Notes on Lisp for CSC173
George Ferguson
Fall 2018 (last revised Fall 2022)

Lisp, from “List Processing,” was invented by John McCarthy in 1958 and first published in 1960 (McCarthy, 1960), making it one of the first programming languages ever. This section provides a very brief introduction to Lisp. I **strongly** urge you to read at least the start of Paul Graham’s “The Roots of Lisp” (see References) which lays it out very well for more modern readers. FYI: Paul Graham is the founder of the famous tech startup incubator Y Combinator, which is named after a functional programming concept. How cool is that?

According to Steele and Sussman (1978), Lisp was based on recursion equations ala Kleene (recusive function theory) more than on Church’s lambda calculus. But Lisp took the `LAMBDA` notation for a way of describing procedures using S-expressions without giving them names via `DEFINE`. If that makes no sense to you at all: read on!

Common Lisp is the ANSI-standard version of Lisp. The definitive reference for Common Lisp is the HyperSpec, which is based on Guy Steele’s books *Common Lisp: The Language*. For an overview of Lisp similar to this document, see Lisp: Syntax and semantics on Wikipedia.

See the section “Using Lisp” at the end of this document for instructions on obtaining and using a Lisp implementation. You will want that as you go through the examples in this document.

Lisp Expressions

In Lisp, expressions are defined *recursively*:

- An *atom* (a sequence of characters other than parentheses and whitespace) is an expression; and
- A *list* of zero or more expressions separated by whitespace and enclosed in parentheses is an expression.

For example (from Paul Graham):

```
foo
()
(foo)
(foo bar)
(a b (c) d)
```

We've seen recursive descriptions of expressions before, right? And BTW: atoms include both *symbols* ("identifiers"), numbers (sequences of digits, *etc.*), and various symbolic atoms like "+" or "<=".

Lisp Functions

List lists can represent the application of a function to its arguments. For example, `(eq x y)` represents the application of the `eq` function (which tests the most basic form of equality) to arguments `x` and `y`. We say that the first element of the list is the *operator* or *functor* and the other elements (possibly none) are the *arguments*.

There are many builtin functions in Lisp. Paul Graham's article describes the "primitive" building block functions of Lisp in detail: `quote`, `atom`, `eq`, `cons`, `car`, `cdr`, and `cond`. You should probably read that now if you haven't already. It then describes `eval`, which is the basis of the amazing function `apply`.

Defining Functions

Function definitions are expressed as list expressions using the `lambda` operator:

`(lambda (`$p_1 \ldots p_n$`)` e`)`

where the atoms p_i are the *parameters* of the function and e is an expression that constiututes the body (or definition) of the function. For example:

```
(lambda (x y) (+ x y))
```

is a function with two parameters, `x` and `y`, whose value is the value of the expression `(+ x y)`, which we will explain shortly, but you can probably guess what it does if I tell you that "+" is the name of a Lisp built-in function.

Naming Functions

It is convenient for several reasons to be able to name functions, like that built-in function "+" and like how you name the functions and methods in your programs in other programming languages. This also allows the definition of recursive functions. There are several ways to do this in Lisp. I will describe only one:

`(defun` f `(`$p_1 \ldots p_n$`)` e`)`

An expression starting with `defun` associates the symbol f with the `lambda` expression with parameters p_i and body e, where f may be used in e.

For example, here are definitions of functions named `plus1` and `factorial`:

```
(defun plus1 (x) (+ 1 x))
(defun factorial (n)
    (* n (factorial (- n 1))))  ; Note: not quite correct...
```

BTW: Comments in Lisp start with semi-colons, as in this example.

Applying (Calling) Functions

An expression whose first element is a lambda expression or a symbol whose value is a `lambda` expression is called a *function application* or *function call*. It represents the application of the function to specific arguments for its parameters.

If function `f` has been defined as follows, for some body expression e:

`(defun f (`$p_1 \ldots p_n$`)` e`)`

Then

`(f` $a_1 \ldots a_n$`)`

represents the application of function `f` to the given arguments a_i.

