Introduction to Lisp on the Lisp Machine

Draft formatted on 11 Jan 84 at 09:26

Draft -- For Internal Use Only

We appreciate any comments on the organization, technical completeness, and technical accuracy of this draft. (Comments about weakness of the product design should go to the appropriate mailing list instead.) Thanks.

Name:________________________, Date:____________

Introduction to Lisp on the Lisp Machine
#

January 1984

This document corresponds to Release 4.5.

This document was prepared by the Documentation Group of Symbolics, Inc.Principal writer(s): Jonathan Balgley, Lois Flynne, and Allan Wechsler

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Table of Contents

Introduction to Lisp Machine Programming

Course Key Notes

Symbolics Educational Services

Succinct summaries of the main points of the lectures and labs, with lots of practice exercises & illustrative examples.

1. LISP WORLD OBJECTS Their Nature & Representation

NUMBERS
SYMBOLS
CONSES
FUNCTIONS

NUMBERS
Magnitude

1.1 Numbers

Numbers, poor dears, are dull beasts. The only interesting thing about them is their **MAGNITUDE**

The **magnitude** of a number is depicted in decimal digits. However, numbers do not have any intrinsic base. Bases have to do with the external representation of numbers.

SYMBOLS
Name
Value
Function Value
Property List

1.2 Symbols

A Symbol is an interesting and versatile creature which has a

- **Name**

and may have a

- **Ward**

- **Functional Ward**

- **Property List**

Symbols are born into the Lisp World with their names enscribed.

Later, a symbol may acquire a ward, or, a functional ward, or a property list, or not. Wards, functional wards, and property lists are relationships that symbols have with other Lisp objects.

All three of these relationships are one-way: that is, given a symbol you can find out what its ward is, but given any old Lisp object you can't find out what symbols it is the ward *of.*

Any of these relationships can be changed. The ward of a symbol may be any Lisp object ... including the symbol itself. The functional ward of a symbol ought to be a function object, but may be any Lisp object. We will ignore the property list for now.

CONSES & LISTS
Car
Cdr

1.3 Conses

Conses are most useful creatures. They have a

- **CAR**

- **CDR**

Conses must have a **CAR** and a **CDR**. The car or cdr of a cons may be any Lisp object, including another cons, or even the cons itself.

Cars and cdrs are relationships that a cons has with other Lisp objects. The relationship is the same as the relationship that symbols have with their wards, except that a cons is required to have both a car and a cdr.

1.3.1 Using Conses

Conses are often used to link or relate lots of objects together. When we use both cars and cdrs to link objects together the resulting structure is called a *tree*. Trees have only conses as their internal "support" and the objects of interest are at the "fringe".

Another way we link objects together is by using cdrs to link conses and make the car of each cons be the object of interest. This structure is called a *list*.

The formal definition of a list is:

- Either the symbol whose name is NIL,

- Or (yes, recursively) a cons whose cdr is a list.

Lists are the backbone of Lisp, which is an acronym for **LIS**t **P**rocessing.

1.3.2 Describing Lists

We will continue the backbone metaphor. The conses which make up a list will be called the *backbone*.

Each individual cons in the backbone will be called a *vertabra*.

The car of each vertabra is an *element of the list*.

The *length* of the list is equal to the number of elements or vertabra in the list.

Note that it would be equivalent to use the cars of the conses as links and the cdrs as the objects of interest. Lisp has a *convention* which asks you to use cdrs.

FUNCTIONS
Definitions
Actions

1.4 Functions

Functions are the actors, of the Lisp World, the movers, the shakers, the changers, the doers, the enquirers, the informers, the examiners, the calculators.

Functions have

- **DEFINITIONS**

which prescribe

- **ARGUMENTS**
- **ACTIONS**

or, "**What is to be done in what way with what to whom.**"

- The function definition specifies the number of arguments, and the action to be taken with those arguments.
- There may be zero, or, more arguments which may be any Lisp World object.
- Functions are *run* or *called* with Lisp objects as arguments.
- The function has a right to expect to be called with the correct number and type of arguments. If this is not the case, an error will be signalled.
- The function performs the action specified by the definition, with the supplied arguments and usually *returns* a *result.*
- The **result** may be any Lisp object.
- If something goes wrong, an *error* will be *signalled.* In this case, a function may not return a result. (It depends on how you choose to handle the error.)
- Some functions are "environmentally clean". They are only used for *value*, that is, for the Lisp object they return. Other functions have an effect on the Lisp world and are not environmentally clean. These functions are used for *effect,* but may also be used for the result that they return.

Definitions may be

- **Interpreted** Definitions, expressed in Lisp Code.
- **Compiled** Definitions, expressed in Compiled Code
- **Micro-Code** Definitions

We won't distinguish between these three kinds of definitions for a while.

1.5 Nomenclature

Some of the words we have used were invented for the purpose of the course, or have been adopted as better than previous words. In this section we list possible words you may see or hear for the same concepts.

Symbol

- *Atom*. The definition of an atom is anything that is not a cons. People will often use "atom" when they really mean "symbol".

Ward

- *Value.* Means exactly the same thing as ward. We prefer "ward" because it eliminates multiple uses of the word "value", and because it implies a relationship.

- *Value cell.* Same as above, but has to do with the Lisp Machine implementation of symbols.

- *Binding*. The traditional word for the ward concept. Unfortunately, it has a double meaning. We will use "to bind" for only one meaning which will be discussed later.

"...has no ward."

- ... *is unbound.* From the word "binding," this is exactly the same as saying "has no ward."

Functional Ward

- *Functional Value* and *Function Value Cell.* Same as above.

- *Function.* Not a good word, because a symbol's functional ward may not actually *be* a function.

Cons

- *Dotted Pair.* This description has to do with the printed representation of conses, which will be discussed soon.

1.6 Games & Practice with LISP WORLD OBJECTS

GAMES & PRACTICE

Simple Manipulations with Lisp World Objects

Figure 1. Lisp World 1.

Question: What is the ward of the car of this cons?

Answer: The ward of the car of this cons is.....

MORE GAMES & PRACTICE

Using Lisp objects to model real objects

Often we use Lisp objects to *model* real world objects. Try to come up with a correlation between some of these real objects and relationships that they have, with Lisp objects and Lisp relationships.

- mice (the little black things with three buttons)
- boats
- trains
- planes
- a deck of cards

Here is a sample of a correlation between a real object - a tiddlywink - and Lisp objects:

Real	Lisp
Tiddlywink	Cons
X-coordinate of tiddlywink	Car of cons
Y-coordinate of tiddlywink	Cdr of cons
Moving the wink	Changing car and/or cdr of cons
Squopping[1] a wink	Making the car and cdr of both winks be the same

[1] *Squopping* means getting one wink to land on another.

2. SIMPLE FUNCTIONS

Fetchers and Examiners
Constructors and Changers
Arithmetizers
Testers

2.1 Essential Characteristics-- Review

- Functions are Lisp World objects.

- They have **definitions** which prescribe what **actions** they will take with what **arguments**.

- The **arguments** of a function may be any Lisp World object. Functions may take zero or more arguments.

- Functions are *called* or *run* with their arguments. These must be of the correct type and number.

- The *called function* carries out its prescribed **action** with the given arguments.

- It *returns a result*, or *signals an error*

- The *result* may be any Lisp object.

- **All functions return a result,** which can be used in some way. (*Called for value*)

- **Some functions effect a change** in the Lisp World as well. (*Called for effect*)

- Functions do not have names.

- The name of the symbol that has the given function as its functional ward is used to refer to the function.

SYMEVAL
FSYMEVAL
CAR
CDR

2.2 Fetchers & Examiners

These are *benign* functions. They effect no changes in the Lisp World.

SYMEVAL & FSYMEVAL

- take **one argument** which must be a **symbol**
- return the **ward** or **functional ward** of the symbol as the result
- signal an error if
  - the number or type of argument supplied is incorrect

    For example, if you run the function **symeval** with two symbols, or if you run it with a cons.
  - the symbol does not have a ward or functional ward

CAR & CDR

- take **one argument** which must be a **cons**
- return the **car** or the **cdr** of the cons as the result
- signal an error if the number or type of argument supplied is incorrect

SET
FSET
MAKUNBOUND
FMAKUNBOUND

2.3 Changers & Constructors

2.3.1 Symbol Values & Function-Values

These functions **produce changes** in the Lisp World. Of course, they also return a result which you can use.

SET & FSET

- take **two** arguments
  - the first argument must be a **symbol**
  - the second argument to **SET** may be **any Lisp World object**
  - the second argument to **FSET** ought to be a **function object**, but this is only a convention. (A strongly encouraged convention, but still a convention.)
- install the second argument as the **ward** or **functional ward** of the first argument, changing the state of the Lisp World
- return the second argument as the result
- signal an error if the number or type of argument supplied is incorrect.

WARNING: No type-check is made to make sure that the second argument to FSET is a function-object.

MAKUNBOUND & FMAKUNBOUND

- take **one argument** which must be a **symbol**
- removes the ward or functional ward, if any
- take no action if the symbol has no ward or functional ward
- return the symbol as the result
- signal an error if the argument is not a symbol or there are too many or too few arguments.

RPLACA
RPLACD
CONS

2.3.2 Conses & Lists

These functions **produce changes** in the Lisp World also.

RPLACA & RPLACD

- take **two arguments**
  - the **first** argument must be a **cons**
  - the second argument may be any Lisp object
- replace the CAR of the first argument or the CDR of the first argument with the second argument
- return the **changed cons** (the first argument) as the result
- signal an error if the first argument is not a cons or if there are too many or too few arguments.

CONS

- takes **two arguments** which may be **any Lisp World objects**
- creates a **new cons** with the first argument as the car and the second argument as the cdr
- returns the **new cons** as the result
- signals an error if there are too many or too few arguments.

+
-
*
//
^

2.4 Arithmetizers

+, -, *, //, ^ are the work-horse arithmetic functions. They all

- take **numbers** as arguments;
  these numbers may be fixed point, or, floating point numbers

- have **no effect on the Lisp World**
  except that sometimes they may have to create a numeric value. This is irrelevant, as the identity of a number is seldom used for anything.

- return a **number** as the result;
  the number will be floating point if any of the arguments was a floating point number. Otherwise, it will be fixed point. // and ^ both truncate towards 0 as opposed to rounding down.

- signal an error if their arguments are of the wrong type or the wrong number

+ and *

- take **zero, or, more numbers** as arguments

- If you give them **no arguments**

    - + returns 0

    - * returns 1

- If you give them **one, or, more arguments**

    - + returns the **sum** of its arguments... + starts with 0 and adds in all of its arguments one at a time

    - * returns the **product** of its arguments... * starts with 1 and multiplies in all of its arguments one at a time

- and //

- take **one, or, more numbers** as arguments
- If you give them **one argument**
    - - returns the negative of its argument
    - // returns the reciprocal of its argument
- With **two, or, more arguments**
    - - subtracts the sum of the rest of its arguments from its first argument and returns that as the result
    - // divides its first argument by the product of the rest of its arguments and reurns that as its result

^

- takes exactly **two numbers** as arguments
- raises the first number to the power of the second number
- returns that as its result

SYMBOLP, NUMBERP, LISTP, FUNCTIONP
EQ, EQUAL, =
NOT, NULL

2.5 Predicates

Predicates are used for comparing two or more Lisp objects in some way. A predicate will return the symbol NIL if the comparison is *not* true. It will return something other than NIL if the comparison *is* true.

A common thing to return is the symbol T. Some predicates may return a more interesting result which you could then use. Remember, anything except NIL means true.

The names of functions which are predicates will often end in **-p**, to indicate to the reader that this is a predicate. (Convention!)

SYMBOLP, NUMBERP, LISTP, and FUNCTIONP

- take one argument, which may be of any type
- are used to determine the type of the argument
- return T if the argument is of the right type, and NIL if it is not
- LISTP does **not** tell if its argument is a list. It will return T if its argument is any **cons**. Yes, it's badly named.

EQ, EQUAL, and =

- all take two arguments

- are use to determine whether the two arguments are equal for some meaning of the word "equal".

EQ is used to determine whether the two arguments it was given are really **the same object**. This is most interesting when you ask this question in terms of relationships of the two objects. For example, it is reasonable to ask whether the ward of one symbol is the same object as the ward of another symbol.

EQUAL is used to determine whether two structures of conses have the same "shape" and elements. This can be formally defined by saying

- objects other than conses are EQUAL only if they are EQ,

- and conses are EQUAL if the respective CARs and CDRs are EQUAL.

Yes, this is non-intuitive. It's much more common to use EQ rather than EQUAL.

= is used to compare two numbers. It is not important to determine whether or not two numbers are EQ, because you should only be concerned with their magnitude.

NOT and NULL

NOT and NULL are synonomous. They invert the meaning of a test. That is, they return the symbol T if their argument is NIL. Otherwise they return NIL. The reason for having both is for clarity of reading.

Other predicates are described in the Lisp Machine Manual, pages 8 - 12 and pages 94 - 96.

2.6 Games & Practice with FUNCTION CALLING

Indicate what Lisp object is returned, and what side effects happen when the following functions are run with the given arguments.
Function: SET

Arguments: RALPH, 5

Function: RPLACD

Arguments: (A . B), NIL
-Some of these exercises will require figures-

3. PRINT & READ & Printed Representations

PRINT
READ

3.1 Print & Read

3.1.1 Communications Functions

PRINT and **READ** are two of the functions that handle communication between the internal Lisp World and the external world.

- A *printed representation* is simply a string of characters used to represent a Lisp World object.
- **PRINT**
    - translates internal Lisp World objects into external *printed representations.*
    - requires at least one argument which may be any Lisp object
    - may take a second argument described below
    - returns its first argument
- **READ**
    - translates external *printed representations* into Lisp World objects
    - creates new objects, if necessary under the rules
    - requires no arguments, but may take one as described below
    - returns the object represented by the *printed representation* that it reads in
- The rules as to what characters represent what are understood by **PRINT** and **READ**.
- Only numbers, symbols and conses have *first class* printed representations.
- Other objects, such as functions, have *second class* printed representations. Such objects can be printed out, but the printed representation cannot be understood by **READ**. These objects need to be attached to a symbol as the ward or functional ward of the symbol. Then you can use the printed representation of the symbol to get to it.

3.1.2 Communications Channels

You can optionally designate which output sink you want PRINT to print on, and which input source you want READ to read from. The defaults are the console display and the keyboard repectively.

3.1.3 Effects on the Lisp World

PRINT is environmentally clean. It has no effects on the Lisp World.

READ, however, usually does produce effects. These effects are summarized by the data type of the printed representation "given" to READ.

- **Symbols:**
    - READ maintains a data base of symbols whose printed representations it has seen.
    - When it reads the printed representation of a symbol, it
        - checks this data base to see if it has seen it before
        - If it hasn't, it creates a new symbol, inscribes the printed representation thereon as the symbol's name and adds it to its data base. This is known as *interning* the symbol.
    - returns either the new symbol, or the symbol that it found in its data base
- **Conses:**
    - READ always creates new conses, even if the printed representations look identical.

Printed representation of NUMBERS

3.2 Numbers

The printed representation of a number is usually the base 10 numeral which represents its magnitude. Scientific notation and other bases can be invoked.

not true !
base 8
unless
base is set differently

Printed representation of SYMBOLS

3.3 Symbols

3.3.1 Simple Printed Representations

In **most cases,** the printed representation of a symbol is simply its name, the set of characters branded on.

Most cases means those symbols whose names are made up of characters chosen from a limited set... upper-case letters, certain punctuation characters, e.g. &%!<>?$=+*, and numbers, as long as there is at least one non-number in the name.

3.3.2 Complicated Representations

In the **rest of the cases,** the printed representation is complicated. The rest of the cases cover the following:

- Symbols whose names include funny characters such as
  - Lower case letters
  - White space characters
  - Characters that have special meanings like ( ). " ' , '/|#
- Symbols whose names might be mistaken for the printed representation of another kind of object, such as a number or a cons.

The **Rules** for handling such cases are as follows:

- **Slashify**
    - If there are any | characters in the name, then a / is placed in front thus /|.
    - Similarly any / characters are "slashified" thus //.

- **Enclose in Bars**

  Finally, if there are other special characters, or the name looks like another type of object, then the whole name is "barred" thus, |59.76|, |It'sIt|, |(fred/|FRED)|.

