Common Lisp

Common Lisp (**CL**) is a dialect of the Lisp programming language, published in ANSI standard document *ANSI INCITS 226-1994 [S2008]* (formerly *X3.226-1994 (R1999)*).[1] The Common Lisp HyperSpec, a hyperlinked HTML version, has been derived from the ANSI Common Lisp standard.[2]

The Common Lisp language was developed as a standardized and improved successor of Maclisp. By the early 1980s several groups were already at work on diverse successors to MacLisp: Lisp Machine Lisp (aka ZetaLisp), Spice Lisp, NIL and S-1 Lisp. Common Lisp sought to unify, standardise, and extend the features of these MacLisp dialects. Common Lisp is not an implementation, but rather a language specification.[3] Several implementations of the Common Lisp standard are available, including free and open-source software and proprietary products.[4] Common Lisp is a general-purpose, multi-paradigm programming language. It supports a combination of procedural, functional, and object-oriented programming paradigms. As a dynamic programming language, it facilitates evolutionary and incremental software development, with iterative compilation into efficient run-time programs. This incremental development is often done interactively without interrupting the running application.

It also supports optional type annotation and casting, which can be added as necessary at the later profiling and optimization stages, to permit the compiler to generate more efficient code. For instance, `fixnum` can hold an unboxed integer in a range supported by the hardware and implementation, permitting more efficient arithmetic than on big integers or arbitrary precision types. Similarly, the compiler can be told on a per-module or per-function basis which type of safety level is wanted, using *optimize* declarations.

Common Lisp includes CLOS, an object system that supports multimethods and method combinations. It is often implemented with a Metaobject Protocol.

Common Lisp is extensible through standard features such as *Lisp macros* (code transformations) and *reader macros* (input parsers for characters).

Common Lisp provides partial backwards compatibility with Maclisp and John McCarthy's original Lisp. This allows older Lisp software to be ported to Common Lisp.[5]

Contents

Common Lisp

Paradigm	Multi-paradigm: procedural, functional, object-oriented, meta, reflective, generic
Family	Lisp
Designed by	Scott Fahlman, Richard P. Gabriel, David A. Moon, Kent Pitman, Guy Steele, Dan Weinreb
Developer	ANSI X3J13 committee
First appeared	1984, 1994 for ANSI Common Lisp
Typing discipline	Dynamic, strong
Scope	Lexical, optionally dynamic
OS	Cross-platform
Filename extensions	.lisp, .lsp, .l, .cl, .fasl
Website	common-lisp.net (http://common-lisp.net/)
Major implementations	
Allegro CL, ABCL, CLISP, Clozure CL, CMUCL, ECL, GCL, LispWorks, Scieneer CL, SBCL, Symbolics Common Lisp	
Dialects	
CLtL1, CLtL2, ANSI Common Lisp	
Influenced by	
Lisp, Lisp Machine Lisp, Maclisp,	

Scheme, Interlisp
Influenced
Clojure, Dylan, Emacs Lisp, EuLisp, ISLISP, *Lisp, AutoLisp, Julia, Moose, R, SKILL, SubL

History

Work on Common Lisp started in 1981 after an initiative by ARPA manager Bob Engelmore to develop a single community standard Lisp dialect.[6] Much of the initial language design was done via electronic mail.[7][8] In 1982, Guy L. Steele, Jr. gave the first overview of Common Lisp at the 1982 ACM Symposium on LISP and functional programming.[9]

The first language documentation was published in 1984 as Common Lisp the Language (known as CLtL1), first edition. A second edition (known as CLtL2), published in 1990, incorporated many changes to the language, made during the ANSI Common Lisp standardization process: extended LOOP syntax, the Common Lisp Object System, the Condition System for error handling, an interface to the pretty printer and much more. But CLtL2 does not describe the final ANSI Common Lisp standard and thus is not a documentation of ANSI Common Lisp. The final ANSI Common Lisp standard then was published in 1994. Since then no update to the standard has been published. Various extensions and improvements to Common Lisp (examples are Unicode, Concurrency, CLOS-based IO) have been provided by implementations and libraries.

Syntax

Common Lisp is a dialect of Lisp. It uses S-expressions to denote both code and data structure. Function calls, macro forms and special forms are written as lists, with the name of the operator first, as in these examples:

```
  (+ 2 2)           ; adds 2 and 2, yielding 4. The function's name is '+'. Lisp has no operators
as such.
```

```
  (defvar *x*)      ; Ensures that a variable *x* exists,
                    ; without giving it a value. The asterisks are part of
                    ; the name, by convention denoting a special (global) variable.
                    ; The symbol *x* is also hereby endowed with the property that
                    ; subsequent bindings of it are dynamic, rather than lexical.
  (setf *x* 42.1)   ; Sets the variable *x* to the floating-point value 42.1
```

```
  ;; Define a function that squares a number:
  (defun square (x)
    (* x x))
```

```
  ;; Execute the function:
  (square 3)        ; Returns 9
```

```
  ;; The 'let' construct creates a scope for local variables. Here
  ;; the variable 'a' is bound to 6 and the variable 'b' is bound
  ;; to 4. Inside the 'let' is a 'body', where the last computed value is returned.
  ;; Here the result of adding a and b is returned from the 'let' expression.
  ;; The variables a and b have lexical scope, unless the symbols have been
  ;; marked as special variables (for instance by a prior DEFVAR).
  (let ((a 6)
        (b 4))
    (+ a b))        ; returns 10
```

Data types

Common Lisp has many data types.

Scalar types

Number types include integers, ratios, floating-point numbers, and complex numbers.[10] Common Lisp uses bignums to represent numerical values of arbitrary size and precision. The ratio type represents fractions exactly, a facility not available in many languages. Common Lisp automatically coerces numeric values among these types as appropriate.

The Common Lisp *character* type is not limited to ASCII characters. Most modern implementations allow Unicode characters.[11]

The *symbol* type is common to Lisp languages, but largely unknown outside them. A symbol is a unique, named data object with several parts: name, value, function, property list, and package. Of these, *value cell* and *function cell* are the most important. Symbols in Lisp are often used similarly to identifiers in other languages: to hold the value of a variable; however there are many other uses. Normally, when a symbol is evaluated, its value is returned. Some symbols evaluate to themselves, for example, all symbols in the keyword package are self-evaluating. Boolean values in Common Lisp are represented by the self-evaluating symbols T and NIL. Common Lisp has namespaces for symbols, called 'packages'.

