lisp dialects of the Maclisp family (Maclisp, Franz Lisp, Zetalisp, and others). The Department of Defense has pushed this standardization effort.

It is a good bet that most manufacturers of Lisp equipment will offer at least a large degree of compatibility with Common Lisp. Competition, different philosophies, and incompatibilities will continue to flourish in areas such as operating systems, network protocols, I/O, graphics, window systems, and other non-computational areas.

Interlisp

Interlisp was originally developed at Bolt, Beranek and Newman Inc., and later moved to Xerox Palo Alto Research Center. It runs on DEC-20's, VAXes, and a family of Lisp machines produced by Xerox. Interlisp is relatively large, and generally runs more slowly than Lisp dialects of the Maclisp family on similar hardware; however, it provides many useful features:

- Editor: An editor for Lisp structures is built-in; the editor itself is written in Lisp.
- File Package: a set of programs for maintaining files of programs and data.
- DWIM ("Do What I Mean") automatically corrects simple mistakes such as spelling errors.
- Break Package: User can put conditional traps and traces inside functions, examine the state of the computation during a break, fix broken functions and continue to run.

- UNDO: A mistaken action can be "undone". Incorrect typed-in commands can be fixed with the editor, then re-executed.

- MASTERSCOPE: Analyzes functions to produce a database that can be queried with English-like commands:

```
WHAT FUNCTIONS USE X FREELY

WHO CALLS APPEND

WHAT FUNCTIONS ON FILE1 ARE CALLED
BY MYFUNCTION

SHOW PATHS FROM DERIV TO SPLUS
```

- CLISP: Translates constructs like those found in more "traditional" programming languages into Lisp:

```
(FOR X IN L DO (PRINT X)
               WHEN (NUMBERP X))
```

Lisp Machines

Improved performance at lower cost can be gotten by designing a computer especially for use with Lisp; such a computer is called a *Lisp machine.* Lisp machines based on "standard" microprocessor chips are also appearing.

Special hardware or microcode can perform certain operations unique to Lisp at high speed:

1. Special data operations such as CDR-coding and reference counting.
2. Lookup of variable bindings.
3. Function calls.
4. Garbage collection.
5. Runtime type checking for error control.

Current Lisp machines also offer high-resolution graphics displays and large collections of software to aid program development and debugging. The graphics interface and programming environment software are *very valuable.*

Choosing A Lisp and Machine

If you can afford it, buy a Lisp machine. The user interface hardware and the program development environment software are extremely valuable.

Shop around to find the best machine for your needs. The Lisp machine market is highly competitive. Speed isn't everything; most time is spent in writing, editing, and debugging programs. The software is a large part of the "value added" of a Lisp machine; consider software carefully as part of the purchase decision.

Shoot high when choosing a machine: by the time you finish your application, the price will have come down.

Try to keep most of your code compatible with Common Lisp. Insulate your code from manufacturer-dependent system functions such as I/O by writing a layer of generic interface functions between your code and the system functions.

Lisp Bibliography

Books:

1. Abelson, Harold and Sussman, Gerald J., ***Structure and Interpretation of Computer Programs***, McGraw-Hill, 1985. Based on the Scheme dialect of Lisp.

2. Allen, John R., ***Anatomy of Lisp.*** McGraw-Hill, 1978. More detail on actual Lisp implementation than we have covered.

3. Charniak, E., Riesbeck, C.K., and McDermott, D.V., ***Artificial Intelligence Programming***, Hillsdale, NJ: Lawrence Erlbaum Associates, 1978. Examples and discussion of more advanced A.I. programming techniques using Lisp.

4. Friedman, Daniel P., ***The Little LISPer***, Science Research Associates, Palo Alto, CA, 1974.

5. ***Interlisp Reference Manual***, Xerox Palo Alto Research Center, 1984. A large Lisp implementation with many useful features.

6. McCarthy, John, et al., ***LISP 1.5 Programmer's Manual***, MIT Press, 1965. For many years the basic reference work on Lisp. Primarily of historical and scholarly interest.

7. Moon, D. and Weinreb, D. ***Lisp Machine Manual.*** Cambridge, MA: Symbolics, Inc., 1981. The manual for Zetalisp (a Maclisp descendant) on the Symbolics Lisp Machine.

8. Novak, Gordon S. Jr., "GLISP User's Manual", Technical Report TR-83-25, Artificial Intelligence Laboratory, Computer Science Dept., University of Texas at Austin, Austin, TX 78712. GLISP is a high-level language with abstract data types built on top of Lisp; it is available free or at nominal charge for several Lisp dialects.

9. Siklossy, Laurent, ***Let's Talk Lisp***, Prentice-Hall, 1976. An introductory text.

10. Steele, Guy L. Jr., ***Common Lisp: The Language***, Digital Press, 1984. The primary reference on the Common Lisp standard language.

11. Touretzky, David D., ***LISP: A Gentle Introduction to Symbolic Computation***, Harper & Row, 1984. Introductory text based on Maclisp and

Common Lisp.

