A Brief Introduction to Common Lisp

Sunil Mishra (smishra@cc.gatech.edu)

What is Lisp?

Lisp is a family of programming languages. The first Lisp was the work of John McCarthy in the late 1950's. It has existed in many forms and dialects since. The most widely used dialects today are Common Lisp and Scheme. In this class we shall work with Common Lisp.

There are many gross characteristics of Common Lisp that make it significantly different from other common languages. Lisp is generally thought of as functional, and for the most part we shall adapt that view in this class. There are many aspects of Lisp that we shall barely touch upon, so the material you will see will contain many gross simplifications, and perhaps a few white lies. This can run you into trouble, and we the TA's shall be around to help you if this should happen.

Common Lisp is Functional

Lisp as it was originally conceived of was a pure

functional language, but that is not necessarily the case any more. Common Lisp offers a variety of programming styles. Programs written in Common Lisp consequently tend to be a pragmatic combination of a variety of programming styles. These tutorials begin with an emphasis on functional programming, and slowly introduce examples of mixing in some imperative programming.

I shall not comment on imperative programming beyond saying that C is a typical example. Functional programming implies a style of programming where functions are first class objects. In other words, they can be passed as arguments, created and used very flexibly. They are as much a data object as are integers. This is perhaps the single most confusing aspect of functional programming. Hopefully you will learn to appreciate functional programming as you gain more experience with it.

Common Lisp Variables are Untyped

Programs written in Common Lisp do not require the programmer to declare the type of data ahead of time. A function written in C is often restricted in the type of argument it can take. A function written in Lisp is not restricted in this way. The data objects themselves however are typed. So the burden of type

checking falls on the Common Lisp environment, not on the programmer. This too can be confusing at first, but there are great advantages in being able to express the intent of the program without having to worry about the particulars of the data.

Common Lisp is Interactive

This is not an inherent property of the language, but a commonly accepted means of working with the language. The programmer interacts with the run-time environment, directly invoking the functions that constitute a program. A language such as C on the other hand requires the programmer to write a program, compile it, and invoke the program in a non-interactive fashion. The program (along with the functions that comprise it) may either be interpreted or compiled.

Common Lisp allows Symbolic Computation

The symbol is perhaps one of the first data types that students learning Common Lisp are taught. The ease and flexibility with which Common Lisp can manipulate symbols is one of the primary reasons why the language is a mainstay of AI research.

Common Lisp has Garbage Collection

Java made garbage collection fashionable, but many languages had it long before. The Lisp family is probably the first among them, and I believe still has the best memory management system you can expect to find. The programmer is freed from the burden of allocating and releasing data. It is all handled by the lisp environment. However, allocation and deallocation still take time. Each time a new object is created, the environment has to spend some processing power in allocating space, and deallocating and reclaiming space when the object is no longer needed.

Common Lisp is BIG

Common Lisp has a large library, comprising over 970 standard functions. It makes the language hard to master, but guarantees that any programs written in Common Lisp are portable. Much like Java.

Components of a Lisp Environment

Common Lisp environments, in their basic form, are quite simplistic, even boring. Most still

present the programmer with a command line as the basic means of interaction. This command line is called the listener. It gives the programmer access to the current state of the run-time environment.

This command line embodies the read-eval-print loop. The command line reads an expression, evaluates it, and prints out a result. Part of the power of lisp comes from the fact that each of these functions is accessible to the programmer. (However, the use of eval is heavily deprecated for a variety of reasons.)

Some modern Common Lisp environments have a fancier graphical interface available. The listener is however still the primary means for interacting with the environment. The remaining components often include editors, inspectors, debuggers, profilers, and a host of other tools. In over six years of working with Lisp I have perhaps a couple of times asked for something more than an emacs editor and the listener.

The Basic Lisp Expression

s-expressions are the primary constituent of Lisp programs. Their syntax tends to be very simple. s-expressions are composed entirely of atoms

and lists, where an atom is (approximately) anything but a list. Atoms may be directly evaluated, while lists involve function calls.

```
s-exp ::= atom | (symbol s-exp*)
atom ::= symbol | number | string | <other things>
```

In other words, an s-expression is an atom or a list expression. Each s-expression returns a value. The definition of an s-expression is recursive. Arbitrarily complex expressions can be created by composing s-expressions in this manner. This can however lead to programs that are hard to read, and is not advisable.

If the s-expression is a single atom, then the action the lisp environment takes depends on the type of atom. If it is a symbol, then the value of the symbol is accessed. Otherwise, atoms usually evaluate to themselves. (5, for example, returns 5 as its value.)

List s-expressions are quite straightforward too. The first component of these s-expressions is always a symbol. Evaluating the s-expression corresponds to executing the function associated with this symbol. The s-expressions following the symbol are also evaluated, and their values are used as arguments to the function. In the example below, the function associated with the symbol + is accessed. The atoms 2 and 3 are evaluated, and their return

values (themselves) are used as arguments to the function +. The value of the s-expression is returned as the atom 5.

```
> (+ 2 3)
5
```

Lisp, as you can see, uses prefix syntax.