A number of functions are available for rounding scalar numeric values in various ways. The function `round` rounds the argument to the nearest integer, with halfway cases rounded to the even integer. The functions `truncate`, `floor`, and `ceiling` round towards zero, down, or up respectively. All these functions return the discarded fractional part as a secondary value. For example, `(floor -2.5)` yields −3, 0.5; `(ceiling -2.5)` yields −2, −0.5; `(round 2.5)` yields 2, 0.5; and `(round 3.5)` yields 4, −0.5.

Data structures

Sequence types in Common Lisp include lists, vectors, bit-vectors, and strings. There are many operations that can work on any sequence type.

As in almost all other Lisp dialects, *lists* in Common Lisp are composed of *conses*, sometimes called *cons cells* or *pairs*. A cons is a data structure with two slots, called its *car* and *cdr*. A list is a linked chain of conses or the empty list. Each cons's car refers to a member of the list (possibly another list). Each cons's cdr refers to the next cons—except for the last cons in a list, whose cdr refers to the `nil` value. Conses can also easily be used to implement trees and other complex data structures; though it is usually advised to use structure or class instances instead. It is also possible to create circular data structures with conses.

Common Lisp supports multidimensional *arrays*, and can dynamically resize *adjustable* arrays if required. Multidimensional arrays can be used for matrix mathematics. A *vector* is a one-dimensional array. Arrays can carry any type as members (even mixed types in the same array) or can be specialized to contain a specific type of members, as in a vector of bits. Usually, only a few types are supported. Many implementations can optimize array functions when the array used is type-specialized. Two type-specialized array types are standard: a *string* is a vector of characters, while a *bit-vector* is a vector of bits.

Hash tables store associations between data objects. Any object may be used as key or value. Hash tables are automatically resized as needed.

Packages are collections of symbols, used chiefly to separate the parts of a program into namespaces. A package may *export* some symbols, marking them as part of a public interface. Packages can use other packages.

Structures, similar in use to C structs and Pascal records, represent arbitrary complex data structures with any number and type of fields (called *slots*). Structures allow single-inheritance.

Classes are similar to structures, but offer more dynamic features and multiple-inheritance. (See CLOS). Classes have been added late to Common Lisp and there is some conceptual overlap with structures. Objects created of classes are called *Instances*. A special case is Generic Functions. Generic Functions are both functions and instances.

Functions

Common Lisp supports first-class functions. For instance, it is possible to write functions that take other functions as arguments or return functions as well. This makes it possible to describe very general operations.

The Common Lisp library relies heavily on such higher-order functions. For example, the `sort` function takes a relational operator as an argument and key function as an optional keyword argument. This can be used not only to sort any type of data, but also to sort data structures according to a key.

```
;; Sorts the list using the > and < function as the relational operator.
(sort (list 5 2 6 3 1 4) #'>)   ; Returns (6 5 4 3 2 1)
(sort (list 5 2 6 3 1 4) #'<)   ; Returns (1 2 3 4 5 6)
```

```
;; Sorts the list according to the first element of each sub-list.
(sort (list '(9 A) '(3 B) '(4 C)) #'< :key #'first)   ; Returns ((3 B) (4 C) (9 A))
```

The evaluation model for functions is very simple. When the evaluator encounters a form `(f a1 a2...)` then it presumes that the symbol named f is one of the following:

1. A special operator (easily checked against a fixed list)
2. A macro operator (must have been defined previously)
3. The name of a function (default), which may either be a symbol, or a sub-form beginning with the symbol `lambda`.

If f is the name of a function, then the arguments a1, a2, ..., an are evaluated in left-to-right order, and the function is found and invoked with those values supplied as parameters.

Defining functions

The macro `defun` defines functions where a function definition gives the name of the function, the names of any arguments, and a function body:

```
(defun square (x)
  (* x x))
```

Function definitions may include compiler directives, known as *declarations*, which provide hints to the compiler about optimization settings or the data types of arguments. They may also include *documentation strings* (docstrings), which the Lisp system may use to provide interactive documentation:

```
(defun square (x)
  "Calculates the square of the single-float x."
  (declare (single-float x) (optimize (speed 3) (debug 0) (safety 1)))
  (the single-float (* x x)))
```

Anonymous functions (function literals) are defined using `lambda` expressions, e.g. `(lambda (x) (* x x))` for a function that squares its argument. Lisp programming style frequently uses higher-order functions for which it is useful to provide anonymous functions as arguments.

Local functions can be defined with `flet` and `labels`.

```
(flet ((square (x)
         (* x x)))
  (square 3))
```

There are several other operators related to the definition and manipulation of functions. For instance, a function may be compiled with the `compile` operator. (Some Lisp systems run functions using an interpreter by default unless instructed to compile; others compile every function).

Defining generic functions and methods

The macro `defgeneric` defines generic functions. Generic functions are a collection of methods. The macro `defmethod` defines methods.

Methods can specialize their parameters over CLOS *standard classes*, *system classes*, *structure classes* or individual objects. For many types, there are corresponding *system classes*.

When a generic function is called, multiple-dispatch will determine the effective method to use.

```
(defgeneric add (a b))
```

```
(defmethod add ((a number) (b number))
  (+ a b))
```

```
(defmethod add ((a vector) (b number))
  (map 'vector (lambda (n) (+ n b)) a))
```

```
(defmethod add ((a vector) (b vector))
  (map 'vector #'+ a b))
```

```
(defmethod add ((a string) (b string))
  (concatenate 'string a b))
```

```
(add 2 3)                    ; returns 5
(add #(1 2 3 4) 7)           ; returns #(8 9 10 11)
(add #(1 2 3 4) #(4 3 2 1)) ; returns #(5 5 5 5)
(add "COMMON " "LISP")       ; returns "COMMON LISP"
```

Generic Functions are also a first class data type. There are many more features to Generic Functions and Methods than described above.

The function namespace

The namespace for function names is separate from the namespace for data variables. This is a key difference between Common Lisp and Scheme. For Common Lisp, operators that define names in the function namespace include `defun`, `flet`, `labels`, `defmethod` and `defgeneric`.

To pass a function by name as an argument to another function, one must use the `function` special operator, commonly abbreviated as `#'`. The first `sort` example above refers to the function named by the symbol `>` in the function namespace, with the code `#'>`. Conversely, to call a function passed in such a way, one would use the `funcall` operator on the argument.

