Lisp (programming language)

From Wikipedia, the free encyclopedia

Lisp (historically, **LISP**) is a family of computer programming languages with a long history and a distinctive, fully parenthesized Polish prefix notation.[1] Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today; only Fortran is older (by one year). Like Fortran, Lisp has changed a great deal since its early days, and a number of dialects have existed over its history. Today, the most widely known general-purpose Lisp dialects are Common Lisp and Scheme.

Lisp was originally created as a practical mathematical notation for computer programs, influenced by the notation of Alonzo Church's lambda calculus. It quickly became the favored programming language for artificial intelligence (AI) research. As one of the earliest programming languages, Lisp pioneered many ideas in computer science, including tree data structures, automatic storage management, dynamic typing, conditionals, higher-order functions, recursion, and the self-hosting compiler.[2]

The name *LISP* derives from "LISt Processing". Linked lists are one of Lisp language's major data structures, and Lisp source code is itself made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or even new domain-specific languages embedded in Lisp.

Lisp

Paradigm(s)	Multi-paradigm: functional, procedural, reflective, meta
Appeared in	1958
Designed by	John McCarthy
Developer	Steve Russell, Timothy P. Hart, and Mike Levin
Typing discipline	Dynamic, strong
Dialects	Arc, AutoLISP, Clojure, Common Lisp, Emacs Lisp, EuLisp, Franz Lisp, Interlisp, ISLISP, LeLisp, LFE, Maclisp, MDL, Newlisp, NIL, Picolisp, Portable Standard Lisp, Racket, Scheme, SKILL, Spice Lisp, T, XLISP, Zetalisp
Influenced by	IPL
Influenced	CLIPS, CLU, COWSEL, Dylan, Falcon, Forth, Haskell, Io, Ioke, JavaScript, Julia, Logo, Lua, Mathematica, MDL, ML, Nu, OPS5, Perl, POP-2/11, Python, Qi, R, Shen, Rebol, Racket, Ruby, Smalltalk, Tcl

The interchangeability of code and data also gives Lisp its instantly recognizable syntax. All program code is written as *s-expressions*, or parenthesized lists. A function call or syntactic form is written as a list with the function or operator's name first, and the arguments following; for instance, a function f that takes three arguments might be called using `(f arg1 arg2 arg3)`.

Contents

History

Lisp was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT). McCarthy published its design in a paper in *Communications of the ACM* in 1960, entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I"[3] ("Part II" was never published). He showed that with a few simple operators and a notation for functions, one can build a Turing-complete language for algorithms.

Information Processing Language was the first AI language, from 1955 or 1956, and already included many of the concepts, such as list-processing and recursion, which came to be used in Lisp.

McCarthy's original notation used bracketed "M-expressions" that would be translated into S-expressions. As an example, the M-expression `car[cons[A,B]]` is equivalent to the S-expression `(car (cons A B))`. Once Lisp was implemented, programmers rapidly chose to use S-expressions, and M-expressions were abandoned. M-expressions surfaced again with short-lived attempts of MLISP[4] by Horace Enea and CGOL by Vaughan Pratt.

John McCarthy and Steve Russell

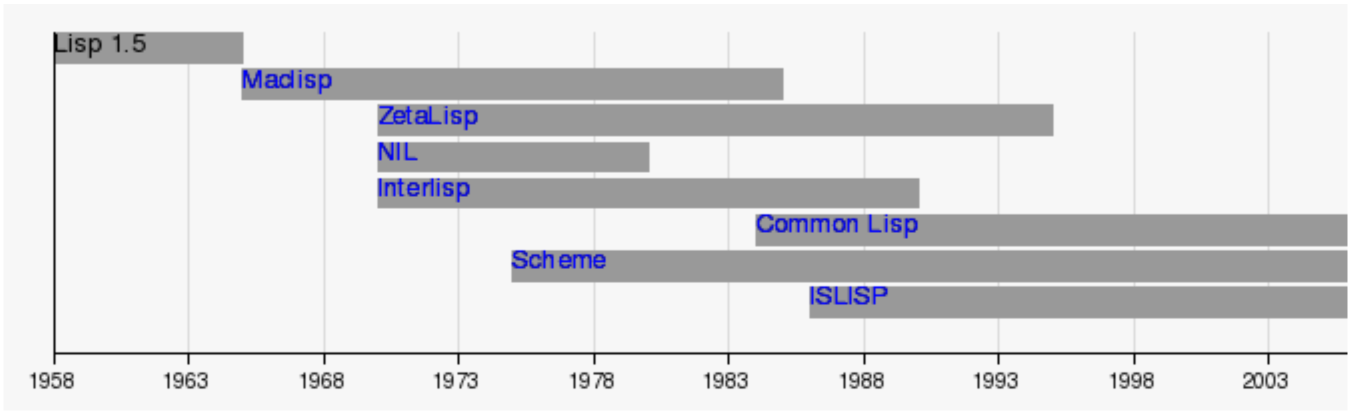
Lisp was first implemented by Steve Russell on an IBM 704 computer. Russell had read McCarthy's paper, and realized (to McCarthy's surprise) that the Lisp *eval* function could be implemented in machine code.[5] The result was a working Lisp interpreter which could be used to run Lisp programs, or more properly, 'evaluate Lisp expressions.'

Two assembly language macros for the IBM 704 became the primitive operations for decomposing lists: `car` (Contents of the Address part of Register number) and `cdr` (Contents of the Decrement part of Register number).[6] From the context, it is clear that the term "Register" is used here to mean "Memory Register", nowadays called "Memory Location". Lisp dialects still use `car` and `cdr` (/ˈkɑr/ and /ˈkʊdər/) for the operations that return the first item in a list and the rest of the list respectively.

The first complete Lisp compiler, written in Lisp, was implemented in 1962 by Tim Hart and Mike Levin at MIT.[7] This compiler introduced the Lisp model of incremental compilation, in which compiled and interpreted functions can intermix freely. The language used in Hart and Levin's memo is much closer to modern Lisp style than McCarthy's earlier code.

Lisp was a difficult system to implement with the compiler techniques and stock hardware of the 1970s. Garbage collection routines, developed by then-MIT graduate student Daniel Edwards, made it practical to run Lisp on general-purpose computing systems, but efficiency was still a problem.[*citation needed*] This led to the creation of Lisp machines: dedicated hardware for running Lisp environments and programs. Advances in both computer hardware and compiler technology soon made Lisp machines obsolete. [*citation needed*]

During the 1980s and 1990s, a great effort was made to unify the work on new Lisp dialects (mostly successors to Maclisp like ZetaLisp and NIL (New Implementation of Lisp)) into a single language. The new language, Common Lisp, was somewhat compatible with the dialects it replaced (the book Common Lisp the Language notes the compatibility of various constructs). In 1994, ANSI published the Common Lisp standard, "ANSI X3.226-1994 Information Technology Programming Language Common Lisp."