Example 1 - Factorial

We shall first go over a simple example to demonstrate lisp syntax before examining more details of the language. Factorial has a simple recursive mathematical definition:

```
fact(0) = 1
fact(n) = n*fact(n-1)
```

Below are two methods for implementing factorial. In general, you will find that there are any number of ways of expressing an idea in lisp, and some of them turn out to be better than others.

Version 1:

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (1- n)))))
```

This is the literal interpretation of the definition

of factorial. Remember that each expression in Lisp returns a value, which allows for a simple translation of a variety of mathematical and logical constructs into equivalent programs.

The entire form is a single s-expression that constructs a function definition. The return value of this s-expression is the name of the function. DEFUN is a symbol in the function slot of the s-expression. It is not a function, but a special form. Unlike a function invocation, it does not necessarily evaluate all its arguments.

Like DEFUN, IF is a special form. It takes a variable number of s-expressions (two or three) in its argument slots. The first argument is necessarily evaluated. It is usually a predicate (a function that returns a boolean value). If the predicate returns true, then the second argument is evaluated. If false, the third argument is evaluated. The value of the entire IF s-expression is either the value of the second argument or the third, depending on which one is evaluated. If the third argument is absent, then NIL is returned.

The remaining s-expressions are mathematical functions that evaluate as you would expect them to. Let us run a few tests:

```
> (factorial 4)
24
```

```
> (factorial 30)
265252859812191058636308480000000
```

Note on the last one that the result value was an exact integer. Common Lisp is limited only by memory in the size of the integers it can work with. You don't have to worry about overflows and such.

Version 2:

```
(defun factorial-1 (n product)
  (if (= n 0)
      product
      (factorial-1 (1- n) (* n product))))

(defun factorial (n)
  (factorial-1 n 1))
```

At first glance, this version of FACTORIAL seems to be inferior to the first. It uses two functions, while the first uses only one. Version 2 is thus more verbose, and potentially more confusing. But this version has a very important advantage, for it allows a type of optimization called tail call elimination.

Consider the stack usage of version 1:

```
(factorial 3)
=> (* 3 (factorial 2))
=> (* 3 (* 2 (factorial 1)))
=> (* 3 (* 2 (* 1 (factorial 0))))
=> (* 3 (* 2 (* 1 1)))
```

```
=> (* 3 (* 2 1))
=> (* 3 2)
=> 6
```

Now let's look at version 2:

```
(factorial 3)
=> (factorial-1 3 1)
=> (factorial-1 2 3)
=> (factorial-1 1 6)
=> (factorial-1 0 6)
=> 6
```

Version 1 requires the lisp environemnt to keep track of the partial results of the recursive calls, while version 2 keeps track of the partial result as an explicit argument. This allows the lisp environment to optimize version 2, by reusing the stack space of the previous call. Thus, version 2 can be easily reduced to a simple loop.

Of course, it is not always possible to reduce a recursive function to a loop. Factorial involves linear recursion, since a call to the function factorial results in at most one new call to factorial. In a tree recursive algorithm, a call to the function can result in multiple new calls to the function, which may make tail call elimination difficult or impossible. We shall see an example of tree recursion with Towers of Hanoi, below.

Common Data Types (and Associated Functions)

Lisp has a fairly extensive and complex type hierarchy. Listed below are many of the major types of data I expect you will have to work with through the course of the quarter. Under each type I have listed a set of useful functions. This list is far from exhaustive, so I suggest you look into a text to get more information.

Lists and Cons

Lists, like symbols, have a special place in the lisp type hierarchy. They are a very convenient default representation when exploring various aspects of a program or problem. They are also the underlying representation for non-atomic s-expressions. Thus, lists provide us with the building blocks for both data and programs. This interchangeability of data and programs affords a great deal of power to lisp programs.

A list in turn is composed of a series of cons cells. A cons cell is abstractly an object made of two slots. More concretely, it can be thought of as a two element array. Each of the slots can hold any kind of value, including other cons cells. Effectively, cons cells give us the ability to construct arbitrary tree structures.

Lists have a recursive definition.

1. () is the empty list.
2. A cons cell with its second component () is a list of length 1.
3. The length of a list is one more than the length of the list in its second component.

Note that:

```
> ()
NIL
```

Thus, NIL is also the empty list.

cons

Allocates a cons cell.

```
> (cons 1 2)
(1 . 2)
```

The result of the above expression is called the dotted pair representation of a cons cell.

consp

Tests if the argument is a cons cell.

```
> (consp (cons 1 3))
T
```

list

This function is the primary list constructor.

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

It automatially allocates and strings together all the necessary cons cells. The above expression is equivalent to

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

Note that the result contains no dotted pairs. That is because for readability the dotted pair representation is compressed to give a compact list representation. Generally, if the second component of a cons cell is another cons cell or (), the cons cell is not printed as a cons cell.

