LISP

Harish Karnick

Harish Karnick teaches in the Department of Computer Science and Engineering at IIT Kanpur. His interests are machine learning, cognition and more generally artificial intelligence. Programming languages, their history and evolution is a hobby and LISP is a particular favourite.

This article discusses the programming language LISP. The language was created by John McCarthy for an artificial intelligence application. LISP was the first functional language based on a mathematical theory of computation. It has been standardized and is still used fairly widely. The language also had a huge influence on later languages. Its implementation introduced interpreted languages, source-level debugging and garbage collection. These facilities are a common denominator for many languages today. We discuss the mathematical foundations of LISP the interpreter and briefly touch on its influence on other languages and its status today.

1. Introduction

Of languages that are still alive LISP is the second oldest. It was created by John McCarthy in 1958 [1], one year after John Backus created FORTRAN. The two languages are very different in conception, design, application domain and finally in how they have evolved over time. McCarthy invented the language as a tool for the Advice Taker program which required a way to express and manipulate symbolic expressions [1].

LISP is fundamentally a functional language inspired by the lambda calculus[1] (or λ-calculus) [2,3] which is a computational formalism based on functions and function application. This is an alternative to the more widely known computational model, the Turing machine [4,5], which is much closer to a computer, the device that sits on our desks.

[1] See *Resonance,* Vol.19, No.4, pp.345–367, 2013.

Keywords
LISP, EVAL, λ-calculus, functional language, McCarthy, garbage collection, interpreter.

In this article we look at the basic computational model at the heart of LISP; the language itself; and finally how LISP has evolved and influenced other languages.

2. λ-Calculus: The Foundation for LISP

Consider simple addition where we are adding x and y, which we often write as the expression $x + y$ where x, y are variables that are substituted with the values that we want to add. Another way to look at this is to think of $+$ as a function of two arguments $+(x,\ y)$; so when we want to add, say 3 and 4 we can write $+(3,\ 4)$ and when we evaluate the function it returns the value 7, the sum of x and y. We can easily generalize this idea and define any function f with a known number of arguments, say n, $f(x_1, x_2, \ldots, x_n)$. Here, $x_1, \ldots, x_n$ are called the formal arguments and when we actually wish to evaluate the function f we provide the 'actual arguments' (like 3 and 4 for $+$ above) and evaluate the function to get the resulting answer. These kind of functions are part of most programming languages in one form or another.

A more interesting observation is to view a function of two arguments as one where if we provide only one actual argument then the function returns as an answer, a function of a single argument with the actual argument frozen to the value provided. For example, let us write the $+$ function as: $+(x)(y)$ and let us say we evaluate $+(3)$; the resulting answer is a function of one argument that adds 3 to its actual second argument. So, if we allow functions themselves to be treated as values that can be returned as answers then we are able to build a computational model with functions having just one argument.

Alonzo Church formalized function construction and function application in a systematic way through the λ-calculus.

Alonzo Church formalized function construction and function application in a systematic way through the λ-calculus. For example, assuming that $+$ is a primitive function we can write the above as: $\lambda x.\lambda y.\ +\ x\ y$ where the λ marks out x and y as formal arguments.

Let us take a brief, but decidedly incomplete, look at the λ-calculus (for a systematic introduction see [6]). Let $\mathcal{V}$ be a set of variables $\{x, y, z, w, \dots\}$ and $\mathcal{C}$ a set of constants $\{a, b, c, \dots\}$. We assume we have as many variables (or constants) as we want; we can symbolize them by using subscripts when we need new ones. Then the following four rules define a λ-expression (we symbolize a λ-expression by capital letters and optionally subscripts, e.g., E_1).

1. Any variable from $\mathcal{V}$ or a constant from $\mathcal{C}$ is a λ-expression.

2. If E_1, E_2 are λ-expressions then E_1E_2 is a λ-expression called an application. This models function application. Think of E_1 as a function and E_2 as an argument.

3. If E_1 is a λ-expression and x is a variable then $\lambda x.E_1$ is a λ-expression called an abstraction. This models function creation with x as a formal argument.

4. If E_1 is a λ-expression then (E_1) is a λ-expression. This is for notational convenience much as we use brackets in algebraic expressions.

Note how the set of all λ-expressions is defined by just four recursive rules.