Connection to artificial intelligence

Since its inception, Lisp was closely connected with the artificial intelligence research community, especially on PDP-10[8] systems. Lisp was used as the implementation of the programming language Micro Planner which was used in the famous AI system SHRDLU. In the 1970s, as AI research spawned commercial offshoots, the performance of existing Lisp systems became a growing issue.[*citation needed*]

Genealogy and variants

Over its fifty-year history, Lisp has spawned many variations on the core theme of an S-expression language. Moreover, each given dialect may have several implementations—for instance, there are more than a dozen implementations of Common Lisp.

Differences between dialects may be quite visible—for instance, Common Lisp uses the keyword `defun` to name a function, but Scheme uses `define`.[9] Within a dialect that is standardized, however, conforming implementations support the same core language, but with different extensions and libraries.

Historically significant dialects

- LISP 1[10] – First implementation.
- LISP 1.5[11] – First widely distributed version, developed by McCarthy and others at MIT. So named because it contained several improvements on the original "LISP 1" interpreter, but was not a major restructuring as the planned LISP 2 would be.
- Stanford LISP 1.6[12] – This was a successor to LISP 1.5 developed at the Stanford AI Lab, and widely distributed to PDP-10 systems running the TOPS-10 operating system. It was rendered obsolete by Maclisp and InterLisp.
- MACLISP[13] – developed for MIT's Project MAC (no relation to Apple's Macintosh, nor to McCarthy), direct descendant of LISP 1.5. It ran on the PDP-10 and Multics systems. (MACLISP would later come to be called Maclisp, and is often referred to as MacLisp.)
- InterLisp[14] – developed at BBN Technologies for PDP-10 systems running the Tenex operating system, later adopted as a "West coast" Lisp for the Xerox Lisp machines as InterLisp-D. A small version called "InterLISP 65" was published for Atari's 6502-based computer line. For quite some

time Maclisp and InterLisp were strong competitors.
- Franz Lisp – originally a Berkeley project; later developed by Franz Inc. The name is a humorous deformation of the name "Franz Liszt", and does not refer to Allegro Common Lisp, the dialect of Common Lisp sold by Franz Inc., in more recent years.
- XLISP, which AutoLISP was based on.
- Standard Lisp and Portable Standard Lisp were widely used and ported, especially with the Computer Algebra System REDUCE.
- ZetaLisp, also known as Lisp Machine Lisp – used on the Lisp machines, direct descendant of Maclisp. ZetaLisp had big influence on Common Lisp.
- LeLisp is a French Lisp dialect. One of the first Interface Builders was written in LeLisp.
- Common Lisp (1984), as described by *Common Lisp the Language* – a consolidation of several divergent attempts (ZetaLisp, Spice Lisp, NIL, and S-1 Lisp) to create successor dialects[15] to Maclisp, with substantive influences from the Scheme dialect as well. This version of Common Lisp was available for wide-ranging platforms and was accepted by many as a de facto standard[16] until the publication of ANSI Common Lisp (ANSI X3.226-1994).

A Lisp machine in the MIT Museum

- Dylan was in its first version a mix of Scheme with the Common Lisp Object System.
- EuLisp – attempt to develop a new efficient and cleaned-up Lisp.
- ISLISP – attempt to develop a new efficient and cleaned-up Lisp. Standardized as ISO/IEC 13816:1997[17] and later revised as ISO/IEC 13816:2007:[18] *Information technology – Programming languages, their environments and system software interfaces – Programming language ISLISP*.
- IEEE Scheme – IEEE standard, 1178–1990 (R1995)
- ANSI Common Lisp – an American National Standards Institute (ANSI) standard for Common Lisp, created by subcommittee X3J13, chartered[19] to begin with *Common Lisp: The Language* as a base document and to work through a public consensus process to find solutions to shared issues of portability of programs and compatibility of Common Lisp implementations. Although formally an ANSI standard, the implementation, sale, use, and influence of ANSI Common Lisp has been and continues to be seen worldwide.
- ACL2 or "A Computational Logic for Applicative Common Lisp", an applicative (side-effect free) variant of Common LISP. ACL2 is both a programming language in which you can model computer systems and a tool to help proving properties of those models.
- Clojure, a modern dialect of Lisp which compiles to the Java virtual machine and handles concurrency very well.

2000-present

After having declined somewhat in the 1990s, Lisp has recently experienced a resurgence of interest. Most new activity is focused around open source implementations of Common Lisp, and includes the development of new portable libraries and applications. A new print edition of *Practical Common Lisp* by Peter Seibel, a tutorial for new Lisp programmers, was published in 2005.[20]

Many new Lisp programmers were inspired by writers such as Paul Graham and Eric S. Raymond to pursue a language others considered antiquated. New Lisp programmers often describe the language as an eye-opening experience and claim to be substantially more productive than in other languages.[21] This increase in awareness may be contrasted to the "AI winter" and Lisp's brief gain in the mid-1990s.[22]

Dan Weinreb lists in his survey of Common Lisp implementations[23] eleven actively maintained Common Lisp implementations. Scieneer Common Lisp is a new commercial implementation forked from CMUCL with a first release in 2002.

The open source community has created new supporting infrastructure: CLiki is a wiki that collects Common Lisp related information, the Common Lisp directory lists resources, #lisp is a popular IRC channel (with support by a Lisp-written Bot), lisppaste supports the sharing and commenting of code snippets, Planet Lisp collects the contents of various Lisp-related blogs, on LispForum users discuss Lisp topics, Lispjobs is a service for announcing job offers and there is a weekly news service, *Weekly Lisp News*. *Common-lisp.net* is a hosting site for open source Common Lisp projects.

50 years of Lisp (1958–2008) has been celebrated at LISP50@OOPSLA.[24] There are regular local user meetings in Boston, Vancouver, and Hamburg. Other events include the European Common Lisp Meeting, the European Lisp Symposium and an International Lisp Conference.