12. Wilensky, Robert, ***LISPcraft***, W. W. Norton, 1984. Introductory text based on the Franz Lisp dialect.

13. Winston, Patrick H. and Horn, Berthold K. P., ***Lisp***, 2nd ed., Addison-Wesley, 1984. An introductory text based on Common Lisp.

Articles:

1. Bobrow, D. G., "Managing Reentrant Structures Using Reference Counts", *ACM Transactions On Programming Languages And Systems*, Vol. 2, No. 3, July 1980, pp. 269-273.

2. Bobrow, D. G. and Clark, D. W., "Compact Encodings of List Structure", *ACM Transactions On Programming Languages And Systems*, Vol. 1, No. 2, October 1979, pp. 266-286.

3. Borning, Alan, “THINGLAB: A Constraint-Oriented Simulation Laboratory”, Technical Report STAN-CS-79-746, Computer Science Department, Stanford University, Stanford, CA 94305. Interesting uses of object-oriented programming (in Smalltalk, not Lisp).

4. Goldberg, Adele, et al., *Byte Magazine*, special issue on Smalltalk, Aug. 1981. Object-oriented programming.

5. McCarthy, John, "History of Lisp". In R. L. Wexelblat (Ed.), *History of Programming Languages*, Academic Press, 1981. Also in *ACM SIGPLAN Notices*, vol 13, no. 8 (Aug. 1978), pp. 217-223.

6. Steele, Guy L. and Sussman, Gerald J., “Design of a LISP-Based Microprocessor”, *Communications of the ACM*, vol. 23, no. 11 (Nov. 1980).

Lisp Exercise #1

This provides exercises in data representation and basic data manipulation in Lisp. The material in this assignment is covered in Winston & Horn, Chapters 2 and 3.

Algebraic representation in Lisp:

1. Write down in regular algebraic notation what the following LISP expressions represent:

```
a. (- (* x 5) y)

b. (* (+ (/ x z) y) (sin (sqrt (+ z (- w)))))

c. (/ (- (* x y) (* z w))
      (+ (/ z (+ y 5)) 1.0))
```

2. Translate the following expressions into Lisp expressions using the mixed-arithmetic functions (+, /, etc.):

a. xy–z

b. (–b + b^2–4ac) / 2a

c. (mx+b)/(–y + |z*w–u|^3) + 7.0

Evaluate the following Lisp expressions. For each expression, write out what the Lisp interpreter would type back to you if you were to type in the expression. In evaluating expressions, Lisp first evaluates the arguments of each function (recursively, so that more deeply nested expressions are evaluated before the expressions which contain them) and then applies the function to the result of evaluating the arguments. The result of the outermost function is typed back to the user. When variable values are set using SETQ, use the new value of the variable in subsequent evaluations. SET and SETQ return the value to which the variable is set. The best way to do this assignment is to answer the questions yourself on paper, then use the computer to check your answers. Try some variations on these examples where appropriate.

```
a. (car nil)
b. (cdr nil)
c. (setq colors '(red yellow ((orange) grey) ((blue) green)))
d. (car colors)
e. (cdr colors)
f. (car 'colors)
g. (cadr colors)
h. (caddr colors)
i. (cdddr colors)
j. (cdaddr colors)
k. (car (caaddr colors))
l. For each atomic color in the list "colors", write the Lisp
   code to extract that color from the list structure.
m. (cons 'cat nil)
n. (cons '(cat mouse) nil)
o. (setq animals '(cat mouse))
o'. animals
p. (cons 'bear animals)
p'. animals
q. (setq animals (cons 'moose animals))
r. animals
s. (car animals)
t. (cons '(bear lion) animals)
u. (append animals '(tiger giraffe))
v. animals
w. (append animals animals animals animals)
x. (setq birds (list 'jay 'sparrow 'eagle))
y. (caddr birds)
z. (list birds animals)
aa. (append birds animals)
bb. (list '(armadillo) birds)
cc. (car (list '(armadillo) birds))
dd. (setq zoo (append animals birds))
ee. (cadr zoo)
ff. (car birds)
gg. (set (car birds) 'favorite)
gg'. jay
hh. (eval (car birds))
ii. (eval '(car birds))
jj. (reverse animals)
```

```
kk. (last birds)
ll. (car (last (reverse birds)))
mm. (subst (car birds) 'rose '(a rose is a rose is a rose))
nn. (subst 'taste 'smell (subst 'lollipop 'rose
         '(a rose by any other name would smell as sweet)))
oo. (eq 'a 'a)
pp. (eq nil '())
qq. (equal 2.0 2.0)
rr. (equal '(a) '(a))
ss. (eq '(a) '(a))
tt. (not t)
uu. (not 'a)
vv. (not '())
ww. (length zoo)
xx. (plus (length (car zoo)) (length (cadr zoo)))
```

3. Write a function EXCHANGE, with parameters DOLLARS and CURRENCY, which will compute the amount of the specified CURRENCY corresponding to the specified amount of DOLLARS. Assume that DOLLARS is a floating-point number, and that CURRENCY is one of POUNDS, DEUTSCHMARKS, FRANCS, PESOS, or YEN.