First each argument expression a_i is evaluated in order. Then e is evaluated with the values of each of the parameters p_i set equal to the value of the corresponding a_i. This is just like how arguments are passed to functions via parameters in other programming languages.

For example, if we have the previous definition of the function named `plus1`:

```
(defun plus1 (x) (+ 1 x))
```

Then `(plus1 0)` returns 1, and `(plus1 99)` returns 100. Try it.

Function applications can be nested, just like in other programming languages:

```
(plus1 (plus1 0))
```

The outermost call to the `plus1` function takes one argument. The argument expression is evaluated first, resulting in the inner call to the `plus1` function being evaluated. Its argument is 0, so that call returns 1. That value is used as the argument to the outer call to the `plus1` function, which then returns 2. So the value of the entire expression is 2.

Functions can also be invoked using the Common Lisp builtins `apply` or `funcall` to invoke a function on some arguments:

```
(apply '+ (list 2 3)) => 5
(funcall '+ 2 3) => 5
(apply 'plus1 (list 99)) => 100
(funcall 'plus1 99) => 100
```

Both of these builtins take a lambda expression or a symbol associated with a lambda expression as their first argument. `apply` takes a **list** of arguments for the function as its second argument. `funcall` takes as many additional arguments as needed for the function, **not in a list**.

Lisp Datatypes

We've already seen that Lisp expressions are made out of *atoms* and *lists*.

And we've seen two types of atoms: *symbols*, such as `x` or `plus1`, and *numbers*, such as `0` and `1`. Symbols are names that can have values associated with them, for example by `defun` or when the parameters of a `lambda` expression are "bound" to the argument values during function application. The value of a numeric atom is the corresponding number and cannot be changed.

We've seen that when Lisp reads an expression that starts with an open parenthesis, it creates a list containing the sub-expressions until the matching parenthesis. In fact, what it is doing is calling the built-in function `list`, which returns a new list containing the values given as arguments (as many as there are, meaning that `list` is a *variadic* function—like a Java method that uses a "`...`" parameter).

For example:

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

returns the list `(1 2 3)`, which is a list containing three numeric values, the numbers 1, 2, and 3. Easy. Try it.

But what if we just write that list directly?

```
(1 2 3)
```

Remember that a Lisp expression represents a function application. So Lisp will try to apply the function that is the value of the first sub-expression to the values of the remaining sub-epxresssions. But the value of the atom `1` is the just the number 1, which is not a function, so Lisp will signal an error. Try it.

Quoting

To write a literal list like that, we have to *quote* it:

```
(quote (1 2 3))
```

The built-in function `quote` tells Lisp: don't evaluate this expression, just read it in and return its value. Since the expression is a list, this will return a list (the list containing the the numbers 1, 2, and 3, like before).

Why is this called `quote`? Well suppose that you are writing a paper in English on the subject of the crazy things that computer people say, and you want to point out the English word “crash.” See: I just quoted the word so that you would know that I’m talking about the word itself, not using the word in a sentence. Without the quotes, it would be “point out the English word crash,” and readers would be left wondering what an “English word crash” was.

Since `quote` is so widely used in Lisp, there is a shorthand for it:

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

A single quote symbol before an expression means: don’t evaluate the following expression, just use it as a value.

If the quoted expression is a symbol, then the value of the quoted expression is the symbol itself, rather than its value. For example, try:

```
'foo
```

The value of this expression (or `(quote foo)`) is the symbol `foo` itself.

This business of quoting is one of the trickiest things to get used to in Lisp. If you want Lisp to do its thing and evaluate an expression, then no quote. But if you want it to use the expression (atom or list) as a literal value, then you quote it, with two exceptions:

- Numbers do not need to be quoted. Their values are fixed.
- And also, in some contexts Lisp does not evaluate expressions, so sort of like auto-quoting. An example is `defun`: the list of arguments following the function name is not evaluated but rather used as a list of symbols, and similarly for the body of the function (which is evaluated when the function is called). This is typically true for so-called *macros* and *special forms* (such as `quote`), but the details are beyond the scope of this introduction.