Calculation in the λ-calculus is done by doing reductions, more precisely beta reduction (or β-reduction). A λ-expression of the form $(\lambda x.E_1)E_2$ is called a β-redex and a β-reduction substitutes E_2 for all 'free' occurrences of variable x in E_1 by E_2. If we assume that variable names never clash (remember we have as many variables as we want and we can rename variables apart) then we can ignore the word 'free'. Let us look at some

Calculation in the λ-calculus is done by doing reductions, more precisely beta reduction (or β-reduction).

examples of reductions ($\underset{\beta}{\rightarrow}$ means that the left-hand side is β-reduced to the right-hand side):

***Example* 1.**

$$(\lambda x.(\lambda y.x\ y))E_1 \underset{\beta}{\rightarrow} \lambda y.E_1 y$$

***Example* 2.**

$$(\lambda x.y)E_1 \underset{\beta}{\rightarrow} y$$

Note that there is no x in the λ-expression which can be substituted by E_1; so we are just left with y.

***Example* 3.**

$$\begin{aligned}(\lambda x.((\lambda y.x\ (y\ y))E_1))E_2 & \\ & \underset{\beta}{\rightarrow}(\lambda y.E_2\ (y\ y))E_1 \\ & \underset{\beta}{\rightarrow}E_2\ (E_1\ E_1)\end{aligned}$$

A β-reduction need not terminate as in the example below.

***Example* 4.**

$$\begin{aligned}(\lambda x.x\ x)(\lambda y.y\ y) & \\ & \underset{\beta}{\rightarrow}(\lambda y.y\ y)(\lambda z.z\ z)\ \text{ renamed } y \text{ with } z \\ & \underset{\beta}{\rightarrow}(\lambda z.z\ z)(\lambda w.w\ w)\ \text{ renamed } z \text{ with } w \\ & \ldots \\ & \ldots\end{aligned}$$

A λ-expression that does not have a β-redex is said to be in *normal form*. So, β-reduction ends when a λ-expression is in normal form. If a λ-expression contains more than one β-redex then the outermost β-redex is reduced first. This is called normal order reduction.

A λ-expression that does not have a β-redex is said to be in *normal form*.

It is amazing that using the above we can calculate any computable function whatsoever. The natural numbers can be represented by λ-expressions and therefore all of arithmetic can be done.

3. LISP: The Language

LISP gives a language dimension to the λ-calculus computational model with some changes and extensions to make it practical to use.

LISP replaces λ-expressions with S-expressions (symbolic expressions). S-expressions are defined much as we did λ-expressions. In the orginal paper [1] McCarthy defines M-expressions (meta expressions) that contain S-functions that act on S-expressions. There are five primitive S-functions and two functional forms, the conditional and recursion, to compose them. In addition, through the 'label' mechanism, a functional form can be named and called recursively within its own M-expression.

Separately, he provides a way to translate M-expressions to S-expressions and defines an universal S-function *apply* that can evaluate the resulting S-expression to give the answer that we would expect from the M-expression. While this appears complicated it is really analogous to the basic von Neumann stored program idea where both program and data are stored in the same memory and a higher level interpreter, the CPU, interprets everything suitably.

We will simplify the exposition and define an interpreter `EVAL` that works on S-expressions. We start by defining an S-expression: Assume we have an infinite set of atomic symbols[2] including numbers and strings of characters. An S-expressions is defined by the following two rules:

1. Any atomic symbol is an S-expression.
2. If S_1 and S_2 are S-expressions then $(S_1.S_2)$ is an S-expression.

[2] Atomic symbols are entities that EVAL treats as atomic (even though they may be composed of parts), for example, numbers (a concatenation of digits) and names (a concatenation of letters).

An S-expression like (A.(B.(C.(D.NIL)))) is often written as a list (A, B, C, D). Here NIL is a special atomic symbol that terminates lists. We will use mostly lists instead of S-expressions since they are easier to read and write.