<u>Read</u>

```
(setq x |ab . cd|)
        |ab . cd|
(setq y |AB . CD|)
        |AB . CD|
(setq z  AB//./CD)
(eq y z)
 T
```

Printed representation of
CONSES & LISTS

3.4 Conses & Lists

Conses may be represented in *dot-notation*, or in *list-notation*. READ understands both notations. PRINT will always use list notation if possible. List notation is preferred.

The printed representation of a cons proceeds by the following algorithm.

1. Dot-Notation Representation

    - Left parenthesis
    - PR of CAR of CONS
    - Space
    - Period
    - Space
    - PR of CDR of CONS
    - Right parenthesis

2. List Notation Representation

    Note that this is just an optimization of the above algorithm.

    - Start with your basic dot notation printed representation
      (A . (B . (C . (D . NIL))))

    - Erase any occurrences of <space>.<space>NIL
      (A . (B . (C . (D))))

    - Whenever you encounter any match for the following pattern..
      .<space>(<somatorother>)
      erase the dot and the leading left parenthesis and the corresponding right parenthesis. (A B C D)

      Example: (X . (Y))=====>> (X Y)
      (X . (Y . Z))======>>(X Y . Z)

Printed representation of FUNCTIONS

3.5 Functions

Functions are totally uncouth and declasse when it comes to printed representations. PRINT will print out a printed representation if you insist. But the printed representation is quite incomprehensible. If you try to type this in, READ will refuse to read it. It's that second class!

Of course, that's why symbols have functional wards. You can use the name of a symbol to get to a particular function.

3.6 Games & Practice with PRINTED REPRESENTATIONS

1. Pretend that you are playing the job of the function READ. The following printed representations are typed in. Draw the graphic representation of the Lisp object that you return.

    a. **(((a . b)) c . d)**

    b. **|abc/d/||**

    c. **(a b (c))~~)~~**

    d. **(a b (c) (d) /1 3)**

    e. **(setq a 5)**

    f. **((((a) b) c) c)**

    g. **//**

2. Now pretend that you are doing the job of the PRINT function. Write the printed representation you will display when given the object shown in each of the following figures.

    -some figures-

4. EVAL & EVALUATION of FORMS

Modelling & Forms
Problem Solving &
Evaluation

4.1 Modelling & Problem Solving-- Forms & Evaluation

Problem solving consists of

- constructing a model of the problem
- manipulating the model
- producing and testing the result

Problem solving in Lisp involves

- creating a model of the problem in the form of a *form*
- evaluating that *form*
- returning a result or signalling an error

4.1.1 Forms

A *form* is any Lisp Object that is intended for *evaluation.*

4.1.2 Evaluation

Evaluation is the process carried out by the function EVAL.

- EVAL is the Lisp Interpreter. It is the very heart of Lisp.
- It takes **one argument,** a *form,* which it evaluates.
- It is the only function that *evaluates* its argument.
- All other functions receive their arguments as they are, usually already evaluated by EVAL.
- EVAL is the only function that understands the *form* model.

Simple Forms
Numbers
Symbols

4.2 Evaluation of *Simple Forms*

4.2.1 Numbers

If the *form* is a number, EVAL returns that **number** as the *value of the form* (or, said another way, as the *result* of running EVAL.)

4.2.2 Symbols

If the *form* is a symbol, EVAL returns the **ward of the symbol** as the *value of the form.*

If the symbol has no ward, an error is signalled.

Compound Forms
Operator &
Operands

4.3 Evaluation of Compound Forms

A *compound form* is a form that is a **list** of one or more elements.

- The first element of a compound form is called the *operator.*

- The rest of the elements, if any, are called the *operands.*

4.3.1 Operators

There are two kinds of operators--

- **functions**

- **special operators**

If the operator is neither a function nor a special operator, EVAL will make the functional ward of the operator be the operator and will keep doing this until it

- gets a function or special operator or

- gets an error.

4.3.2 Functions

Functions are Lisp World objects as described. Usually, they are referenced through a symbol name. Therefore, the first element of the form handed to EVAL is most often a symbol that has a function as its functional ward. Sometimes it is a list which constitutes an anonymous function definition.

4.3.3 Special Operators

Special operators should not be confused with functions, though symbol names are used to reference special operators, and the functional ward of the symbol is used.

Special operators are like irregular verbs. The evaluation rules differ from case to case. There is no general rule as there is for functions.

There are two main types of special operators.

- **Wired-in Special Operators**
  EVAL maintains "exception" rules for each special operator of this class.

- **Macros**
  The functional ward of the symbol indicates that the operator is a macro. In this case, the form is expanded into a new form according to the particular rules of that macro, and then the new form is evaluated.

  We will not discuss this further. Just understand that macros, like all other special forms, each have their own rules of evaluation.

4.3.4 Operands

Rules for the evaluation of *operands*, if any, are prescribed by the type of operator.

- If the operator is a **function**
  - Operands, if any, are evaluated, **in order**, following the same rules for evaluation of simple and compound forms
  - If all operands are successfully evaluated without error, the results of evaluating each operand are handed to the function as its arguments
  - The function is then run with these arguments to produce the result of the evaluation of the compound form.
- If the operator is a **wired-in special operator**, the rules for that given operator are followed with respect to evaluation of operands. For example,
  - The special operator QUOTE requires that the operand be left unevaluated
  - The special operator SETQ requires the first operand be left unevaluated, while the second operand is evaluated.
- If the operator is a **macro**, evaluation of operands is dependent upon the form to which the macro expands.
- For example, a form in which the operator is the macro IF expands to a form in which the operator is the wired-in special operator COND.

READ
EVAL
PRINT

4.4 Ring-around the READ-EVAL-PRINT Rosy

- **READ**
    - accepts input from the currently designated input source which by default is the keyboard
    - reads until it has a valid printed representation of a Lisp object
    - expands read-macro characters such as ’ ‘ # if any
    - converts the external printed representation into a Lisp object
    - hands this to EVAL

EVAL

- looks at the *form* it has been handed
    - If the form is a **number, the result is the number**
    - If the form is a **symbol, the result is the ward of the symbol,** or, if the symbol has no ward, **an error is signalled**
    - If the form is a **cons,**
        - the cons better be a **list**
        - if the **car of the cons** (or its functional ward, etc.) is a **function,**
            1. the operands of the form are EVALuated
            2. the function is called with the results of the EVALuation of the operands as its arguments

          the result is the result of the function call or **the function signals an error.**
        - if the **car of the cons** is a **wired-in special operator,**
            1. the operands of the form may or may not be evaluated, depending on the operator

          the result depends on the unique rules of the special operator
        - if the **car of the cons** is a **macro**
            1. the form is expanded according to the given macro definition
            2. the macro expansion is EVALuated

          the result is the result of the evaluation of the macro expansion
        - if the **car of the cons** is NOT a function, a wired-in special operator, or a macro, then **EVAL signals an error**
- hands the result of evaluating the form to PRINT

PRINT

- builds an external printed representation of the object that is the result of the evaluation

- diplays that printed representation on the currently designated output device

4.5 Games & Practice with EVAL & EVALUATION of FORMS

For each of the following groups of forms,

- write in the printed representation that will be displayed if you had typed in that form to a **READ-EVAL-PRINT** loop

- assume that you type in the forms in order

- assume that the machine is cold-booted between the groups, but not between the individual forms

- if you get an error, ignore the rest of the forms in the group and go to the next group

If you're not sure what would happen

- try drawing the graphic representation of what the world looks like before and after the evaluation of the form

- and try evaluating these forms on a machine

1. **(setq a 3)**

2. **(+ a 4)**

3. **(setq b 'c)**

4. **(+ a b)**

1. **(setq herman 'joe)**

2. **(setq joe herman)**

3. **joe**

4. **(symeval joe)**

5. **(symeval herman)**

6. **(makunbound herman)**

7. **herman**

1. **(setq list '(a b c))**
2. **(setq big-list (list list list))**
3. **(rplaca list 'b)**
4. **big-list**
5. **(setq big-list-2 (list '(a b c) '(a b c)))**
6. **(eq (car big-list-2) (cadr big-list-2))**

1. **(setq l '(a b c))**
2. **(rplacd (last l) l)**

5. FUNCTION DEFINITION

Defining Functions

Lisp Code/Interpreted

Compiled/Machine Code

5.1 Extending the Lisp World

Programming in Lisp consists of writing Lisp code that extends the existing Lisp World, creating new objects and new relationships between objects.

Defining new functions is a major part of Lisp Machine programming. It brings new actors to the stage, and advances the action played thereon.

Function definitions start out being written in **Lisp code**.

The **Lisp code version** is usually then **compiled** into **machine code**.

The **compiled version** naturally runs **faster**.

LAMBDA EXPRESSIONS

DEFUN

5.2 Lambda Expressions & DEFUNs

5.2.1 Lambda Expressions

A *lambda expression* **is a function**

A *lambda expression* is made up of the following elements:

- The symbol **LAMBDA,** which is a keyword recognized by EVAL.
- A **parameter list,** which is a list of zero or more symbols.
- A **body,** which consists of zero or more *forms*

((lambda (c) (car c)) '(7 9))

Evaluation of forms containing Lambda Expressions

When EVAL is handed a compound form, in which the operator is a list,

1. EVAL checks to see if the CAR of the operator of the form is the keyword LAMBDA. If it is not, it signals an error.

2. It then checks to see that the number of symbols in the parameter list matches the number of operands in the form. If a mismatch is found, it signals an error.

3. These operands are evaluated and the result of the evaluation of each operand becomes the ward of the corresponding symbol in the parameter list. The old ward of the symbol, if any, is saved away, and the new ward is attached temporarily.

   This processing is known as *binding*. The symbol is now said to be *bound*, or *temporarily bound*.

4. EVAL then proceeds to evaluate each of the forms in the body of the lambda expression.

5. The temporary ward of each symbol in the lambda parameter list is detached and the old ward, if any, of the symbol is restored.

   This is known as *unbinding*. However, when a symbol goes through the unbinding process, it does not necessarily become *unbound*. Unbound is technical term meaning *the symbol has no ward*. Going through the unbinding process merely brings the symbol back to its previous state, which may or may not be unbound.

6. The result of evaluating the last form in the body is returned as the result of the evaluation.

setq a

5.2.2 DEFUN & Lambda Expressions

Lambda Expressions are created by a *Compound Form* which is a list of the following elements:

- The special operator **DEFUN**

- **The Name of the function.**
  Strictly speaking, this is the name of the symbol which will be used to invoke the lambda expression

- **Lambda Parameter List**

- **Body**

(Describe a)

The special operator **DEFUN** causes **EVAL** to

- create a *lambda expression,*

- attach it to a symbol as the *functional ward* of the symbol, and

- return the symbol as the result

(Describe 'Henry)

Henry is the function (Named-LAMBDA
HENRY (C). . .): (C)

(Named-Lambda Henry (C) (car C) is a list

5.3 Compiling Lisp Code Definitions

For speed and efficiency, one normally **compiles** Lisp Machine programs.

compile 'henry
HENRY

COMPILE takes

henry is the function
DTP-compiled function
HENRY

- a **single required argument** which must be a function specification of some kind. For now, the only thing a *function specification* can be is a **symbol with a functional ward,** which must be some kind of function.

- an **optional second argument,** which ought to be a Lambda Expression. If this second argument is given, then this will replace the former functional ward of the symbol in the first argument. (compile 'adolphus '(lambda(K)(car K)))

The **compiler** converts the *interpreted function* into a FEF (old term) or *compiled function,* which becomes the functional ward of the symbol.

There are three ways to invoke the **compiler**

1. from a Lisp Listener by a call to **COMPILE** as above.

2. from within the Editor which has various commands to read code from a buffer and compile it.

3. by a call to **compiler:compile-file** which translates a source code file saved on the file system into a **BIN,** or machine code, file. This function can be invoked from a Lisp Listener or from the Editor, using the Compile File command.

You will use the compiler when you do the first exercises on the Lisp Machine.

(describe 'adolphus)

:

extra info : interpreted-definition
(lambda (K) (car K))

(adolphus '(7 9))

5.4 Parameter List Keywords

(defun elephant (x &optional (y 1))
(+ x y))

You can define functions that take optional arguments or any number of arguments as well as required arguments. This is done by inserting special keywords in the parameter list. Here are the four most useful ones:

&optional This says that all the parameters following the word "**&optional**" are optional. If you don't supply arguments corresponding to these parameters, they will take on a ward of **NIL**. You can force a default ward for the parameter (if you want something other than **NIL**) by specifying the parameter as a list:

```
(defun my-function (&optional (argument-1 t) arg-2)
  ...)
```

In the body of the above function, **argument-1** has a *default value* of **T**, and arg-2 has a default value of **NIL**. You can override these defaults just by supplying the function with the values you want when you run the function.

&rest This keyword is used when you want to get any number of arguments. There can only be one **&rest** argument. It must follow any required or optional arguments. The ward of the **&rest** argument is a *list* of the arguments supplied.

```
(defun my-function (&rest all-of-em)
  (do-something-to (car all-of-em))
  ...)

(defun other-function (one-argument
                       &optional (arg 34.2)
                       &rest all-the-rest)
  (....))
```

&aux The parameters following **&aux** are not supplied at all. They are just used as *local variables*. Their wards get saved away and restored just like the other parameters, so you can use them without being ecologically unclean. You can supply default values in the same way as with **&optional** arguments.

```
(defun user-program (argl arg2 &aux (temp nil))
  (setq temp (* argl arg2))
  ...)
```

&key

&key is a lot like &optional. &key arguments do not have to be supplied. Their important feature is that they don't have to be supplied *in order*. You specify which argument you are supplying by using a *keyword*. This is best explained by using an example:

```
(defun foo (&key one (two 2) three)
  (terpri)
  (print one)
  (print two)
  (print three))

;the keywords don't need to be quoted in Release 5+

(foo ';three 9 ':one 1)  ===>

1
2
9


(foo) ===>

NIL
2
NIL
```

Keyword arguments must be supplied in pairs, the keyword and the value. In Release 5+, keywords don't need to be quoted because they evaluate to themselves. (They are symbols that have themselves as their ward.) All keywords begin with a colon.

: one 9

5.5 Games & Practice with FUNCTION DEFINITION

1. Write a function called AVE which takes two numbers and returns their average.

2. Using the above function, write a function called AVE-4 which takes four arguments and returns their average.

3. (You won't be able to do this one with just the information we've given you so far.) Write a function that takes any number of arguments and returns their average.

use c-sh-A in ed & ll

6. FLOW OF CONTROL-- CONDITIONALS, RECURSION & ITERATION

WHEN/UNLESS

IF

COND, AND, OR

LOOP

6.1 Which forms to evaluate, when and how

The **body** of a function definition consists of zero, or, more forms. Each form may be evaluated, and the result of the evaluation of the last form returned as the result of the function call.

There is a family of **special operators,** referred to as *conditionals* which **control** the flow of form evaluation based on the outcome of some *logical* test.

These include:

- **one-way conditionals** such as WHEN and UNLESS.
- **two-way conditionals** such as IF
- **multi-way conditionals** such as COND, AND and OR

There is another family of special operators that control *iteration.* We will only discuss **LOOP,** since it can do everything that the others can do.

6.1.1 AND/OR

AND and OR are *special operators.*

You can use **and** and **or** both for controlling the flow of a program, and for combining tests.

Both AND and OR forms have an indefinite number of operands which get EVALuated as follows:

- Evaluation of AND's operands continues until
    1. A NIL result is encountered, in which case the result returned is NIL, or
    2. All operands are evaluated, in which case the result returned is the result of evaluating the last operand.
- Evaluation of OR's operands continues until
    1. A NON-NIL result is encountered, in which case the result returned is that result
    2. All operands are evaluated, in which case the result returned is NIL

(and) => t
(or) => nil

can be used as
a conditional
& print

6.1.2 IF

implemented with cond

IF is a *special operator*, to be precise a *macro*. **IF** deals with conditional operations of the *if-test-thenthis* and the *if-test-thenthis-elsethat...* kind.

IF

```
(defun mike (x)
(if (= x 9) 1 2 3))
  (mike 9)
   (mike 2)
       3
```

- requires at least two arguments.

  1. a **test-form** which evaluates to NIL or NON-NIL
  2. a single **then-form** which gets evaluated if the evaluation of the test-form returns NON-NIL[2]. In this case, the result of evaluating the **then-form** is returned as the result.

- accepts an indefinite number of additional arguments...
  else-form1...**else-formN**. These get evaluated in turn if the **test-form** evaluates to NIL. The result of evaluating the last **else-form** is returned as the result.

If there is no **else-form** explicitly given, an **else-form** of NIL is assumed by default. (Or you can think of it as returning the result of the test-form.)