Other Datatypes

Originally, Lisp had only atoms and lists. Everything else could be built out of these, at least in principle.

More modern Lisp implementations have lots of other datatypes, including:

- Arrays (you should know the difference between lists and arrays already).
- Characters and strings (types `character` and `string`). Strings in Lisp are vectors (one-dimensional arrays) of characters.
- Rational and complex numbers (types `ratio` and `complex`, respectively)
- Structured representations: `defstruct` (like `structs` in C but with many more features)
- First-class objects defined using `defclass` and with very powerful features including generic functions and methods (different from Java's) and a true meta-object protocol.

The definitive reference for all of these is the HyperSpec, which is the official specification of the ANSI Common Lisp standard.

That's All Folks

That's as far as I'm going to go. You have enough to get you started even if you've never programmed in Lisp before (and most people haven't). The next section gives some tips for installing and running Lisp.

Using Lisp

There are several free implementations of Lisp (and some commercial options). I recommend that you try `abcl`, Armed Bear Common Lisp, from abcl.org. Because `abcl` runs on the Java Virtual Machine (JVM), it runs on Linux, Mac, and Windows just like Java. You will need to run it from a command shell (a.k.a. terminal). If you aren't comfortable with your computer's command line yet, now's a good time to learn.

Download one of the "binary" packages from abcl.org. Unpack the archive to get a folder with `abcl.jar` among other files.

Next open a terminal window (Linux: `xterm` or equivalent; Mac: `Terminal`; Windows `CMD.COM` or `PowerShell` or other).

Change your working directory to the folder that you just unpacked. For example, if you downloaded the archive to your "Downloads" folder and then unpacked it to create the sub-folder "abcl-bin-1.9.2", you would type the following, followed by Return:

```
cd Downloads/abcl-bin-1.9.2
```

You can list the contents of the current directory (folder) with the `ls` command (`dir` in `CMD.COM`). If you need a tutorial on using the command shell, there are plenty on the web.

All you need to do to run `abcl` is to tell Java to run the `jar` (Java Archive) that you downloaded:

```
java -jar abcl.jar
```

You should see some startup messages, and finally a prompt from the main read-eval-print loop (REPL). Here's what I see:

```
Armed Bear Common Lisp 1.9.0
Java 18.0.2.1 Homebrew
OpenJDK 64-Bit Server VM
Low-level initialization completed in 0.098 seconds.
Startup completed in 0.457 seconds.
Type ":help" for a list of available commands.
CL-USER(1):
```

Try entering just `1` followed by Return. You should see the value 1, which is the value of the expression you entered.

Try evaluating `(+ 1 2)`. (Don't type the period!) You should see the result of applying the function + (plus) to arguments 1 and 2, yielding result 3.

Try evaluating `"hello world"` (that is, a string enclosed in double-quotes). It should be echoed back to you, since that's the value of a string expression.

Try evaluating `(defun plus1 (x) (+ x 1))`. You should see the name `PLUS1` since you just defined a function with that name (Lisp is case-insensitive, that is, not case-sensitive, by default).

Then try evaluating `(plus1 8)`. You should see the value 9, which is the result of evaluating your function `PLUS1` with argument 8.

Type `:exit` to exit `abcl`. Type `:help` for more REPL commands.

You are ready to write some Lisp code! You can type at the REPL, which is good for testing and debugging, but usually you type your code into a file and tell Lisp to load it:

```
    (load "project3")
```

This will load and run the code in the file named `project3.lisp` and return you to the REPL if there were no errors.

References

Graham, P. (2001). "The Roots of Lisp". http://paulgraham.com/rootsoflisp.html. I have uploaded a PDF of the `jmc.ps` document to BlackBoard for CSC173.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 3(4), pp. 184–195.

Steele, G. L., Jr., and G. J. Sussman. (1978) *The Art of the Interpreter or, The Modularity Complex (Parts Zero, One, and Two).* Massachusetts Institue of Technology AI Lab Memo No. 453.