4. Write a function NNUMS that will compute the number of numbers (i.e., numeric atoms) in an arbitrary list structure. Test your function on enough test cases that you make up to convince yourself that it works. Can you give a formal proof that your function will always work?

Lisp Exercise #2

Symbolic Differentiation[1]

A symbolic differentiation program finds the derivative of a given formula with respect to a specified variable, producing a new formula as its output. In general, symbolic mathematics programs manipulate formulas to produce new formulas, rather than performing numeric calculations based on formulas. In one sense, symbolic manipulation programs are more powerful, since a formula provides answers to a class of problems while a numeric answer applies to only a single problem.

Symbolic differentiation makes a good exercise in Lisp programming because it is easy to do and because it takes advantage of some of the unique features of Lisp. Symbolic differentiation is an easy problem in Lisp because it can be solved using the *Divide and Conquer Strategy* (also called *Problem Reduction Search*):

1. If the problem to be solved is an easy problem, solve it at once.

2. If the problem to be solved is a hard problem, try to solve it in terms of the solutions to smaller *subproblems*; use the problem-solver itself recursively to solve the subproblems. Combine the solutions to the subproblems into a solution for the larger problem.

Symbolic differentiation is guaranteed to be solved by this strategy because:

1. Solutions to the subproblems are guaranteed to provide a solution to the larger problem.

2. The subproblems are always smaller (less difficult) that the original problem; this guarantees that the solution process will terminate.

3. The subproblems are independent, that is, the solution to one subproblem cannot interfere with the solution to another subproblem.

For this exercise, you are to write a function named DERIV with arguments FORM (the formula to be differentiated) and VAR (the variable with respect to which the derivative is taken). FORM is written in Lisp notation, that is, a mathematical operation is written as a Lisp list with the function name first and arguments following.

[1]It is ***not*** necessary for you to know differential calculus in order to do this problem.

Assume that the usual Lisp operators (+ - * /) are used, as well as mathematical functions (SQRT, LOG, EXP, SIN, COS, and TAN). EXPT raises its first argument to the power specified by its second argument. For simplicity, we will assume that all of the binary algebraic operators have exactly two arguments.

The following rules of differential calculus apply; the notation used is d/dx[form] where x is the variable with respect to which the derivative is taken and form is the formula whose derivative is desired.

1. d/dx[c] = 0 , where c is a numeric constant.
2. d/dx[x] = 1
3. d/dx[v] = 0 , where v is a variable other than x.
4. d/dx[u + v] = d/dx[u] + d/dx[v]
5. d/dx[u - v] = d/dx[u] - d/dx[v]
6. d/dx[-v] = - d/dx[v]
7. d/dx[u * v] = u * d/dx[v] + v * d/dx[u]
8. d/dx[u / v] = (v * d/dx[u] - u * d/dx[v]) / v^2
9. d/dx[u^c] = c * u^{c-1} * d/dx[u] , where c is constant.[2]
10. d/dx[sqrt(u)] = (1/2) * d/dx[u] / sqrt(u)
11. d/dx[log(u)] = (d/dx[u]) / u
12. d/dx[exp(u)] = exp(u) * d/dx[u]
13. d/dx[sin(u)] = cos(u) * d/dx[u]
14. d/dx[cos(u)] = - sin(u) * d/dx[u]
15. d/dx[tan(u)] = (1 + $(\tan(u))^2$) * d/dx[u]

Write your functions in such a way that the main function, DERIV, solves the easy cases directly; for each more complex case, DERIV should determine what the operator

[2]We will only consider the case where c is constant.

is and call an appropriate function to take the derivative of a formula involving that operator. (It may be convenient to let one function handle multiple operators whose derivatives are similar.)

Test your program on the following test cases, and on others which you make up. In these examples, the derivative is taken with respect to X.

1. x
2. 3
3. y
4. x+7
5. x*5
6. 5*x
7. $x^3 + 2*x^2 - 4*x + 3$
8. $sqrt(x^2 + 2)$

Values of Derivatives

Write a function DERIV-VAL, with arguments FORM, VAR, and VAL, which will take the derivative of FORM with respect to VAR and return the numerical value of the derivative when VAR has the value VAL. We will assume that any variables other than VAR have values assigned to them outside the call to DERIV-VAL. Test your function on the above examples with the value 7 for x. Note: DERIV-VAL is a *very short* function.

Symbolic Simplification

As you may have noticed, the values returned by DERIV, while mathematically correct, aren't very useful because they contain so much extraneous algebra. It is apparent that in addition to the DERIV program, we need a *symbolic simplifier* which will simplify its result using well-known mathematical identities. In general, algebraic simplifiers may be very complex. However, it is easy to catch some common cases at the point of generation.