```
> (cons 1 (cons 2 3))
(1 2 . 3)
```

The first cons cell is not printed as a dotted pair, while the second is.

car

This accesses the first component of a cons cell.

```
> (car (cons 1 2))
1
```

cdr

This accesses the second component of a cons

cell.

```
> (cdr (cons 1 2))
2

> (cdr (list 1))
NIL
```

caar, cadr, cdar, cddr...

These functions are abbreviations for a series of applications of CAR and CDR. Thus,

```
(caar x) <-> (car (car x))
(cadr x) <-> (car (cdr x))
```

While convenient, excessive reliance on CAR and CDR can quickly make a program unreadable.

first, second... tenth

These access the first, second... upto the tenth element of a list.

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

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

nth

This retrieves the nth element of a list. Note that

the elements of a list are zero based, rather than one based.

```
(nth 0 x) <-> (first x)
```

nthcdr

This function retrieves the nth cons cell of a list. The elements are again zero based.
(car (nthcdr 0 x)) <-> (nth 0 x)

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

last

This function returns the last cons cell of a list.

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

length

This function returns the length of a list (and other objects that have a length).

listp

This function tests if the object is a list. It returns true if the object is either a cons cell or the empty list. CONSP on the other hand tests only if the object is a cons cell.

```
> (listp ())
T

> (consp ())
NIL
```

endp

This function tests if we are at the end of a list. In other words, it tests if the object is ().

```
> (endp ())
T

> (endp nil)
T
```

In other words, ENDP performs precisely the same function as NULL. There are two different functions for the same purpose, since they communicate different information back to the reader. ENDP indicates that the argument is a list, while NULL does not. Consequently, ENDP should be used in preference to NULL when lists are involved.

Symbols

Symbols are special objects in Common Lisp. They are used for naming a variety of objects, including variables and functions. They consequently play a special role in lisp programs.

A symbol is in fact an object with some internal structure. Two of these slots are of consequence to the programmer. They store the variable value and the function value of the symbol. In other words, a symbol can simultaneously have distinct variable and function values. The variable value of a symbol can be accessed by typing in the name of the symbol at the lisp prompt. For example,

```
> most-negative-single-float
-3.4028238E38
```

A symbol can be almost any string of characters. For example, +, A, A1, 1A, $ and $%* are all symbols. There are a few exceptions though, involving characters that play a special role in lisp syntax. Some of them are:

- A single . is not a valid symbol.
- A symbol may not begin with a #.
- A symbol may not contain | or ' or ` or ,.
- (and) cannot be in a symbol.

symbolp

This predicate tests if an object is a symbol. It returns T if the object is a symbol, and NIL otherwise. Look below for more information about T and NIL.

```
> (symbolp nil)
```

```
T

> (symbolp 4)
NIL
```

T/NIL

These are probably the most overloaded and confusing symbols in lisp. Both of them are special in similar ways, so they are listed together.

As I have noted above, symbols can have variable values. T and NIL are special, in that they are constant objects that have themselves as variable values. In other words

```
> nil
NIL

> t
T
```

One of their many functions is to act as booleans. NIL means false, and T means true. However, true is in actuality a lot more fuzzy, since anything not NIL is also true. This can be very handy, but can get rather confusing as additional meanings of NIL are introduced.

null

Tests if an object is nil. If so, returns T,

otherwise NIL.

Exercise: What is (null (null t))?

Atoms

Atoms are anything that are not cons cells. The rest of the types listed below (including strings) are thus classified as atoms.

atom

This predicate tests if an object is an atom. It can quite literally be defined in terms of consp.

```
(atom x) <=> (not (consp x))

> (atom 1)
T

> (atom (list 1))
NIL

> (atom nil)
T
```

Numbers

We are all familiar with numbers. We know of integers, rationals, reals, complex numbers, etc. Lisp provides a rich vocabulary of numbers. We shall only discuss integers for now.

As noted under the factorial example, numbers in lisp are not limited by the underlying machine architecture. They can be of arbitrary size, only limited by the available memory. All arithmetic operators function as one would expect on these numbers.

arithmetic operators (+, -, *, /, 1+, 1-)

The function + without any arguments returns 0. With one argument it returns the number it was given. With more than one argument it returns the sum of all of its arguments.

The function - requires at least one argument. If given only one argument, the return value is the argument's negation. If given more than one argument, the return value is the first argument minus the sum of the remaining arguments.

The function * without any arguments returns 1. With one argument the result is the argument itself. With more than one argument the product of all the arguments is returned.

The function / requires at least one argument. With one argument it returns the inverse of the argument. When more than one argument is supplied, the result is the first argument divided by the product of the remaining arguments.

1+ and 1- add 1 and subtract 1 from their arguments, respectively.

integerp

Tests if the argument is an integer.

oddp/evenp

Tests if the argument is odd or even, respectively.

Functions

Functions, as has been noted, are first class objects in Common Lisp. We can construct them, pass them around, invoke them, and test for them.

function

This is a special form, not a function. It returns the function value of the symbol supplied, and an error if none exists.