We start by defining five primitive functions on S-expressions. These are `car, cdr, cons, atom, eq`. The first two are used to disassemble S-expressions, the third is used to construct or assemble an S-expression and the last two are predicates. Note that these are partial functions and may give the undefined value $\perp$, if they get the wrong arguments. The primitive functions `car` and `cdr` are so named because the first interpreter for LISP implemented on IBM 704 used the address register to point to the first element of the list and the data register to point to the rest of the list. They stand for *contents of address register* and *contents of data register* respectively. Notationally, we have written the functions in a typewriter font for clarity but for all practical purposes they are S-expressions. The interpreter `EVAL` interprets them suitably. We will describe later exactly how `EVAL` works. The primitive functions have the following meaning (note that a prefix-based function notation is used, so the function is the first element of the list):

- `car` gives the head or first element of an S-expression or list.
Example: (`car` (A.B)) is A, (`car`(A,B,C)) is also A.

- `cdr` gives the second element of an S-expression or the tail of a list.
Example: (`cdr` (A.B)) is B, (`cdr`(A,B,C)) is the list (B,C).

- (`cons` A B) constructs the S-expression (A.B). Similarly, (`cons` A (B, C)) constructs the list (A, B, C) and (`cons` A NIL) gives the list (A).

The primitive functions `car` and `cdr` are so named because the first interpreter for LISP implemented on IBM 704 used the address register to point to the first element of the list and the data register to point to the rest of the list.

One way to understand cond is as a nested if-else-statement. For example: if (C1 is true) then S1 else if (C2 is true) then S2 .... else if (Cn is true) then Sn else S.

• (`atom` S_1) is true (T) if S_1 is an atomic S-expression and false (F) otherwise. For example, (`atom` A) is true but (`atom` (A.B)) or (`atom` (A,B)) is false.

• (`eq` S_1 S_2) is defined if and only if S_1 and S_2 are atomic else it is undefined. It is true if they are identical atoms otherwise it is false.
Example: (`eq` A B) is false, (`eq` A A) is true, (`eq` (A.B) A) or (`eq` (A B) B) give $\perp$.

The basic functional forms in LISP are composition, the conditional and recursion. Of these only the conditional needs a bit of explanation.

(`cond` (C1 S1)(C2 S2)...(Cn Sn)(T S))

The interpreter evaluates each conditional S-expression C1, C2, etc. in turn and for the first condition, say Ci, that is true, it evaluates the corresponding Si and returns the resulting value as the answer. If all the conditions evaluate to false then the answer is the value obtained on evaluating the S-expression S since its condition is trivially true. If (T S) is absent then the value is the atom NIL.

To make recursive functions easy to use there is a naming facility which binds a name to any arbitrary S-expression. In particular this expression can be the body of a function. So, this allows one to call a function within itself.

Let us see some examples. All LISP code is in typewriter font for clarity. This is distinct from the use of the typewriter font earlier to distinguish between S-functions and S-expressions. The naming facility uses `defun` to bind function values to names. We will freely use logical connectives like `and`, `or`, `not` to combine predicates.

***Example* 5.**

```
;;;; returns T if S1 and S2 are identical
;; S-expressions F otherwise
(defun equal (S1 S2)
  (cond
     ((atom S1)(cond ((atom S2) (eq S1 S2))
       (T NIL)))
     ((atom S2) NIL) ; S1 is not an atom so
       ;S1, S2 are not equal if S2 is an atom
     (T (and (equal (car S1)(car S2))(equal
       (cdr S1) (cdr S2))))
  )
)
```

What `defun` does is bind the S-expression that represents the body of the function definition (see below) to the name `equal` (atomic S-expression). An S-expression that represents the function is given below. Note that `Lambda` signals that the S-expression is a function definition. This is followed by the formal arguments which in turn is followed by the body of the function definition.

`Lambda` signals that the S-expression is a function definition.

```
(Lambda (S1 S2)
  (cond
     ((atom S1)(cond ((atom S2) (eq S1 S2))
       (T NIL)))
     ((atom S2) NIL) ; S1 is not an atom so
       ;S1, S2 are not equal if S2 is an atom
     (T (and (equal (car S1)(car S2))(equal
       (cdr S1) (cdr S2))))
  )
)
```

Below are a few more examples of LISP function definitions:

***Example* 6.**

```
;;;;Checks if a list is null or not
(defun null (L)
  (and (atom L) (eq L NIL))
)

;;;;Finds length of list L. Uses the null
;;function above.
(defun length (L)
  (cond
    ((null L) 0)
    (T (plus 1 (length (cdr L))));plus adds
      its two arguments
  )
)
```