Scheme's evaluation model is simpler: there is only one namespace, and all positions in the form are evaluated (in any order) – not just the arguments. Code written in one dialect is therefore sometimes confusing to programmers more experienced in the other. For instance, many Common Lisp programmers like to use descriptive variable names such as *list* or *string* which could cause problems in Scheme, as they would locally shadow function names.

Whether a separate namespace for functions is an advantage is a source of contention in the Lisp community. It is usually referred to as the *Lisp-1 vs. Lisp-2 debate*. Lisp-1 refers to Scheme's model and Lisp-2 refers to Common Lisp's model. These names were coined in a 1988 paper by Richard P. Gabriel and Kent Pitman, which extensively compares the two approaches.[12]

Multiple return values

Common Lisp supports the concept of *multiple values*,[13] where any expression always has a single *primary value*, but it might also have any number of *secondary values*, which might be received and inspected by interested callers. This concept is distinct from returning a list value, as the secondary values are fully optional, and passed via a dedicated side channel. This means that callers may remain entirely unaware of the secondary values being there if they have no need for them, and it makes it convenient to use the mechanism for communicating information that is sometimes useful, but not always necessary. For example,

- The `TRUNCATE` function[14] rounds the given number to an integer towards zero. However, it also returns a remainder as a secondary value, making it very easy to determine what value was truncated. It also supports an optional divisor parameter, which can be used to perform Euclidean division trivially:

```
(let ((x 1266778)
      (y 458))
  (multiple-value-bind (quotient remainder)
      (truncate x y)
    (format nil "~A divided by ~A is ~A remainder ~A" x y quotient remainder)))

;;;; => "1266778 divided by 458 is 2765 remainder 408"
```

- `GETHASH`[15] returns the value of a key in an associative map, or the default value otherwise, and a secondary boolean indicating whether the value was found. Thus code which does not care about whether the value was found or provided as the default can simply use it as-is, but when such distinction is important, it might inspect the secondary boolean and react appropriately. Both use cases are supported by the same call and neither is unnecessarily burdened or constrained by the other. Having this feature at the language level removes the need to check for the existence of the key or compare it to null as would be done in other languages.

```
(defun get-answer (library)
  (gethash 'answer library 42))

(defun the-answer-1 (library)
  (format nil "The answer is ~A" (get-answer library)))
;;;; Returns "The answer is 42" if ANSWER not present in LIBRARY

(defun the-answer-2 (library)
  (multiple-value-bind (answer sure-p)
      (get-answer library)
    (if (not sure-p)
```

```
        "I don't know"
    (format nil "The answer is ~A" answer))))
;;;; Returns "I don't know" if ANSWER not present in LIBRARY
```

Multiple values are supported by a handful of standard forms, most common of which are the `MULTIPLE-VALUE-BIND` special form for accessing secondary values and `VALUES` for returning multiple values:

```
(defun magic-eight-ball ()
  "Return an outlook prediction, with the probability as a secondary value"
  (values "Outlook good" (random 1.0)))

;;;; => "Outlook good"
;;;; => 0.3187
```

Other types

Other data types in Common Lisp include:

- *Pathnames* represent files and directories in the filesystem. The Common Lisp pathname facility is more general than most operating systems' file naming conventions, making Lisp programs' access to files broadly portable across diverse systems.
- Input and output *streams* represent sources and sinks of binary or textual data, such as the terminal or open files.
- Common Lisp has a built-in pseudo-random number generator (PRNG). *Random state* objects represent reusable sources of pseudo-random numbers, allowing the user to seed the PRNG or cause it to replay a sequence.
- *Conditions* are a type used to represent errors, exceptions, and other "interesting" events to which a program may respond.
- *Classes* are first-class objects, and are themselves instances of classes called metaobject classes (metaclasses for short).
- *Readtables* are a type of object which control how Common Lisp's reader parses the text of source code. By controlling which readtable is in use when code is read in, the programmer can change or extend the language's syntax.

Scope

Like programs in many other programming languages, Common Lisp programs make use of names to refer to variables, functions, and many other kinds of entities. Named references are subject to scope.

The association between a name and the entity which the name refers to is called a binding.

Scope refers to the set of circumstances in which a name is determined to have a particular binding.

Determiners of scope

The circumstances which determine scope in Common Lisp include:

- the location of a reference within an expression. If it's the leftmost position of a compound, it refers to a special operator or a macro or function binding, otherwise to a variable binding or something else.

- the kind of expression in which the reference takes place. For instance, `(go x)` means transfer control to label `x`, whereas `(print x)` refers to the variable `x`. Both scopes of `x` can be active in the same region of program text, since tagbody labels are in a separate namespace from variable names. A special form or macro form has complete control over the meanings of all symbols in its syntax. For instance, in `(defclass x (a b) ())`, a class definition, the `(a b)` is a list of base classes, so these names are looked up in the space of class names, and `x` isn't a reference to an existing binding, but the name of a new class being derived from `a` and `b`. These facts emerge purely from the semantics of `defclass`. The only generic fact about this expression is that `defclass` refers to a macro binding; everything else is up to `defclass`.
- the location of the reference within the program text. For instance, if a reference to variable `x` is enclosed in a binding construct such as a `let` which defines a binding for `x`, then the reference is in the scope created by that binding.
- for a variable reference, whether or not a variable symbol has been, locally or globally, declared special. This determines whether the reference is resolved within a lexical environment, or within a dynamic environment.
- the specific instance of the environment in which the reference is resolved. An environment is a run-time dictionary which maps symbols to bindings. Each kind of reference uses its own kind of environment. References to lexical variables are resolved in a lexical environment, et cetera. More than one environment can be associated with the same reference. For instance, thanks to recursion or the use of multiple threads, multiple activations of the same function can exist at the same time. These activations share the same program text, but each has its own lexical environment instance.

To understand what a symbol refers to, the Common Lisp programmer must know what kind of reference is being expressed, what kind of scope it uses if it is a variable reference (dynamic versus lexical scope), and also the run-time situation: in what environment is the reference resolved, where was the binding introduced into the environment, et cetera.

Kinds of environment

Global

Some environments in Lisp are globally pervasive. For instance, if a new type is defined, it is known everywhere thereafter. References to that type look it up in this global environment.