The Scheme community actively maintains over twenty implementations. Several significant new implementations (Chicken, Gambit, Gauche, Ikarus, Larceny, Ypsilon) have been developed in the last few years. The Revised^5 Report on the Algorithmic Language Scheme[25] standard of Scheme was widely accepted in the Scheme community. The Scheme Requests for Implementation process has created a lot of quasi standard libraries and extensions for Scheme. User communities of individual Scheme implementations continue to grow. A new language standardization process was started in 2003 and led to the R^{6}RS Scheme standard in 2007. Academic use of Scheme for teaching computer science seems to have declined somewhat. Some universities, such as MIT, are no longer using Scheme in their computer science introductory courses.[26][27]

There are several new dialects of Lisp: Arc, Nu, Clojure, Liskell, LFE (Lisp Flavored Erlang), and Shen.

Major dialects

The two major dialects of Lisp used for general-purpose programming today are Common Lisp and Scheme. These languages represent significantly different design choices.

Common Lisp is a successor to MacLisp. The primary influences were Lisp Machine Lisp, MacLisp, NIL, S-1 Lisp, Spice Lisp, and Scheme.[28] It has many of the features of Lisp Machine Lisp (a large Lisp dialect used to program Lisp Machines), but was designed to be efficiently implementable on any personal computer or workstation. Common Lisp has a large language standard including many built-in data types, functions, macros and other language elements, as well as an object system (Common Lisp Object System or shorter CLOS). Common Lisp also borrowed certain features from Scheme such as lexical scoping and lexical closures.

Scheme (designed earlier) is a more minimalist design, with a much smaller set of standard features but with certain implementation features (such as tail-call optimization and full continuations) not necessarily found in Common Lisp.

Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme. Scheme continues to evolve with a series of standards (Revised^n Report on the Algorithmic Language Scheme) and a series of Scheme Requests for Implementation.

Clojure is a recent dialect of Lisp that principally targets the Java Virtual Machine, as well as the CLR, the Python VM, the Ruby VM YARV, and compiling to JavaScript. It is designed to be a pragmatic general-purpose language. Clojure draws considerable influences from Haskell and places a very strong emphasis on immutability.[29] Clojure is a compiled language, as it compiles directly to JVM bytecode, yet remains completely dynamic. Every feature supported by Clojure is supported at runtime. Clojure provides access to Java frameworks and libraries, with optional type hints and type inference, so that calls to Java can avoid reflection and enable fast primitive operations.

In addition, Lisp dialects are used as scripting languages in a number of applications, with the most well-known being Emacs Lisp in the Emacs editor, AutoLisp and later Visual Lisp in AutoCAD, Nyquist in Audacity. The small size of a minimal but useful Scheme interpreter makes it particularly popular for embedded scripting. Examples include SIOD and TinyScheme, both of which have been successfully embedded in the GIMP image processor under the generic name "Script-fu".[30] LIBREP, a Lisp interpreter by John Harper originally based on the Emacs Lisp language, has been embedded in the Sawfish window manager.[31]

Language innovations

Lisp was the first homoiconic programming language: the primary representation of program code is the same type of list structure that is also used for the main data structures. As a result, Lisp functions can be manipulated, altered or even created within a Lisp program without extensive parsing or manipulation of binary machine code. This is generally considered one of the primary advantages of the language with regard to its expressive power, and makes the language amenable to metacircular evaluation.

The ubiquitous *if-then-else* structure, now taken for granted as an essential element of any programming language, was invented by McCarthy for use in Lisp, where it saw its first appearance in a more general form (the cond structure). It was inherited by ALGOL, which popularized it.

Lisp deeply influenced Alan Kay, the leader of the research on Smalltalk, and then in turn Lisp was influenced by Smalltalk, by adopting object-oriented programming features (classes, instances, etc.) in the late 1970s. The Flavours object system (later CLOS) introduced multiple inheritance.

Lisp introduced the concept of automatic garbage collection, in which the system walks the heap looking for unused memory. Most of the modern sophisticated garbage collection algorithms such as generational garbage collection were developed for Lisp.[32]

Edsger W. Dijkstra in his 1972 Turing Award lecture said,

> "With a few very basic principles at its foundation, it [LISP] has shown a remarkable stability. Besides that, LISP has been the carrier for a considerable number of in a sense our most sophisticated computer applications. LISP has jokingly been described as “the most intelligent way to misuse a computer”. I think that description a great compliment because it transmits the full flavour of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts."[33]

Largely because of its resource requirements with respect to early computing hardware (including early microprocessors), Lisp did not become as popular outside of the AI community as Fortran and the ALGOL-descended C language. Because of its suitability to complex and dynamic applications, Lisp is currently enjoying some resurgence of popular interest.

Syntax and semantics

> ***Note**: This article's examples are written in Common Lisp (though most are also valid in Scheme).*

Symbolic expressions (S-expressions)

Lisp is an expression-oriented language. Unlike most other languages, no distinction is made between "expressions" and "statements"; all code and data are written as expressions. When an expression is *evaluated*, it produces a value (in Common Lisp, possibly multiple values), which then can be embedded into other expressions. Each value can be any data type.

McCarthy's 1958 paper introduced two types of syntax: S-expressions (Symbolic expressions, also called "sexps"), which mirror the internal representation of code and data; and M-expressions (Meta Expressions), which express functions of S-expressions. M-expressions never found favor, and almost all Lisps today use S-expressions to manipulate both code and data.

The use of parentheses is Lisp's most immediately obvious difference from other programming language families. As a result, students have long given Lisp nicknames such as *Lost In Stupid Parentheses*, or *Lots of Irritating Superfluous Parentheses*.[34] However, the S-expression syntax is also responsible for much of Lisp's power: the syntax is extremely regular, which facilitates manipulation by computer. However, the syntax of Lisp is not limited to traditional parentheses notation. It can be extended to include alternative notations. XMLisp, for instance, is a Common Lisp extension that employs the metaobject-protocol to integrate S-expressions with the Extensible Markup Language (XML).

The reliance on expressions gives the language great flexibility. Because Lisp functions are themselves written as lists, they can be processed exactly like data. This allows easy writing of programs which manipulate other programs (metaprogramming). Many Lisp dialects exploit this feature using macro systems, which enables extension of the language almost without limit.

Lists

A Lisp list is written with its elements separated by whitespace, and surrounded by parentheses. For example, `(1 2 foo)` is a list whose elements are three *atoms*: the values `1`, `2`, and `foo`. These values are implicitly typed: they are respectively two integers and a Lisp-specific data type called a "symbolic atom", and do not have to be declared as such.

The empty list `()` is also represented as the special atom `nil`. This is the only entity in Lisp which is both an atom and a list.