6.1.3 COND

COND is a *special operator* that deals with **multi-way** conditional branches.

COND forms have an indefinite number of operands, each of which must be a list. This list is made up of one, or, more forms.

The CAR of each operand is checked.

- If it evaluates to NON-NIL, each of the remaining forms in the list are evaluated, and the result of evaluating the last such form is returned as the value of the COND.
- If it evaluates to NIL, then the rest of the forms in the list are ignored, and evaluation moves to the next operand.
- Evaluation proceeds in this way until

  1. either the CAR of one of the operands evaluates to NON-NIL, or
  2. there are no more operands, in which case, the result returned is NIL

[2] This is a good place to use the special operator **PROGN**. **PROGN** takes any number of arguments, and returns the last one. This lets you combine any number of forms into a single form.

6.2 Recursion

Recursion is simply using a function from within that same function. You should use recursion when the problems seems to lend itself to recursion - like when dealing with binary trees, or problems with recursive definitions. Don't feel like you have to use recursion just because you're programming in Lisp. Iteration is just as good, if not better. Use the method that feels right.

6.3 Games & Practice with RECURSIVE DEFINITIONS

Here are some functions that use recursion.

```
;;; This function takes a number and returns that element of the
;;; Fibonacci series.  The Fibonacci series begins like this:
;;; 0,1,1,2,3,5... The zeroth Fibonacci number is 0, the first
;;; one is 1, and the sixth is 8.

(defun fib (n)
  (if (or (= n 0)              ;are we at the "bottom"?
          (= n 1))
      n                        ;yes, just return the argument
      (+ (fib (- n 1))         ;no, add up the fibs of n-1 and n-2
         (fib (- n 2)))))
```

```
;;; This function will return a list of all the leaves in the tree,
;;; excluding all occurences of NIL.  The argument "tree" can be any
;;; structure built of conses.  It uses one &aux argument as a local
;;; variable -  this makes it almost completely ecologically
;;; clean.

(defun make-list-from-tree (tree &aux (list nil))
  (if (listp tree)

      ;; Now we have a node in the tree.  Recursively call ourself
      ;; on the car and cdr, "appending" the result onto our running
      ;; list.

      (progn
        (setq list (nconc list (make-list-from-tree (car tree))))
        (setq list (nconc list (make-list-from-tree (cdr tree)))))

      ;; Now we're at a leaf.  For no particular reason, we've decided
      ;; not to put NIL's into the list.

      (when tree    ;don't push NIL on the list
                    ;[you might want to, though,
                    ;     depending on the application]
        (setq list (cons tree list))))

  ;; Now return the list as the result.  This is necessary both to get
  ;; the final answer, and for intermediate results

  list)
```

Here are some exercises that can either be best solved by writing a recursive function, or at least solved equally as well with recursion or iteration.

1. Write a function that takes a cons as an argument, and prints out all the "leaves" of the tree. That is, look at the car and cdr of the cons, and print them if they're anything except a cons. If either of them **is** a cons, do the same thing for that cons. A good name for this function is **print-fringe**.

2. Write a function that takes a list of numbers as its single argument and returns the sum of all the numbers in the list.

3. Write a function that uses either recursion or iteration to find the largest number in a list of numbers. Note: Yes, there is a very simple way of doing this, using the **MAXIMIZE** keyword of **LOOP**. Don't do it that way, this is an exercise.

6.4 Iteration & Loop

LOOP

LOOP is a special operator (actually a macro) that allows you to do all kinds of iteration. There are so many different ways of writing loops using **LOOP** that the syntax of it is often called *loop language*.

You might say that **LOOP** is a very special operator, and that there are very complex rules for what is evaluated and what isn't, and in what order. The *keywords* that **LOOP** uses are not evaluated, everything else usually is.

So that we don't confuse you right away, we're just going to discuss the simple kinds of iteration.

LOOP takes a bunch of keywords and interprets them in order to do the right thing. The syntax of **LOOP** is basically:

```
(loop iteration clauses
      DO
      body)
```

The *body* (any number of forms) is executed according to the instructions in the *iteration clauses*. When any iteration clause finishes, the body stops being executed.

The **DO** separates the body from the iteration clauses. Remember to always include the **DO**! It is a common mistake. This should be one of the first things you look for it a loop doesn't seem to be working properly.

LOOP returns **NIL** unless you do something special (see below.)

There are examples of all of the simple kinds of iteration in the **Games and Practice** section.

6.4.1 Numeric Iteration

Normal run-of-the-mill numeric iteration is accomplished by the following format:

```
(loop for i from 1 to 100 by 5
      do
      (print i) ...more body forms...)
```

The symbol following the **for** is a local variable within the loop. It is incremented each time through by the *step*, the amount following **by**. The step is optional and defaults to 1. When the counter is greater than the end, the body stops being executed, and **LOOP** returns **NIL**.

The *start*, *end*, and *step* operands **are** evaluated, so you can use a variable in their place.

6.4.2 Going through the elements of a list

To do something for each element in a list we do:

```
(loop for each-element in '(a b c d)
      do
      (print each-element))
```

Notice how close the keywords in this example are to the English words in the sentence above. This is very much the spirit of **LOOP**.

Again, the symbol after **for** is a local variable. It gets temporarily bound to an element of the given list.

The list is the thing following the keyword **in**. The *list* operand is evaluated, so you can have a form (often a symbol) that evaluates to a list.

6.4.3 Local variables and arbitrary exiting

You can get extra local variables by using the keyword **with**.

You can force a **LOOP** to exit by using the **RETURN** function. It takes one argument, which is the value to return from the whole loop. You can also use **RETURN** in any of the more primitive functions that **LOOP** is made of, but we won't talk about them.

```
(loop with foo = nil
      with bar = t
      do
      (if bar (return foo))  ;see if bar's value has been changed
      (....)
      ...more forms...)
```

```
(loop with x=9
        for i from 1 to 3 do
          (print (+ i x)))
```

6.5 Games & Practice with ITERATION & LOOP

6.5.1 Examples of LOOP

Examples of functions that use simple LOOPs. [Not LOOPS]

These examples show how to do

- simple numeric iteration,
- simple iteration through the elements of a list, and
- a simple way to do completely arbitrary iteration and finishing.

```
;;; This function takes a list as an argument and prints
;;; each element.

(defun print-elements-of-list (list)
  (LOOP FOR element IN list
        DO
        (print element)))

;;; This function adds the numbers in a list together,
;;; returning the sum.
```

local variable? sum to 0

```
(defun add-em-up (list-of-numbers)
  (let ((sum 0))    ;local variable SUM to keep running total
    (LOOP FOR i IN list-of-numbers DO
          (setq sum (+ i sum)))
    sum))           ;return value of SUM as value of LET,
                    ;LOOP returns NIL.


;;; Here is a slightly different implementation. (How do these two
;;; functions differ?  Hint: Try to run each of them.)

(defun other-add-em-up (&rest numbers)
  ;;local variable SUM to keep running total
  (let ((sum 0))
    (LOOP FOR i IN numbers DO
          (setq sum (+ i sum)))
    sum))           ;return value of SUM as value of LET,
                    ;LOOP returns NIL.


;;; This function prints the squares of all numbers from 0 to N.
;;; How many numbers will this function print?
```

n+1

```
(defun print-squares (n)
  (LOOP FOR i FROM 0 TO n DO
        (print (^ i 2))))
```

```
;;; This function prints the numbers from 1 to n squared with
;;; n numbers on a line.  Note use of nested loops, and BELOW.

(defun print-board (size)                    initializes
  (terpri)
  (LOOP WITH total = 0                                     stop when
        FOR i FROM 0 BELOW size DO                         i ≥ size
        (LOOP FOR j FROM 0 BELOW size DO
              (prinl (setq total (1+ total))))
        (terpri)))  {writeln}

;;; This function takes a list of numbers as an argument.  It
;;; prints the sequence of numbers that begins at zero and
;;; goes up by each given amount in turn.  Note use of explicit
;;; return to exit function on some arbitrarily weird condition.

(defun series (list)
  (setq list (copylist list)) ;so we don't get any surprises later
  (rplacd (last list) list)   ;this makes the list circular
  (LOOP WITH total = 0 DO
        (setq total (+ total (car list)))
        ;; Now check for some random condition we happen to be
        ;; interested in.  If it's true, return some random thing.
        (if (= total 319.) (RETURN 'uh-oh-spaghettios))
        (print total)
        (setq list (cdr list))))


;;; This function uses both iteration and recursion.  It takes a
;;; "tree" of numbers, that is, a list whose elements are either
;;; a number or, recursively, a tree of numbers.  We use
;;; iteration to go through each list, and recursion to go down
;;; all the levels of the tree.

(defun add-up-tree (tree &aux (sum 0))
  (LOOP FOR thing IN tree DO
        (setq sum (+ sum (if (listp thing)
                                 (add-up-tree thing)
                                 thing))))
  sum)  ;return sum as value of whole function
```

6.5.2 Exercises

Here are some exercises that are best solved by using LOOP. Some of these exercises lend themselves to a recursive solution as well. Try some of them this way too. It will give you an interesting perspective on when to use iteration and when to use recursion.

1. Write a function that takes one argument, a number, prints out all multiples of that number. For example, if you give the function 5 as an argument, it will print out 0,5,10,15...etc. c-abort

2. Write a function that takes (a list of) an arbitrary number of numbers, and returns the average of those numbers.

3. Write a function that takes a list containing both symbols and numbers and returns the number of symbols in the list. Here is an example of running it:

```
    (number-of-symbols '(a b 1 2 c 9 d e))
    5
```

4. Write a function to print all the divisors of a given number.

5. Write a function which takes a list of numbers. For each number in the list, print that many asterisks on a single line. Use a different line for each number in the list. Helpful Hint: The form **(PRINC "*")** will display a single star. The function *TERPRI* will perform a carriage return.

6. Write a function which prints the numbers from 1 to 1000 (or more). However, print the word "buzz" if the number either contains a 7 as a digit, or is divisible by seven.

7. The Lisp Machine

Booting

Logging in

The Lisp Listener

The Editor

7.1 Games and Practice with the Lisp Machine

This is an exercise designed to give your fingers some practice. If you follow these steps you will see a fast introduction to a lot of the simpler features of the Lisp Machine programming environment.

There is a lot of information here that is not written down anywhere else, or at least is not gathered together in a sequential manner. Please read through this exercise even if you have used the Lisp Machine before.

You should note that the appropriate procedure is usually documented in the *Lisp Machine Summary.*

What we're going to do in these steps is to:

1. Show you how to start the machine and login

2. Practice typing a few forms to the **READ-EVAL-PRINT** loop

3. Write a function and save it in a file

Do exactly as this guide says. Every time we use the word "should" (when we are explaining something that *should* happen), we mean that if it doesn't happen that way, you should get your instructor to help you figure out what went wrong. But above all,

DON'T WORRY!

Step 1. Don't worry.

Unless you really know what you are doing and try very hard, you can't break anything and you can't do anybody any harm. Your attitude should be one of, "What the heck...let's try it!"

However, do try especially hard to type things exactly as we tell you to here. While it's not a big deal, you'll probably need some help to get back on the the right path.

Previous experience has shown that working in pairs is beneficial. It helps to have two pairs of eyes watching for interesting things. If you decide to work with someone else, switch off between the person typing and the person kibitzing.

Read through the description of each step before doing it.

Step 2. Cold-boot the machine.

If the machine says **Cold booted** on the bottom line, right hand side, then you don't need to do this step. You can do it anyway, it won't hurt.

There are some sub-steps that you may have to do, depending on the current state of the machine.

1. *See if there is someone already logged in.* Right next to the time (one space over,) there may be a name. If there is just two inches or so of blank space next to the time, then nobody is logged in. If there is a name, ask that person or your instructor whether it's okay to boot the machine.

2. *Get to a Lisp Listener window.* All interaction with the Lisp Machine is done "through" a window. *Lisp Listener* windows run a **READ-EVAL-PRINT** loop. *Windows* are rectangular areas of the screen, usually surrounded by a black border. Windows usually have labels in their lower left hand corner.

   a. If there is a window visible that has a label that says "Lisp Listener" and a number, and the cursor in that window is blinking, then you're all set for the moment.

   b. If those two conditions are not true, then you have to do something to make it true.

      - Press the SELECT key, and let go. On the bottom line, in the center, it should say "System-" or "Select:".

- Now type a single "L".

This should bring up a Lisp Listener window. Get your instructor to help you if it didn't.

no! (si:halt)

3. *Type in the form* **(si:%halt)**. **si:%halt** is a function that takes no arguments and whose contract is to halt the Lisp processor and bring you to the *Fep* (Front End Processor) command level. At the top of the screen, overwriting anything that might be there, should be the prompt, **Fep>**.

4. *Type a "b" and a return.* "B" stands for boot. Actually you can type out the whole word, "boot", but you don't have to. This completely reinitializes the machine. It resets everything. The last user's work will be lost unless it was explicitly saved away, so you should always check that somebody is finished with a machine before you cold-boot it.

At the bottom of the screen, black dashes called ***run bars*** flash. After a minute, the screen clears and the system types out the ***herald***. Wait until things settle down. The run bars stop flashing, and the word "Tyi" appears in the middle of the bottom line of the screen.

The bottom line is called the ***who line***.

On the right hand side of the who line it should say "Cold booted." This means that the machine is completely untouched. The Lisp World is in its pristine, original state. **The only time you can be sure of this is when you see that message.** As you do the next step, you will see that as soon as you type the first character, the "Cold booted" message goes away.

Step 3. Log in.

'Grgic

or

If your user name happens to be GRGIC, type **(login "GRGIC")**. If your user name is *not* GRGIC, just substitute your name for GRGIC in that incantation. (If you don't know what your user name is, try your last name, or ask your instructor.) Watch the bottom of the display. Various mildly entertaining things will happen down there. Then things should quiet down. The run bars will stop flashing, and the word "Tyi" will reappear in the center of the who line.

The only things moving on the screen should be the flashing rectangular ***blinker*** and the clock at the lower left-hand corner of the screen.

It ought to say **T** right under where you typed **(login ...)**. If it doesn't, and especially if an error message is being displayed, call your instructor. After the instructor has fixed things up, go back to step 1.

In case you're really curious, the who line shows, from left to right, the date, the time, your user name, the *current package* (forget it), and the current machine state, which is **Tyi** whenever the machine is waiting for you to **TY**pe **I**n.

The run bars below the who line flash whenever the machine is doing something. The one on the right (the *run bar*) means the Lisp processor is running, and the one on the left (the *page par*) means that the disk is running. You can make the run bar light up by moving the mouse around. (~~However, the mouse won't move until you type something on the keyboard.)~~

Function - m - /

Step 4. Do a simple calculation.

Type the following Lisp form: **(* 365 24 60 60)**

Note that you don't need to type a return. (Some of you will say to yourselves "Of course!", for the rest of you: why not?)

If you make mistakes when you're typing to the Lisp Listener, don't despair. Of course, you can use the RUB OUT key to erase mistakes. But Lisp Listeners (and almost any other place that ask you to type things in) have a feature known as the *input editor* or *rubout handler*. This allows you to use most simple editor commands, even though you're not running the editor. Try control-B to go back, and control-F to go forward characters.

There is another interesting feature of the input editor. If yo~~u type control-C,~~ it will bring c-m-y back the last thing that you typed, omitting the last character. After one ~~control-C,~~ you can c-m-y then type ~~meta-C~~ to rotate through all the last few things you typed. This is known as *yanking in text.* m-y

c-D c-m-y

When you've typed in the whole form, the **READ-EVAL-PRINT** loop should print 31536000, which is almost the number of seconds in a year.

Problem 1: A solar year has 365 days, 5 hours, 48 minutes, and 46 seconds. Use the Lisp Listener to calculate the number of seconds in a solar year. Try to use the ~~control-C~~ c-m-c (yank) feature to minimize typing. 31556926

Step 5. Get an error message and recover.

Look at the following Lisp form: **(+ 7 'A)** The variable A is unbound

Figure out what's wrong with it. What kind of error will you get when you try to evaluate it? Now try it and see how close your guess was.

Look at the error message. You should be able to understand at least part of it. Notice that the blinker is blinking to the right of an arrow. The arrow prompt means that you are talking to the *debugger*, a fancy tool for analyzing and fixing errors. Debugger commands are mostly single keystrokes. The simplest debugger command is ABORT, which means, "Give up

S-A try 8
(setq m 19)
Then do w S-A 4 A
m

Then use S-B or m-C Then do A
Then retry
(setq A nil)

return a value from the +- internal instruction
means provide replacement form

completely on trying to evaluate this form. Just get me back to the Lisp Listener." Try that now.