Write functions SPLUS, SDIFF, SMINUS, STIMES, and SQUOT to simplify expressions whose top operator is +, -, unary -, *, and /, respectively. Each of these functions will have two arguments, LHS and RHS (left hand side and right hand side). If no simplification can be performed, each function will return as its value the list (<op> LHS RHS), where <op> is the operator associated with the simplifier function. In certain cases, however, a simpler form can be returned. For example, if LHS and RHS are both numbers, the operation can be performed at once[3]. You can also make use of mathematical identities such as x+0=x, x*0=0, x*1=x, etc. Change your derivative functions so that they use the simplifier functions in computing their answers; try the modified version on the test cases given earlier.

Note that it is often easiest to test Lisp functions individually on small test cases before testing them as part of a larger system. For example, the following test cases might be used for SPLUS:

1. (splus 2 3) = 5
2. (splus 'a 'b) = (+ A B)
3. (splus 'x 0) = X
4. (splus 0 'x) = X
5. (splus '(- x) 'x) = 0

The last example above suggests that a really good simplifier would be required to cover a lot of cases. Can the job of writing a simplifier ever be finished? That is, is it possible to write a simplifier that will simplify *any* arithmetic formula into its simplest form?

[3]This is called *constant folding*.

Lisp Exercise #3

Iterative Functions

Recursive definitions of functions which work on (possibly) long lists are elegant; however, they may be computationally expensive because:

1. Function calls take a great deal of time in most languages, including most Lisp implementations.

2. Each *invocation* of a Lisp function requires allocation of a new *stack frame* on the function call stack. With a recursive definition of a list-manipulation program, the stack space required will be proportional to the length of the list.

Using the PROG construct, write iterative versions of the functions LAST, MEMBER, REVERSE, and ASSOC. (Use different names for your versions so that you do not destroy the system versions!) Test your versions to make sure they work. Consider the problems you would encounter if you tried to write (a) APPEND and (b) COPY in the same style as the above functions.

Natural Language Processing

Lisp is the language most often used in research on computer understanding of natural languages (e.g., English). Applications of natural language processing include use of English-like languages for communicating with computer programs (e.g., database question answering, airline reservation systems), and machine translation between different natural languages.

For this exercise, write a set of functions that can parse (assign structure to) simple declarative English sentences. Sentences will be represented as lists of words, e.g.,

```
(THE BIG BLACK DOG CHASED SEVEN CATS)
```

The parts of speech of each word will be stored as a list on the property list of the word under the property PART-OF-SPEECH. Parts of speech include the following:

1. DET -- Determiner (A, AN, THE)

2. ADJ -- Adjective

3. NOUN -- Noun

4. VERB -- Verb

5. VAUX -- Verb auxiliary (IS, ARE, WAS, WERE, BE, BEEN, BEING, HAS, HAVE, HAD, DO, DOES, DID, WILL, SHALL)

6. PREP -- Preposition

Write a function DEFPOS (DEFine Part-Of-Speech) to help you set up the property-list entries needed for the "dictionary" of your parsing program. DEFPOS should take a list of sub-lists as its argument; each sub-list should be a part-of-speech followed by words which have that part-of-speech. For example,

```
    (DEFPOS '((DET A AN THE)
              (ADJ BIG BLACK SEVEN) ...))
```

Remember that the parts-of-speech for each word are stored in a list, since a word may have more than one part-of-speech.

In writing the parsing program, use Global variables SENT for the remaining portion of the sentence and WORD for the current word. Write a function NEXT that will remove the top word from SENT and set WORD to the new current word. Write a function CAT (CATegory) of one argument that will test whether the current word, WORD, has the specified part-of-speech. For example, `(CAT 'NOUN)` will test whether the current word is a noun.

We will use the following simplified grammar for English. In this notation, the left-hand-side of a *production* specifies the name of a phrase, and the right-hand side specifies the components of such a phrase. Phrase names and word categories are specified inside angle brackets. Square brackets around a construct mean that the construct is optional. A * following a construct (called a "Kleene star") means that the construct may be present zero or more times.

```
<S>     =>    <NP> <VP>
<NP>    =>    [<det>] <adj>* <noun> <PP>*
<PP>    =>    <prep> <NP>
<VP>    =>    <vaux>* <verb> <PP>* [<NP>]
```

The first production means that a Sentence consists of a Noun Phrase followed by a Verb Phrase. A Noun Phrase consists of an optional Determiner, followed by zero or more Adjectives, followed by a Noun, followed by zero or more Prepositional Phrases. For simplicity, we will assume that ADJ and NOUN are disjoint and that VAUX and VERB are disjoint. (Other assumptions of disjoint sets may be made as needed for simplicity in this exercise.) We will also assume that the first legal parsing of a sentence that is found by your program is the correct one.

You should write a Lisp function to parse each of the phrases S, NP, VP, and PP. The value of each of these should be NIL if no such phrase is found at the current position of SENT, or an output list if a phrase was successfully parsed; in addition, SENT should be updated so that any phrase which has been parsed is removed from the front of SENT. Your top-level function should be PARSE, which accepts a sentence list as its argument and returns a parse structure for its output.