Dynamic

One type of environment in Common Lisp is the dynamic environment. Bindings established in this environment have dynamic extent, which means that a binding is established at the start of the execution of some construct, such as a `let` block, and disappears when that construct finishes executing: its lifetime is tied to the dynamic activation and deactivation of a block. However, a dynamic binding is not just visible within that block; it is also visible to all functions invoked from that block. This type of visibility is known as indefinite scope. Bindings which exhibit dynamic extent (lifetime tied to the activation and deactivation of a block) and indefinite scope (visible to all functions which are called from that block) are said to have dynamic scope.

Common Lisp has support for dynamically scoped variables, which are also called special variables. Certain other kinds of bindings are necessarily dynamically scoped also, such as restarts and catch tags. Function bindings cannot be dynamically scoped using `flet` (which only provides lexically scoped function bindings),

but function objects (a first-level object in Common Lisp) can be assigned to dynamically scoped variables, bound using `let` in dynamic scope, then called using `funcall` or `APPLY`.

Dynamic scope is extremely useful because it adds referential clarity and discipline to global variables. Global variables are frowned upon in computer science as potential sources of error, because they can give rise to ad-hoc, covert channels of communication among modules that lead to unwanted, surprising interactions.

In Common Lisp, a special variable which has only a top-level binding behaves just like a global variable in other programming languages. A new value can be stored into it, and that value simply replaces what is in the top-level binding. Careless replacement of the value of a global variable is at the heart of bugs caused by the use of global variables. However, another way to work with a special variable is to give it a new, local binding within an expression. This is sometimes referred to as "rebinding" the variable. Binding a dynamically scoped variable temporarily creates a new memory location for that variable, and associates the name with that location. While that binding is in effect, all references to that variable refer to the new binding; the previous binding is hidden. When execution of the binding expression terminates, the temporary memory location is gone, and the old binding is revealed, with the original value intact. Of course, multiple dynamic bindings for the same variable can be nested.

In Common Lisp implementations which support multithreading, dynamic scopes are specific to each thread of execution. Thus special variables serve as an abstraction for thread local storage. If one thread rebinds a special variable, this rebinding has no effect on that variable in other threads. The value stored in a binding can only be retrieved by the thread which created that binding. If each thread binds some special variable `*x*`, then `*x*` behaves like thread-local storage. Among threads which do not rebind `*x*`, it behaves like an ordinary global: all of these threads refer to the same top-level binding of `*x*`.

Dynamic variables can be used to extend the execution context with additional context information which is implicitly passed from function to function without having to appear as an extra function parameter. This is especially useful when the control transfer has to pass through layers of unrelated code, which simply cannot be extended with extra parameters to pass the additional data. A situation like this usually calls for a global variable. That global variable must be saved and restored, so that the scheme doesn't break under recursion: dynamic variable rebinding takes care of this. And that variable must be made thread-local (or else a big mutex must be used) so the scheme doesn't break under threads: dynamic scope implementations can take care of this also.

In the Common Lisp library, there are many standard special variables. For instance, all standard I/O streams are stored in the top-level bindings of well-known special variables. The standard output stream is stored in *standard-output*.

Suppose a function foo writes to standard output:

```
(defun foo ()
  (format t "Hello, world"))
```

To capture its output in a character string, *standard-output* can be bound to a string stream and called:

```
(with-output-to-string (*standard-output*)
  (foo))
```

```
-> "Hello, world" ; gathered output returned as a string
```

Lexical

Common Lisp supports lexical environments. Formally, the bindings in a lexical environment have lexical scope and may have either an indefinite extent or dynamic extent, depending on the type of namespace. Lexical scope means that visibility is physically restricted to the block in which the binding is established. References which are not textually (i.e. lexically) embedded in that block simply do not see that binding.

The tags in a TAGBODY have lexical scope. The expression (GO X) is erroneous if it is not embedded in a TAGBODY which contains a label X. However, the label bindings disappear when the TAGBODY terminates its execution, because they have dynamic extent. If that block of code is re-entered by the invocation of a lexical closure, it is invalid for the body of that closure to try to transfer control to a tag via GO:

```
(defvar *stashed*) ;; will hold a function

(tagbody
  (setf *stashed* (lambda () (go some-label)))
  (go end-label) ;; skip the (print "Hello")
 some-label
  (print "Hello")
 end-label)
-> NIL
```

When the TAGBODY is executed, it first evaluates the setf form which stores a function in the special variable *stashed*. Then the (go end-label) transfers control to end-label, skipping the code (print "Hello"). Since end-label is at the end of the tagbody, the tagbody terminates, yielding NIL. Suppose that the previously remembered function is now called:

```
(funcall *stashed*) ;; Error!
```

This situation is erroneous. One implementation's response is an error condition containing the message, "GO: tagbody for tag SOME-LABEL has already been left". The function tried to evaluate (go some-label), which is lexically embedded in the tagbody, and resolves to the label. However, the tagbody isn't executing (its extent has ended), and so the control transfer cannot take place.

Local function bindings in Lisp have lexical scope, and variable bindings also have lexical scope by default. By contrast with GO labels, both of these have indefinite extent. When a lexical function or variable binding is established, that binding continues to exist for as long as references to it are possible, even after the construct which established that binding has terminated. References to lexical variables and functions after the termination of their establishing construct are possible thanks to lexical closures.

Lexical binding is the default binding mode for Common Lisp variables. For an individual symbol, it can be switched to dynamic scope, either by a local declaration, by a global declaration. The latter may occur implicitly through the use of a construct like DEFVAR or DEFPARAMETER. It is an important convention in Common Lisp programming that special (i.e. dynamically scoped) variables have names which begin and end with an asterisk sigil * in what is called the "earmuff convention".[16] If adhered to, this convention effectively creates a separate namespace for special variables, so that variables intended to be lexical are not accidentally made special.

Lexical scope is useful for several reasons.

Firstly, references to variables and functions can be compiled to efficient machine code, because the run-time environment structure is relatively simple. In many cases it can be optimized to stack storage, so opening and closing lexical scopes has minimal overhead. Even in cases where full closures must be generated, access to the closure's environment is still efficient; typically each variable becomes an offset into a vector of bindings, and so a variable reference becomes a simple load or store instruction with a base-plus-offset addressing mode.

Secondly, lexical scope (combined with indefinite extent) gives rise to the lexical closure, which in turn creates a whole paradigm of programming centered around the use of functions being first-class objects, which is at the root of functional programming.