Expressions are written as lists, using prefix notation. The first element in the list is the name of a *form*, i.e., a function, operator, macro, or "special operator" (see below). The remainder of the list are the arguments. For example, the function `list` returns its arguments as a list, so the expression

```
(list '1 '2 'foo)
```

evaluates to the list `(1 2 foo)`. The "quote" before the arguments in the preceding example is a "special operator" which prevents the quoted arguments from being evaluated (not strictly necessary for the numbers, since 1 evaluates to 1, etc.). Any unquoted expressions are recursively evaluated before the enclosing expression is evaluated. For example,

```
(list 1 2 (list 3 4))
```

evaluates to the list `(1 2 (3 4))`. Note that the third argument is a list; lists can be nested.

Operators

Arithmetic operators are treated similarly. The expression

```
(+ 1 2 3 4)
```

evaluates to 10. The equivalent under infix notation would be "`1 + 2 + 3 + 4`". Arithmetic operators in Lisp are variadic (or *n-ary*), able to take any number of arguments.

"Special operators" (sometimes called "special forms") provide Lisp's control structure. For example, the special operator `if` takes three arguments. If the first argument is non-nil, it evaluates to the second argument; otherwise, it evaluates to the third argument. Thus, the expression

```
(if nil
    (list 1 2 "foo")
    (list 3 4 "bar"))
```

evaluates to `(3 4 "bar")`. Of course, this would be more useful if a non-trivial expression had been substituted in place of `nil`.

Lambda expressions and function definition

Another special operator, `lambda`, is used to bind variables to values which are then evaluated within an expression. This operator is also used to create functions: the arguments to `lambda` are a list of arguments, and the expression or expressions to which the function evaluates (the returned value is the value of the last expression that is evaluated). The expression

```
(lambda (arg) (+ arg 1))
```

evaluates to a function that, when applied, takes one argument, binds it to `arg` and returns the number one greater than that argument. Lambda expressions are treated no differently from named functions; they are invoked the same way. Therefore, the expression

```
((lambda (arg) (+ arg 1)) 5)
```

evaluates to `6`.

Named functions are created by storing a lambda expression in a symbol using the defun macro.

```
(defun foo (a b c d) (+ a b c d))
```

`(defun f (a) b...)` defines a new function named `f` in the global environment. It is a shorthand for the expression:

```
(place-in-function-definition-slot-of-symbol 'f #'(lambda (a) b...))
```

Atoms

In the original **LISP** there were two fundamental data types: atoms and lists. A list was a finite ordered sequence of elements, where each element is in itself either an atom or a list, and an atom was a number or a symbol. A symbol was essentially a unique named item, written as an alphanumeric string in source code, and used either as a variable name or as a data item in symbolic processing. For example, the list `(FOO (BAR 1) 2)` contains three elements: the symbol FOO, the list `(BAR 1)`, and the number 2.

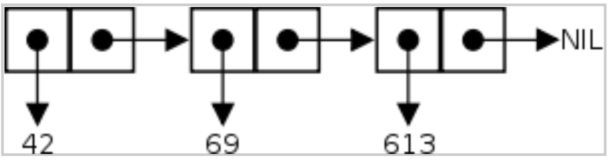
The essential difference between atoms and lists was that atoms were immutable and unique. Two atoms that appeared in different places in source code but were written in exactly the same way represented the same object[*citation needed*], whereas each list was a separate object that could be altered independently of other lists and could be distinguished from other lists by comparison operators.

As more data types were introduced in later Lisp dialects, and programming styles evolved, the concept of an atom lost importance.[*citation needed*] Many dialects still retained the predicate *atom* for legacy compatibility[*citation needed*], defining it true for any object which is not a cons.

Conses and lists

Main article: Cons

A Lisp list is a singly linked list. Each cell of this list is called a *cons* (in Scheme, a *pair*), and is composed of two pointers, called the *car* and *cdr*. These are equivalent to the `data` and `next` fields discussed in the article *linked list*, respectively.



Box-and-pointer diagram for the list (42 69 613)

Of the many data structures that can be built out of cons cells, one of the most basic is called a *proper list*. A proper list is either the special `nil` (empty list) symbol, or a cons in which the `car` points to a datum (which may be another cons structure, such as a list), and the `cdr` points to another proper list.

If a given cons is taken to be the head of a linked list, then its car points to the first element of the list, and its cdr points to the rest of the list. For this reason, the `car` and `cdr` functions are also called `first` and `rest` when referring to conses which are part of a linked list (rather than, say, a tree).

Thus, a Lisp list is not an atomic object, as an instance of a container class in C++ or Java would be. A list is nothing more than an aggregate of linked conses. A variable which refers to a given list is simply a pointer to the first cons in the list. Traversal of a list can be done by "cdring down" the list; that is, taking successive cdrs to visit each cons of the list; or by using any of a number of higher-order functions to map a function over a list.

Because conses and lists are so universal in Lisp systems, it is a common misconception that they are Lisp's only data structures. In fact, all but the most simplistic Lisps have other data structures – such as vectors (arrays), hash tables, structures, and so forth.

S-expressions represent lists

Parenthesized S-expressions represent linked list structures. There are several ways to represent the same list as an S-expression. A cons can be written in *dotted-pair notation* as `(a . b)`, where `a` is the car and `b` the cdr. A longer proper list might be written `(a . (b . (c . (d . nil))))` in dotted-pair notation. This is conventionally abbreviated as `(a b c d)` in *list notation*. An improper list[35] may be written in a combination of the two – as `(a b c . d)` for the list of three conses whose last cdr is `d` (i.e., the list `(a . (b . (c . d)))` in fully specified form).

List-processing procedures

Lisp provides many built-in procedures for accessing and controlling lists. Lists can be created directly with the `list` procedure, which takes any number of arguments, and returns the list of these arguments.

```
(list 1 2 'a 3)
;Output: (1 2 a 3)
```

```
(list 1 '(2 3) 4)
;Output: (1 (2 3) 4)
```

Because of the way that lists are constructed from cons pairs, the `cons` procedure can be used to add an element to the front of a list. Note that the `cons` procedure is asymmetric in how it handles list arguments, because of how lists are constructed.

```
(cons 1 '(2 3))
;Output: (1 2 3)
```

```
(cons '(1 2) '(3 4))
;Output: ((1 2) 3 4)
```

The `append` procedure appends two (or more) lists to one another. Because Lisp lists are linked lists, appending two lists has asymptotic time complexity $O(n)$

```
(append '(1 2) '(3 4))
```

```
;Output: (1 2 3 4)
```

```
(append '(1 2 3) '() '(a) '(5 6))
;Output: (1 2 3 a 5 6)
```

Shared structure

Lisp lists, being simple linked lists, can share structure with one another. That is to say, two lists can have the same *tail*, or final sequence of conses. For instance, after the execution of the following Common Lisp code:

```
(setf foo (list 'a 'b 'c))
(setf bar (cons 'x (cdr foo)))
```

the lists `foo` and `bar` are `(a b c)` and `(x b c)` respectively. However, the tail `(b c)` is the same structure in both lists. It is not a copy; the cons cells pointing to `b` and `c` are in the same memory locations for both lists.