The parse structure returned by each parsing function should be a list consisting of the phrase name, followed by the main word of the phrase, followed by modifiers:

```
(S SUBJ <NP> PRED <VP>)
(NP <noun> DET <determiner> ADJ <adjectives> MODS <PPs>)
(VP <verb> AUX <vauxes> OBJ <NP> MODS <PPs>)
(PP <prep> OBJ <NP>)
```

For example, the sentence (THE OLD MAN BY THE SEA EATS RAW FISH WITH KETCHUP) should return the following structure:

```
    (S SUBJ (NP MAN DET THE ADJ (OLD)
                MODS ((PP BY OBJ (NP SEA DET THE
                                      ADJ NIL MODS NIL))))
       PRED (VP EATS AUX NIL
                OBJ (NP FISH DET NIL ADJ (RAW)
                        MODS ((PP WITH OBJ (NP KETCHUP DET NIL
                                                ADJ NIL MODS NIL))))
                MODS NIL))
```

Test your program on the following examples:

1. (FISH SWIM)

2. (JOHN LOVES MARY)

3. (A DOG CHASED THE CAT)

4. (JOHN SWAM IN THE SEA)

5. (A BIRD IN HAND IS WORTH TWO BIRDS IN THE BUSH)

6. (I SAW THE MAN ON THE HILL WITH THE TELESCOPE FROM MACYS)

The parsing program necessary to do the above exercise is quite simple. Expand your program to do more. Some suggested expansions are:

1. Eliminate the NIL entries in the output when the corresponding forms are not present in the input sentence. (Easy.)

2. Expand the grammar embodied by your program so that more complex English constructions can be parsed.

3. Write the inverse of the parser: a *generator* program that will produce English sentences from a *parse tree* as produced by your parser. (Easy!)

The general problem of computer understanding of natural language is very difficult. Some of the problems include:

1. Lexical ambiguity. Many words have multiple meanings, as in "The angry pitcher sat on the bench" and "The empty pitcher sat on the table."

2. Ambiguity in parsing. In the sentence "I saw the man on the hill with the telescope from Macys," many different readings are possible depending on what the prepositional phrases are assumed to modify.

3. Definite reference. In the sequence "I had car trouble today. Two tires were flat." the phrase "two tires" is a definite reference to the tires of the car. However, the tires have not been mentioned previously; the reader must have knowledge about cars in order to connect the second sentence to the first.

References:

1. Woods, W. A., "Transition Network Grammars for Natural Language Analysis", *Communications of the ACM,* pp. 591-606, October, 1970.

2. Simmons, R. F. and Slocum, J., "Generating English Discourse from Semantic Networks", *Communications of the ACM,* pp. 891-905, 1972.

3. Novak, G. S., "Representations of Knowledge in a Program for Solving Physics Problems", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 286-291, 1977.

4. Hendrix, G. G., Sacerdoti, E. D., Sagalowicz, D., and Slocum, J., "Developing a Natural Language Interface to Complex Data", *ACM Transactions on Database Systems*, pp. 105-147, 1978.

Lisp Exercise #4

1. Write the function (SUBSET <predicate> <list>), which will return a new list consisting of those elements of <list> that satisfy <predicate>, as a Macro that expands into MAPCAN.

2. In one respect, Lisp programs are rather low-level: access to data structures must be done by particular functions whose names depend on the actual structure of the data (CAR, CADR, GET, etc.). This makes Lisp programs hard to read (since the names are non-mnemonic), error-prone (since the programmer may use the wrong access function), and difficult to change (since a change to data structures requires changes to code in many places).

One way to make Lisp programs relatively independent of the actual formats of Lisp data structures is to write access functions for all accesses to the structures. Another way is to define a language for describing Lisp data and write functions that will translate a request for access to a specified field of a structure into code to perform the access. This exercise is to write a set of functions to perform such a translation. Conceptually, this is similar to the declaration and use of Record structures in languages like Pascal. The program you are to write is comparable to the part of the Pascal compiler that generates code for access to Records.[4]

First, we must define a language for describing Lisp structures, analogous to the language for describing record types in Pascal. Our Structure Description Language is defined recursively as follows:

1. Each of the following Basic Types is a legal Structure Description:

ATOM	
INTEGER	
REAL	
NUMBER	(either INTEGER or REAL)
STRING	
BOOLEAN	(either T or NIL)

2. Composite Structures: Structured data types composed of other structures

[4]This exercise specifies part of the GLISP compiler. It is not a difficult program, but students usually find it hard to understand; perhaps it is described badly. At any rate, it is included here for those who might be interested.

are described using the following structuring operators:

a. **(CONS** $<sd_1>$ $<sd_2>$)
The CONS of two structures whose descriptions are $<sd_1>$ and $<sd_2>$.

b. **(LIST** $<sd_1>$ $<sd_2>$ ... $<sd_n>$)
A list of exactly the elements whose descriptions are $<sd_1>$ $<sd_2>$ ... $<sd_n>$.

c. **(LISTOF** <sd>)
A list of zero or more elements, each of which has the description <sd>.

d. **(ALIST** ($<name_1>$ $<sd_1>$) ... ($<name_n>$ $<sd_n>$))
An association list in which the atom $<name_i>$, if present, is associated with a structure whose description is $<sd_i>$.

e. **(ATOM** **(PROPLIST** ($<pname_1>$ $<sd_1>$) ...
($<pname_n>$ $<sd_n>$)))
This describes an atom with its property list. Each property name $<pname_i>$ is treated as a property list indicator as well as the name of the substructure.