Now analyze each of the following erroneous Lisp forms, guess what the error will be, and then compare your guess with the real error message. As you get used to popping into and out of the debugger, notice that the debugger gives you other options besides simply aborting. You might try using these to fix each situation. Remember you can always get out of the debugger, back to the Lisp Listener, by typing ABORT.

```
(+ 12 'A)          Look at arg 0, asks for a replacement form

(+ 12 A)

(// 4 0)           replace entire form

(++ 13 14)         (fsymeval '+)

((* 2 3) (* 4 12)) (fsymeval '+) replace (* 2 3) ∴ get 48
                                  would have returned to LL

(SETQ 'X 5)        'X   if you return, get right return but X is
                                                        unbound

(CAE '(1 2 3))     (fsymeval 'car)
```

You should play with the Lisp Listener for a while. Try some of the things you learned in class.

Now you will do some things that are a microcosm of the standard program development process on the Lisp Machine. You are going to "develop" a factorial program. You should pretend that this is an enormous program and that between cold-boots are full days of work.

Step 6. Enter the editor and get into Lisp Mode.

Key-board

Press the SELECT key and then an E. The **SELECT** key isn't a shift key; you ***don't*** hold it down while pressing the E. The shift keys (the ones you do hold down while pressing other keys) all have brown printing on them.

The screen will change. The new display shows you that you are now running the ***Zmacs editor.***

Type m–X. Watch the display change near the bottom of the screen as you do this. The meta-X command begins an *extended command.* The editor is now prompting you for the name of the extended command.

Type in the words "lisp mode". (No quotes.) Watch what happens when you type the space. There is a *completion* facility for extended commands and other long inputs. When enough of a word has been typed in for the command or part of command to be uniquely specified, you can just go on to the next word.

Type return to finish the command.

Just to learn something else, try typing m–x "lis m" and then a return. The cursor jumps to the spot that you have to fill in to make the command be unique. Press the HELP key, and read the display. Remember that the HELP key can give different help in different contexts - if you're confused at any time, try typing HELP - it can't hurt. Now type a "p" and then another return.

Step 7. Type a short program with a bug in it.

c-m-q to indent

You can use this one or invent your own. Look up editor commands in the red *Lisp Machine Summary* booklet.

note flashing (
return line

Type 1st line
Type 2nd line

```
(DEFUN FACTORIAL (N)
  (LET ((PRODUCT 1))
    (DOTIMES (FACTOR N)
      (SET PRODUCT (* PRODUCT FACTOR)))))
```

product

1

(dotimes (factor 7)
(print factor))

0
:
6

Loop for i from 2 to n do

c-m-@ marks a form

Step 8. Compile the buffer and go back to the interpreter.

There's one command that does all of this. It's m-Z.

There is other common ways to compile your buffer and go to a Lisp Listener:

- **meta-x Compile Buffer** will compile the whole buffer
- control-shift-C will compile the nearest function definition or form or region
- **meta-x Compile Changed Definitions of Buffer** Well, you can probably guess what this does. You don't have to type the whole name in. Just type m-x C SPACE C RETURN. wrong (also will compile not new functions)
- There are also comparable commands for evaluating code.
- SELECT L will get you to a Lisp Listener from anywhere
- suspend BREAK will give you a little "Lisp Listener-like" window within your editor. You should use this when you want to try out a function that is known to work, but you're just not sure about what it returns or something like that. If you're debugging your own functions, use the Lisp Listener. If your function gets an error while in the Lisp Listener, you don't have to abort out of it to go look at the code in the editor.
- You can also change the size and position of the editor and/or Lisp Listener windows so that both are showing at the same time. Do this by clicking the right mouse button twice fast to get up the *system menu*, and then click left on "Edit ~~Screen~~". read mouse line or split screen

  explain & mouse highlighting

  sh-r

  To switch between the editor and Lisp Listener you can still use the select key, but you can also click left on the window you want to type to.

Step 9. Test the program and get an error message.

Try **(FACTORIAL 6)**. Look at the error message carefully.

Step 10. Exit the debugger and return to the editor.

Now get out of the debugger (use ABORT), and go back to Zmacs. Can you figure out what caused the error? If you're stumped, ask your instructor.

Step 11. Try again.

When you have figured out the problem, fix it (one inserted character will do it). Then recompile and return to the Lisp Listener.

Step 12. Test the program again.

Note that it still doesn't work. Pretend that you can't find the problem and are giving up for the day.

Step 13. Return to the editor and write the buffer out to a file.

Use the red booklet to figure out how. You should call the file something that ends with ".lisp".

C X
C W

A *pathname* is a way to get at a particular file. Pathnames have at least four components:

- a *host name* which is followed by a colon (:) H: or F: Hydra:
- directories, which are specified using the syntax that the host supports (">" on Lisp Machines, "/" on Unix systems, etc.)
- file name
- file *type* or extension
- Sometimes a version number, and sometimes a device are included, but this is rare.

The various file commands prompt you with a default filename. You only have to type in what you want to change about the default. For example, while you're in this course, you'll almost never want to change the default host. So if the the default pathname given was **acadia:>your-name>foo.lisp** and you want to write the buffer into **acadia:>your-name>test.lisp**, then you only have to type in "test" and it will keep the default host, directories, and file type.

Step 14. Log out and cold boot.

Get to a Lisp Listener and type the form **(logout)**. Then follow the boot process described above. It's good etiquette to boot your machine when you're done. That way the next person to use the machine doesn't have to wonder whether or not you were finished.

Pretend to go home for the day.

Step 15. Log in, enter the editor, and read in the file. CX CF [illegible]
or S-F then [illegible]

Use the red booklet to find the command for reading in a file. The same defaulting mechanisms exist here.

Additionally, you can get completion of file names. Type COMPLETE to get as much of the pathname completed as is unique, or END to complete the pathname and get the file if the partial text can uniquely specify a file. RETURN says don't complete the name at all, just get the file as I've typed it, creating a new file if necessary.

lisp- is enough to return
H:>wolf>lisp-probs.lisp
lisp
H:>wolf>lisp

Step 16. Debug the program.

When it works, go on to the next step.

Step 17. Write out the modified buffer.

Step 18. Compile the file.

Use the command **m-X Compile File**. This creates a *binary* file that contains the equivalent of the text in your regular file. You can then use the **load** function to bring the contents of the file back into the Lisp World.

Step 19. Log out and cold boot.

This is the end of the second pretend day.

Step 20. Log in and load the compiled file.

Select F

Use the **LOAD** function. It has a single required argument, which is the only one usually used. The argument is a pathname, which you enclose in double quotes. Note (for future reference) that if you're loading in a file from a Unix system, you must double the slashes in the pathname. Now you can use the factorial function that you wrote "yesterday".

8. Lists

List Manipulation

Lists as Tables

8.1 List Manipulation

8.1.1 Review of Terminology

known as list

- A *list* is either the symbol **NIL**, or a cons whose cdr is a list.
- The conses that make up the list are collectively known as the *backbone*, and individually known as *vertabrae*.
- The car's of the vertabrae are known as *elements.*
- There are equal numbers of vertabrae and elements.

8.1.2 List Manipulation Functions

These are the most common functions used for dealing with lists.

obsolete

See *Lisp Machine Manual*, pp. 53 - 63, for more details on these and other related functions.

Making lists

LIST &rest elements
takes any number of things, and makes a list with those things as elements.

COPYLIST *list* returns a list which is **EQUAL** but not **EQ** to *list.*

to return
the direct
use this)

Examining lists

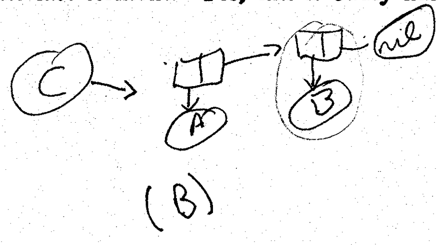
LENGTH *list* returns the length, or number of elements in *list.*

FIRST, SECOND, THIRD, ... SEVENTH *list*
get that element of *list.*

NTH *n list* gets the nth element of *list.*

LAST *list* returns the last **cons** of *list*, **not** the last element. Yes, this is badly named.

(Defun lastel (list)
(First (last list)))



Adding to lists

NCONC &rest *lists*

concatenates all the lists together, by changing making? the ~~car~~ cdr of the last cons in each list except last to be the first cons in the next list.

is this correct? first list still exists

APPEND &rest *lists*

makes a list which is a concatenation of all the lists, without modifying any lists. It does this by copying all the lists *except for the last one.*

Adding things to the front of lists

Can be done using **CONS,** but somehow you have to get a hold on the new cons. Usually this is done with **SETQ.**

```
(setq stack (cons 'new-top stack))
```

Reversing lists

2 **NREVERSE** *list* reverses *list* by changing the order of conses by manipulating cdr's. draw use w setq as above note what happens if you don't

1 **REVERSE** *list* makes a reversed copy of *list* draw..

8.2 Lists as Tables

There are basically four families of functions here. Each family has three variants on a single theme. The variants specify what predicate is being used for comparison for the theme. *

The ASS family - **ASSQ, ASSOC, ASS**

Used for looking up associations between one thing and something else. The functions take two arguments, an *item* and an *alist* or *association list.*

An item is just any Lisp object. An association list is a list of conses. The car of each element (each cons) is the *key* that we use to look up the cdr.

```
(setq grade-alist '((a . 100)
                    (b . 80)
                    (c . 70)
                    (d . 60)
                    (f . 0)))

(assq 'd grade-alist) ===> (d . 60)
```

A common trick is to say:

```
(cdr (assq 'd grade-alist)) ===> 60
```

This gives you the answer or NIL.

member 7 '(9 7 6)
(7 6)
memq is same
also mem 'eq

The MEM family - **MEMQ, MEMBER, MEM**

Take an *item* and a list, and see if the item is an element of the list. Returns the sublist whose car is the first occurence of *item* or NIL.

The REM family - **REMQ, REMOVE, REM**

Take an item and a list, and remove all (or a specified number) of the occurences of the item in the list. Makes a copy of the list.

with items removed ?

copies

basis of comparison

(rem 'eq 'a b)

(remq 'a b) remove any eq to a from b

changes

The **DEL** family - **DELQ, DELETE, DEL**

Take an item and a list and actually modify the list to remove the item. These functions should not be used only for effect. Instead, you must use them *both for value and effect.*

```
(setq my-list '(a b c d))

(remq 'a my-list) ===> (b c d)

my-list ===> (a b c d)

(delq 'b my-list) ===> (a c d)

my-list ===> (a c d)

(delq 'a my-list) ===> (c d)

my-list ===> (a c d)
```

The last **delq** did not give us the desired effect. We wanted to have no more **a**'s in the ward of **my-list**. It didn't work because **delq** doesn't change the fact that the ward of **my-list** is a particular cons. That cons is not part of the list that **delq** returns, but it remains the ward of **my-list**.

To get the desired effect:

```
(setq my-list (delq 'a my-list))
```

note 3/6/84
~~The model fails here~~

push
pop

8.3 Games and Practice Using Lists

Use Lisp list functions, preferably on the Lisp Machine, to do the following:

1. Make a list of U.S. presidents **NIXON, FORD, CARTER** and **REAGAN**. Call the list **PRES**. (That is, create a list of four elements, the symbols **NIXON, FORD, CARTER** and **REAGAN**; and make the list be the ward of the symbol **PRES**.)

   We'll give you this first one.

   ```
   (setq pres (list 'nixon 'ford 'carter 'reagan))
   ```

   Del

2. Remove Nixon from the list. How many elements does the ward of **PRES** have? It should have three.

3. Add Johnson to the list. Now there should be four elements in the ward of **PRES**.

4. Make a new list called **DENT** consisting of the second and last elements of **PRES**, as well as Truman, Eisenhower, and Kennedy.

5. Reverse the order of elements in **DENT** and attach it to the front of **PRES**. There are a number of ways to do this: Do you use **NREVERSE** or **REVERSE**? **APPEND** or **NCONC**? In this exercise, it doesn't matter which way you do it, *as long as you understand the difference*. Each of the four functions above has a very different effect.

   4 make the new list the ward of Pres
   or "alter dent."

6. Alter **PRES** so that the second occurrence of Ford becomes Nixon. You don't have to write a general way of doing it, just count over and splice it out by hand.

7. Create three new lists called P-1, P-2, and P-3. Each of these is a copy of **PRES** but with

   (or dent if new list above is dent)

   a. Both occurrences of Reagan removed

   b. The first occurrence of Reagan removed

   c. The last occurrence of Reagan removed

   Is there any difference between using **DELQ** and **REMQ** for these three operations?

8. Write a function called **SHUFFLE** which takes two lists as arguments, and returns a single list containing alternating elements of the first list and of the second list.

   ```
   (shuffle '(a b c) '(1 2 3))  ===>

   (A 1 B 2 C 3)
   ```

9. Write a function called **UNSHUFFLE** that takes a single list, and returns a list which might have been equal to the first argument to **SHUFFLE.**

```
(unshuffle '(a foo b bar c d)) ===>

(A B C)
```

9. Graphics

Graphic operations

9.1 Windows, Modes, and Graphics

Just so we can do some fun things right away, these are incantations you can use to get various graphical things to happen on the screen.

No explanation will be made as to what these incantations do exactly, until Flavors and message-passing is explained. Then all this will be obvious.

- All of the graphic operations happen on a *window*.
- Windows have a Cartesian coordinate system that is *upside-down*. That is, the point (0,0) is the upper left corner of the window. Y coordinates increase as you go *down*, but X coordinates increase *normally*, to the right.
- Each square between integral points is referred to as a *pixel*. Pixels are the smallest addressable units of the screen.
- The symbol **terminal-io** has as its ward the "current" (in some sense, anyway) window.
- All graphics are drawn in a particular *mode*, referred to as an **alu**. Alu's are just numbers, but are usually referred to by using a symbol whose ward is the proper number.
    - **tv:alu-ior** means to turn on all the pixels being drawn, no matter what was there before
    - **tv:alu-andca** means to erase all the pixels being drawn, no matter what was there before
    - **tv:alu-xor** means to flip all the pixels being drawn, depending on what was there before
- All graphic incantations begin with:

  ```
  (send terminal-io
  ```

  then they have additional operands to determine what kind of thing to draw, and where, and how to overwrite existing displays.

Send is just a function, although it does some complex things. This means that the evaluation of these "incantations" proceeds just like the evaluation of any other normal form. All the operands are evaluated.

9.2 The Incantations

Here are the other operands:

(send terminal-io ':draw-point x y) & optional Alu 29
draws the point at (x,y) in the default mode. You can supply an optional last argument to specify the mode, like this
(send terminal-io ':draw-point x y tv:alu-xor).

(send terminal-io ':draw-line x1 y1 x2 y2) & optional alu
draws a line from (x1,y1) to (x2,y2). There are two optional arguments, for specifying the mode and drawing the end-point. This last one defaults to **T**, meaning draw it. 30

(send terminal-io ':draw-rectangle width height x y &optional alu)
draws a filled-in rectangle with dimensions *width* by *height* with its upper left corner at (x,y). Note that this only allows you to draw horizontal-vertical rectangles - for other ones, use :draw-triangle twice. And, no, you don't have to type "&optional". 31

(send terminal-io ':draw-triangle x1 y1 x2 y2 x3 y3 &optional alu) 31
draws a triangle with corners at (x1,y1), (x2, y2), and (x3,y3).

(send terminal-io ':draw-circle center-x center-y radius &optional alu) 31
draws the outline of a circle of the specified radius and center.

(send terminal-io ':draw-filled-in-circle center-x center-y radius &optional alu) 31
draws a filled-in circle of the specified radius and center.

(send terminal-io ':clear-screen)
Clears the screen.

