Lesson #4: Lisp

What's It All About?

A story is told, in very broad strokes, of the origin, development, and significance of Lisp. The main features of Lisp are presented. S-Expressions are introduced. The defining functions of historical Lisp are presented, and it is observed that historical Lisp is a "pure Lisp".

About Lisp

These notes are adapted from "Concepts in Programming Languages", pages 18-21, by John Mitchell. Mostly, this is all common knowledge, but this particular presentation is certainly informed by a reading of these pages.

1. The Lisp programming language was developed at MIT by John McCarthy in the late 1950s for research on AI and symbolic computation.
2. The strength of Lisp is its simplicity and flexibility. It is widely used for "exploratory programming", a style of software development in which systems are built incrementally, with potentially dramatic changes being made as a result of experimental evaluation.
3. Lisp stands for "List Processor", an apt name considering the fact that its fundamental data type is the list.
4. There have been many implementations of Lisp over the years which have contributed to the development of the language and to the field of AI. That said, Scheme and Common Lisp are particularly worthy of mention. Scheme, developed by Guy Steele and Gerald Sussman at MIT in the 1970s, is an elegant blend of Lisp and Algol. It is commonly used in educational circles, perhaps more than ever in its Racket cloak. Common Lisp was developed by committee in the 1980s, and consists of an eclectic mix of powerful computational mechanisms. It is the language of choice for many AI researchers today.
5. McCarthy presented the ideas on which Lisp is based in a 1960 CACM paper called "Recursive functions of symbolic expressions and their computation by machine". If you would like to take a look, you might go here: `http://jmc.stanford.edu/articles/recursive/recursive.pdf`
6. The Lisp language itself was first presented in the "LISP 1.5 Programmers Manual", written by McCarthy and others, and published by MIT Press.
7. The motivating applications for the development of the language were the Advice Taker, a common sense reasoning system with a logical foundation, and symbolic calculation with respect to functions (e.g., integration and differentiation).
8. Lisp basically runs on an abstract Machine called `eval`, which Steven Russell made real within a matter of days by implementing McCarthy's program, which was originally meant just to be read, on an IBM 709 computer.
9. The theoretical foundation of Lisp is the lambda calculus, which can be construed as meaning that Lisp has a particularly strong pedigree.
10. Successful language design efforts generally share three important characteristics: (1) a motivating application, (2) an abstract machine, and (3) theoretical foundations. These three characteristics are abundantly clear with respect to the development of Lisp!
11. **According to Alan Kay: "Most people who graduate with CS degrees don't understand the significance of Lisp. Lisp is the most important idea in computer science."**

Features of Lisp

These things tend to come to mind when a Lisper thinks about Lisp. Thanks to Paul Graham, in “Revenge of the Nerds”, for many of the articulations.

1. **Recursive Functions.** Fortran, the only other major language at the time that Lisp was proposed, did not have recursive functions. These were an essential element of Lisp.
2. **Conditional Expressions.** McCarthy introduced conditional expressions in Lisp, which were not fully implemented in Fortran. Soon after, he lobbied successfully for their inclusion in ALGOL.
3. **Function Type.** In Lisp, functions are a data type, just like integers or strings. They have a literal representation, can be stored in variables, can be passed as arguments, and can be returned from functions as values. This innovation afforded programmers the use of higher order functions and all of the powerful programming patterns that are now associated with them.
4. **Symbol Type.** Symbols are effectively pointers to strings stored in a hash table. So you can test equality by comparing a pointer, instead of comparing each character.
5. **Homoiconicity (Same Representation).** A programming language is homoiconic if its data and its code have the same representation. In Lisp, that representation is the S-Expression. A language which is homoiconic facilitates “reflection”, which means that programs written in the language can examine, introspect, and modify the structure and behavior of programs written in the language at runtime. Homoiconic languages facilitate certain kinds of “metaprogramming”, and afford powerful “macro” writing mechanisms for achieving syntactic transformations. Machine language is also homoiconic, in that bits represent both instruction codes and data. But Lisp was the first homoiconic higher level language.
6. **Dynamic Typing** In Lisp, all variables are effectively pointers. Values are what have types, not variables, and assigning or binding variables means copying pointers, not what they point to.
7. **Garbage Collection.** According to Wikipedia: In computer science, garbage collection (GC) is a form of automatic memory management. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced - also called garbage. Garbage collection was invented by American computer scientist John McCarthy around 1959 to simplify manual memory management in Lisp. Garbage collection relieves the programmer from performing manual memory management where the programmer specifies what objects to deallocate and return to the memory system and when to do so.
8. **REPL, which stands for Read-Eval-Print-Loop** In REPL, the tokens “read”, “eval”, and “print” are sequenced within a “loop” construct. These days, many languages provide Lisp’s REPL. By way of emphasizing the nature of the construct, here is a Common Lisp implementation, along with a demo:

```
[1]> 9
9
[2]> "cat"
"cat"
[3]> ( car '( blue yellow red ) )
BLUE
[4]> ( cdr '( blue yellow red ) )
(YELLOW RED)
[5]> ( defun repl ()
       ( loop ( print-x ( eval ( read-x ) ) ) )
     )
REPL
[6]> ( defun print-x (s) ( prin1 s ) ( terpri ) )
```

```
PRINT-X
[7]> ( defun read-x () ( princ "> " ) ( read ) )
READ-X
[8]> ( repl )
> 9
9
> "cat"
"cat"
> ( car '( blue yellow red ) )
BLUE
> ( cdr '( blue yellow red ) )
(YELLOW RED)
>
```

S-expressions

The basic conceptual data type in Lisp is the “S-expression”, or “symbolic-expression”.