```
> (function +)
#<Function +>
```

The special form FUNCTION is hardly ever used. Special syntax provided in Common Lisp gives us a handy short cut.

```
> #'+
#<Function +>
```

functionp

This predicate tests if the supplied argument is a function.

```
> (functionp #'+)
T
```

funcall

This function invokes the function supplied as its first argument on the arguments supplied.

```
> (funcall #'+ 1 2 3)
6
```

Thus, the above function call is equivalent to (+ 1 2 3). It is useful when a function has been passed as an argument, and cannot be executed in the usual manner.

apply

APPLY is like FUNCALL, in that it executes a function. However, it takes arguments in a different format. The last argument to APPLY is interpreted as a list of arguments to be given to the function in the first argument's place.

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

```
6

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

All arguments other than the last are interpreted as individual arguments for the indicated function.

Strings

Strings are vectors of characters. Literal strings are indicated by double quotes surrounding a series of characters, and evaluate to themselves.

```
> "foo"
"foo"
```

stringp

This function returns T if its argument is a string.

length

This function returns the length of the string. (You may recall that this function also returns the length of a list.)

Booleans and Predicates

eql

EQL tests if two objects are the same. Integers are usually EQL if they are the same number. Symbols are always EQL to each other. Complex objects that involve allocation may or may not be EQL. A cons cell is always EQL to itself, but never EQL to another cons cell that merely has identical contents.

```
> (eql t t)
T

> (eql nil t)
NIL

> (eql 'a 'a)
T

> (eql 4 4)
T

> (eql (cons 1 2) (cons 1 2))
NIL

> (let ((x (cons 1 2)))
    (eql x x))
T
```

equal

EQUAL, unlike EQ, ignores the issue of cons cell identity. It simply requires that the two objects have the same content. In other words, it is

more general than EQL in its application.

```
> (equal (cons 1 2) (cons 1 2))
T
```

=

The = predicate is intended specifically for comparing numbers. Giving it non-numeric arguments will result in an error. It can take multiple arguments, in which case it will test if all the arguments have the same numerical value.

```
> (= 6 (+ 2 4) (* 2 3))
T
```

typep

As has been stated earlier, data rather than variables in Common Lisp are typed. TYPEP is a predicate intended to check the type of the data.

```
> (typep 5 'number)
T

> (typep nil 'list)
T

> (typep nil 'cons)
NIL
```

Control Structures

In addition to functions, Common Lisp has a variety of special forms. A call to a function results in all the arguments being evaluated before the function is executed. Special forms are different, in that their arguments are selectively evaluated. They may also produce a variety of other effects that functions cannot. Thus, special forms are an essential component for controlling the path along which a computation proceeds. All special forms in Common Lisp, like functions, have a return value.

defun

This form is responsible for constructing globally accessible functions. It returns the name of the function as its value. The full syntax is:

```
(defun <function-name> <argument-list>
  <form1>
  <form2>
  ...
  <formN>)
```

<function-name> has to be a valid symbol. (There are some exceptions that you should not concern yourself with just yet.) This symbol is returned as a result of the DEFUN expression. The <argument-list>, properly called the lambda list, is in its simplest form a list of

symbols to which the arguments will be bound. When the function is executed, a new lexical context is created within which these bindings exist. It is in this context that <form1> through <formn> are evaluated, and the return value of the function is the result of the evaluation of <formn>.

We have already seen the FACTORIAL example and a sample use of a DEFUN expression.

lambda

DEFUN is limited in that it can only construct globally accessible functions that are named. For taking advantage of functional programming paradigms, it is necessary to be able to create functions at run-time. The LAMBDA form provides this functionality. It creates an anonymous (unnamed) function that is otherwise identical to regular functions. The syntax of the LAMBDA expression is also identical to the syntax of a regular function definition. For example,

```
> (funcall #'(lambda (x) (if (oddp x) (1+ x) x)) 5)
6
```

As has been stated earlier, functions in Common Lisp are simply objects. We have seen two methods of creating functions so far. One is through DEFUN, which creates a named

function, and the second is through LAMBDA, which creates an anonymous function. The main difference is that the symbol that names a function can be at the head of a complex s-expression, while an anonymous function must be invoked using FUNCALL or APPLY.

It is important to note the commonalities and differences between the two forms as well. The structure of the function definition is identical in both cases. A function definition requires the programmer to give a lambda (argument) list followed by a series of forms to evaluate. Both DEFUN and LAMBDA proceed to create function objects that behave identically when invoked. In fact, there are other special forms that create function objects. We shall not discuss them in the course of this tutorial.

quote

The QUOTE special form instructs lisp to take the argument literally. That is,

```
> (quote x)
X
```

Without the QUOTE, Common Lisp would attempt to access the variable value of the symbol. This form is used very often, to the point that it has a syntactic short-cut, the single quote.

```
> 'x
X
```

Symbols are not the only objects that may be quoted.