There are other ones as well, but they are harder to explain, and not necessary at this point in your Lisp life. See *Introduction to Using the Window System*, pages 28 - 32; and the *Release 4.0 Release Notes.*

9.3 Games & Practice with Graphics

9.3.1 Examples

Here is program that moves a rectangle across the screen.

```
(defun move ()
  (loop for x from 100. to 1000. do
        (send terminal-io ':draw-rectangle
              40. 40. x 100. tv:alu-xor)
        (send terminal-io ':draw-rectangle
              40. 40. x 100. tv:alu-xor)))
```

Here is a recursive function that draws a neat picture. Try running it with **(squares 400 200 200)**.

```
(defun squares (size x y)
  (cond
        ;;just return nil when the square is very small

        ((< size 2) nil)

        ;; otherwise draw the square, shrink the size and
        ;; corner, and keep going

        (t (send terminal-io ':draw-rectangle
                 size size x y tv:alu-xor)
           (squares (- size 10.) (+ x 5.) (+ y 5.)))))
```

This function draws a line given a starting point, length and angle, instead of taking the endpoints. This will be useful for the exercises. The arguments are:

- x and y for a starting point
- the length of a line to draw
- the angle (in degrees) to draw it at,
- and optionally the mode to draw it in.

```
(defun angle-line (x y length angle &optional (alu tv:alu-ior))
  (send terminal-io ':draw-line
        x
        y
        (fixr (+ x (* length (cosd angle))))
        (fixr (+ y (* length (sind angle))))
        alu))
```

9.3.2 Exercises

Now here are some exercises. You might want to use the function **angle-line** for some of them.

1. Write a function called POLYGON that draws a regular polygon (not filled in) given four arguments: the X and Y coordinates of the center of the polygon, the radius of the circumscribed circle, and the number of sides.

   Extra credit: Have it draw a five-pointed star when given 2.5 as the number of sides. In order to get the extra credit you must also figure out why this is a reasonable thing to expect!

2. The Arcturian Stribbage-Tree begins life as a small seed. During the first year of life it grows a tall stalk exactly 27 meters high. Then it sends out two symmetrical branches from the top of the stalk at an angle of 45 degrees from the vertical. These grow for a half-year, until they are each 13.5 meters long. Then the branches divide into sub-branches which grow for a quarter of a year... Write a program that draws a picture of an Arcturian Stribbage-Tree that is almost two years old.

3. An order-zero Dragon Curve is a line segment. If you know how to draw an order-N Dragon Curve, it's easy to draw an order-N+1 Dragon Curve. Just draw an order-N and draw another order-N *backwards*, from the end of the first order-N curve. When you draw a curve "backwards," turn 90 degrees to the right before drawing it. If you're going forwards, just draw the order-N. Write a program to draw Dragon Curves of any order.

Hold on that
[illegible] A here
was [illegible]
To the [illegible]

4. (This one requires a little math.) Snowflakes are complicated many-sided polygons. You can make an order-N+1 Snowflake from an order-N Snowflake by gluing an equilateral triangle to the middle third of each side. (For this to work, the sides of the triangle must be one-third of the length of the side you are gluing it to.) Oh, by the way, an order-0 Snowflake is just a single equilateral triangle. Write a program to draw Snowflakes of any order.

10. MORE LISP WORLD OBJECTS Their Nature & Representation

Arrays

Strings

Dimensions

Cells

Array-Types

Lists vs. Arrays

10.1 Arrays & Strings

An *array* is a Lisp object that has

- *dimensions* and
- *elements* or *cells*

An array may have one through seven *dimensions*, and each dimension may have one or more *elements*, or, *cells*.

In each dimension [illegible]

The *cells* elements of an array are numbered 0 through n-1, where n is the number of elements of a given dimension.

Each *cell* of an array may connect to **any Lisp World object**, including an array.

An array of any type may be the **ward** of a **symbol** or the **car** or **cdr** of a **cons**. It is a bona-fide Lisp World object, and can be used anyplace where any other Lisp object can be used.

A *string* is a one dimensional array (of type art-string) wherein each element or cell is an eight-bit unsigned fixnum which represents a *character*.

10.1.1 Types of Arrays

The types of arrays include:

- **art-q**-arrays whose elements may be any Lisp World Object.

- **art-*n*b**-arrays whose elements are *n-bit* fixnums, where *n* ranges from 1 to 32. These elements, however, are not real Lisp Objects since they are **truncated**. Their advantage is that they occupy less storage than art-q type arrays. Arrays of this type are used a lot by the window system.

- **art-string**-arrays are very similar to **art-8b** arrays. Both of these types of arrays have elements which are fixnums representing characters. There are many differences in effects between arrays of type **art-8b** and type **art-string**, but the elements of both kinds of arrays are the same.

  We will discuss strings separately from the other types of arrays.

Various other array types exist.

10.1.2 Printed Representation

The *printed representation* of arrays is second class.

10.1.3 Arrays vs. Lists

Lists and arrays are used when you have collections of data.

Lists are useful when you are going to be

- adding to or subtracting from such collections

- doing a lot of interpolating

- manipulating stuff that occurs early in the list

Arrays are what you want when

- you are NOT going to be changing the number of elements, and,

- you want to refer to a particular item by its location.

For accessing the first three elements, a list, however, is faster than an array.

Array Makers

Array Accessors

Array Changers

10.2 Array & String Manipulating Functions

As with other Lisp World objects, there is a collection of functions that deal with arrays in general, and strings in particular.

These functions fall into the categories of

- **Constructors & Changers**

- **Fetchers & Examiners**

10.2.1 Constructors & Changers

Array Makers

Arrays are **created** by a call to **MAKE-ARRAY.**

MAKE-ARRAY takes

- **one required argument,**
  –a list of one up to seven numbers specifying the number of elements for each dimension of the array.

- **a variety of optional arguments**
  –each consists of a *keyword* specifying the nature of the argument, and the actual argument.
  :TYPE is one such keyword for which the appropriate argument is the array type specification name, such as ART-Q, or, ART-STRING, or, ART-1B, etc.

Since arrays have only second class printed representations, they are normally brought into being as the values of symbols. The symbol name is then used to refer to the array. For example,

```
(setq my-array (make-array '(10 10) ':type 'art-1b))
```

Array Setters

Objects become elements of an array by a call to **ASET**.

ASET takes

- **three, or, more required arguments**

  1. **the object to be assigned**

  2. **the array object**

  3. **the subscripts of the element that is being assigned**

10.2.2 Fetchers & Examiners

An element of an array can be accessed by a call to **AREF**.

AREF takes

- **two, or, more required arguments**

  1. **the array object**

  2. **the subscripts of the element being referenced**

10.2.3 String Particulars

The *printed representation* of a string is any collection of characters enclosed in double quotes ("). | (vertical bar), / (slash) and " (double quote) are slashified. When you type a "string", **READ** calls **make-array with :type art-string.**

A 101_8 / 65_{10}

READ has a macro-abbreviation (like ') for reading in integers that are represented by characters. The syntax #/ followed by a single character actually means the integer that that character represents.

Example:

```
(SETQ S "This is a string")  ===> "This is a string"

(AREF S 0) ===> 84.

(ASET #/u S 13.)  ====> 117.

S ====> "This is a strung"
```

124_8

```
(SETQ STRNG
      (MAKE-ARRAY 5 ':TYPE 'ART-STRING))

(ASET #/S STRNG 0)
(ASET #/T STRNG 1)
(ASET #/R STRNG 2)
(ASET #/N STRNG 3)
(ASET #/G STRNG 4)

(PRINT STRNG) ====> "STRNG"
```

Special characters such as space, CR, LF, C-A, etc. are handled by another character macro...#\SP, #\CR, #\C-A, etc. This way it's obvious what character you mean to use, rather than using #/ format with an invisible character. (#/ works with invisible characters, of course.)

Like numbers (but not like other types of arrays), strings evaluate to themselves.

Strings are numbered not by the elements in the string but by the *crack numbering* system.

Example:

```
     F    R    E    D

  0    1    2    3    4
```

Crack 1 is just to the left of character 1 (See how this might be confusing? "Fencepost" errors are much more common in strings, where you're often trying to get a chunk from the middle. Better to say "the substring between crack 0 and crack 1," so that you know exactly which characters you mean.)

```
(SUBSTRING "FRED" 1 3)======> "RE"

(SUBSTRING "FRED" 1)========> "RED"
```

General form examples:

```
(STRING[-reverse]-search[[-not]-char or -set]
                   <key> <string> <start> <stop>)

(STRING-SEARCH #/E "FREDDY")---->2

(STRING-SEARCH-SET '(#/E #/F) "FREDDY")-----> 0
```

These functions can be used with READLINE to break input into components..

```
(defun get-and-parse-sentence ()
  (let* ((sentence (readline))
         (first-non-space
           (string-search-not-char #\sp sentence))
         (command (substring
                    sentence string
                    first-non-space start
                    (string-search-char #\sp sentence end
                                        first-non-space))))
    ;; Start processing based on the command
    ...
    ))
```

10.3 Games & Practice with ARRAYS & STRINGS

10.3.1 Exercises

1. An excellent exercise, recommended for everyone, is the game of Life. This is a simple simulation of an environment of single-cell organisms. The cells can survive if they have a few neighbors, for mutual protection from the elements. The cells die if it gets too crowded.

   The game is played on a rectagular board of squares. The size of the board does not matter. The exact rules of survival are as follows:

   - A cell survives to the next generation if it has two or three neighbors, out of a possible eight.

   - A cell dies if it has more than three neighbors or less than two.

   - A cell is born in an empty square if there are exactly three neighbors.

   - All "checking" for birth, survival, and death happens for all cells before any new cells are born or existing cells die. That is, all births and deaths happen simultaneously.

   You get to choose how the board is drawn, and how the initial configuration of cells gets entered.

   There are lots of optimizations possible. Make sure to get a working version before attempting to do any optimization.

2. Write a function that takes a string as an argument. The string will be a sentence. Your function should return a list of strings, representing the words in the sentence.

3. Write a function that takes a string as an argument. The string will contain a line from a Lisp program. Your function should return a string containing the comment, or NIL if there is none. Try to account for at least one of the special cases, like checking to see whether the semi-colon is contained in a string - in which case it doesn't make a comment.

4. Write a function called string-singularize, which takes a string and returns a string which is a good guess as to what its singular form is.

11. Additional Lisp Machine Features

Debugging Techniques

Other Editor Commands

11.1 Debugging Tools

The most important debugging tools are your brain and your eyes.

Most of the tools that the Lisp Machine provides will just gather more information about the state of the Lisp World for your brain to interpret.

11.1.1 Using the Lisp Machine and your own code to help you.

- Remember that *any function can be called by hand* at top-level in a Lisp Listener. You don't *have* to have your "mainline" call your "subroutines". If you suspect a certain function, try

    - going to another Lisp Listener, (SELECT c-L if you need a new one)

    - and running the function, giving it the necessary arguments by hand.

- *Split the screen into an Editor window and a Lisp Listener window,* if that seems more convenient. From the system menu (click right twice, or hold the SHIFT key down and click right once), use

    - Split Screen - Allows you to evenly divide up the screen among some windows. Note: Click on "Existing Window", not "Edit" or "Lisp" to get an ~~your~~ already existing editor or Lisp window.

    - Edit Screen - this is the general screen editor that allows you to change the size and position of any window.

11.1.2 Modifying your code to help you.

Wrapping a **PRINT** *form around a piece of suspected code.*
Since **PRINT** just returns its argument, you can stick a call to **PRINT** in anywhere without affecting how the function runs.

The function **BEEP**.
If you're not sure whether your program gets to a certain point, or if your program deals with graphics, you can insert a **(beep)**. It flashes the screen and returns **NIL**.

Breakpoints. You can get your program to stop at a particular place so you can poke around by using **BREAK** or **DBG**.

```
(break here)  ;break is a special operator
              ;the argument is a label and is not
              ;evaluated

(dbg)         ;dbg is just a function
```

BREAK puts you into a read-eval-print loop. You can look at the values of symbols (or special variables), but not *local variables*. This means that **BREAK** is not so good for gathering information about compiled code.

DBG puts you in the debugger. Use the **"DBG:"** functions to look at local variables and arguments.

Type RESUME to get out of either type of breakpoint and to continue your function's execution.

11.1.3 The Debugger

The debugger has lots of commands for information gathering and for manipulation of the execution of your program. If you go into the debugger and type C-HELP, it will list out all the possible commands. There's about 40 of them, here we've cut that to about 15 of the most useful ones:

Special commands For each kind of error, there are a few special commands dealing with the particular error. One of these will be useful if you want to "help the program along", and let it keep going.

ABORT Always used for "forget about this, go back to the previous top-level".

RESUME Used for the most commonly used special command.

c-N Goes down the stack, to the *function that called this one*, and displays information about that function call.

c-P Goes up the stack, to the *function that this one called*, and displays information about that function call.

c-L, or REFRESH Clears the screen, and redisplays the error message plus a *backtrace of the stack* (a list of all the functions that were called to reach this one).

m-L Clears the screen, and displays more extensive (but not necessarily more useful) information about this function call.

c-B Displays a backtrace of the stack.

(dbg:arg n) Accesses the nth argument to this function.

You can use the macro **SETF** to change the values of the arguments, like this:

```
;;; makes the value of the zeroth
;;; argument be 32.5

(setf (dbg:arg 0) 32.5)
```

We'll be discussing **SETF** soon.

(dbg:loc n) Accesses the nth local variable in this function. You can use **SETF** on this, too.

`c-R` Returns from this function. It prompts you for a value to return if one is necessary. Useful when you know about the error you got, and want to ignore it and go on.

`c-m-R` Reinvoke this function. Useful when you've gone into the editor, found the bug, recompiled, and now you want the program to continue.

`c-E` Edits the current function. Goes into the editor, reads in the file that this function was defined in, and locates the function within the file.

`c-M` Goes into Zmail, and sets up a bug report to be sent to the right place (based on your site).

`c-m-W` Goes into the Window Debugger, which some people like better. It has some of the same features as the regular debugger, plus it displays some extra information nicely.

11.1.4 Monitoring your code without modifications

The Stepper. The **STEP** function will evaluate a form, stopping at the beginning and ~~a~~end of each function call. It *only works for interpreted code.* This makes it not very useful.

Tracing This feature allows you to observe the execution of your program when a particular function is entered (called) or exited. The special operator **trace** will enable this feature.

```
(trace get-number-of-neighbors)
```

You can also get fancier handling by using the *TRACE* option in the *system menu*. Click the mouse button twice (or hold the shift key down while clicking once) to get the system menu. Click left on *trace.* This will pop up a little window asking you for a function to trace. Then a menu will appear giving you a choice of fancy options.

Use the special operator **UNTRACE** to untrace a function, or all functions if you call it with no arguments

```
;;;stop tracing this function

(untrace get-number-of-neighbors)


;;;stop tracing all functions

(untrace)
```

Helpful hint: Avoid tracing standard Lisp functions. Naturally some of the background processes use these functions, but they never expect to have to type anything out. If they do have to type something out, the process will hang until you let it type out. Needless to say, this will cause you grief. So just trace your own functions, and leave the driving to us.

11.1.5 Looking at data

DESCRIBE **DESCRIBE** is a function that takes one argument, and will display some information based on the type of the argument. **DESCRIBE** knows about symbols, conses, numbers, functions, named defstructs (you have to use the **:NAMED** option), and other things.

The inspector. The inspector is a utility that allows you to browse through data structures. You can invoke the inspector by:

1. SELECT I, or

2. the function **inspect**. It takes an optional argument which is the first thing to inspect.

In the inspector, you can type in a form and its value will be inspected. Move the mouse over any displayed thing, and click left to inspect the highlighted thing. There is a list of all things that have been inspected, so you can go back to any of them.

11.2 More Zmacs Commands

11.2.1 Debugging Commands

m-X Find Unbalanced Parentheses
will make a good guess as to how your parentheses are unbalanced. It only counts them, though, so if you have the right number but they're in the wrong place - tough luck.

m-X Edit Compiler Warnings
will split the screen into two windows, showing the compiler warnings for each function in the top window, and the function in the bottom one. Use `c-.` to get to the next function.

11.2.2 Typing Lisp code

`m-.` is "edit definition". Allows you to type in the name of a function, and then proceeds to edit that function. It will read in the proper file, if necessary, and go to the right location in that file. *This is the best way to read through the system sources.*

`c-sh-A` when typed in the middle of a form, will display the arguments of the operator. *This command works in the input editor as well.*

m-X Lisp Mode gets you into the mode where all of the rest of these commands work. If you the file you edit has a *type* (or extension) of "lisp" (**.lisp, .l, .lsp**), then you get Lisp mode automatically. ? ?,

`LINE` will go down one line and space over to the right place. (Hopefully, you knew this one already.)

`TAB` will indent the current line to right place. If you don't like the "right" place, `c-TAB` will try some other good places.

`c-m-Q` will completely indent the following Lisp form. ?,

m-X Fill Long Comment
will fill up lines containing comments. The comment should begin at the beginning of the line.

m-X Auto Fill Mode
works in Lisp mode, as well as Text mode. It does its best to automatically indent.

`c-;` makes a comment at the end of the current line. Use `c-m-;` to get rid of a comment at the end of the current line.

