MCS-385 **Notes on Lisp** February 8, 2003
David Wolfe (adapted from Stuart Russell)

These notes review the basic ideas of symbolic computation and functional programming as embodied in LISP. We will cover the basic data structures (s-expressions); the evaluation of functional expressions; recursion as the expression of repetition; binding and equality; user-defined data structures (defstructs). With minor differences which will be pointed out, LISP and Scheme (which you should recall from MCS-177 and 178) are very similar. These notes intended mainly as a refresher for students who have seen Scheme a while back. For a thorough introduction and a complete reference, I strongly recommend Paul Graham's *ANSI Common Lisp*; a copy is in the lab.

Characteristics of LISP

The main characteristic of LISP is its capability for *symbolic computation*. Symbols (atoms) are the principal data type. The operations that can be performed on symbols include equality testing and building up symbol structures. Putting two symbols together creates a structure, which can then be accessed and taken apart.

Typical applications:

- Language processing, using words as symbols; lists for sentences, trees for grammatical structure.
- Mathematics, involving expressions and equations; trees for expressions.
- Manipulating programs — these are just pieces of (formal) language.

LISP programs are just symbol structures, so LISP programs can modify or create LISP programs. Hence programs can implement the results of *learning* by writing new code for themselves; it is also easy to write interpreters for new languages in LISP.

LISP is a *functional* language: compute by evaluating nested functional expressions. (Pure) programs have no *side-effects*, so they are modular. Simple semantics (i.e., it's easy to tell what a program does) allows for powerful program development environments.

Evaluating functional expressions

Read-eval-print: LISP reads the expressions you type, evaluates them and prints the value.

Expressions are as follows:

Atoms can have a value associated with them, e.g. `X` might have the value `5`.

Numbers (which are also atoms) evaluate to themselves.

Functional expressions are delimited by matching parens: (fn arg1 ... argn) applies `fn` to the arguments as follows:

1. *Evaluate* each argument in turn, then
2. *Apply* the *function definition* of `fn` to the results.

Note the difference from Scheme: in Scheme, every position including the `fn` position is evaluated; in LISP, symbols can have a function definition pointer distinct from the value pointer. More on this later.

Sometimes we want to pass an argument directly, without evaluation. To do this we need an identity function, which doesn't evaluate ITS argument. `QUOTE` serves this purpose.

- `(QUOTE A)` or `'A` evalues to `A`
- `(+ '4 '4)` returns 8, but `(+ '(+ 2 2) '(+ 1 3))` is an error.

`(+ 2 2)` is just a piece of *list structure*. The next section discusses how to build list structures from atoms.

Operations on list structure

List structure is made by putting symbols together. The function that puts things together is `CONS`.

`(cons (cons 'a 'b) 'c)` evaluates to `((a . b) . c)`

The parts of cons-pairs are accessed using `car` and `cdr`:

`(car (cons 'a 'b))` is `a`; `(cdr (cons 'a 'b))` is `b`.

Lists, of which the functional expressions used above are examples, are special kinds of cons-expressions whose rightmost element is the special atom `NIL`. they have a special printed representation:

`(cons 'a (cons 'b NIL))` evaluates to `(a b)`

(Note that the missing "." indicates a list, a special kind of cons pair.) CDRing down a list eventually returns `NIL`. `NIL` is the empty list.

Lists can be constructed using `list` as well as `cons`:
`(list 'a 'b '(c d))` evaluates to `(a b (c d))`.

Defining functions, conditionals and temporary variables

The `defun` function is used to associate a *function definition* with a symbol (not the same as giving the symbol a value). Note that `defun`, since it effects the global environment, is a function with side-effects. (`defun` is like Scheme's `define`, except it only effects a symbol's function pointer.) The following code defines a function `1/` which calculates $1/x$.

```
(defun 1/ (x &optional (checkp nil))  ;comments can follow semicolons like this
  (if (and checkp (zerop x))
     most-positive-single-float
     (/ 1 x)))
```

Note the use of *optional arguments*. Here, `checkp` does not have to be provided in the calling expression, and if not then it defaults to `nil`.