3. The name of an atom which has a Structure Description stored on its property list under the property STRUCTURE is a valid type name for use in a structure description. For example, if the VECTOR structure has been defined, as illustrated below, then **VECTOR** is a legal structure description.

4. Any substructure can be given a name by enclosing it in a list prefixed by the name: (<name> <sd>). The names used in forming composite types (given above) are treated as reserved words and may not be used as names.

Write a function DEFSTR that accepts zero or more named Structure Descriptions, each of which is assumed to be quoted (i.e., the user does *not* quote it), and stores the Structure Description on the property list of the structure name under the indicator STRUCTURE. DEFSTR should be written using the &rest argument keyword. An example of a call to DEFSTR with examples of legal structure descriptions is shown below:

```
(DEFSTR

(VECTOR  (CONS (X INTEGER) (Y INTEGER)) )

(CIRCLE  (LIST (START VECTOR) (RADIUS REAL) (COLOR ATOM)) )

(STUDENT (ATOM (PROPLIST (NAME STRING)
                         (MAJOR ATOM)
                         (GRADES (LISTOF INTEGER)))) )
  )
```

The first declaration above states that a VECTOR structure consists of the CONS of two items, X, which is an integer, and Y, which is an integer. A CIRCLE object is a LIST of three items, the START, which is a VECTOR, the RADIUS, which is a real number, and the COLOR, which is an atom. An example of a VECTOR, as printed by Lisp, is `(3 . 4)`; an example of a CIRCLE is `((20 . 30) 10.0 BLUE)` .

Write a function STRGET, with arguments STRUCTURE and FIELD, which will produce Lisp code to get the field FIELD from an object whose structure is described by STRUCTURE. STRGET should produce as its result a list of two items:

1. Code to get FIELD from the value of the "variable" *OBJ* . For example, if STRGET were called with arguments `VECTOR` and `Y`, it should return `(CDR *OBJ*)` as its code result.

2. The Structure Description of the result. For the above example, the result type would be `INTEGER`.

If FIELD cannot be found within STRUCTURE, STRGET should return NIL. STRGET should be designed to call itself recursively; SUBST can be used to combine results.

Write a function STRPUT with arguments STRUCTURE, FIELD, and VALUE. STRPUT will generate code to store VALUE into FIELD of *OBJ*, which has the structure STRUCTURE. STRPUT should use STRGET, then modify the result produced by STRGET to store a value rather than retrieve one. Note that there is a unique "put" function corresponding to each "get" function (for example, RPLACA is the "put" function which corresponds to CAR), and that only the outermost function call of the "get" code needs to be converted into "put" code. You will need to write a "get" and "put" function for the ALIST structure; you may assume that an ALIST structure will never be NIL.

Write Macros `(GETSTR <code> <structure> <field>)` and `(PUTSTR <code> <structure> <field> <value>)` that will produce code to perform the specified operations. The arguments will be assumed to be quoted, i.e., the user will not quote them. The <code> argument to these functions will usually be a variable name. GETSTR and PUTSTR will call STRGET and STRPUT, and substitute <code> for *OBJ* in the result. We will assume that the result returned by the code generated by PUTSTR will not be used, i.e., may be arbitrary.

The following examples illustrate the results to be generated by your functions:

```
(STRGET 'VECTOR 'X)              =>   ((CAR *OBJ*) INTEGER)

(STRGET 'CIRCLE 'COLOR)          =>   ((CAR (NTHCDR 2 *OBJ*)) ATOM)

(STRGET 'STUDENT 'GRADES)        =>   ((GET *OBJ* (QUOTE GRADES))
                                        (LISTOF INTEGER))

(STRPUT 'VECTOR 'X 3)            =>   (RPLACA *OBJ* 3)

(GETSTR V VECTOR X)              =>   (CAR V)

(PUTSTR V VECTOR X 5)            =>   (RPLACA V 5)

(GETSTR S STUDENT GRADES)        =>   (GET S (QUOTE GRADES))

(PUTSTR C CIRCLE COLOR 'BLUE)    =>   (RPLACA (NTHCDR 2 C)
                                              (QUOTE BLUE))

(PUTSTR S STUDENT MAJOR 'CS)    =>   (SETF (GET S (QUOTE MAJOR))
                                             (QUOTE CS))
```

Hints:

1. Write STRGET so that it works for any structure using only the CONS structuring operator. Next, add the LIST operator, and then add the rest.