12. VARIABLE BINDINGS & SCOPING ISSUES

Variables and Binding

Scoping

12.1 Variables

Programming languages use variables to hold a value. In Lisp, we've used symbols as variables, because they seemed best suited for it. Sometimes though, we used the car of a cons, or an element of an array to hold a value.

The compiler's contract is to take a function (usually an *interpreted* function) and write a function that works exactly the same way, except faster.

One of the ways it does this is by not using symbols as variables. Instead it uses a fancy thing called a *local variable*. We've used this term loosely before, but now it's a technical term.

inconsistent language

A *local variable* is not a Lisp object. It is just a place that the compiler reserves to hold a value. The compiler only uses local variables to replace the use of symbols that get *bound*. Remember that binding is a temporary saving away of a symbol's (*variable's*) ward, to be restored at a later time.

The main point is that **variables that get bound** are represented differently in compiled code (local variables) from the way they're represented in interpreted code (symbols). This can cause some difficulties.

Here is an example of where you can run into difficulty: (from the **Lisp Machine Manual,** page 15.)

```
(setq a 2)              ;set the ward of the
                        ;symbol A to be 2


(defun foo ()           ;define a function
  (let ((a 5))          ;bind the variable A to be 5
    (bar)))             ;call BAR


(defun bar ()           ;define BAR
  a)                    ;just return the value of the
                        ;variable A

(foo)  ===> 5           ;does what we expect

(compile 'foo)          ;The compiler now makes the A in
                        ;FOO be a local variable.

(foo)  ===> 2           ;It doesn't work.  Or does it?
```

* The function **BAR** makes a *free reference* to the variable A. This means that the function has used the value of the variable, but has not *bound* it. **BAR** doesn't even know whether the variable **A** *has* a value.

The important point here is that **you can't make a free reference to a local variable.** That is, for the technical meaning of the the term "local variable." If you make a free reference, you will always be referring to a *symbol,*

Note that this problem does not come up unless you make a free reference.

12.2 How the Compiler Decides What to Make Into Local Variables

Any symbol that gets bound, by

- **lambda** binding
- **let** binding
- **loop with** binding

is a candidate for getting turned into a local variable. If the only thing about the symbol that gets used by the function is the ward, then the compiler will use a local variable to represent that variable. In any other case, the compiler will use a symbol or *special variable.*

A *special variable* is just a variable that is implemented by a symbol. The thing that makes special variables special is that you can make free references to them.

If the compiler doesn't make a variable into a local variable, then it *declares it special.* You've probably seen this message. Declaring it to be special just means that the compiler is using a symbol.

12.3 How You Can Make Free References Work

You can declare variables special, too. The special operator **DECLARE** does this. It tells the compiler to ignore its rules about local variables, and to use a symbol whenever it compiles this variable.

DECLARE's should not be done within a function, but at top-level - usually the top-level in a file. For example,

```
(declare (special *a*))

(setq a 2)

(defun bar ()
  a))
```

Most people don't use **DECLARE,** though. There is a special operator called **DEFVAR** which is used to simultaneously declare a variable to be special, and to give it an initial value. Also, the editor knows about forms that begin with "def" so you can look up this variable without knowing what file originally declared it. **DEFVAR**'s shouldn't be done within functions either.

```
(defvar a 2)

(defun bar ()
  a)
```

12.4 Why You Should And Shouldn't Use Special Variables

Both the advantage and disadvantage of using special variables is that any function can make a free reference to their value.

This means that it saves you some hassle because you don't have to pass this variable as an argument to every function that needs to use it.

This also means that functions that you didn't write have access to this variable and can maliciously or unintentionally change its value.

12.4.1 Protecting Your Special Variables

- By convention, names of special variables begin and end with "*". This way, nobody will do something like

  ```
  (setq *my-variable* (test-function))
  ```

  It's assumed that if you change the value of a variable named "*...*", then you better know what you're doing.

- Bind the special variable in a top-level function. Make sure that all the functions that refer to the special variable get called in the context of this binding.

12.5 Terminology and Future Changes

Special variables and all variables in interpreted code are said to be *dynamically* scoped.

Local variables are *lexically* scoped.

Dynamic scope refers to the fact that the value of a dynamically scoped variable depends on the *runtime environment.* The value of lexically scoped variables can be determined just from looking at the *text* of the code.

By about 1985 or so, the interpreter will be lexically scoped as well. The *Common Lisp* dialect requires this. Symbolics will be supporting Common Lisp.

12.6 Games & Practice with VARIABLE BINDINGS

12.6.1 Chess Program Example

This is the board. Each element is either NIL or a "piece", which contains some kind of information about what kind of piece it is, and other information particular to specific piece. For example, whether or not the king is in check, or whether or not a rook can castle.

```
(defvar *chess-board* (make-array '(8 8)))
```

This function moves a piece. You tell it which piece to move by referencing the position on the board. You don't have to pass the board as an argument because the function makes a free reference to *chess-board*. There is only one board, so there is no need for all functions that want to use the board to have get it as an argument.

```
(defun move-piece (from-rank from-file to-rank to-file)
  (let ((piece (aref *chess-board* from-rank from-file)))
    (aset piece *chess-board* to-rank to-file)
    (aset nil *chess-board* from-rank from-file)))
```

Here's another useful function:

```
(defun set-up-chess-board ()
  (aset ... *chess-board* 0 0)
  (aset ... *chess-board* 0 1)
  ...
  (aset ... *chess-board* 7 6)
  (aset ... *chess-board* 7 7))
```

Now consider this situation, or a similar one where it might be more appropriate to have two copies of the same program running at once:

1. You start up a chess game not by using **chess-top-level** (below), but by doing the appropriate internal things: (set-up-chess-board) (loop ...). The intention is to debug the program, but it seems to be working pretty well, and you play for an hour.

2. Then you get a bug. You decide that you don't want to ruin your almost-finished game, so you go into another Lisp Listener, and start up another game.

3. You fix the bug.

4. Then you go back to the first Lisp Listener, and ... oh no! You're playing the second game! Since both games referred to the same board (***chess-board***), the second game clobbered the first.

If instead, you had started up a game using this function:

```
(defun chess-top-level ()
  ;the LET is the key thing to notice here
  (let ((*chess-board* (make-array '(8 8))))
    (set-up-chess-board)
    (loop ...
          do
          (move-piece ...)
          ...)))
```

then you won't clobber any games if you have more than one, since you'll always be playing with the "local" chess board.

13. PACKAGES

Avoiding name conflicts

Keywords

13.1 Avoiding Name Conflicts

When more than one person is working on a program, there is the possibility of name conflicts.

Packages allow you have your own set of names that you can use, without fear of a conflict with someone else.

A package is (mostly) just the database that **READ** uses to look up symbol names.

There is always a *current package.* This is the one that is displayed near the middle of the who-line at the bottom of the screen. (The thing that says "USER:")

When **READ** gets the printed representation of a symbol, it looks in the current package to see if it already knows about it. If it doesn't find the symbol name there, it goes to the package's *superior package,* and looks there. It keeps going until it reaches the highest package. If it wasn't found in any of them, then it gets interned in the current package.

The hierarchy of packages is normally very flat. All existing packages are inferiors of the **GLOBAL** package.

* The easiest way to make a package is when writing your program in a file, make the first line (the *mode line*) say:

```
;;; -*- Package: (your-package-name global 300); -*-
```

followed by m-x reparse attribute list *

can also m-x set package, but then have to put in the global 300 by hand)

ask if [illegible] it still create it & set for attribute list

You can change the current package either temporarily or permanently.

- When you're typing to **READ,** you can temporarily change the current package for the next printed representation by typing in a package name and a colon.

- You can change the current package permanently by using the function **PKG-GOTO.** It takes one argument, a package name.

```
(pkg-goto 'user)

(pkg-goto 'tv)
```

attached only to files for which I indicate update attribute list during the session.

13.2 Some Packages

USER where little programs usually go. The default package for people who don't have to worry about packages.

GLOBAL contains symbols that you want to be able to access from within any package, like all the standard Lisp symbols.

* inverse is update attribute list.

KEYWORD The keyword package is for interning symbols that you just use for their name. For example, keyword arguments like :TYPE for **make-array** are just used for their names. You never want to evaluate a keyword. The keyword package's name is also the empty string.

In Release 5, all symbols in the keyword package automagically have themselves as their wards. This is so you don't have to quote them - they evaluate to themselves.

In Release 4, there is no package called "keyword", but you still use the empty-string convention. The **USER** package has abbreviation.

TV contains window system stuff

SI contains low-level system stuff (System Internals)

There are lots of other packages.

13.3 Games and Practice with Packages

1. Put one of your programs into its own package.

14. SMART MACROS

14.1 SETF & Friends

For every kind of Lisp World Object there are functions and special operators to

- **construct & change**

- **fetch & examine**

the object.

SETF is a *macro* that essentially handles some of the busy work of **accessing** and **altering** objects and relationships in a general way.

SETF takes **two** arguments.

- The first argument must be a form.

- It examines that form to see what function you are using.

- It then works out what you would need to do to make the result of a call on that form be the same thing as the second argument to SETF.

- Example:

- (SETF (CAR FOO) 3) is the same as (RPLACA FOO 3)

- (SETF (CDR FOO) 3) is the same as (RPLACD FOO 3)

- (SETF (THIRD LIST) 'BAR) is the same as (RPLACA (CDDR LIST) 'BAR)

- (SETF (AREF fred 2 3) 67) is the same as (ASET 67 fred 2 3)

INCF is similar to **SETF**.

However, its second argument is the amount by which to increase the result of evaluating its first argument.

This increased amount then replaces the old amount.

Example:

(INCF (AREF fred 2 3) 5) is the same as (ASET (+ (AREF fred 2 3) 5) fred 2 3)

14.2 Plist & Information Stored on the Property List of Symbols

The information used by **SETF** and **INCF** to determine which alterant function is appropriate given the accessor function used in the first argument is located on the *property list* of the symbol used to name the function.

PLIST is the function used to examine the property list of a symbol.

14.3 Games & Practice with SMART MACROS

15. Projects

One of the goals of this course is for you to write a good-sized Lisp program. At this point, you've probably either finished, or are getting finished with the game of Life. This is a good first program.

For your project, we'd like you to do something a *little* more ambitious. Here are some guidelines.

15.1 Criteria for good projects

Not too hard The goal should be achievable within the course schedule. If you don't reach a point of "closure" you may go away feeling frustrated, even if you have learned a lot. We want to show you that you are capable of producing finished programs with the machine.

Not too easy This isn't so important if there are obvious extensions. If you finish way ahead of time, your instuctor can always suggest extensions or modifications. Some students feel insecure, and select an easy project for that reason. This is fine. Just go for whatever you feel comfortable with, and make additions later.

Graphics Graphics are so cool and so universally useful that it would be a shame to leave them out of any project completely. Fancy hackery with text in different fonts can be an adequate substitute, if the project does not depend on the font editor to do all the work. Graphics are fun – don't miss out!

Nontrivial user interface

A "trivial" user interface is one where the user just types Lisp forms and the program just types back Lisp objects. We encourage: mouse commands, menus, single-keystroke commands, non-lisp command languages (not too hairy, though).

Data abstraction and modelling

If the program manipulates abstract objects of any kind, try to model those abstract objects with Lisp objects. We encourage the use of **DEFSTRUCT**, ~~and for students who feel confident, the use of flavors for data abstraction.~~

Fun, i.e. not too work-related or drudgy

Part of our job is to make you feel "on vacation" from "real work". If we can make it seem like Lisp isn't "real work" you will learn it better. Ideas on how to encourage this "holiday from work" feeling are solicited. But one of the best things we can do is to encourage fun projects. You should do a project that you can laugh about if it breaks.

Working on projects that directly relate to work should be discouraged for two reasons. First, it makes learning Lisp too much like work. Second, work-inspired projects are hard for the instructor to evaluate because the student is an expert in the area and the instructor (generally) is not.

In the early stages of learning Lisp, it isn't important exactly what applications one works on. You will get at least as much out of a pool-table simulation as they will get out of a sewage-treatment plant simulation.

We're not adamant. We're reasonable. If you insist on doing sewage-treatment, fine. You paid.

Instructable Please don't insist on doing a project that is in a problem domain that the instructor is not familiar with. If you do, the instructor will have to make a choice of whether to try to become an instant expert in neuropathology (or whatever) or whether to regretfully inform you that he or she will be unable to offer much support. We favor the latter, in general. The time the instructor spends learning neuropathology could be spent helping other students learning Lisp. You will have been warned that the instructor won't be much help. If you insist on doing neuropathology, well, you paid.

15.2 Ideas for good projects

- Finishing the game of Life, if you haven't done so, or giving it a fancy mouse interface. (Only do this if you want to. If you haven't finished the Life program and are bored with it, do something else.)

- Calculator simulation

- Turtle graphics

- Games:

  - Tic Tac Toe

  - Othello

  - Checkers

  Note: unless the game has a real easy strategy, the actual AI part of the program may be beyond the scope of the course. But games are still great projects. You can write a program that acts as a smart game board, so two people can sit at the console and play against each other, with the machine taking care of captures and things like that.

- Video Games

  A simple Space Invaders is actually within the range of possibility. We've seen a couple of creative and interesting video games come out of this course.

- Computer-aided design:

  From the most abstract sort of design tool ("Put a big triangle over there. Erase that square.") to real applications like plumbing, wiring, and mechanical design, there are zillions of possible projects in this domain.

- A graphic representation of the Lisp World (reasonably difficult).

If you can't think of anything, talk to your instructor. There are a number of projects that have been done already which are interesting, but not unique.

Comments

Watch out for vagueness. Try to have a detailed vision. Try to imagine exactly how you want your program to work when somebody uses it.

Don't be overambitious

We encourage grand visions, but we want to make sure that you have an achievable subgoal. Here is one scenario:

> "I want to do a CAD program for designing radios," says a student. "You'll draw a schematic with the mouse, and then be able to simulate the response of the radio to real signals. It'll be able to draw graphs of selectivity and stuff like that."
>
> The instructor responds, "That sounds great – really ambitious, and exactly the kind of thing that this machine is intended for. The actual circuit entry sounds most interesting to me – how about concentrating on that first, and doing the simulation stuff if there's enough time?"

In this case, the student would be gently advised that it would be best to start with only a couple of kinds of circuit elements, or maybe only one. This way the student can concentrate on mouse clicking, basic graphics, and the structure of the circuit database, and can get the major pieces of the circuit entry program into place.

If the circuit-entry program works for only one kind of circuit element, say resistors, it can easily be extended to other circuit elements. Keeping this in mind will encourage the student to write extensible, modular code.

15.3 Your mission Jim, if you should choose to accept it...

Write a short description of the project you are going to do. It should be a description of how it will look to a user. Give this to the instructor for comments. When the two of you have agreed on a specification, do it. This book will self-destruct in five seconds.

16. STRUCTURES & DATA ABSTRACTION

Representations

Structure Macros

Structure Makers

Structure Accessors

16.1 Representations & Structure Macros

Lisp World Objects are used to **model,** or, **represent** real world objects and their relationships.

A **good** representation bears a strong, easy to remember, easy to use resemblance to the original.

The higher the level of abstraction, the better.

16.1.1 Structure Macros

Structure Macros extend the representation capabilities of the Lisp World by providing a mechanism to make easy mappings from Lisp World objects to real world objects.

They do **NOT** create new types of Lisp World Objects.

They simply enable you to call existing Lisp objects by different names, and to access and change them by operations that are closer to the real world operations.

Structure Macros do much of the busy work of creating the **tools** for

- **constructing**
- **accessing** &
- **altering**

the modelled objects, without you having to worry about exactly how they are implemented.

16.1.2 DEFSTRUCT

DEFSTRUCT is the name of the **structure macro**. As a *macro* it is a *special operator.*

It enables you to create objects of some (potentially unknown) Lisp world type, and refer to that kind of object by a real world name, and parts of that kind‸of object by real world names. The parts of the object are called *slots.*

It offers a large number of options that specify a variety of attributes the structure and structure tools may possess.

The DEFSTRUCT form is a list consisting of

- **DEFSTRUCT**
- **a list consisting of**
    - the **name** of the structure
    - zero or more **options** such as
        - the **:type** option which specifies what type of object to use to implement the structure. An **art-q array** is the **default**.
        - the **:conc-name** feature which prefixes the name of all accessor macros with the name of the structure.
        - the **:include** option which **includes** an earlier structure definition as part of a new definition.
- one or more **slot-descriptions** that are descriptions of the attributes of the structure being defined. A slot description may be:
    - a symbol, whose name will be used for the slot-name
    - a list of a symbol and an initial value to put in the slot

All **DEFSTRUCT** does is create some new macros:

- a constructor macro
- an alterant macro
- and an accessor macro for each slot

16.1.3 Constructor Macro

The name of this macro is a hyphenation of MAKE and the **name** of the structure.