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

> '(1 . 2)
(1 . 2)

> (car '(1 . 2))
1
```

if

IF is one of the many conditional evaluation forms in Common Lisp. Its syntax is:

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

On evaluation, the IF form first evaluates the <test> expression. If the value is non-NIL, the IF form returns the result of evaluating the <then-form>. Otherwise, the <else-form> is evaluated.

We have seen an example of the IF expression in FACTORIAL.

cond

COND can be thought of as a series of IF expressions. The full syntax is:

```
(cond (<test1> <form11> <form12> ... <form1K>)
      (<test2> form21> ... <form2L>)
      ...
      (<testN> <formN1> <formN2> ... <formNM>))
```

The evaluation of a COND begins with the environment evaluating <test1>. If the test returns non-nil, then the forms following <test1> are evaluated, and COND returns the value of the last form under the test. If <test1> returns NIL, the environment proceeds to the next test. If all the tests are exhausted, then NIL is the return value of the COND.

A COND expression may have its last test (<testN> above) be T. This then is the default clause, whose forms are evaluated if all the other tests fail.

```
> (let ((x 50))
    (cond ((oddp x) (1+ x))
          ((> x 10) (/ x 2))
          (t x)))
25
```

when/unless

WHEN and UNLESS share a common syntax:

```
(when/unless <test>
  <form1> ... <formN>)
```

For a WHEN expression, the <test> is evaluated. If it returns non-NIL, <form1> through <formN> are evaluated, and the return value of the WHEN expression is the return value of <formN>. UNLESS differs from WHEN in that it evaluates <form1>... <formN> when <test> returns NIL.

```
> (when t 5)
5

> (when nil 5)
NIL

> (unless t 5)
NIL

> (unless nil 5)
5
```

let/let*

LET creates a new lexical scope along with the specified bindings, and evaluates a series of forms. The full syntax of LET is

```
(let (<binding1>... <bindingN>)
  <form1>... <formN>)
```

where each binding has the form

```
(<symbol> <value-expression>)
```

The return value of the LET expression is the result of evaluating <formN>. It is instructive to

compare LET with DEFUN. The symbols that LET binds are loosely equivalents of the arguments specified in a function definition. Like the arguments of a function, the values of the bindings are all evaluated in parallel. Then the forms that follow are evaluated, and the last form's value is returned.

LET* is different from LET in that it evaluates its bindings sequentially, so that the variable in <binding1> is available when the <value-expression> for <binding2>... <bindingN> are evaluated.

```
> (let ((x 2)
        (y 3))
    (+ x y))
5

> (let ((x 2)
        (y (* x 2)))
    (+ x y))
ERROR x is unbound ...

> (let* ((x 2)
         (y (* x 2)))
    (+ x y))
6

> (let ((x 2))
    (let ((x 4)
          (y (* x 2)))
      (+ x y)))
8
```

```
> (let ((x 2))
    (let* ((x 4)
           (y (* x 2)))
      (+ x y)))
12
```

or

OR takes multiple arguments. It evaluates its arguments until it finds one that is non-NIL. This value is its return value.

```
> (or nil 5 t)
5
```

and

AND, like OR, takes multiple arguments. It evalutes its arguments until it finds one that is NIL. If one is found, the return value of the AND expression is NIL, otherwise it is the value of the last argument evaluated.

```
> (and nil 5 t)
NIL

> (and 5 t 'x)
X
```

not

NOT takes a single argument. If the argument is NIL, it returns T, otherwise it returns NIL.

```
> (not 'x)
NIL

> (not nil)
T
```

defvar

DEFVAR defines a global variable. It returns the name of the variable as its value.

```
> (defvar *foo* 5)
*FOO*
```

The circumstances in which global variables are used are few, and excessive use can make programs difficult to read and debug. Use them sparingly.

setq

SETQ destructively modifies the variable value of a symbol. It's full syntax is:

```
(setq <symbol1> <value1>
      <symbol2> <value2>
      ...
      <symbolN> <valueN>)
```

It does not establish a new lexical scope, but simply modifies the value of a variable within the current scope. It should be used as sparingly as possible. The return value of SETQ is the result of evaluating <valueN>.

```
> (let ((x 5))
    (setq x (1+ x))
    x)
6
```

setf

SETF destructively modifies the value in a place. A place is an abstract concept. A symbol's variable value is an example of a place, so SETF is more general than SETQ in its scope.