2. No substructures can be retrieved from a LISTOF structure; if a LISTOF is reached, STRGET should return NIL.

3. To quote something in your generated code, enclose it in a list with QUOTE as its first element: `(QUOTE x)`.

4. Use functions in addition to the ones required by this assignment as you see fit.

Test your functions on the above examples, and enough additional examples to verify that they work.

Those who want an additional challenge may write a function to interactively acquire a data structure from a user based on its structure description.

Lisp Exercise #5

1. Write a function that will destructively insert a new item into an existing sorted list. The function should be called `(DINSERT NEW L AFTERFN)` , where `NEW` is the new item, `L` is the existing list, and `AFTERFN` is a function that compares two items and returns T iff the first argument should appear after the second in the list. Examples of functions that could be used as AFTERFN are $>$ and $<$. Test your function on lists of numbers and on lists of atoms, using alphabetic sorting for atoms; test it for both ascending and descending order. Make sure it works for all possible cases!

2. Ordinary programming is *Procedure-Centered*, i.e., a program consists primarily of calls to named procedures, each of which expects arguments of relatively fixed types. In *Object-Centered Programming* (or *Object-Oriented Programming*), data objects are considered to be active entities that are manipulated by sending *messages* to them. For example, in ordinary Lisp one would call the function PRINT to print a data object; in object-centered programming, one would send to the object a message that says (in effect) "print yourself". Object-centered programming originated in the programming language SIMULA, which is used for computer simulations; one may think of the data contained in the object as representing the *state* of the object, and the set of messages an object can understand as representing its *behavior* in response to changes in its environment. Object-centered programming has been popularized by the language SMALLTALK (*BYTE Magazine*, August, 1981; *SMALLTALK: Language and Implementation* by Goldberg and Robson, 1983).

In pure object-centered programming, the only way to access an object is through messages. The internal implementation of the object is hidden. Instead of extracting a piece of data, say `X`, from the data structure of an object, one sends it a message asking "what is your `X`?", and the `X` value is returned as the value of the message. In order to set the value of the object's data, one likewise sends it a message saying "please set your `X` to <value>". The advantage of this is that objects of very different kinds can be "plugged together" if they understand the appropriate messages, even though their internal implementations might be very different. For example, one implementation of a Vector might store X and Y directly; another implementation might store R and THETA and compute X and Y from R and THETA. With object-centered programming, a program that uses vectors need not worry about which method of implementing the vectors is used. New kinds of objects can be added to an existing system in a modular way. For example, in an object-centered system, a new type of printable object can be added to the system by making that object understand the PRINT message; in an ordinary Lisp system, it would be necessary to modify the system PRINT routine.

In a large system, there might be many different objects of the same kind. For efficiency, it is natural to store the *state* (i.e., data values) with each individual object, but to group together the set of common *behavior* (i.e., message responses) that they all share. A *Class* is a data structure that describes a class of similar objects. A Class structure will contain a list of the names of "variables" associated with each instance of the class, and an association list that associates message names (or *selectors*) with the names of functions that implement the messages.

Additional power can be obtained by having a *hierarchy* of classes. For example, if there are classes DOG and CAT, it may be useful to have a class MAMMAL that contains the behaviors common to all mammals. MAMMAL is a *superclass* of DOG and CAT; each class will have a list called SUPERS that lists all its superclasses. A distinguished class, OBJECT, will be assumed to be the superclass of all classes that do not have any SUPERS specified by the user. While a class may have multiple superclasses, each instance object will be a member of exactly one class.

In case you haven't guessed, you will be privileged to write an object-centered system within Lisp for this exercise.

We will assume that an Object is implemented as a Lisp atom (usually a newly generated atom) with the "variable" values stored on the property list of the atom using the "variable" names as property list indicators. The Class of an object will be stored under the property CLASS; each object will have exactly one class. (We will assume that CLASS cannot be a variable name.) The name of an Object will be its Class name followed by an integer that is incremented for each new object generated. (GENSYM can generate such atoms.)

Each Class will be represented by the atom that is the Class name. Properties of each class will be stored on its property list, as follows:

1. SUPERS is a list of the superclasses of the class; a class can have multiple superclasses.

2. VARIABLES is a list of the names of "instance variables" contained in each object that is an instance of the class.

3. MESSAGES is an association list that associates message *selectors* with the names of corresponding Lisp functions that execute the actions associated with the messages.