"Conditional branching" used in imperative languages is replaced in LISP by conditional evaluation. The `if`-expression is evaluated just like any other, but the returned value depends on whether the value of the first argument to `if` is `nil` or not. (Truth values in LISP are `nil` for false, anything else counts as true. `t` is used as a readable default symbol for true. Both `t` and `nil` evaluate to themselves, like numbers.)

Complex test expressions can be formed using the functions `and or not`:

- `and` returns a non-null value if all its arguments are non-null.
- `or` returns a non-null value if any of its arguments are non-null.
- `not` returns a non-null value if its argument is `nil`.

When you want to return one of several different values depending on several different conditions, use `cond` (see example below).

Sometimes, the same expression will be used several times in the same function definition. To simplify the code, and save time, one should define a temporary variable to stand for the value of the expression:

```
(defun age-group (person)
  (let ((n (age person)))
    (cond ((< n 2) 'baby)
          ((< n 18) 'child)
          ((< n 120) 'adult)
          (t 'dead))))
```

For temporary variables that are defined at the beginning of the function, as above, one can also use `&aux` variables in the parameter list:

```
(defun age-group (person &aux (n (age person)))
  (cond ((< n 2) 'baby)
        ((< n 18) 'child)
        ((< n 120) 'adult)
        (t 'dead))))
```

Recursion

The simplest way to get repetitive execution in LISP is to use *recursion*, wherein one uses the function being defined in the definition of the function itself. The key to thinking clearly about this is the *recursion relation* that holds for the problem at hand.

For example:

- *The length of a list is one more than the length of its cdr*
- *The number of atoms in a tree is the sum of the numbers in the left and right-hand sides*
- *The number of digits in an integer is one more than the number of digits in the integer part of one-tenth of the integer.*

The other thing to take care of is the cases where the recursion relation is *false*. For example, an empty list doesn't have a cdr; a tree that is just an atom doesn't have left and right hand sides.

```
(defun count (x)  ;; returns number of atoms in list structure x
  (if (atom x)
     1
     (+ (count (car x)) (count (cdr x)))))
```

Although recursion is often elegant, deeply nested recursion takes a lot of space in some cases, so we also use mapping and iterative constructs.

Mapping and Iteration

In effect, mapping is a way of constructing a big operator out of a little one. The little one works on objects; the big one works on *lists* of those objects. In LISP mapping is done with `mapcar`, which is the same as Scheme's `map`. LISP it's `mapcar.`:

```
(mapcar #'1/ '(1 2 3))
(1 1/2 1/3)
```

For now, you can think of the `#'` as a special kind of quote used on functions. *This is different from Scheme, where you would simply use the function symbol.* If you write functions that take functions as arguments, you need to be careful in writing the expressions that use the function parameter (unlike SCHEME, where the thing in functional position is evaluated like anything else). For example, if we wanted to write `mapcar`:

```
(defun mapcar (f l)
  (if (null l)
      nil
    (cons (funcall f (car l)) (mapcar f (cdr l)))))
```

`funcall` takes a function and some arguments to apply the function to. `apply` takes a function and a *list* of arguments. For example, if you want to find the sum of a list of numbers, use

```
(apply #'+ '(1 2 3 4 5))
```

Sometimes, one needs to map over a list using a function that doesn't have a name. For this, and other occasions demanding dynamically-created functions, we use the special λ-expression:

```
(mapcar  #'(lambda (x) (* x x)) '(1 2 3))
(1 4 9)
```

Once again, note the use of `#'`, unlike Scheme.

`mapc` is like `mapcar`, but doesn't gather up the results of the operations into a list.

Then there are the repetitive tasks for which one just can't come up with a nice clean way to say it using recursion or mapping. For this, one uses the general `do` construct, or the specialized `dolist` and `dotimes`.

`do` is too complicated to explain and you'll only forget it anyway, so look it up in the book (this is what I do). (If you think `do` is complicated, wait till you see `loop`, which I cannot wholeheartedly recommend.)

`(dotimes (x 100) <body>)` executes the body 100 times, with the index variable `x` ranging from 0 to 99. The index variable is only bound inside the `dotimes` expression.