***Example* 7.**

```
;;;;Appends two lists L1, L2.
(defun append (L1 L2)
  (cond
    ((null L1) L2)
    ((null L2) L1)
    (T (cons (car L1) (append (cdr L1) L2)))
  )
)
;;;;Reverses list L
(defun rev (L)
  (cond
    ((null L) L)
    (T (append (reverse (cdr L)) (cons (car L)
      NIL)))
  )
)

;;;A much better way to reverse a list
(defun revBetter (L)
```

```
  (cond
    ((null L) L)
    (T (revfn L NIL)))
)
;;;;revfn does all the work
(defun revfn (L R)
  (cond
    ((null L) R)
    (T (revfn (cdr L) (cons (car L) R)))
  )
)
```

The above examples show how one can build complex functions by composing simpler functions. A typical modern LISP system has hundreds of functions already defined as part of the LISP system.

4. EVAL: The LISP Interpreter

The LISP interpreter EVAL is a function that takes an S-expression, evaluates it and returns the result obtained. This is the equivalent of β-reduction in the λ-calculus.

Broadly, (EVAL L) works as follows: If L is an atom then it returns the value bound to that atom. In some cases the atom itself is the value, e.g., when the atom is a number or NIL (representing the empty list). If L is a list and the first element signals that L is to be treated specially, e.g., defun, then it evaluates it in a special way; otherwise it evaluates each element of the list and then applies the function associated with the first element of the list to the arguments represented by the rest of the list. The interpreter EVAL is available to the LISP programmer, usually as the function eval. So, it is possible to dynamically create a function at run time and apply it to arguments by using eval. Two special functions quote and setq are very useful in this context. When (quote S) is evaluated it returns its argument literally (as is) as the answer. The function setq allows

The interpreter EVAL is available to the LISP programmer, usually as the function eval.

one to bind an S-expression to an atom. So, `(setq A 10)` will bind atom A (think of it as a variable) to 10. Note that the first argument of `setq` is not evaluated but treated literally. For a second example consider: `(setq B (quote (car C)))` will bind the atom B to the S-expression (car C).

The following example shows what happens when `eval` is used explicitly:

```
(setq L (quote (length L1)))
;length is the  function defined in example 6
(setq L1 (quote (1,2,3,4)))
(eval L)
```

The above will evaluate to 4 which is the length of list L1. Note how we have constructed the list L at run time which will be bound to the *S*-expression (length L1) where L1 is bound to (1,2,3,4). When `eval` is called explicitly we are actually evaluating twice. The argument of `eval` will evaluate to (length L1) which on being evaluated again will give the length of list L1 which is 4.

It is this ability of LISP to construct programs on the fly by manipulating data that attracted the early AI (artificial intelligence) researchers to this language. In the roughly two and a half decades from 1960 to 1985, LISP was the language of choice when working on AI problems. Actually, multiple companies developed special-purpose hardware in the form of LISP machines which ran LISP natively. However, these companies withered when LISP implementations on standard hardware improved substantially and standard hardware rapidly gained in power.

It is this ability of LISP to construct programs on the fly by manipulating data that attracted the early AI (artificial intelligence) researchers to this language.

The interpreter `EVAL` can be easily written as a function in LISP. Actually, it will look like a `cond` with a large number of clauses that will handle each primitive

function and the special functions (like `defun, setq, quote`). The default clause in `cond` will evaluate all arguments of list L (its lone argument) and then apply the function represented by the first argument to the remaining evaluated list elements as actual arguments. Notice that this differs from β-reduction in the λ-calculus where the evaluation is normal order and the first β-redex chosen for reduction is the outermost one. However, `EVAL` evaluates the actual arguments first before applying the function. In λ-calculus this is called applicative order reduction.

McCarthy's orginal description of the interpreter is less than two pages. It is probably the shortest description of the semantics of a programming language.

5. LISP: Evolution, Influence and Prospects

In the early days of LISP it was actively developed and many dialects that did not inter-operate came into existence. The two major dialects were MACLISP and INTERLISP. Consequently, in 1984 Guy Steele with help from a large number of Lispers created the book *Common Lisp. The Language*. A second edition came out in 1990 [7]. In 1994 Common Lisp was standardized as an ANSI standard. Objects have been added to LISP through CLOS (the Common Lisp Object System) and this is also part of the ANSI standard. CLOS differs from standard Object Oriented (OO) languages[3] like Java and C++ in significant ways. It has a dynamic object system that allows class creation and change at execution time. LISP also has a powerful, dynamic debugger that makes it a much easier language to use.