Thirdly, perhaps most importantly, even if lexical closures are not exploited, the use of lexical scope isolates program modules from unwanted interactions. Due to their restricted visibility, lexical variables are private. If one module A binds a lexical variable X, and calls another module B, references to X in B will not accidentally resolve to the X bound in A. B simply has no access to X. For situations in which disciplined interactions through a variable are desirable, Common Lisp provides special variables. Special variables allow for a module A to set up a binding for a variable X which is visible to another module B, called from A. Being able to do this is an advantage, and being able to prevent it from happening is also an advantage; consequently, Common Lisp supports both lexical and dynamic scope.

Macros

A *macro* in Lisp superficially resembles a function in usage. However, rather than representing an expression which is evaluated, it represents a transformation of the program source code. The macro gets the source it surrounds as arguments, binds them to its parameters and computes a new source form. This new form can also use a macro. The macro expansion is repeated until the new source form does not use a macro. The final computed form is the source code executed at runtime.

Typical uses of macros in Lisp:

- new control structures (example: looping constructs, branching constructs)
- scoping and binding constructs
- simplified syntax for complex and repeated source code
- top-level defining forms with compile-time side-effects
- data-driven programming
- embedded domain specific languages (examples: SQL, HTML, Prolog)
- implicit finalization forms

Various standard Common Lisp features also need to be implemented as macros, such as:

- the standard `setf` abstraction, to allow custom compile-time expansions of assignment/access operators
- `with-accessors`, `with-slots`, `with-open-file` and other similar `WITH` macros
- Depending on implementation, `if` or `cond` is a macro built on the other, the special operator; `when` and `unless` consist of macros
- The powerful `loop` domain-specific language

Macros are defined by the *defmacro* macro. The special operator *macrolet* allows the definition of local (lexically scoped) macros. It is also possible to define macros for symbols using *define-symbol-macro* and *symbol-macrolet*.

Paul Graham's book On Lisp describes the use of macros in Common Lisp in detail. Doug Hoyte's book Let Over Lambda extends the discussion on macros, claiming "Macros are the single greatest advantage that lisp has as a programming language and the single greatest advantage of any programming language." Hoyte provides several examples of iterative development of macros.

Example using a macro to define a new control structure

Macros allow Lisp programmers to create new syntactic forms in the language. One typical use is to create new control structures. The example macro provides an `until` looping construct. The syntax is:

```
(until test form*)
```

The macro definition for *until*:

```
(defmacro until (test &body body)
  (let ((start-tag (gensym "START"))
        (end-tag   (gensym "END")))
    `(tagbody ,start-tag
              (when ,test (go ,end-tag))
              (progn ,@body)
              (go ,start-tag)
              ,end-tag)))
```

tagbody is a primitive Common Lisp special operator which provides the ability to name tags and use the *go* form to jump to those tags. The backquote ` provides a notation that provides code templates, where the value of forms preceded with a comma are filled in. Forms preceded with comma and at-sign are *spliced* in. The tagbody form tests the end condition. If the condition is true, it jumps to the end tag. Otherwise the provided body code is executed and then it jumps to the start tag.

An example of using the above *until* macro:

```
(until (= (random 10) 0)
  (write-line "Hello"))
```

The code can be expanded using the function *macroexpand-1*. The expansion for above example looks like this:

```
(TAGBODY
 #:START1136
 (WHEN (ZEROP (RANDOM 10))
   (GO #:END1137))
 (PROGN (WRITE-LINE "hello"))
 (GO #:START1136)
 #:END1137)
```

During macro expansion the value of the variable *test* is *(= (random 10) 0)* and the value of the variable *body* is *((write-line "Hello"))*. The body is a list of forms.

Symbols are usually automatically upcased. The expansion uses the TAGBODY with two labels. The symbols for these labels are computed by GENSYM and are not interned in any package. Two *go* forms use these tags to jump to. Since *tagbody* is a primitive operator in Common Lisp (and not a macro), it will not be expanded into something else. The expanded form uses the *when* macro, which also will be expanded. Fully expanding a source form is called *code walking*.

In the fully expanded (*walked*) form, the *when* form is replaced by the primitive *if*:

```
(TAGBODY
 #:START1136
 (IF (ZEROP (RANDOM 10))
     (PROGN (GO #:END1137))
   NIL)
 (PROGN (WRITE-LINE "hello"))
 (GO #:START1136))
 #:END1137)
```

All macros must be expanded before the source code containing them can be evaluated or compiled normally. Macros can be considered functions that accept and return S-expressions – similar to abstract syntax trees, but not limited to those. These functions are invoked before the evaluator or compiler to produce the final source code. Macros are written in normal Common Lisp, and may use any Common Lisp (or third-party) operator available.

Variable capture and shadowing

Common Lisp macros are capable of what is commonly called *variable capture*, where symbols in the macro-expansion body coincide with those in the calling context, allowing the programmer to create macros wherein various symbols have special meaning. The term *variable capture* is somewhat misleading, because all namespaces are vulnerable to unwanted capture, including the operator and function namespace, the tagbody label namespace, catch tag, condition handler and restart namespaces.

Variable capture can introduce software defects. This happens in one of the following two ways:

- In the first way, a macro expansion can inadvertently make a symbolic reference which the macro writer assumed will resolve in a global namespace, but the code where the macro is expanded happens to provide a local, shadowing definition which steals that reference. Let this be referred to as type 1 capture.
- The second way, type 2 capture, is just the opposite: some of the arguments of the macro are pieces of code supplied by the macro caller, and those pieces of code are written such that they make references to surrounding bindings. However, the macro inserts these pieces of code into an expansion which defines its own bindings that accidentally captures some of these references.

The Scheme dialect of Lisp provides a macro-writing system which provides the referential transparency that eliminates both types of capture problem. This type of macro system is sometimes called "hygienic", in particular by its proponents (who regard macro systems which do not automatically solve this problem as unhygienic).

In Common Lisp, macro hygiene is ensured one of two different ways.

One approach is to use gensyms: guaranteed-unique symbols which can be used in a macro-expansion without threat of capture. The use of gensyms in a macro definition is a manual chore, but macros can be written which simplify the instantiation and use of gensyms. Gensyms solve type 2 capture easily, but they are not applicable to type 1 capture in the same way, because the macro expansion cannot rename the interfering symbols in the surrounding code which capture its references. Gensyms could be used to provide stable aliases for the global symbols which the macro expansion needs. The macro expansion would use these secret aliases rather than the well-known names, so redefinition of the well-known names would have no ill effect on the macro.

Another approach is to use packages. A macro defined in its own package can simply use internal symbols in that package in its expansion. The use of packages deals with type 1 and type 2 capture.