Write a function DEFCLASSES that will define a set of object classes. DEFCLASSES will take a list of class descriptions, which are not evaluated. Each class description consists of the name of the class, a list of its SUPERS, a list of its VARIABLES, and its association list of MESSAGES. If the SUPERS specification for a class is NIL, DEFCLASSES should fill in the list (OBJECT). The following example illustrates some class definitions.

```
(DEFCLASSES

(PHYSICAL-OBJECT NIL NIL ((DENSITY PHYSICAL-OBJECT-DENSITY)))

(SPHERE NIL NIL ((VOLUME SPHERE-VOLUME) (AREA SPHERE-AREA)))

(PLANET (PHYSICAL-OBJECT SPHERE)
        (MASS RADIUS)
        NIL)

(ORDINARY-OBJECT (PHYSICAL-OBJECT)
                 NIL
                 ((MASS ORDINARY-OBJECT-MASS)))

(PARALLELEPIPED NIL NIL ((VOLUME PARALLELEPIPED-VOLUME)))

(BRICK (ORDINARY-OBJECT PARALLELEPIPED)
       (WEIGHT LENGTH WIDTH HEIGHT)
       NIL)

(BOWLING-BALL (ORDINARY-OBJECT SPHERE)
              (TYPE WEIGHT)
              ((RADIUS BOWLING-BALL-RADIUS)))
)
```

In this example, the class PHYSICAL-OBJECT defines a single message, whose selector is DENSITY; the density is computed by the function PHYSICAL-OBJECT-DENSITY. A PLANET has PHYSICAL-OBJECT and SPHERE as its superclasses; it has two variables whose values comprise its state, MASS and RADIUS.

Write a function NEW that will make a new object of a specified class. The arguments to NEW are the class name (not evaluated) followed by variable/value pairs; the variable names will not be evaluated, while the values will be evaluated. An example call to new is:

```
(NEW BOWLING-BALL WEIGHT 5.0 TYPE 'ADULT)
```

NEW should make a new atom from the class name, put in the CLASS pointer, and add the specified variable values to the new atom.

The sending of messages to objects will be specified by calls to the function SENDM. The format of a call to SENDM is:

```
    (SENDM <object> <selector> <arg₁> ... <argₙ>)
```

where `<object>` is the object to which the message is sent, `<selector>` is the name of the message selector, and the `<arg_i>` are optional arguments of the message. All arguments of SENDM are evaluated. SENDM will first try to find the appropriate *response* function to respond to a message with the given `<selector>` for the class of the specified object; then, SENDM will execute the response function, giving it as arguments the `<object>` to which the message was sent (the "SELF" argument) and the arguments `<arg_i>` specified in the SENDM call. For example, if the variable `P` has been set to point to a PLANET object, the message `(SENDM P 'DENSITY)` would cause the message selector DENSITY to be looked up starting in the class of P (i.e., PLANET); since DENSITY is not defined for PLANET, it would then be looked up for superclasses of PLANET, and would be found in the class PHYSICAL-OBJECT. The corresponding response function, PHYSICAL-OBJECT-DENSITY, would then be executed with the argument specified by P (i.e., P's value, the PLANET instance object).

The functions that implement messages will often use SENDM themselves. For example, PHYSICAL-OBJECT-DENSITY might be implemented as:

```
    (DEFUN PHYSICAL-OBJECT-DENSITY (SELF)
       (/ (SENDM SELF 'MASS) (SENDM SELF 'VOLUME)))
```

This message function expresses the general rule that the density of an object is defined as its mass divided by its volume. By writing this function in object-centered form, it can work for objects that are implemented in many different ways. For example, we will assume that a BOWLING-BALL can have two TYPEs, ADULT and CHILD, and that the radius of a BOWLING-BALL is 0.1 meters if it is an ADULT model or 0.07 meters if it is a CHILD model; the RADIUS message function for BOWLING-BALL will define this convention. You should write the remaining message functions shown in the DEFCLASSES example above; we will assume the use of metric (MKS) units for measurements. Test your functions by executing the DENSITY message on objects that are instances of PLANET, BRICK, and BOWLING-BALL.

It is conventional in object-centered systems to let "variable" names be the selectors of messages that return the "variable" values; for example, `(SENDM P 'RADIUS)` would return the value of the RADIUS for a PLANET object P. Likewise, a message whose selector is a variable name followed by a special character (say, asterisk) may be used to set the value of a variable, for example, `(SENDM P 'RADIUS* 4000000.0)`. The user could be required to write a pair of message functions for each variable name and to define the corresponding message selectors for each class; however, this would be a nuisance. Modify your SENDM procedure(s) so that a message whose selector is a variable name or a variable name followed by an asterisk is interpreted as accessing or redefining the value of the variable, respectively. The order of testing should be to look for a given name first as a message selector, then to look for it as a "variable" name, then to look for it in superclasses of the current class (depth-first). We will ignore the possibility that the same message selector might be defined in more than one superclass of a given class; the first matching name that is found will be assumed to be the correct one.

The radical view of object-oriented programming is to treat *all* data items as objects -- even simple data such as integers. Modify your function SENDM to permit this for Lisp data, as follows. If the <object> argument to SENDM is not an Object (i.e., a Symbol atom with a CLASS property), make the class of the object default to the name of its Lisp data type. (Common Lisp provides functions for testing types, such as INTEGERP, FLOATP, STRINGP, etc.) Use DEFCLASSES to define the correspondence between message names and system function names for some of the basic types so that messages to basic data objects will be interpreted appropriately as calls to the corresponding system functions. After you have done this, for example, it should be possible to execute:

```
    (SENDM 3 '+ 4)
```

and get the expected value, 7.