Running the **constructor-macro** results in the creation of a concrete example of the structure.

The **values** of any or all **attributes** named in the **slot-descriptions** may be specified at this time.

16.1.4 Alterant Macro

The name of this macro is a hyphenation of ALTER and the structure **name**.

This macro allows you to change the value in any of the slots in a particular structure.

A call to this macro consists of a list as follows:

1. The name of the alterant macro

2. The concrete structure example to be altered

3. One or more pairs consisting of an **attribute name** followed by the corresponding **value** to be assigned.

16.1.5 Accessor Functions

For each attribute named in the slot-descriptions of the abstract structure, a macro is created that fetches the value of the given attribute for any concrete instance.

The name of the macro may be simply the name of the attribute, or, a hyphenation of the structure name with the attribute name. This will depend on whether the :conc-name option was invoked in defining the structure.

An accessor macro requires one argument which must be a form that evaluates to an instance of the given structure.

16.1.6 DESCRIBE-DEFSTRUCT

DESCRIBE-DEFSTRUCT is a function that allows an all inclusive view of a structure instance.

It takes two arguments

- a structure instance

- the structure name

16.2 Games & Practice with STRUCTURES & DATA ABSTRACTION

16.2.1 Examples of using DEFSTRUCT

```
(DEFSTRUCT (SHIP)
  SPEED
  CREW
  CAPTAIN)

(DEFSTRUCT (PERSON)
  AGE
  SEX
  NATURE)

(SETQ BLIGH (MAKE-PERSON
              AGE 50
              SEX 'MALE
              NATURE 'TYRANNICAL))

(SETQ BOUNTY (MAKE-SHIP CAPTAIN BLIGH))

(NATURE (CAPTAIN BOUNTY))  ===> TYRANNICAL
```

Here is an improved version:

```
(DEFSTRUCT (SHIP :CONC-NAME)
  SPEED
  CREW
  CAPTAIN)

(DEFSTRUCT (PERSON :CONC-NAME)
  AGE
  SEX
  (NATURE 'GOOD))

(DEFSTRUCT (SAILOR :CONC-NAME (:INCLUDE PERSON))
  HOME-PORT
  RANK)

(SETQ BLIGH (MAKE-SAILOR AGE 50
                         SEX 'MALE
                         NATURE 'TYRANNICAL
                         RANK 'CAPTAIN
                         HOME-PORT 'UNKNOWN))

(SETQ BOUNTY (MAKE-SHIP CAPTAIN BLIGH
                        CREW ....
                        CAPTAIN BLIGH))

(SAILOR-RANK BLIGH)  ===> CAPTAIN

(PERSON-AGE BLIGH)  ===> 50
```

? why repeated?

16.2.2 Exercises

Pretend you are writing an airline reservation system. Think of some of the objects you would have to model in such a program. Write some reasonable **DEFSTRUCT's** for the objects. Then write one function that the big program might need. For example, you might model airports and airplanes. Write a function that gets run whenever a flight leaves an airport and adds the plane to the destination airport's data, and removes the plane from the origin.

Don't get too, too fancy.

17. Flavors

Mechanics

Motivation

The *Flavor system* is an extension of Lisp. It is fully integrated into the language so you can write code that uses both flavor stuff, and all the regular stuff we've talked about so far. But you can certainly write Lisp programs without using Flavors.

First we will talk about the mechanics of using the flavor system, and then we'll explain the motivation.

17.1 Mechanics of the Flavor System

17.1.1 Flavor Instances

Flavor instances:

- are a new kind of Lisp object.
- have instance variables, similar to slots in a structure.
- have a message handler. A message handler is a table of *message names* and *methods.*
- receive (*are sent*) messages.

A *message* is just a symbol. It should be interned in the keyword package.

A *method* is (mostly) just a function. It is not the functional ward of any symbol, though. The only way it can be referenced is through the message handler.

17.1.2 SEND

The **send** function is used to *send a message to an instance.*

SEND

1. scans the instance's message handler for method to use
2. "binds" symbols to instance variable values (sort of, anyway)
3. binds the variable **SELF** to the instance itself
4. runs the method in this context
5. "unzips" the instance variable bindings
6. returns what the method returns

We've already used the **send** function when we did graphics. The ward of **terminal-io** is an instance.

17.1.3 How do instances learn new behaviors?

Or, how do we add a new message-method pair to an instance's message handler?

- Usually we have lots of instances of the same kind, so we'd like to teach all of them at once.
- "Kind" = **Flavor**
- A **flavor** gives you
    - a template to stamp out new instances with
    - message handler that all the instances of that flavor share
- instances learn new methods by adding message-method pairs to the message handler

DEFMETHOD

- defines a method for a particular flavor

DEFFLAVOR

- defines a template for the "size and shape" of instances
- creates a message handler that is shared by all instances of this flavor
- may create some initial methods

MAKE-INSTANCE

- makes an instance of a particular flavor

17.1.4 Flavor combination

- Flavors can be added together to get behavior of all combined flavors

Now the following question comes up: Which flavor's method gets to run if more than one included flavor has a method for a particular message?
The answer lies in:

The order of flavor combination

Note that the following is only the effect of how flavors are combined, not how they *actually* do it.

When an instance receives a message, it:

- Searches through its message handler looking for the message

- If it finds the message, it runs the corresponding method

- If not:

  - It goes to leftmost included flavor (as defined in the **DEFFLAVOR**) and searches *its* message handler

  - If the message is not found in this flavor, Keep going "down and to the left" until you reach the bottom (i.e. the leftmost, bottom-most flavor). If still not found, go up one level and one flavor to the right. This is a left-to-right, depth-first search. Loops in the flavor structure are allowed and cause no problem.

  - Use the first method that you find, or signal an error if none is found.

So the order in which you combine methods becomes crucial when more than one flavor has a method for a particular message. If there is no overlap of messages, then the order won't matter.

Here is a simple example of flavor combination, showing which method will run when more than one flavor defines a method for the same message.

```
;;; None of these flavors have any instance variables.


(defflavor a ()
           ())

(defflavor b ()
           ())

(defflavor c ()
           (b a))

(defflavor d ()
           (a))

(defflavor e ()
           (d c))

(defflavor f ()
           (c d))

;;; All of these flavors define the :doit message, or include a flavor
;;; that defines the :doit message

(defmethod (a :doit) ()
  (print "This is flavor A"))

(defmethod (b :doit) ()
  (print "This is flavor B"))

(defmethod (d :doit) ()
  (print "This is flavor D"))
```

```
(setq a (make-instance 'a))  ; A defines :doit

(setq b (make-instance 'b))  ; B defines :doit

(setq c (make-instance 'c))  ; C includes B and A

(setq d (make-instance 'd))  ; D defines :doit

(setq e (make-instance 'e))  ; E includes D and C

(setq f (make-instance 'f))  ; F includes C and D


(send a ':doit) ===> "This is flavor A"

(send b ':doit) ===> "This is flavor B"

(send c ':doit) ===> "This is flavor B"

(send d ':doit) ===> "This is flavor D"

(send e ':doit) ===> "This is flavor D"

(send f ':doit) ===> "This is flavor B"
```

17.1.5 Advanced Method Combination

This section may not be covered in every instance (pun intended) of the course.

- This is more advanced stuff, that you don't need to understand in order to understand anything else in flavors.

- Typically only one method per message wil get to run. You might, in certain circumstances, want all the flavors to add their little bit to the effect.

- For example, you might want all the included flavors' methods to run. Or you might want all the methods to try to run until one works properly.

- The most common (and default) way of combining methods is called DAEMON combination. This only lets one method run, as we saw before, but lets you define additional methods that run either before or after the PRIMARY method runs.

- You can write daemon methods by simply specifying **:before** or **:after** before the message-name in the **defmethod**.

- Other ways of combining methods are PROGN, AND, OR, and LIST. These are more complex, and are almost never used. If you want to see an example of this, the **:mouse-click** methods are combined using OR.

17.2 Motivation for Using Flavors

The motivation for using flavors usually arises in large programs, so it's difficult for us to provide small examples.

When you want to have lots of particular kinds of things.
This is very similar to why you want to use **DEFSTRUCT**.

When abstract types of things share behavior.
This is similar to **DEFSTRUCT's include** facility, except that including flavors is more flexible, because you don't have to be strictly hierarchecal.

When you can separate out a particular behavior that different types of things share.
This is why we have "mixins". You define a flavor that can be mixed into completely different kinds of flavors. If the behavior that it specifies is appropriate for mixing in with more than one other flavor, there's no problem of type-checking or anything like that.

When you want to do something whenever you create an "instance" of a type of thing.
You can specify an **:after :init** method that does something whenever an instance is created.

When you want to define a protocol that different programs can use.
The protocol for output streams is a good example. You can easily write "device-independent" programs by just doing output to a stream. The

messages are the same, no matter how the actual output is implemented. On the other side, a person implementing a new kind of output device can just implement methods for all the messages that generic streams handle. Then all the existing programs can work on the new output device too.

17.3 Games and Practice Using Flavors

17.3.1 Examples

Here are some examples of **DEFFLAVOR's** and **DEFMETHOD's** for defining tiddlywinks.

```
(defflavor wink (x y)    ;instance variables for location
         ()
  :settable-instance-variables)  ;settable makes instance
                                 ;variables gettable and
                                 ;initable too.


;;; This method allows a wink to move all at once.

(defmethod (wink :move-to) (new-x new-y)  ;don't call arguments
                                          ;x and y
  (setq x new-x)
  (setq y new-y))

;;; Another way of doing this is like this, in some cases this
;;; way is preferred:

(defmethod (wink :move-to) (new-x new-y)
  (send self ':set-x new-x)
  (send self ':set-y new-y))
```

A program that used the **wink** flavor might look like this:

```
(defun play-game ()
  (let ((winks (loop with list = nil
                     for i from 1 to 10
                     do
                     (push (make-instance 'wink
                                          ':x (random 10)
                                          ':y (random 10))
                           list)
                     finally (return list))))
    ....
    (send (first winks) ':move-to (random 3) (random 3))
    ....))
```

We could add colors to the winks by just adding some more code.

```
(defflavor (color-mixin) (color-table current-color)
           ()
  :settable-instance-variables)
```

We call this flavor **color-mixin** because it is intended to be *mixed in* with other flavors, and not instantiated by itself. That is, you should never see:

```
(make-instance 'color-mixin
               ':color-table '(black white)
               ':current-color 'white)
```

Then we make a new flavor, and slightly redefine **play-game**:

```
(defflavor color-wink ()
           (color-mixin wink))


(defun play-game ()
  (let ((winks (loop with list = nil
                     for i from 1 to 10
                     do
                     ;; all changes are in the make-instance
                     (push (make-instance 'color-wink
                                          ':x (random 10)
                                          ':y (random 10)
                                          ':color-table
                                          '(black white)
                                          ':current-color 'black)
                           list)
                     finally (return list))))
    ....
    (send (first winks) ':move-to (random 3) (random 3))
    ....))
```

There's no point in having a new flavor if it either

- doesn't add any new instance variables itself, or
- doesn't add any new methods itself, or
- include other flavors

color-wink adds a new flavor which includes new instance variables, and a new method, called **:flip**. The reason why we defined **:flip** as a method for **color-wink**, and not **color-mixin,** is because *flipping* is something that only colored winks do. **Color-mixin** should remain as general as possible, so that we could include it into other flavors. For example, some of the celestial objects in the solar system program on the next page could include **color-mixin** and could change color when they get eclipsed. (Or something.)

```
;;; This method flips a wink over.  It assumes that there are
;;; only two colors in the color table.
(defmethod (color-wink :flip) ()
  (if (eq current-color (first color-table))
      (setq current-color (second color-table))
      (setq^color (first color-table))))
```

current-

The following example is a complete small program that uses flavors. You will need to ask your instructor how to run it, if you want to see it go.

```
;;; -*- Base: 10 -*-

;;; Flavor examples:

;;; These examples are a simple way to get animation.  Each instance
;;; keeps a list of characters and a font.  The characters are assumed
;;; to be a sequence of images that form a repeatable motion.

;;; The :animate message tells the instance how to move.  A high-level
;;; function will be sending the :animate message to a bunch of
;;; "animated" objects.

;;; First we define a basic object that all objects should include.
;;; Then we define the character-display-mixin that allows basic-objects
;;; to be actively displayed.  Character-display-mixin keeps track of
;;; updating the display each time the :animate message is sent.

;;; Finally we define two flavors that define the motion of particular
;;; kinds of objects.


;;; While the flavors we define are generic to any image, using any
;;; font, you should load the solar font to get the right font for the
;;; high level solar system functions.  Find out what file this font is
;;; in from your instructor, and use the LOAD function.

;;; Use m-x Display Font Solar to see what characters are available
```

