Pure LISP

LISP is a unique language in the following ways:

- symbolic rather than numeric computation.
- functional/applicative style.
- indefinitely extensible.
- interpreted/interactive rather than compiled. LISP can be compiled after debugging.
- uniform data representation. Programs are data, which means LISP programs can modify themselves at run-time.
- LISP is written in LISP. This *bootstrapping* means that the LISP evaluation mechanism and compiler are easily available to the programmer for modification and customization.

Pure LISP excludes most of the programming ideas which lead to poor code. Most programming language innovations (such as garbage collection, streams, closures and continuations, symbol packages, first-class errors, object orientation, provability) were pioneered in LISP. Pure LISP does not allow:

- destructive data operations
- gotos
- 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)

Primitives

LISP has a very small kernel of primitive functions. These are:

nil	empty list, false, nothing to return
atom	predicate to determine valid labels
eq	predicate to test equality of atoms
car, cdr	selectors/accessors of list data structure
cons	constructor for list data structures
cond	basic logic function
eval, quote	controlling the difference between program and data

Special functions have a non-standard format. Some important special functions include:

setq	setting or assigning labels to the results of functions
list	constructing a list

`defun`	defining named functions
`lambda`	constructing an unnamed function
`let`	defining the scope of variables

LISP 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 (i.e. Scheme)

Storage Reclamation

Most programming languages use *explicit erasure* to reclaim storage cells. This is a bad idea since it makes a low-level maintenance chore the responsibility of the programmer. As well, it violates security. Suppose the value in a cell is erased, but the cell is still referenced by some data structures. These *dangling pointers* are unprotected and undocumented, and the source of difficult to trace errors.

Once automated way to keep track of memory usage is *reference counting*. Whenever a cell is used, or referred to, by part of a program, the reference count of that cell is increased by one. When a cell has no existing references, that cell is not accessible to the current program, and is thus on the list of free cells.

Another approach is *garbage collection*. Here inaccessible cells are simply abandoned. When the list of free cells is exhausted, the processor interrupts normal computation and enters a garbage collection phase. A mark-and-sweep garbage collector passes twice over all memory cells. On the first pass, inaccessible cells are marked as such. On the second pass, the marked cells are returned to free storage. A serious problem for garbage collection is *nonuniform response time*, in that processing halts while garbage collection is occurring. If there are many cells to be reclaimed, this interrupt may be several seconds.

Some Observations about the LISP Language

• 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.

• 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.

• 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. In general, programs that look good are good.

• 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).