Sharing structure rather than copying can give a dramatic performance improvement. However, this technique can interact in undesired ways with functions that alter lists passed to them as arguments. Altering one list, such as by replacing the `c` with a `goose`, will affect the other:

```
(setf (third foo) 'goose)
```

This changes `foo` to `(a b goose)`, but thereby also changes `bar` to `(x b goose)` – a possibly unexpected result. This can be a source of bugs, and functions which alter their arguments are documented as *destructive* for this very reason.

Aficionados of functional programming avoid destructive functions. In the Scheme dialect, which favors the functional style, the names of destructive functions are marked with a cautionary exclamation point, or "bang"—such as `set-car!` (read *set car bang*), which replaces the car of a cons. In the Common Lisp dialect, destructive functions are commonplace; the equivalent of `set-car!` is named `rplaca` for "replace car." This function is rarely seen however as Common Lisp includes a special facility, `setf`, to make it easier to define and use destructive functions. A frequent style in Common Lisp is to write code functionally (without destructive calls) when prototyping, then to add destructive calls as an optimization where it is safe to do so.

Self-evaluating forms and quoting

Lisp evaluates expressions which are entered by the user. Symbols and lists evaluate to some other (usually, simpler) expression – for instance, a symbol evaluates to the value of the variable it names; `(+ 2 3)` evaluates to `5`. However, most other forms evaluate to themselves: if you enter `5` into Lisp, it returns `5`.

Any expression can also be marked to prevent it from being evaluated (as is necessary for symbols and lists). This is the role of the `quote` special operator, or its abbreviation `'` (a single quotation mark). For instance, usually if you enter the symbol `foo` you will get back the value of the corresponding variable (or an error, if there is no such variable). If you wish to refer to the literal symbol, you enter `(quote foo)` or, usually, `'foo`.

Both Common Lisp and Scheme also support the *backquote* operator (known as *quasiquote* in Scheme), entered with the `` ` `` character (grave accent). This is almost the same as the plain quote, except it allows expressions to be evaluated and their values interpolated into a quoted list with the comma `,` *unquote* and comma-at `,@` *splice* operators. If the variable `snue` has the value `(bar baz)` then `` `(foo ,snue) `` evaluates to `(foo (bar baz))`, while `` `(foo ,@snue) `` evaluates to `(foo bar baz)`. The backquote is most frequently used in defining macro expansions.[36][37]

Self-evaluating forms and quoted forms are Lisp's equivalent of literals. It may be possible to modify the values of (mutable) literals in program code. For instance, if a function returns a quoted form, and the code that calls the function modifies the form, this may alter the behavior of the function on subsequent iterations.

```
(defun should-be-constant ()
  '(one two three))

(let ((stuff (should-be-constant)))
  (setf (third stuff) 'bizarre))   ; bad!

(should-be-constant)   ; returns (one two bizarre)
```

Modifying a quoted form like this is generally considered bad style, and is defined by ANSI Common Lisp as erroneous (resulting in "undefined" behavior in compiled files, because the file-compiler can coalesce similar constants, put them in write-protected memory, etc.).

Lisp's formalization of quotation has been noted by Douglas Hofstadter (in *Gödel, Escher, Bach*) and others as an example of the philosophical idea of self-reference.

Scope and closure

The modern Lisp family splits over the use of dynamic or static (aka lexical) scope. Clojure, Common Lisp and Scheme make use of static scoping by default, while Newlisp, Picolisp and the embedded languages in Emacs and AutoCAD use dynamic scoping.

List structure of program code; exploitation by macros and compilers

A fundamental distinction between Lisp and other languages is that in Lisp, the textual representation of a program is simply a human-readable description of the same internal data structures (linked lists, symbols, number, characters, etc.) as would be used by the underlying Lisp system.

Lisp uses this to implement a very powerful macro system. Like other macro languages such as C, a macro returns code that can then be compiled. However, unlike C macros, the macros are Lisp functions and so can exploit the full power of Lisp.

Further, because Lisp code has the same structure as lists, macros can be built with any of the list-processing functions in the language. In short, anything that Lisp can do to a data structure, Lisp macros can do to code. In contrast, in most other languages, the parser's output is purely internal to the language implementation and cannot be manipulated by the programmer.

This feature makes it easy to develop *efficient* languages within languages. For example, the Common Lisp Object System can be implemented cleanly as a language extension using macros. This means that if an application requires a different inheritance mechanism, it can use a different object system. This is in stark contrast to most other languages; for example, Java does not support multiple inheritance and there is no reasonable way to add it.

In simplistic Lisp implementations, this list structure is directly interpreted to run the program; a function is literally a piece of list structure which is traversed by the interpreter in executing it. However, most substantial Lisp systems also include a compiler. The compiler translates list structure into machine code or bytecode for execution. This code can run as fast as code compiled in conventional languages such as C.

Macros expand before the compilation step, and thus offer some interesting options. If a program needs a precomputed table, then a macro might create the table at compile time, so the compiler need only output the table and need not call code to create the table at run time. Some Lisp implementations even have a mechanism, `eval-when`, that allows code to be present during compile time (when a macro would need it), but not present in the emitted module.[38]

Evaluation and the read–eval–print loop

Lisp languages are frequently used with an interactive command line, which may be combined with an integrated development environment. The user types in expressions at the command line, or directs the IDE to transmit them to the Lisp system. Lisp *reads* the entered expressions, *evaluates* them, and *prints* the result. For this reason, the Lisp command line is called a "read–eval–print loop", or *REPL*.

The basic operation of the REPL is as follows. This is a simplistic description which omits many elements of a real Lisp, such as quoting and macros.

The `read` function accepts textual S-expressions as input, and parses them into an internal data structure. For instance, if you type the text `(+ 1 2)` at the prompt, `read` translates this into a linked list with three elements: the symbol +, the number 1, and the number 2. It so happens that this list is also a valid piece of Lisp code; that is, it can be evaluated. This is because the car of the list names a function—the addition operation.

Note that a `foo` will be read as a single symbol. `123` will be read as the number one hundred and twenty-three. `"123"` will be read as the string "123".