However, packages don't solve the type 1 capture of references to standard Common Lisp functions and operators. The reason is that the use of packages to solve capture problems revolves around the use of private symbols (symbols in one package, which are not imported into, or otherwise made visible in other packages). Whereas the Common Lisp library symbols are external, and frequently imported into or made visible in user-defined packages.

The following is an example of unwanted capture in the operator namespace, occurring in the expansion of a macro:

```
;; expansion of UNTIL makes liberal use of DO
(defmacro until (expression &body body)
  `(do () (,expression) ,@body))

;; macrolet establishes lexical operator binding for DO
(macrolet ((do (...) ... something else ...))
  (until (= (random 10) 0) (write-line "Hello")))
```

The `until` macro will expand into a form which calls `do` which is intended to refer to the standard Common Lisp macro `do`. However, in this context, `do` may have a completely different meaning, so `until` may not work properly.

Common Lisp solves the problem of the shadowing of standard operators and functions by forbidding their redefinition. Because it redefines the standard operator `do`, the preceding is actually a fragment of non-conforming Common Lisp, which allows implementations to diagnose and reject it.

Condition system

The *condition system* is responsible for exception handling in Common Lisp.[17] It provides *conditions*, *handler*s and *restart*s. *Condition*s are objects describing an exceptional situation (for example an error). If a *condition* is signaled, the Common Lisp system searches for a *handler* for this condition type and calls the handler. The *handler* can now search for restarts and use one of these restarts to automatically repair the current problem, using information such as the condition type and any relevant information provided as part of the condition object, and call the appropriate restart function.

These restarts, if unhandled by code, can be presented to users (as part of a user interface, that of a debugger for example), so that the user can select and invoke one of the available restarts. Since the condition handler is called in the context of the error (without unwinding the stack), full error recovery is possible in many cases, where other exception handling systems would have already terminated the current routine. The debugger itself can also be customized or replaced using the `*debugger-hook*` dynamic variable. Code found within *unwind-protect* forms such as finalizers will also be executed as appropriate despite the exception.

In the following example (using Symbolics Genera) the user tries to open a file in a Lisp function *test* called from the Read-Eval-Print-LOOP (REPL), when the file does not exist. The Lisp system presents four restarts. The user selects the *Retry OPEN using a different pathname* restart and enters a different pathname (lispm-init.lisp instead of lispm-int.lisp). The user code does not contain any error handling code. The whole error handling and restart code is provided by the Lisp system, which can handle and repair the error without terminating the user code.

```
Command: (test ">zippy>lispm-int.lisp")

Error: The file was not found.
       For lispm:>zippy>lispm-int.lisp.newest

LMFS:OPEN-LOCAL-LMFS-1
   Arg 0: #P"lispm:>zippy>lispm-int.lisp.newest"

s-A, <Resume>: Retry OPEN of lispm:>zippy>lispm-int.lisp.newest
s-B:           Retry OPEN using a different pathname
s-C, <Abort>:  Return to Lisp Top Level in a TELNET server
s-D:           Restart process TELNET terminal

-> Retry OPEN using a different pathname
Use what pathname instead [default lispm:>zippy>lispm-int.lisp.newest]:
```

```
    lispm:>zippy>lispm-init.lisp.newest

 ...the program continues
```

Common Lisp Object System (CLOS)

Common Lisp includes a toolkit for object-oriented programming, the Common Lisp Object System or CLOS, which is one of the most powerful object systems available in any language. For example, Peter Norvig explains how many Design Patterns are simpler to implement in a dynamic language with the features of CLOS (Multiple Inheritance, Mixins, Multimethods, Metaclasses, Method combinations, etc.).[18] Several extensions to Common Lisp for object-oriented programming have been proposed to be included into the ANSI Common Lisp standard, but eventually CLOS was adopted as the standard object-system for Common Lisp. CLOS is a dynamic object system with multiple dispatch and multiple inheritance, and differs radically from the OOP facilities found in static languages such as C++ or Java. As a dynamic object system, CLOS allows changes at runtime to generic functions and classes. Methods can be added and removed, classes can be added and redefined, objects can be updated for class changes and the class of objects can be changed.

CLOS has been integrated into ANSI Common Lisp. Generic functions can be used like normal functions and are a first-class data type. Every CLOS class is integrated into the Common Lisp type system. Many Common Lisp types have a corresponding class. There is more potential use of CLOS for Common Lisp. The specification does not say whether conditions are implemented with CLOS. Pathnames and streams could be implemented with CLOS. These further usage possibilities of CLOS for ANSI Common Lisp are not part of the standard. Actual Common Lisp implementations use CLOS for pathnames, streams, input–output, conditions, the implementation of CLOS itself and more.

Compiler and interpreter

A Lisp interpreter directly executes Lisp source code provided as Lisp objects (lists, symbols, numbers, ...) read from s-expressions. A Lisp compiler generates bytecode or machine code from Lisp source code. Common Lisp allows both individual Lisp functions to be compiled in memory and the compilation of whole files to externally stored compiled code (*fasl* files).

Several implementations of earlier Lisp dialects provided both an interpreter and a compiler. Unfortunately often the semantics were different. These earlier Lisps implemented lexical scoping in the compiler and dynamic scoping in the interpreter. Common Lisp requires that both the interpreter and compiler use lexical scoping by default. The Common Lisp standard describes both the semantics of the interpreter and a compiler. The compiler can be called using the function *compile* for individual functions and using the function *compile-file* for files. Common Lisp allows type declarations and provides ways to influence the compiler code generation policy. For the latter various optimization qualities can be given values between 0 (not important) and 3 (most important): *speed*, *space*, *safety*, *debug* and *compilation-speed*.

There is also a function to evaluate Lisp code: `eval`. `eval` takes code as pre-parsed s-expressions and not, like in some other languages, as text strings. This way code can be constructed with the usual Lisp functions for constructing lists and symbols and then this code can be evaluated with the function `eval`. Several Common Lisp implementations (like Clozure CL and SBCL) are implementing `eval` using their compiler. This way code is compiled, even though it is evaluated using the function `eval`.

The file compiler is invoked using the function *compile-file*. The generated file with compiled code is called a *fasl* (from *fast load*) file. These *fasl* files and also source code files can be loaded with the function *load* into a running Common Lisp system. Depending on the implementation, the file compiler generates byte-code (for

example for the Java Virtual Machine), C language code (which then is compiled with a C compiler) or, directly, native code.