LISP was also the first language to introduce a powerful macro system that encouraged many users to create new control structures and entire domain-specific languages on top of LISP (see [8]).

LISP was also the first language that had a garbage

[3] Object oriented languages began with SIMULA. Smalltalk was also an early and influential OO language. In such languages one can define classes which are templates or types for a group of objects. An object is some encapsulated data along with functions (often called methods) that define the behaviour of the object. Good examples are common data structures like stack, queue and binary tree. Many older languages have acquired OO features – for example C (as C++), Python, Lisp, Fortran (Fortran95 and later).

LISP was used widely for artificial intelligence programming in its early days. Two well-known early programs are Terry Winograd's SHRDLU and Carl Hewitt's Micro-Planner.

LISP was the first functional programming language and as such is a precursor of more recent functional languages like SML and Haskell.

collector. Garbage is the name given to memory that was once used by the program but cannot be accessed from the running program anymore. Since lists are continuously created and destroyed in LISP programs it is necessary to reclaim memory that is garbage. Otherwise, it is quite possible that the program will run out of memory and thereby crash. OO programs also behave in the same way where objects are created and then become inaccessible. Most modern functional and OO languages have garbage collectors (e.g., Java, Smalltalk, SML, Haskell), but C++ (also C) is an exception. It expects the programmer to manage the memory by explicitly deleting objects that will not be required any longer. This is the single major source of bugs in C++ programs – the memory leak. While garbage is being collected the program does not do useful work. So a lot of effort has been expended in designing efficient garbage collection algorithms and in identifying garbage at the point at which it is created so that it is cheaply collected. Much of this research was done in the context of LISP since it was the first mainstream language with garbage collection.

LISP was the first functional programming language and as such is a precursor of more recent functional languages like SML and Haskell. These languages are strongly typed with a type inference system unlike LISP. However, they are largely functional languages and lack the multi-paradigm dimension of LISP.

LISP is still used in AI research though it is not the dominant language in that area. It is also the base language of the EMACS editing system[4]. While it is unlikely that it will achieve the prominence it had in its earlier years, modern LISP is a standardized, multi-paradigm language that has many good quality implementations (some of them free) that are actively maintained. It has many books and good documentation support and remains a language that one can profitably choose to write

[4] EMACS is a very versatile editing system that has ELISP (or EMACSLISP) as its scripting language. This makes it possible to customize EMACS to do some very complex things like create an integrated program development environment. Another widely used program that uses a dialect of LISP as a scripting language is AutoCAD.

large systems, especially exploratory ones that require rapid prototyping and powerful language features.

Suggested Reading

[1] John McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part i, *Communications of ACM*, Vol.3, pp.184–195, 1960.

[2] Alonzo Church, *The Calculii of Lambda Conversion*, Princeton University Press, 1985.

[3] Stephen Cole Kleene, *Introduction to Metamathematics*, 7th Ed., North-Holland, 1974, Seventh printing.

[4] Alan M Turing. On computable numbers, with an application to the entscheidungs problem, *Proc. of the London Mathematical Society 2*, Vol.42, pp.230–265, 1937.

[5] Christos Papadimitriou and Harry R Lewis, *Elements of the Theory of Computation*, Prentice-Hall, 2nd Ed., 1997.

[6] J Roger Hindley and Jonathan P Seldin, *Lambda Calculus and Combinators, an Introduction*, Cambridge University Press, 2008.

[7] Guy L Steele, *Common Lisp, the Language*, Digital Press, 2nd Ed., 1990.

[8] Eugene Charniak, Christopher K Riesbeck, Drew V McDermott, and James R Meehan, *Arti cial Intelligence Programming*, Psychology Press, 2nd Ed., 2013. Updated for Common Lisp, First Edition 1978.

Address for Correspondence
Harish Karnick
Department of Computer Science and Engineering
IIT Kanpur
Kanpur 208 016, India.
Email: hk@cse.iitk.ac.in