The `eval` function evaluates the data, returning zero or more other Lisp data as a result. Evaluation does not have to mean interpretation; some Lisp systems compile every expression to native machine code. It is simple, however, to describe evaluation as interpretation: To evaluate a list whose car names a function, `eval` first evaluates each of the arguments given in its cdr, then applies the function to the arguments. In this case, the function is addition, and applying it to the argument list `(1 2)` yields the answer `3`. This is the result of the evaluation.

The symbol `foo` evaluates to the value of the symbol foo. Data like the string "123" evaluates to the same string. The list `(quote (1 2 3))` evaluates to the list (1 2 3).

It is the job of the `print` function to represent output to the user. For a simple result such as `3` this is trivial. An expression which evaluated to a piece of list structure would require that `print` traverse the list and print it out as an S-expression.

To implement a Lisp REPL, it is necessary only to implement these three functions and an infinite-loop function. (Naturally, the implementation of `eval` will be complicated, since it must also implement all special operators like `if` or `lambda`.) This done, a basic REPL itself is but a single line of code: `(loop (print (eval (read))))`.

The Lisp REPL typically also provides input editing, an input history, error handling and an interface to the debugger.

Lisp is usually evaluated eagerly. In Common Lisp, arguments are evaluated in applicative order ('leftmost innermost'), while in Scheme order of arguments is undefined, leaving room for optimization by a compiler.

Control structures

Lisp originally had very few control structures, but many more were added during the language's evolution. (Lisp's original conditional operator, `cond`, is the precursor to later `if-then-else` structures.)

Programmers in the Scheme dialect often express loops using tail recursion. Scheme's commonality in academic computer science has led some students to believe that tail recursion is the only, or the most common, way to write iterations in Lisp, but this is incorrect. All frequently seen Lisp dialects have imperative-style iteration constructs, from Scheme's `do` loop to Common Lisp's complex `loop` expressions. Moreover, the key issue that makes this an objective rather than subjective matter is that Scheme makes specific requirements for the handling of tail calls, and consequently the reason that the use of tail recursion is generally encouraged for Scheme is that the practice is expressly supported by the language definition itself. By contrast, ANSI Common Lisp does not require[39] the optimization commonly referred to as tail call elimination. Consequently, the fact that tail recursive style as a casual replacement for the use of more traditional iteration constructs (such as `do`, `dolist` or `loop`) is discouraged[40] in Common Lisp is not just a matter of stylistic preference, but potentially one of efficiency (since an apparent tail call in Common Lisp may not compile as a simple jump) and program correctness (since tail recursion may increase stack use in Common Lisp, risking stack overflow).

Some Lisp control structures are *special operators*, equivalent to other languages' syntactic keywords. Expressions using these operators have the same surface appearance as function calls, but differ in that the arguments are not necessarily evaluated—or, in the case of an iteration expression, may be evaluated more than once.

In contrast to most other major programming languages, Lisp allows the programmer to implement control structures using the language itself. Several control structures are implemented as Lisp macros, and can even be macro-expanded by the programmer who wants to know how they work.

Both Common Lisp and Scheme have operators for non-local control flow. The differences in these operators are some of the deepest differences between the two dialects. Scheme supports *re-entrant continuations* using the `call/cc` procedure, which allows a program to save (and later restore) a particular place in execution. Common Lisp does not support re-entrant continuations, but does support several ways of handling escape continuations.

Frequently, the same algorithm can be expressed in Lisp in either an imperative or a functional style. As noted above, Scheme tends to favor the functional style, using tail recursion and continuations to express control flow. However, imperative style is still quite possible. The style preferred by many Common Lisp programmers may seem more familiar to programmers used to structured languages such as C, while that preferred by Schemers more closely resembles pure-functional languages such as Haskell.

Because of Lisp's early heritage in list processing, it has a wide array of higher-order functions relating to iteration over sequences. In many cases where an explicit loop would be needed in other languages (like a `for` loop in C) in Lisp the same task can be accomplished with a higher-order function. (The same is true of many functional programming languages.)

A good example is a function which in Scheme is called `map` and in Common Lisp is called `mapcar`. Given a function and one or more lists, `mapcar` applies the function successively to the lists' elements in order, collecting the results in a new list:

```
(mapcar #'+ '(1 2 3 4 5) '(10 20 30 40 50))
```

This applies the + function to each corresponding pair of list elements, yielding the result `(11 22 33 44 55)`.

Examples

Here are examples of Common Lisp code.

The basic "Hello world" program:

```
 (print "Hello world")
```

Lisp syntax lends itself naturally to recursion. Mathematical problems such as the enumeration of recursively defined sets are simple to express in this notation.

Evaluate a number's factorial:

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

An alternative implementation, often faster than the previous version if the Lisp system has tail recursion optimization:

```
(defun factorial (n &optional (acc 1))
  (if (<= n 1)
      acc
      (factorial (- n 1) (* acc n))))
```

Contrast with an iterative version which uses Common Lisp's `loop` macro:

```
(defun factorial (n)
  (loop for i from 1 to n
        for fac = 1 then (* fac i)
        finally (return fac)))
```

The following function reverses a list. (Lisp's built-in *reverse* function does the same thing.)

```
(defun -reverse (list)
  (let ((return-value '()))
    (dolist (e list) (push e return-value))
    return-value))
```

Object systems

Various object systems and models have been built on top of, alongside, or into Lisp, including:

- The Common Lisp Object System, CLOS, is an integral part of ANSI Common Lisp. CLOS descended from New Flavors and CommonLOOPS. ANSI Common Lisp was the first standardized object-oriented programming language (1994, ANSI X3J13).
- ObjectLisp[41] or Object Lisp, used by Lisp Machines Incorporated and early versions of Macintosh Common Lisp
- LOOPS (Lisp Object-Oriented Programming System) and the later CommonLOOPS
- Flavors, built at MIT, and its descendant New Flavors (developed by Symbolics).
- KR (short for Knowledge Representation), a constraints-based object system developed to aid the writing of Garnet, a GUI library for Common Lisp.
- KEE used an object system called UNITS and integrated it with an inference engine[42] and a truth maintenance system (ATMS).