`(dolist (x l) <body>)` executes the body repeatedly as `x` ranges over all the elements of the list `l`.

Equality and Binding

Equality between things other than numbers (use = for that) is tricky. You are bound to run into bugs caused by using the wrong equality test at some point. There are two kinds of equality (at least):

1) Identity: two things are `eq` if they are the same thing. Atoms are always `eq` to themselves: `(eq 'x 'x)`. Similarly, the value of `x` is the same thing as the value of `x`, so `(eq x x)` is true.

2) Structural equality: two things are `equal` if they have the same 'structure'; more or less, if they look the same when printed. `(equal (list 1 2) (list 1 2))` is true; `(eq (list 1 2) (list 1 2))` is false, since each call to `list` creates a new (and therefore different) piece of list structure.

In testing membership of an object in a list, for example, it is important to distinguish the two types of equality: `(member x l)` tries to find something in `l` that is `equal` to `x`; if you wanted to find something `eq`, as for example in deciding whether or not the list is circular, then use a *keyword argument*: `(member x l :test #'eq)`. This asks if there is a member of `l` that is `eq` to `x`. If you want to find out if some member of `l` has a square root that is numerically equal to `x`, you can use a `:key`:

```
(member x l :test #'= :key #'sqrt)
```

So how do atoms get their values anyway? We've seen binding for atoms as parameters and in `let` and `lambda` expressions. Yes, there is such a thing as a global variable.

`(setq x <expr>)` sets the value of the atom `x` to the value of the expression. Thus `(setq a b)` causes `a` and `b` to have the same value; i.e., `(eq a b)` becomes true. Note that `setq` doesn't evaluate its first argument; `set` does.

More generally, you can use `setf` to cause a `place` to have a new value. The value cell of an atom is one kind of place; there are places in the cells of an array; in the fields of a defstruct (see below); in the cons cells of a list structure, and so on. More or less anything that you can `access` with a lisp expression can be changed using a call to `setf`. For example:

```
%  (setq x '(1 (2 3) 4))
(1 (2 3) 4)
%  (setf (caadr x) 'ding)
ding
%  x
(1 (ding 3) 4)
```

Warning: if you really want to use a global variable, then for efficiency you should declare it as such using `defvar`, instead of just doing a setq to initialize it. This will let the compiler know what kind of thing it is. Also, the standard is to use asterisks around the name like `*this*`.

Once again, let me emphasize that a symbol's *value* and *function* can be different. The following expression evaluates to 6:

```
(setf f 3)
(setf (symbol-function 'f) #'+) ;; or (defun f (a b) (+ a b))
(f f f)
```

Complex data types

Common LISP also provides a very useful mechanism, called `defstruct`, for creating your own data abstractions. (This feature doesn't exist in Scheme.)

`defstruct` defines a data type with the given field names, and *automatically* creates the associated access and constructor functions. Thus to define a data type for storing information about prisoners, we could use

```
(defstruct prisoner
              name
              number
              crime
              term
              entry-date)
```

which will define the constructor function `make-prisoner`. We can use this to associate a prisoner data structure with a particular atom:

```
(defvar manson (make-prisoner :name "charles manson" :crime 'mayhem :term 'life))
```

Note that in the call to `make-prisoner`, the fields have colons.

The access functions `prisoner-name`, `prisoner-number` etc. will be created when the defstruct is executed. The value of a data field is changed using `setf`, e.g.,

```
(setf (prisoner-number manson) 4424275684)
```

Arrays are sometimes useful for multidimensional data tables. For example, you make a 3x4 array by calling `(make-array '(3 4))`. There are various optional keyword arguments for initialization. The access function is `aref`. Arrays are 0-indexed. Thus to set the first element of a two-dimensional array `*table*`:

```
(setf (aref *table* 0 0) 4)
```

Using Common Lisp

`gcl` or `lisp` invokes Gnu Common Lisp. I recommend you use split-screen emacs with one half running `lisp` in a shell. Type `C-c l` or `M-x run-lisp` in Emacs.