BNF Description of S-expressions

An S-expression can be described by the following BNF rules:

```
<sexp> ::= <atom> | <dotted-pair> | <list>
<dotted-pair> ::= ( <sexp> . <sexp> )
<list> ::= ( <dbd-sexp-list> )
<dbd-sexp-list> ::= <empty> | <sexp> <dbd-sexp-list>
```

What is an atom? We generally think of an atom as a data item that we don’t plan to decompose into parts. In traditional Lisp, symbols and numbers are atoms. In certain circumstances we might think of strings as atoms. The Racketeers likes to include images to be among the atoms. The meaning of the term will vary with the Lisp that you are thinking about, and maybe even on your intentions with respect to the data that you are processing.

Example S-expressions

- The following S-expressions are atoms:
  1. `coffee`
  2. `496`
- The following S-expressions are dotted-pairs:
  1. (one . 1)
  2. ((red . blue) . purple)
  3. (poema . ((canaro . 1935) . tango))
- The following S-expressions are lists:

    1. (a b c)
    2. (1 2 4 8 16)
    3. (1 red 2 blue 3 yellow)
    4. ((one un) (two deux) (three trois) (four quatre))

- The following are also S-expressions, the first a list of dotted pairs, and the second a dotted pair of lists:
    1. ((red . r) (blue . b) (yellow . y))
    2. ((a b c) . (1 2 3))

Relationship between Lists and Dotted Pairs

Two facts about List notation:

1. Any S-expression represented as a list can be represented as a dotted pair. The inverse is not true. For example:
    - `(a b c)` can be written as `( a . ( b . ( c . () ) ) )`
    - `( a . b )` cannot be written as a list.
2. When Lisp prints an S-expression, it will always print it in the most list like fashion possible.

```
; language = Common Lisp (clisp)

[]> '( a . ( b . ( c . () ) ) )
(A B C)
[]> '( ( red . blue ) . purple )
((RED . BLUE) . PURPLE)
[]>

; language = Racket

> '( a . ( b . ( c . () ) ) )
'(a b c)
> '( ( red . blue ) . purple )
'((red . blue) . purple)
>
```

Conversion between Lists and Dotted Pairs

How do you convert lists to dotted pairs, and dotted pairs, as far as possible, to lists? The key to doing this lies in the definitions of the classic Lisp functions functions `car` and `cdr` for dotted pairs and for lists.

- The **car** of a dotted pair is the first component of the pair. The **cdr** of a dotted pair is the second component of the pair. Thus, the `car` of `( red . blue )` is `red`, and the `cdr` of `( red . blue )` is `blue`.
- The **car** of a nonempty list is the first element of the list. The **cdr** of a nonempty list is the list with the `car` removed. Thus, the `car` of `( red blue )` is `red`, and the `cdr` of `( red blue )` is `( blue )`.

A dotted pair and a list represent the same S-expression if (1) they have the same `car` values, and (2) they have the same `cdr` values.

Exercise: Drawing Parse Trees for S-expressions

1. Draw a parse tree for `( a . b )`
2. Draw a parse tree for `( a b )`
3. Draw a parse tree for `( ( a . b ) c () )`

Exercise: Checking for equality between Lists and Dotted Pairs

1. Is the dotted pair `( a . () )` equal to the list `( a )`?
2. Is the dotted pair `( a . b )` equal to the list `( a b )`?
3. Is the dotted pair `( () . a )` equal to the list `( a )`?
4. Is the dotted pair `( a . ( b . ( c . () ) ) )` equal to the list `( a b c )`?
5. Is the dotted pair `( ( a . b ) . c )` equal to the list `( ( a . b ) c )`?
6. Is the dotted pair `( a . ( ( b . c ) ) )` equal to the list `( a ( b . c ) )`?

(The answer to the prime numbered questions is **no**. The answer to the other questions is **yes**.)

The Defining Functions of Historical Lisp

The defining functions of historical Lisp are:

- the operations `car`, `cdr`, `cons`, `eq`, and `atom`, which operate on S-expressions, together with
- the general programming functions `cond`, `lambda`, `define`, `quote`, and `eval`.

Arithmetic operators and relational operators, among others, fill in the gaps in essential infrastructural ways, but they are not the salient, defining functions of historical Lisp.

The functions `car` and `cdr` have already been introduced. The remainder are best discussed, I think, in the context of some "pure Lisp" programming, which will be explored quite soon.

Historical Lisp is a Pure Lisp

Historical Lisp is a **pure Lisp**, pure in the sense that expressions in this Lisp are "well-defined", and do not have "side effects". In other words: (1) an expression will always return the same value for given inputs, and (2) evaluating an expression does not change the state of the machine beyond the scope of the expression.

Most Lisp dialects, for example Scheme and Common Lisp, afford opportunities to write impure code. Some measure of impurity is generally needed to do real work (barring very imaginative, very broad conceptions of function application), but it is good to be able to identify a pure language when you see it.