Common Lisp implementations can be used interactively, even though the code gets fully compiled. The idea of an Interpreted language thus does not apply for interactive Common Lisp.

The language makes a distinction between read-time, compile-time, load-time, and run-time, and allows user code to also make this distinction to perform the wanted type of processing at the wanted step.

Some special operators are provided to especially suit interactive development; for instance, `defvar` will only assign a value to its provided variable if it wasn't already bound, while `defparameter` will always perform the assignment. This distinction is useful when interactively evaluating, compiling and loading code in a live image.

Some features are also provided to help writing compilers and interpreters. Symbols consist of first-level objects and are directly manipulable by user code. The `progv` special operator allows to create lexical bindings programmatically, while packages are also manipulable. The Lisp compiler is available at runtime to compile files or individual functions. These make it easy to use Lisp as an intermediate compiler or interpreter for another language.

Code examples

Birthday paradox

The following program calculates the smallest number of people in a room for whom the probability of unique birthdays is less than 50% (the birthday paradox, where for 1 person the probability is obviously 100%, for 2 it is 364/365, etc.). The answer is 23.

By convention, constants in Common Lisp are enclosed with + characters.

```
(defconstant +year-size+ 365)

(defun birthday-paradox (probability number-of-people)
  (let ((new-probability (* (/ (- +year-size+ number-of-people)
                               +year-size+)
                            probability)))
    (if (< new-probability 0.5)
        (1+ number-of-people)
        (birthday-paradox new-probability (1+ number-of-people)))))
```

Calling the example function using the REPL (Read Eval Print Loop):

```
CL-USER > (birthday-paradox 1.0 1)
23
```

Sorting a list of person objects

We define a class `person` and a method for displaying the name and age of a person. Next we define a group of persons as a list of `person` objects. Then we iterate over the sorted list.

```
(defclass person ()
  ((name :initarg :name :accessor person-name)
```

```
   (age  :initarg :age  :accessor person-age))
  (:documentation "The class PERSON with slots NAME and AGE."))

(defmethod display ((object person) stream)
  "Displaying a PERSON object to an output stream."
  (with-slots (name age) object
    (format stream "~a (~a)" name age)))

(defparameter *group*
  (list (make-instance 'person :name "Bob"   :age 33)
        (make-instance 'person :name "Chris" :age 16)
        (make-instance 'person :name "Ash"   :age 23))
  "A list of PERSON objects.")

(dolist (person (sort (copy-list *group*)
                      #'>
                      :key #'person-age))
  (display person *standard-output*)
  (terpri))
```

It prints the three names with descending age.

```
Bob (33)
Ash (23)
Chris (16)
```

Exponentiating by squaring

Use of the LOOP macro is demonstrated:

```
(defun power (x n)
  (loop with result = 1
        while (plusp n)
        when (oddp n) do (setf result (* result x))
        do (setf x (* x x)
                 n (truncate n 2))
        finally (return result)))
```

Example use:

```
CL-USER > (power 2 200)
1606938044258990275541962092341162602522202993782792835301376
```

Compare with the built in exponentiation:

```
CL-USER > (= (expt 2 200) (power 2 200))
T
```

Find the list of available shells

WITH-OPEN-FILE is a macro that opens a file and provides a stream. When the form is returning, the file is automatically closed. FUNCALL calls a function object. The LOOP collects all lines that match the predicate.

```
(defun list-matching-lines (file predicate)
  "Returns a list of lines in file, for which the predicate applied to
 the line returns T."
  (with-open-file (stream file)
    (loop for line = (read-line stream nil nil)
```

```
          while line
          when (funcall predicate line)
          collect it)))
```

The function AVAILABLE-SHELLS calls above function LIST-MATCHING-LINES with a pathname and an anonymous function as the predicate. The predicate returns the pathname of a shell or NIL (if the string is not the filename of a shell).

```
(defun available-shells (&optional (file #p"/etc/shells"))
  (list-matching-lines
   file
   (lambda (line)
     (and (plusp (length line))
          (char= (char line 0) #\/)
          (pathname
           (string-right-trim '(#\space #\tab) line))))))
```

Example results (on Mac OS X 10.6):

```
CL-USER > (available-shells)
(#P"/bin/bash" #P"/bin/csh" #P"/bin/ksh" #P"/bin/sh" #P"/bin/tcsh" #P"/bin/zsh")
```

Comparison with other Lisps

Common Lisp is most frequently compared with, and contrasted to, Scheme—if only because they are the two most popular Lisp dialects. Scheme predates CL, and comes not only from the same Lisp tradition but from some of the same engineers—Guy L. Steele, with whom Gerald Jay Sussman designed Scheme, chaired the standards committee for Common Lisp.

Common Lisp is a general-purpose programming language, in contrast to Lisp variants such as Emacs Lisp and AutoLISP which are extension languages embedded in particular products (GNU Emacs and AutoCAD, respectively). Unlike many earlier Lisps, Common Lisp (like Scheme) uses lexical variable scope by default for both interpreted and compiled code.

Most of the Lisp systems whose designs contributed to Common Lisp—such as ZetaLisp and Franz Lisp—used dynamically scoped variables in their interpreters and lexically scoped variables in their compilers. Scheme introduced the sole use of lexically scoped variables to Lisp; an inspiration from ALGOL 68. CL supports dynamically scoped variables as well, but they must be explicitly declared as "special". There are no differences in scoping between ANSI CL interpreters and compilers.

Common Lisp is sometimes termed a *Lisp-2* and Scheme a *Lisp-1*, referring to CL's use of separate namespaces for functions and variables. (In fact, CL has *many* namespaces, such as those for go tags, block names, and `loop` keywords). There is a long-standing controversy between CL and Scheme advocates over the tradeoffs involved in multiple namespaces. In Scheme, it is (broadly) necessary to avoid giving variables names which clash with functions; Scheme functions frequently have arguments named `lis`, `lst`, or `lyst` so as not to conflict with the system function `list`. However, in CL it is necessary to explicitly refer to the function namespace when passing a function as an argument—which is also a common occurrence, as in the `sort` example above.

CL also differs from Scheme in its handling of boolean values. Scheme uses the special values #t and #f to represent truth and falsity. CL follows the older Lisp convention of using the symbols T and NIL, with NIL standing also for the empty list. In CL, *any* non-NIL value is treated as true by conditionals, such as `if`, whereas in Scheme all non-#f values are treated as true. These conventions allow some operators in both

languages to serve both as predicates (answering a boolean-valued question) and as returning a useful value for further computation, but in Scheme the value '() which is equivalent to NIL in Common Lisp evaluates to true in a boolean expression.