See also

- Fexpr
- Maxima
- mod_lisp
- P convention
- Prolog

References

1. ^ Edwin D. Reilly (2003). *Milestones in computer science and information technology* (http://books.google.com/books?id=JTYPKxug49IC&pg=PA157). Greenwood Publishing Group. pp. 156–157. ISBN 978-1-57356-521-9.
2. ^ Paul Graham. "Revenge of the Nerds" (http://www.paulgraham.com/icad.html). Retrieved 2013-03-14.
3. ^ John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" (http://www-formal.stanford.edu/jmc/recursive.html). Retrieved 2006-10-13.
4. ^ David Canfield Smith. "MLISP Users Manual" (http://www.softwarepreservation.org/projects/LISP/stanford/Smith-MLISP-AIM-84.pdf). Retrieved 2006-10-13.
5. ^ According to what reported by Paul Graham in *Hackers & Painters*, p. 185, McCarthy said: "Steve Russell said, look, why don't I program this *eval*..., and I said to him, ho, ho, you're confusing theory with practice, this *eval* is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the *eval* in my paper into IBM 704 machine code, fixing bug, and then advertised this as a Lisp interpreter, which it certainly was. So at that point Lisp had essentially the form that it has today..."
6. ^ John McCarthy. "LISP prehistory - Summer 1956 through Summer 1958" (http://www-formal.stanford.edu/jmc/history/lisp/node2.html). Retrieved 2010-03-14.
7. ^ Tim Hart and Mike Levin. "AI Memo 39-The new compiler" (ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-039.pdf). Retrieved 2006-10-13.
8. ^ The 36-bit word size of the PDP-6/PDP-10 was influenced by the usefulness of having two Lisp 18-bit pointers in a single word. Peter J. Hurley (18 October 1990). "The History of TOPS or Life in the Fast ACs (news:84950@tut.cis.ohio-state.edu)". alt.folklore.computers (news:alt.folklore.computers). Web link (http://groups.google.com/group/alt.folklore.computers/browse_thread/thread/6e5602ce733d0ec/17597705ae289112). "The PDP-6 project started in early 1963, as a 24-bit machine. It grew to 36 bits for LISP, a design goal.".
9. ^ Common Lisp: `(defun f (x) x)`
   Scheme: `(define f (lambda (x) x))` or `(define (f x) x)`

10. ^ McCarthy, J.; Brayton, R.; Edwards, D.; Fox, P.; Hodes, L.; Luckham, D.; Maling, K.; Park, D. et al. (March 1960). *LISP I Programmers Manual* (http://history.siam.org/sup/Fox_1960_LISP.pdf). Boston, Massachusetts: Artificial Intelligence Group, M.I.T. Computation Center and Research Laboratory Accessed May 11, 2010.
11. ^ McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P.; Levin, Michael I. (1962; 2nd Edition, 15th printing, 1985). *LISP 1.5 Programmer's Manual* (http://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf). MIT Press. ISBN 0-262-13011-4.
12. ^ Quam, Lynn H.; Diffle, Whitfield. *Stanford LISP 1.6 Manual* (http://www.softwarepreservation.org/projects/LISP/stanford/SAILON-28.6.pdf) (PDF).
13. ^ "Maclisp Reference Manual" (http://web.archive.org/web/20071214064433/http://zane.brouhaha.com/~healyzh/doc/lisp.doc.txt). March 3, 1979. Archived from the original (http://zane.brouhaha.com/~healyzh/doc/lisp.doc.txt) on 2007-12-14.
14. ^ Teitelman, Warren (1974). *InterLisp Reference Manual* (http://www.bitsavers.org/pdf/xerox/interlisp/1974_InterlispRefMan.pdf) (PDF).
15. ^ Steele, Guy L., Jr. "Purpose" (http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node6.html). *Common Lisp the Language* (2nd ed.). ISBN 0-13-152414-3.
16. ^ Kantrowitz, Mark; Margolin, Barry (20 February 1996). "History: Where did Lisp come from?" (http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/lisp/part2/faq-doc-13.html). *FAQ: Lisp Frequently Asked Questions 2/7*.
17. ^ "ISO/IEC 13816:1997" (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22987). Iso.org. 2007-10-01. Retrieved 2013-11-15.
18. ^ "ISO/IEC 13816:2007" (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=44338). Iso.org. 2013-10-30. Retrieved 2013-11-15.
19. ^ "X3J13 Charter" (http://www.nhplace.com/kent/CL/x3j13-86-020.html).
20. ^ Siebel, Peter (2005). *Practical Common Lisp* (http://gigamonkeys.com/book/). Apress. ISBN 978-1-59059-239-7.
21. ^ "The Road To Lisp Survey" (http://wiki.alu.org/The_Road_To_Lisp_Survey). Retrieved 2006-10-13.
22. ^ "Trends for the Future" (http://www.faqs.org/docs/artu/ch14s05.html). Faqs.org. Retrieved 2013-11-15.
23. ^ Weinreb, Daniel. "Common Lisp Implementations: A Survey" (http://common-lisp.net/~dlw/LispSurvey.html). Retrieved 4 April 2012.
24. ^ "LISP50@OOPSLA" (http://www.lisp50.org/). Lisp50.org. Retrieved 2013-11-15.
25. ^ Documents: Standards: R5RS (http://www.schemers.org/Documents/Standards/R5RS/). schemers.org (2012-01-11). Retrieved on 2013-07-17.
26. ^ "Why MIT now uses python instead of scheme for its undergraduate CS program" (http://cemerick.com/2009/03/24/why-mit-now-uses-python-instead-of-scheme-for-its-undergraduate-cs-program/). *cemerick.com*. March 24, 2009. Retrieved November 10, 2013.
27. ^ Broder, Evan (January 8, 2008). "The End of an Era" (http://mitadmissions.org/blogs/entry/the_end_of_an_era_1). *mitadmissions.org*. Retrieved November 10, 2013.
28. ^ Chapter 1.1.2, History, ANSI CL Standard
29. ^ An In-Depth Look at Clojure Collections (http://www.infoq.com/articles/in-depth-look-clojure-collections), Retrieved 2012-06-24
30. ^ Script-fu In GIMP 2.4 (http://www.gimp.org/docs/script-fu-update.html), Retrieved 2009-10-29
31. ^ librep (http://sawfish.wikia.com/wiki/Librep) at Sawfish Wikia, retrieved 2009-10-29
32. ^ Lieberman, Henry; Hewitt, Carl (June 1983), "A Real-Time Garbage Collector Based on the Lifetimes of Objects" (http://web.media.mit.edu/~lieber/Lieberary/GC/Realtime/Realtime.html), *CACM* **26** (6): 419–429, doi:10.1145/358141.358147 (http://dx.doi.org/10.1145%2F358141.358147)
33. ^ Edsger W. Dijkstra (1972), *The Humble Programmer (EWD 340)* (http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html) (ACM Turing Award lecture).
34. ^ "The Jargon File - Lisp" (http://www.catb.org/~esr/jargon/html/L/LISP.html). Retrieved 2006-10-13.
35. ^ NB: a so-called "dotted list" is only one kind of "improper list". The other kind is the "circular list" where the cons cells form a loop. Typically this is represented using #n=(...) to represent the target cons cell that will have multiple references, and #n# is used to refer to this cons. For instance, (#1=(a b) . #1#) would normally be printed as ((a b) a b) (without circular structure printing enabled), but makes the reuse of the cons cell clear. #1=(a . #1#) cannot normally be printed as it is circular, the CDR of the cons cell defined by #1= is itself.