load the font

```
;;; This flavor just gives us a point to do or place
;;; things at.
(defflavor basic-object (x y)
           ()
  ;; :settable makes i.v.'s initable and gettable too!
  :settable-instance-variables)
```

```
;;; This flavor allows animation using a list of characters that is
;;; looped through (forever).
(defflavor character-display-mixin
        (font
         start-char         ;index of first char in sequence
         nchars             ;number of chars in sequence
         (char-pointer 0)   ;where we are in sequence
         (x-offset nil))    ;offset due to font size
           ()
  :settable-instance-variables
  (:required-flavors basic-object)
  (:required-methods :animate))
```

```
;;; Keep a list of all of these things so we can animate all of them.
(defvar *list-of-displayable-objects* nil)

;;; Get what appears to be the center of the object
(defmethod (character-display-mixin :x) ()
  (+ x x-offset))

;;; This might be needed in the future
(defmethod (character-display-mixin :y) ()
  y)



;;; Get the actual coordinates

(defmethod (character-display-mixin :real-x) ()
  x)

(defmethod (character-display-mixin :real-y) ()
  y)



(defmethod (character-display-mixin :after :init) (&rest ignore)
  ;; This adds the new instance to the list
  (push self *list-of-displayable-objects*)
  ;; This sets up the right displacement for the x-coordinate
  (setq x-offset (// (font-blinker-width font) 2)))
```

```
;;; Primitive graphics

(defmethod (character-display-mixin :draw-self) ()
  (send terminal-io ':draw-char
        font (+ start-char char-pointer) x y tv:alu-ior))

(defmethod (character-display-mixin :erase-self) ()
  (send terminal-io ':draw-char
        font (+ start-char char-pointer) x y tv:alu-andca))



;;; Erase the character before it moves
(defmethod (character-display-mixin :before :animate) ()
  (send self ':erase-self))


;;; Some other flavor defines the motion using the :animate method.
;;; This method should just update the X and Y instance variables
;;; according to how the object wants to move each time.


;;; Draw the character after the object has been moved.
(defmethod (character-display-mixin :after :animate) ()
  (if (= char-pointer (1- nchars))
      (setq char-pointer 0)
      (setq char-pointer (1+ char-pointer)))
  (send self ':draw-self))
```

```
;;; This flavor defines an :animate method that moves an object in a
;;; circle, centered around another (possibly moving) object.
;;;
(defflavor revolving-object-mixin
      ((theta 0) ;changed at animate time
         (delta-theta nil) ;defined at init time
         radius    ;user defined
         mother)   ;user defined, must be an
                   ;instance of basic-object
                   ;with has-satellites-mixin
           ()
  :settable-instance-variables
  (:required-flavors basic-object)) ;must have x and y instance
                                    ;variables for :animate


(defvar *arc* 30)  ;How far objects move along the
                   ;circumference of the circle


;;; This method defines the size of the angle that the instance moves
;;; each time it is ":animated". All "revolving objects" move the same
;;; distance each time, but the angle varies with the circumference of
;;; the orbit.
;;;
(defmethod (revolving-object-mixin :after :init) (&rest ignore)
  (setq delta-theta (// (* 360 *arc*) (* 2 3.14159 radius))))


;;; Here is the important method.  It's main job is to change X and Y in
;;; such a way that the object moves in a circle.
(defmethod (revolving-object-mixin :animate) ()
  (setq theta (+ theta delta-theta)) ;move along the circle
  (when (> theta 360.)               ;fix up angle to look nice
    (setq theta (- theta 360.)))
  ;; change x
  (setq x (+ (send mother ':x) (fix (* radius (cosd theta)))))
  ;; change y
  (setq y (+ (send mother ':y) (fix (* radius (sind theta))))))
```

```
;;; This mixin is for objects that have satellites.  Each time it moves,
;;; it has to move all the satellites too.

(defflavor has-satellites-mixin (old-x
                                 old-y
                                 (satellites nil))
           ()
  :settable-instance-variables
  (:required-flavors character-display-mixin)
  (:required-methods :animate))

(defmethod (has-satellites-mixin :before :animate) ()
  (setq old-x x old-y y))


(defmethod (has-satellites-mixin :after :animate) ()
  (loop for s in satellites
        do
        (send s ':erase-self)
        (send s ':set-x (+ (send s ':real-x)
                           (- x old-x)))
        (send s ':set-y (+ (send s ':real-y)
                           (- y old-y)))
        (send s ':draw-self)))
```

```
;;; This is an instantiable flavor.
;;;
(defflavor satellite (name)
           (has-satellites-mixin   ;some can have satellites
            revolving-object-mixin
            character-display-mixin
            basic-object)
  :settable-instance-variables)

(defmethod (satellite :after :init) (&rest ignore)
  ;;the mother better have  has-satellites-mixin
  (push self (send mother ':satellites)))



;;; This is an instantiable flavor also.
;;;
(defflavor fixed-object (name)
           (has-satellites-mixin
            character-display-mixin
            basic-object)
  :settable-instance-variables)



;;; No motion for a fixed object.  You don't actually need to
;;; define a primary method, but we put one in for
;;; completeness.

(defmethod (fixed-object :animate) ()
  nil)



;;; This function actually does the animation.
;;;
(defun doit (&optional (sleep 10))
  (send terminal-io ':clear-screen)
  (format t "Type any character to stop:")
  (loop do
        ;(process-sleep sleep)
        (loop for object in *list-of-displayable-objects* do
              (send object ':animate))
        (when (send terminal-io ':tyi-no-hang)
          (return *list-of-displayable-objects*))))
```

```
;;; This function is good if you just want to see what happens.

(defun solar-demo (&optional (sleep 10))
  (let* ((*list-of-displayable-objects* nil)
         (sol (make-instance 'fixed-object
                             ':name "Sol"
                             ':x 350.
                             ':y 350.
                             ':font fonts:solar
                             ':start-char #/⊃
                             ':nchars 3))
         (earth (make-instance 'satellite
                               ':name "Earth"
                               ':mother sol
                               ':radius 150.
                               ':x (send sol ':x)
                               ':y (+ 150. (send sol ':y))
                               ':font fonts:solar
                               ':start-char #/E
                               ':nchars 2))
         (saturn (make-instance 'satellite
                                ':name "Saturn"
                                ':mother sol
                                ':radius 250.
                                ':x (send sol ':x)
                                ':y (+ 250. (send sol ':y))
                                ':font fonts:solar
                                ':start-char #/S
                                ':nchars 2))
         (moon-1 (make-instance 'satellite
                              ':name "Moon"
                              ':mother earth
                              ':radius 30.
                              ':x (send earth ':x)
                              ':y (+ 30. (send earth ':y))
                              ':font fonts:solar
                              ':start-char #/m
                              ':nchars 1))
         (moon-2 (make-instance 'satellite
                              ':name "Moon"
                              ':mother earth
                              ':radius 60.
                              ':x (send earth ':x)
                              ':y (+ 60. (send earth ':y))
                              ':font fonts:solar
                              ':start-char #/m
                              ':nchars 1)))
    (doit sleep)))
```

```
;;; Try this on the value that SOLAR-DEMO returns:

(loop for o in * do (describe o))
```

fonts described in window system manual

(in editor)

```
;;;  These two functions create new objects with a nice
;;;  user-interface.  As a side-effect, the objects are added
;;;  to the list of objects.  Note that if you have DOIT
;;;  running in one Lisp Listener, and you run one of the
;;;  following functions in another, the object gets
;;;  dynamically added to the display.

(defun make-fixed-object ()
  (format t
   "~&Please enter the following pieces of information.~%")
  (let ((name (prompt-and-read ':string "Name: "))
        (x (prompt-and-read ':number "X coordinate: "))
        (y (prompt-and-read ':number "Y coordinate: "))
        (char (progn (format t "~&Starting character: ")
                     (send terminal-io ':tyi)))
        (nchars (prompt-and-read ':number
                                 "Number of chars in sequence: ")))
    (make-instance 'fixed-object
                   ':name name
                   ':x x
                   ':y y
                   ':font fonts:solar
                   ':start-char char
                   ':nchars nchars)))
```

~~cpt~~ F:cptfont

```
(defun make-satellite ()
  (format t
   "~&Please enter the following pieces of information.~%")
  (let
    ((name (prompt-and-read ':string "Name: "))
     (mother
       (prompt-and-read
         ':eval-form
         "Form to evaluate and use for satellite's mother: "))
     (radius (prompt-and-read ':number "Radius: "))
     (char (progn (format t "~&Starting character: ")
                  (send terminal-io ':tyo
                        (send terminal-io ':tyi))))
     (nchars (prompt-and-read ':number
                              "Number of chars in sequence: ")))
    (make-instance 'satellite
                   ':name name
                   ':mother mother
                   ':radius radius
                   ':x (send mother ':x)
                   ':y (+ radius (send mother ':y))
                   ':font fonts:solar
                   ':start-char char
                   ':nchars nchars)))
```

```
;;;*********************************************************

;;; Now we can add some new things easily.  For example, it's
;;; easy to make a shooting gallery target that walks back and
;;; forth.  We just define a new flavor that defines its
;;; animation behavior, (and new font/character sequence) and
;;; mix it in with basic-object and character-display-mixin.

(defflavor target-object-mixin
        (height         ;the height that it walks at
        (direction 1)   ;start going to right
         (incr 1))      ;default to one pixel at a time

           ()
  :settable-instance-variables
  (:required-flavors basic-object))



;;; This :animate method ignores the Y coordinate, and just
;;; moves X back and forth across the window.  It has to take
;;; the width of the font into account so that the display
;;; doesn't go off the edge of the window.

(defmethod (target-object-mixin :animate) ()
  (when (or
          ;; solar font is 50. wide.
          (≥ x (- (send terminal-io ':inside-width) 50.))
          (≤ x 0))
    (setq direction (- direction)))
  (setq x (+ x (* direction incr))))


;;; This is an instantiable flavor to make shooting gallery
;;; targets.

(defflavor target (name)
           (target-object-mixin
            character-display-mixin
            basic-object)
  :settable-instance-variables)
```

```
(defun make-target ()
  (format t
   "~&Please enter the following pieces of information.~%")
  (let ((name (prompt-and-read ':string "Name: "))
        (height (prompt-and-read ':number "Height: "))
        (incr (prompt-and-read
                ':number
                "Distance to move each loop (1<n<25): ")))
    (make-instance 'target
                   ':incr incr
                   ':name name
                   ':height height
                   ':x 1
                   ':y height
                   ':font fonts:solar
                   ':start-char #/c
                   ':nchars 4)))

;;; ********************************************************

;;; It's important to note that revolving-object-mixin and
;;; target-object-mixin were made separate flavors.  This way
;;; we can mix in these flavors to create new ones.

;;; Suppose that we didn't want to use
;;; character-display-mixin, but instead had a way of
;;; displaying planets in the real world, hooked up by motors
;;; and cables.  Then we could mix basic-object,
;;; revolving-object-mixin and a new flavor, 3d-display-mixin,
;;; to manage the real display.

;;; Revolving-object-mixin just defines the revolving
;;; behavior.  It should not be necessarily tied to display on
;;; the screen.  You should be able to mix this flavor into any
;;; other flavor that you want to be able to revolve.  For
;;; example, you might be implementing a program that deals
;;; with motor shafts or merry-go-rounds and then you can use
;;; revolving-object-mixin.
```

```
(defun s-demo (&optional (sleep 10)) (let* ((*list-of-displayable-objects* nil) (sol (make-instance
’fixed-object ’:name "Sol" ’:x 350.  ’:y 350.  ’:font fonts:solar ’:start-char #/§ ’:nchars 3)) (moon
(make-instance ’satellite ’:name "Moon" ’:mother sol ’:radius 10.  ’:x (send sol ’:x) ’:y (+ 10.
(send sol ’:y)) ’:font fonts:solar ’:start-char #/m ’:nchars 1))) (doit sleep)))
```

Index

Table of Contents

1. Input and Output

Streams

Files

Formatted Output

Querying the User

1.1 Streams

- All input and output is done through objects called *streams.* A stream is *a source or sink* (or both) *of characters.*
- Streams are almost always flavor instances.
- Windows are streams. (They have a stream flavor mixed in.)
- The input and output functions that we've seen so far (like **PRINT** and **READ**) take a stream as an optional argument.
- Streams accept the following messages:

:TYI	Get the next character from the stream
:TYO *char*	Put the character out to the stream

And lots of others. Usually you won't need to use stream messages other than these two.

1.2 Files

Files are nothing special. You just have to get a stream to the file.

The **open** function takes a pathname as an argument, (and optional options,) and returns a stream to that file.

You have to use the **close** function on the stream when you're done with it.

Use **with-open-file** to automatically close the stream.

1.3 Formatted Output

You can do all kinds of fancy formatted output by using the FORMAT function.

FORMAT takes as arguments:

stream - A stream or NIL or T. If you supply a stream, the output goes on that stream. If you supply NIL, then format will do **no output**, and just return the string it would have printed. If you supply T, it will go to **standard-output**, which is usually the same as **terminal-io** (the current window).

control-string - A string which is displayed on the stream. It is interpreted specially by **format**. Characters following tilde's (~'s) are *directives* to **format**. Some of them will take an argument from any further arguments supplied.

format-args - see above and below

A *directive* in a format control string means "do some display other than this exact text". Sometimes doing this doesn't need something to "do it on", like going to a new line, or clearing the screen. Other times you do need something to do it on, like when you say "display *the answer* in octal."

Here are some examples that should make this clearer. We're using a first argument of NIL so FORMAT will return a string and not do any output. If we supplied something else, **format** would display the "result" on the appropriate stream.

The **~o** and **~d** directives take an argument.

```
(setq ans 10.)

(format nil "The answer is ~o (octal)." ans)  ===>

"The answer is 12 (octal)."


(format nil
        "The answer, ~d., squared is: ~d (all in decimal)"
        ans
        (^ ans 2))  ===>

"The answer, 10., squared is: 100 (all in decimal)"
```

The **~%** directive does not take an argument

```
(format nil "~%The answer is ~o (octal)." ans)  ===>

"
The answer is 12 (octal)."
```

Other useful directives are:

	~F	Floating point notation
	~E	Scientific notation
	~&	Fresh line. Makes sure you're at the beginning of a line. Doesn't give you a newline if you don't need it.
*	~A	Prints a Lisp object like **PRINC**
	~S	Prints a Lisp object like **PRIN1**

(format nil "your response was ~a"|Jon|)
. . . Jon
. . . ~s"|Jon|)
. . . |Jon|

Even more format directives are described in the *Lisp Machine Manual*. Just about any output you'd like to do can be done with **format**.

* when used w readline just returns the string given
slashification returns various slashified results

1.4 Querying the User

Of course, you can ask questions of the user by using a combination of **format** and **read** or **readline**. But these functions are designed to do some of the more common things you need.

y-or-n-p Used as a predicate to ask the user whether to go on or not. It takes an optional argument, a string to be displayed, and waits for the user to type a "y" or an "n". It returns T or NIL based on the response.

```
(defun game-top-level ()
  (loop do
        (play-game)
        while (y-or-n-p "Play again? ")))
```

yes-or-no-p is the same as **y-or-n-p**, but requires that the user type in "yes" or "no" with a return. This is meant for questions that are more serious, and should require conscious thought on the part of the user.

```
(defun clear-out-all-known-data ()
  (when (yes-or-no-p
          "Are you sure you know what you're doing? ")
    (ok-lets-really-do-it)))
```

fquery is a general facility for asking questions that have a small, finite number of possible answers. Both **y-or-n-p** and **yes-or-no-p** use **fquery** internally.

see 4
Release
Notes p 22

Sometimes it's not clear as to whether to use **fquery** or a menu. Use a menu when either you have a lot of choices to make at once, or if all the other interaction is being done with the mouse.

fquery needs two arguments, a list of options, and a prompt. The prompt is a format control string. You can supply any number of format arguments. The first argument, the options, are usually used like this:

```
;;; Almost all uses of fquery look a lot like this.

(setq rain-force-multiplier
      (fquery '(:type :tyi
                :choices (((1 "Lightly") #/L)
                          ((2 "Pretty hard") #/P)
                          ((3 "Cats and dogs") #/C #/d)))
                "~&How hard is it raining? "))        ===>


;;; User types help, then o, then c.

How hard is it raining? (L, P, or C)
(Type L (Lightly), P (Pretty hard),or C (Cats and dogs))
How hard is it raining? (L, P, or C)
How hard is it raining? (L, P, or C) Cats and dogs
3


;;; The :readline :type is much less common, because the
;;; user has to type in the whole response.

(setq type
      (fquery '(:type :readline
                :choices ((saved "File")
                          (saved "Tape")
                          (lost "Hardcopy")
                          (lost "Screen")))
              "What happened to the data set number ~d? "
              (data-set-number *current-data*)))
```

prompt-and-read

is for, obviously enough, prompting the user and getting a response. It allows you to specify the "kind" of reading, and to specify the type of thing that must be read in.

Arguments to **prompt-and-read** are similar to those for **fquery**. It takes a *type*, which is the type of thing which must be entered by the user, and a format control string with format arguments.

Here are some examples.

```
(defun print-squares-1 ()
  (format t "~%Type 0 to stop.~%~%")
  (loop
    as n = (prompt-and-read
             :number
             "~%Type a number to square: ")
    do
    (unless (= n 0) (format t
                       "~&The square of ~d is ~d."
                       n (^ n 2)))
    while (not (= n 0))))
```

```
(defun print-squares-2 ()
  (format t "~%Type <end> to stop.~%~%")
  (loop
    as n = (prompt-and-read
             '(:number :or-nil t)
             "~%Type a number to square: ")
    do
    (when n (format t
               "~&The square of ~d is ~d."
               n (^ n 2)))
    while n))
```

The most common options for **prompt-and-read**

Option	*Action*
eval-form	Reads a Lisp form. Evaluates it and returns the value.
expression	Reads the printed representation of a Lisp object, and returns the object (without evaluating it.)
:number	Reads a number.
:string	Reads a string, terminated by RETURN or END. enter a sequence of chars
:pathname	Reads a pathname.

All of the above options have a way of returning NIL if the user just hits END. See *Release 5.0 Release Notes* for more details, or ask your instructor.

1.5 Games and Practice Using Input and Output

1. Write a function that has a nice user interface for inputting numbers and returning their average. It should allow you to type in as many numbers as you want, then hit END to finish. Then it should display the average and ask you if you want to do some more averaging.

2. Make the above function have an option to write the information out to a file when it's done.

3. Make the first function have an option that asks the user what kind of calculation should be done on the data: average, median or mode.

1.6 Answers to Some Exercises

```
(defun average-1 ()
  (let (numbers
        ave)
    (format t "~&Average program.  Type in the numbers to be averaged.~%~%")
    (loop
      as
      new-number = (prompt-and-read
                     '(:number :or-nil t)
                     "Type a number, or <end> to end: ")
      do (when new-number (push new-number numbers))
      until (null new-number))
    (setq ave (// (loop for n in numbers
                        summing n)
                  (float (length numbers))))
    (format t "~%~%The average of the numbers is: ~D." ave)))
```

```
(defun average-2 ()
  (let (numbers
        ave)
    (format t "~&Average program.  Type in the numbers to be averaged.~%~%")
    (loop
      as
      new-number = (prompt-and-read
                     '(:number :or-nil t)
                     "Type a number, or <end> to end: ")
      do (when new-number (push new-number numbers))
      until (null new-number))
    (setq ave (// (loop for n in numbers
                        summing n)
                  (float (length numbers))))
    (format t "~%~%The average of the numbers is: ~D.~%" ave)
    (when (y-or-n-p "Write this information to a file?")
      (with-open-file (str (prompt-and-read :pathname "Name of file: ")
                           ':out)
        (format str "~%;;; Format of file:  number of pieces of data, ~
                           then data, then average of data~%~%~%")
        (format str "~d  " (length numbers))
        (loop for n in (reverse numbers)
              do
              (format str "~d  " n))
        (format str "~d  " ave)))))
```