```
> (let ((x (cons 1 2)))
    (setf (car x) 4)
    x)
(4 . 2)
```

Lexical Scopes and Closures

As has been noted, a number of Common Lisp forms produce variable bindings. Such bindings are called lexical bindings, since they are determined at load/compile time. Their extent is dependent upon the structure of the program, rather than the nature of the execution of the program. For example, consider

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

The LET form defines a lexical scope, within which variables X and Y are bound to 1 and 2,

respectively. The expression (+ X Y) within this lexical scope accesses these bindings when it is evaluated. A lexical scope is roughly equivalent to a block in C.

Closures are one of the trickier topics for students new to lisp. A closure is simply a lexical environment within which a function has been defined. A closure then preserves all the bindings of the lexical environment for the function to use as needed. Consider the following:

```
> (let ((x 5))
    (defun foo ()
      x))
FOO

> (foo)
5
```

The value of the binding X is now completely inaccessible. It cannot be modified or accessed in any way, except by calling the function FOO. We have just constructed a lexical closure around FOO. Closures are rarely useful in this form, but there are a variety of other situations where they can be very powerful. Consider:

```
(defun make-foo-predicate (arg)
  (let ((processed-arg (some-complicated-processor arg)))
    #'(lambda (x)
        (foo-same-p x processed-arg))))
```

This paradigm represents a method for constructing a closure that specifies the context for a customized predicate. This style of abstraction is often useful for providing abstractions otherwise difficult to provide.

Programming Style

Be Expressive

Use descriptive names. Spending a little time here can save a lot of time down the line debugging and documenting code. If done right, code becomes essentially self-documenting. This is one of Lisp's strengths.

Doing this right can be difficult. The ability to arbitrarily compose expressions is confusing to some, but is ultimately a great benefit when authoring a program.

Also, use the most specific possible function when you have a choice of forms to use. This is useful in many ways. It provides a form of self documenting code, in that the particular function used can tell the reader what data you had expected. When testing for end of list, it is better to use ENDP than NULL for this reason. NOT is preferable to NULL when you are expecting a boolean value. The other advantage

is that you can catch errors at the right point, if you use a more restrictive function. For that reason it is better to use NTH than ELT if you want to refer to a particular element of a list.

Finally, you should start wondering about your abstractions if you find yourself using a lot of CAR and CDR and other list destructuring operations while accessing data.

Use Appropriate Formatting

Lisp has an extensive set of programming rules that have developed over time. The structure of a program is generally expressed through appropriately indenting lisp code. Use emacs (xemacs), for they can generally do a good job of appropriately indenting your code.

Never use excessive whitespace. Compare:

```
(defun factorial-1 (n product)
  (if (= n 0)
      product
      (factorial-1 (1- n) (* n product))))
```

with

```
( defun factorial-1 ( n product )
  ( if ( = n 0 )
       product
       ( factorial-1 ( 1- n ) ( * n product ) )
       ) ; end if
  ) ; end defun
```

The latter for an experienced lisp programmer is really hard to read. The extra whitespace is confusing. The parentheses on lines by themselves even more so. In the first version, all you have to do is look at the indenting, and you can correctly identify the extent of each expression. In the second version, this process becomes very difficult.

Comments and Documentation

Use comments sparingly, only to point out things that cannot be directly and explicitly demonstrated in the code itself. Excessive comments are a sign that the code you are writing is not well structured. Consider rewriting parts of it.

A comment begins with a ; and extends for the remainder of the line. There are three generally accepted levels of comments. A single ; is an in-line comment, generally indicating a minor point relevant to a single line of code. Two ;; are used for comments inserted between lines of a function, and are for describing higher level issues within a function. Three ;;; are for comments between functions, and are for describing high level aspects of the program.

Finally, block comments are indicated by #| ... |# pairs. They are equivalent to /* ... */ in how

they work, with one major advantage--they can be nested.

Each function is allowed a documentation string. For example,

```
(defun factorial-1 (n product)
  "Helper function for calculating the factorial of a nur
  (if (= n 0)
      product
      (factorial-1 (1- n) (* n product))))
```

This documentation string immediately follows the function lambda list, and should concisely describe the function. For major top-level functions the documentation string is often much more elaborate.

cond/if/when/unless

These special forms are intended to control the flow of a program, and each should be used in particular circumstances.

WHEN and UNLESS indicate that you only care about the consequent case, and not the alternate. IF on the other hand indicates that you care about both the consequent and the alternate cases. IF however is restricted to a single expression for the alternate and the consequent. A programmer that requires either of these to be multiple lines, or requires to

specify else-if cases, should consider using COND.

let/let*

LET and LET* are useful for introducing intermediate bindings. LET introduces bindings in parallel, while LET* introduces them sequentially. By default, you should use LET. If you find yourself in a situation where you are writing code like this:

```
(let ((x ...)
      (y ...))
  (let ((z (f x)))
    ...))
```

Consider switching to LET* in this situation. Your code would then reduce to