36. ^ "CSE 341: Scheme: Quote, Quasiquote, and Metaprogramming" (http://www.cs.washington.edu/education/courses/cse341/04wi/lectures/14-scheme-quote.html). Cs.washington.edu. 1999-02-22. Retrieved 2013-11-15.
37. ^ Quasiquotation in Lisp (http://repository.readscheme.org/ftp/papers/pepm99/bawden.pdf), Alan Bawden
38. ^ Time of Evaluation - Common Lisp Extensions (http://www.gnu.org/software/emacs/manual/html_node/cl/Time-of-Evaluation.html). Gnu.org. Retrieved on 2013-07-17.
39. ^ 3.2.2.3 Semantic Constraints (http://www.lispworks.com/documentation/HyperSpec/Body/03_bbc.htm) in *Common Lisp HyperSpec* (http://www.lispworks.com/documentation/HyperSpec/Front/index.htm)
40. ^ 4.3. Control Abstraction (Recursion vs. Iteration) in Tutorial on Good Lisp Programming Style (http://www.cs.umd.edu/~nau/cmsc421/norvig-lisp-style.pdf) by Pitman and Norvig, August, 1993.
41. ^ pg 17 of Bobrow 1986
42. ^ Veitch, p 108, 1988

Further reading

- McCarthy, John (1979-02-12). "The implementation of Lisp" (http://www-formal.stanford.edu/jmc/history/lisp/node3.html). *History of Lisp*. Stanford University. Retrieved 2008-10-17.
- Steele, Jr., Guy L.; Richard P. Gabriel (1993). "The evolution of Lisp" (http://www.dreamsongs.com/NewFiles/HOPL2-Uncut.pdf). *The second ACM SIGPLAN conference on History of programming languages*. New York, NY: ACM, ISBN 0-89791-570-4. pp. 231–270. ISBN 0-89791-570-4. Retrieved 2008-10-17.
- Veitch, Jim (1998). "A history and description of CLOS". In Salus, Peter H. *Handbook of programming languages*. Volume IV, Functional and logic programming languages (first ed.). Indianapolis, IN: Macmillan Technical Publishing. pp. 107–158. ISBN 1-57870-011-6
- Abelson, Harold; Sussman, Gerald Jay; Sussman, Julie (1996). *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press. ISBN 0-262-01153-0.
- My Lisp Experiences and the Development of GNU Emacs (http://www.gnu.org/gnu/rms-lisp.html), transcript of Richard Stallman's speech, 28 October 2002, at the International Lisp Conference
- Graham, Paul (2004). *Hackers & Painters. Big Ideas from the Computer Age*. O'Reilly. ISBN 0-596-00662-4.
- Berkeley, Edmund C.; Bobrow, Daniel G., eds. (March 1964). *The Programming Language LISP: Its Operation and Applications* (http://www.softwarepreservation.org/projects/LISP/book/III_LispBook_Apr66.pdf). Cambridge, Massachusetts: MIT Press.
- Weissman, Clark (1967). *LISP 1.5 Primer* (http://www.softwarepreservation.org/projects/LISP/book/Weismann_LISP1.5_Primer_1967.pdf). Belmont, California: Dickenson Publishing Company Inc.

External links

History

- History of Lisp (http://www-formal.stanford.edu/jmc/history/lisp/lisp.html) – John McCarthy's history of 12 February 1979
- Lisp History (http://www8.informatik.uni-erlangen.de/html/lisp-enter.html) – Herbert Stoyan's history compiled from the documents (acknowledged by McCarthy as more complete than his own, see: McCarthy's history links (http://www-formal.stanford.edu/jmc/history/))
- History of LISP at the Computer History Museum (http://www.softwarepreservation.org/projects/LISP/)

Associations and meetings

- Association of Lisp Users (http://www.alu.org/)

- European Common Lisp Meeting (http://www.weitz.de/eclm2009/)
- European Lisp Symposium (http://european-lisp-symposium.org/)
- International Lisp Conference (http://www.international-lisp-conference.org/)

Books and tutorials

- *Casting SPELs in Lisp (http://www.lisperati.com/casting.html)*, a comic-book style introductory tutorial
- *On Lisp (http://paulgraham.com/onlisptext.html)*, a free book by Paul Graham
- *Practical Common Lisp (http://www.gigamonkeys.com/book/)*, freeware edition by Peter Seibel
- Lisp for the web (https://leanpub.com/lispweb)
- Land of Lisp (http://landoflisp.com/)
- Let over Lambda (http://letoverlambda.com/)

Interviews

- Oral history interview with John McCarthy (http://purl.umn.edu/107476) at Charles Babbage Institute, University of Minnesota, Minneapolis. McCarthy discusses his role in the development of time-sharing at the Massachusetts Institute of Technology. He also describes his work in artificial intelligence (AI) funded by the Advanced Research Projects Agency, including logic-based AI (LISP) and robotics.
- Interview (http://www.se-radio.net/2008/01/episode-84-dick-gabriel-on-lisp/) with Richard P. Gabriel (Podcast)

Resources

- CLiki: the common lisp wiki (http://www.cliki.net/)
- Common Lisp directory (http://www.cl-user.net/)
- Lisp FAQ Index (http://www.faqs.org/faqs/lisp-faq/)
- lisppaste (http://paste.lisp.org/)
- Planet Lisp (http://planet.lisp.org/)
- Weekly Lisp News (http://lispnews.wordpress.com/)
- Lisp (http://www.dmoz.org/Computers/Programming/Languages/Lisp/) on the Open Directory Project

Retrieved from "http://en.wikipedia.org/w/index.php?title=Lisp_(programming_language)&oldid=592201559"

Categories: 1958 in computer science | Academic programming languages | American inventions | Articles with example Lisp code | Dynamically typed programming languages | Functional languages | Lisp programming language | Lisp programming language family | Programming languages created in 1958 | Programming languages created in the 1950s | Extensible syntax programming languages

---