# Comments on LISP

The “artificial” in Artificial Intelligence means that the “intelligence” is implemented on a computer. Almost all AI implementations are in LISP; that is, LISP and AI are synonymous. The content of AI is defined by what you can write in the LISP language.

In this class, you will learn how to think in LISP, not how to be a LISP programmer. (This has made choosing a LISP primer difficult.) We will use a small subset of LISP called Pure Lisp.

PURE LISP = ideal programming style

Pure LISP excludes all of the programming ideas which lead to bad code. Pure LISP does not allow:

destructive data operations
go tos
explicit pointers and dereferencing
side effects (only the direct results of the function being processed)
unbound and global variables
do loops (use recursion instead, this rule is not firm)
block structure (functions provide grouping)

LISP is a unique language in the following ways:

- symbolic rather than numeric computation. This permits an algebraic approach.

- functional/applicative style rather than procedural style. Program structure is defined by function composition, and by the formal system of lambda calculus. The normal evaluation regime is successive function evaluation, but this can be freely modified

- indefinitely extensible. Most programming language innovations (such as garbage collection, streams, closures and continuations, symbol packages, first-class errors, object orientation, provability) were pioneered in LISP.

- interpreted/interactive rather than compiled. LISP can be compiled when debugging is completed. When you enter an expression into the LISP environment, the evaluation is immediate. LISP programming is a dialog.

- uniform data representation. Programs are data, which means LISP programs can modify themselves at run-time.

- LISP is written in LISP. The process of LISP is written in the same language as the rest of LISP. Bootstrapping means that the LISP evaluation mechanism and compiler are easily available to the programmer for modification and customization. In the days of LISP machines (1980s), the entire operating system was accessible. Emacs is a text editor written in LISP, the entire editing environment is accessible.

- very small kernel. The LISP primitives are:

| | |
|---|---|
| nil | empty list, false, nothing to return |
| atom | predicate to determine elementary data |

| | |
|---|---|
| eq | predicate to test equality of atoms |
| car, cdr | selectors/accessors of data structure |
| cons | constructor for data structures |
| cond | basic logic function |
| eval, quote | controling the difference between program and data |

• important functions

| | |
|---|---|
| setq | setting or assigning names to the results of functions |
| list | constructing a list |
| defun | defining named functions |
| lambda | constructing an unnamed function |
| let | defining the scope of variables |

• debugging tools include

| | |
|---|---|
| whitespace | ignored by the evaluator |
| trace | follow the evaluation sequence |
| pprint | print data in pretty form |
| read-eval-print | the basic evaluation process |

• disadvantages of LISP and their solutions

*hard to read syntax with lots of parentheses*
redefine the syntax to look the way you want it to

*one data type*
build the data types you want and wrap them in an abstraction barrier

*inefficiency*
no longer true, LISP runs at 95% the speed of C. It is possible to write inefficient LISP programs, but the rules to avoid this are straight forward and can be learned with practice. It is easier to write inefficient programs in other languages.

*many dialects*
the community has standardized on Common LISP. Dialects built from the same foundation are a good idea.

*no first class functions*
dialects for higher order programming are available (ie Scheme)

## How To Think in LISP

What follows is a list of principles and ideas, not a formula for thinking.

• All valid expressions are valid programs. This provides arbitrary granularity. Programming consists of building up hierarchical languages built on a solid foundation.

• You are always in control of what is data and what is process. Programming is building data, then testing processes on it, then making those processes into data, and so on.

• Computation consists of applying functions to data. Function composition lets you apply functions to the results of other functions, which themselves turn data into different data.

• All defined functions are provable, that is they are data structures you can talk about, and the way to talk about them is to assert their correctness.

• The programmer is always part of the computation. The read-eval-print loop can be seen as an interactive dialog. Read means listen to what the person says. Eval means do what the person asks you. Print means tell the person the results of the request.

• All objects are the same. There are base objects (atomic data) and compound objects built from atomic objects. Atomic objects (atoms) are the pieces of a program, the bricks. Function composition is the cement holding the atoms together. Nothing else is happening. Atoms define your conceptualization, the pieces of the world. Functions just define bigger pieces. Object-orientation is function composition turned inside out.

• Variables are just convenient and arbitrary names for compound objects. So a variable is meaningful only when it is in the same context as the object it names. This is called scoping.

• Function names are also variables. You can rename functions at any time, and you should always use names that are meaningful to you. Write languages not programs. Think like a human, not like a computer and write code that matches human thought.

• Recursion means stating a base case (what is true in the simplest circumstance) and a general case (what holds invariant across all objects being manipulated). The latter is not easy. Recursion makes control structure and data structure look the same, thus all cases of a particular data domain do not need to be examined exhaustively.

• There are always two levels when programming, the syntactic and the semantic: what you see and what you mean. Representation and value. Try to align the two by defining the look of a program to remind you of its meaning.

• Control is just telling the function how to look for what you what. The state space model is the space being searched. You can look in three different ways:

* depth first (follow the chosen path to its end, then go back to the beginning to try other paths
* breadth first (look one step down each path choice)
* dependent (choose a path that meets your current objectives, then go back to the next best path to meet those objectives)

• Formulate knowledge in terms of patterns, and look for those patterns. Patterns can be abstract, with many things of the same class fitting a particular spot.

• Formulate operations as functions. Operations can be abstract, with many functions fitting the same operation. Use operations that address all objects at the same time. For example, rather than explicitly checking each object for a property (by writing a DO loop), just ask if the property is true for everything (using the function EVERY).