```
(let* ((x ...)
       (y ...)
       (z (f x)))
  ...)
```

Global Variables

In short, avoid them like the plague. They make programs difficult to debug and follow. And they are generally unnecessary, especially in LISP. If you do have to use them, make the really obvious. The accepted convention is to surround a global variable with *. For example, *foo*

would indicate a global variable, while foo would be a local binding.

Example 2 - Towers of Hanoi

The Towers of Hanoi is a relatively simple problem that turns out to be very difficult to solve. To recap, the problem involves moving rings from a start peg to a destination peg, using a third peg as spare space. The only constraints are that only one peg at a time may be moved, and a larger peg may never be on top of a smaller peg.

The problem is conceptually fairly simple to solve. To move n rings from the start to the destination, move n-1 from the start to the spare, then move the nth ring from the start to the destination. Finally, move the n-1 rings on the spare to the destination, using the start peg as the spare. Our task is to write a function that tells us what moves to make to get all n rings from the start peg to the destination peg. From this description alone, it is possible to deduce that the most natural way of expressing the solution is as a recursive function, and that the recursion will be tree recursion rather than linear recursion.

In typical lisp style, we want to write a function

TOWERS-OF-HANOI which takes one argument, the number of rings on the start peg. Our return value should be a list of moves. Given the nature of the problem, a very simple system suffices for describing the move. We can only move one ring, the top one, at a time, so it is sufficient to state the source peg and the destination peg to fully describe a move. A cons cell is ideally suited for storing such a pair. We thus have a data input and output specification in lisp terms, and a conceptually straight-forward algorithm. This yields the following implementation:

```
(defun toh (height start-peg-name extra-peg-name end-peg
  (cond ((= height 0) nil)
        ((= height 1) (list (cons start-peg-name end-peg
        (t (append (toh (1- height) start-peg-name end-pe
                        extra-peg-name)
                   (toh 1 start-peg-name extra-peg-name e
                   (toh (1- height) extra-peg-name start
                        end-peg-name)))))

(defun towers-of-hanoi (height)
  (toh height 'start-peg 'extra-peg 'end-peg))
```

There are only a handful of issues to point out here. Note the use of COND rather than nested IF statements. COND is preferentially used when such nesting is necessary. The other is the consistent application of the choice of representation. Every return value that TOH can possibly return is a list, so it is trivial to apply the APPEND operator to the results of the

different recursive steps. As you might guess, APPEND sticks together the contents of all its argument lists into a single list. It allocates new cons cells as necessary.

Exercise: Try writing a definition for APPEND.

Sample runs (formatted for neatness):

```
> (tower-of-hanoi 3)
((START-PEG . END-PEG) (START-PEG . EXTRA-PEG) (END-PEG
 (START-PEG . END-PEG) (EXTRA-PEG . START-PEG) (EXTRA-PE(
 (START-PEG . END-PEG))

> (tower-of-hanoi 4)
((START-PEG . EXTRA-PEG) (START-PEG . END-PEG) (EXTRA-PE(
 (START-PEG . EXTRA-PEG) (END-PEG . START-PEG) (END-PEG
 (START-PEG . EXTRA-PEG) (START-PEG . END-PEG) (EXTRA-PE(
 (EXTRA-PEG . START-PEG) (END-PEG . START-PEG) (EXTRA-PE(
 (START-PEG . EXTRA-PEG) (START-PEG . END-PEG) (EXTRA-PE(
```

Note the exponential growth of the size of the solution. I had read once that Vietnamese tradition had it that the world would end when a puzzle of 64 rings, each made of stone, was solved. My apologies if this is a bit of urban legend I happened to pick up. The analysis that followed demonstrated that such a feat would take an impossibly long time. Something like half the age of the universe if a ring was moved every second.

Example 3 - Matrix

Transpose

This is a relatively simple problem that nevertheless demonstrates many different aspects of solving problems in Lisp.

We shall use the tools we have picked up so far for the task. The only data structure we have to represent a sequence of objects is a list, so our choice is rather limited. (I would not recommend that anyone use lists to represent matrices though. Lisp has multi-dimensional arrays.) We will use a row-major representation, thus:

```
Empty matrix => nil
m by n matrix => ((x11 x12 ... x1n)
                  (x21 x22 ... x2n)
                  .
                  .
                  (xm1 xm2 ... xmn))
```

The transpose problem can be viewed as follows:

1. Turn the row into a column.
2. Add it to the transpose of the rest of the matrix.

For the first step, we may assume that rows and columns have the same representation - a list. So The first step is a non-issue. The second step is a little more complex. There are two cases we

must consider--add a column to the empty matrix, and add a column to an existing matrix. Both of these problems are ideally suited for the functional approach.

First, consider adding a column to an empty matrix. The following transformation can serve as an example:

```
(add-column (list 1 2) nil) => ((1) (2))
```

Each row has exactly one element, and the number of rows is the same as the length of the column. Looking at this operation from the perspective of list operations, we have simply enclosed each element of the column in its own list. This is called a mapping operation, where the same transformation is applied to all the elements of a collection. The functional style provides convenient abstractions for performing such mappings, through the function MAPCAR.

Adding a column to an existing matrix is similarly easy. Conceptually, we want to add each element of the column to the corresponding row in the matrix. That is, we want a mapping from the column and existing matrix to a new matrix that combines the two.

Here is the final implementation:

```
(defun empty-matrix-p (m)
```

```
  (null m))

(defun add-column (column m)
  (if (empty-matrix-p m)
      (mapcar #'list column)
      (mapcar #'cons column m)))

(defun matrix-transpose (m)
  (if (empty-matrix-p m)
      m
      (add-column (car m) (matrix-transpose (cdr m)))))
```

Note the definition of EMPTY-MATRIX-P. It simply checks if its argument is NIL. It is generally inadvisable to write functions that do so little, unless it is for providing an appropriate abstraction. The advantage in creating such abstractions is that the underlying representation can be modified, without having to worry very much about the effect on the remainder of the program. This level of abstraction has not been used uniformally in MATRIX-TRANSPOSE, which is a potential problem.

Some examples, again formatted for legibility:

```
> (matrix-transpose nil)
NIL

> (matrix-transpose (list (list 1 2) (list 3 4)))
((1 3)
 (2 4))
```

Example 4 - mapcar

The MATRIX-TRANSPOSE example introduced the notion of mapping operations. I have included this example to demonstrate that there is little magic behind this kind of a mapping function. MAPCAR can be implemented entirely in Common Lisp, using facilities common to any programmer. It is a fairly tricky function to understand, so I shall first demonstrate a special case which maps the items in one list. I shall then construct a more general implementation of MAPCAR that can map the elements of multiple lists to constuct one resulting list.

Single list argument

The single list case is quite simple. It takes two arguments. The first is a function that maps individual elements. The second is a list of elements to map. What we want is a function that attaches the result of mapping the first element of the list to the result of mapping the remaining elements.

Below is the implementation of this simple linear recursive function:

```
(defun my-mapcar (fn list)
  (if (endp list)
      nil
```

```
(cons (funcall fn (car list))
      (my-mapcar fn (cdr list)))))
```

Multiple list arguments

There are quite a few new issues that appear when we try to implement the general case, where the mapping is from the elements of multiple lists. The algorithm is still linear recursive, even though the implementation appears to be tree recursive. We ask the same questions that we would when writing any recursive producdure:

What is the base case?

The answer turns out to be surprisingly tricky. Given a single list, the base case is when the list is empty. There are no more elements left to map. With multiple lists, it is possible that some of the input lists are longer than others. The simplest solution is to terminate mapping as soon as any of the lists is exhausted. This also makes intuitive sense, since exhausting even one list makes collecting the arguments for the mapping function.

What is the recursive step?

We want to attach the result of applying the function to the first element of every list, to the

result of mapping the rest of every list. This is directly analogous to the recursive step of the single list case.

The implementation

```
(defun my-mapcar (fn first-l &rest rest-l)
  (if (or (endp first-l) (some #'endp rest-l))
      nil
      (cons (apply fn (car first-l) (my-mapcar #'car rest
            (apply #'my-mapcar fn (cdr first-l) (my-mapca
```

This is the first function we have written that takes multiple arguments, so it deserves special attention. Note the &REST in the lambda list of MY-MAPCAR. &REST is a special keyword that instructs the environment to collect the remaining arguments supplied to a function into a list, which is bound to the supplied name. A simpler example is given below:

```
> (defun my-list (&rest l)
    l)
MY-LIST

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

This function replicates the functionality of the function LIST, though this is not the recommended implementation. Note that individual items supplied to MY-LIST are collected into a list.

Let us return now to MY-MAPCAR. Note that we require at least one list to be given as an argument, so we have both a regular list argument and an &REST argument.

The next step is to implement the predicate for the base case. Testing for the end of a single list is the same as it was for the single list case. The predicate ENDP gives us what we need. We use the function SOME to test for the end of a list of lists. SOME is a specialized mapping function, that tests if the predicate returns non-NIL for some element of the argument list. (It too can take multiple lists as arguments.)

Exercise: Look up the definition of SOME. Try to implement it.

The recursive test is again complicated by the existence of multiple lists. Now that we have established that none of the lists is empty, we want to obtain the result of calling the function with the first element of each of the lists. The first element of a single list can be trivially obtained through CAR. Obtaining the first element of a collection of lists is again a mapping operation, for which we again use MY-MAPCAR, where we fetch the CAR of each element in the collection of lists. We then invoke the mapping function using APPLY, since it can take a list of values as arguments to apply, while

FUNCALL cannot.

The recursive call to MY-MAPCAR again involves a similar set of collection functions. Make sure you understand how it works.

Last modified: Tue Apr 6 16:59:40 EDT 1999