Lastly, the Scheme standards documents require tail-call optimization, which the CL standard does not. Most CL implementations do offer tail-call optimization, although often only when the programmer uses an optimization directive. Nonetheless, common CL coding style does not favor the ubiquitous use of recursion that Scheme style prefers—what a Scheme programmer would express with tail recursion, a CL user would usually express with an iterative expression in `do`, `dolist`, `loop`, or (more recently) with the `iterate` package.

Implementations

See the Category Common Lisp implementations.

Common Lisp is defined by a specification (like Ada and C) rather than by one implementation (like Perl). There are many implementations, and the standard details areas in which they may validly differ.

In addition, implementations tend to come with extensions, which provide functionality not covered in the standard:

- Interactive Top-Level (REPL)
- Garbage Collection
- Debugger, Stepper and Inspector
- Weak data structures (hash tables)
- Extensible sequences
- Extensible LOOP
- Environment access
- CLOS Meta-object Protocol
- CLOS based extensible streams
- CLOS based Condition System
- Network streams
- Persistent CLOS
- Unicode support
- Foreign-Language Interface (often to C)
- Operating System interface
- Java Interface
- Threads and Multiprocessing
- Application delivery (applications, dynamic libraries)
- Saving of images

Free and open-source software libraries have been created to support extensions to Common Lisp in a portable way, and are most notably found in the repositories of the Common-Lisp.net[19] and CLOCC (Common Lisp Open Code Collection)[20] projects.

Common Lisp implementations may use any mix of native code compilation, byte code compilation or interpretation. Common Lisp has been designed to support incremental compilers, file compilers and block compilers. Standard declarations to optimize compilation (such as function inlining or type specialization) are proposed in the language specification. Most Common Lisp implementations compile source code to native

machine code. Some implementations can create (optimized) stand-alone applications. Others compile to interpreted bytecode, which is less efficient than native code, but eases binary-code portability. Some compilers compile Common Lisp code to C code. The misconception that Lisp is a purely interpreted language is most likely because Lisp environments provide an interactive prompt and that code is compiled one-by-one, in an incremental way. With Common Lisp incremental compilation is widely used.

Some Unix-based implementations (CLISP, SBCL) can be used as a scripting language; that is, invoked by the system transparently in the way that a Perl or Unix shell interpreter is.[21]

List of implementations

Commercial implementations

Allegro Common Lisp
: for Microsoft Windows, FreeBSD, Linux, Apple macOS and various UNIX variants. Allegro CL provides an Integrated Development Environment (IDE) (for Windows and Linux) and extensive capabilities for application delivery.

Liquid Common Lisp
: formerly called Lucid Common Lisp. Only maintenance, no new releases.

LispWorks
: for Microsoft Windows, FreeBSD, Linux, Apple macOS, iOS, Android and various UNIX variants. LispWorks provides an Integrated Development Environment (IDE) (available for all platforms, but not for iOS and Android) and extensive capabilities for application delivery.

mocl
: for iOS, Android and macOS.

Open Genera
: for DEC Alpha.

Scieneer Common Lisp
: which is designed for high-performance scientific computing.

Freely redistributable implementations

Armed Bear Common Lisp (ABCL)
: A CL implementation that runs on the Java Virtual Machine.[22] It includes a compiler to Java byte code, and allows access to Java libraries from CL. It was formerly just a component of the Armed Bear J Editor.

CLISP
: A bytecode-compiling implementation, portable and runs on several Unix and Unix-like systems (including macOS), as well as Microsoft Windows and several other systems.

Clozure CL (CCL)
: Originally a free and open-source fork of Macintosh Common Lisp. As that history implies, CCL was written for the Macintosh, but Clozure CL now runs on macOS, FreeBSD, Linux, Solaris and Windows. 32 and 64 bit x86 ports are supported on each platform. Additionally there are Power PC ports for Mac OS and Linux. CCL was previously known as OpenMCL, but that name is no longer used, to avoid confusion with the open source version of Macintosh Common Lisp.

CMUCL
: Originally from Carnegie Mellon University, now maintained as free and open-source software by a group of volunteers. CMUCL uses a fast native-code compiler. It is available on Linux and BSD for Intel x86; Linux for Alpha; macOS for Intel x86 and PowerPC; and Solaris, IRIX, and HP-UX on their native platforms.

Corman Common Lisp

for Microsoft Windows. In January 2015 Corman Lisp has been published under MIT license.[23]

Embeddable Common Lisp (ECL)
ECL includes a bytecode interpreter and compiler. It can also compile Lisp code to machine code via a C compiler. ECL then compiles Lisp code to C, compiles the C code with a C compiler and can then load the resulting machine code. It is also possible to embed ECL in C programs, and C code into Common Lisp programs.

GNU Common Lisp (GCL)
The GNU Project's Lisp compiler. Not yet fully ANSI-compliant, GCL is however the implementation of choice for several large projects including the mathematical tools Maxima, AXIOM and (historically) ACL2. GCL runs on Linux under eleven different architectures, and also under Windows, Solaris, and FreeBSD.

Macintosh Common Lisp (MCL)
Version 5.2 for Apple Macintosh computers with a PowerPC processor running Mac OS X is open source. RMCL (based on MCL 5.2) runs on Intel-based Apple Macintosh computers using the Rosetta binary translator from Apple.

ManKai Common Lisp (MKCL)
A branch of ECL. MKCL emphasises reliability, stability and overall code quality through a heavily reworked, natively multi-threaded, runtime system. On Linux, MKCL features a fully POSIX compliant runtime system.

Movitz
Implements a Lisp environment for x86 computers without relying on any underlying OS.

Poplog
Poplog implements a version of CL, with POP-11, and optionally Prolog, and Standard ML (SML), allowing mixed language programming. For all, the implementation language is POP-11, which is compiled incrementally. It also has an integrated Emacs-like editor that communicates with the compiler.

Steel Bank Common Lisp (SBCL)
A branch from CMUCL. "Broadly speaking, SBCL is distinguished from CMU CL by a greater emphasis on maintainability."[24] SBCL runs on the platforms CMUCL does, except HP/UX; in addition, it runs on Linux for AMD64, PowerPC, SPARC, MIPS, Windows x86[25] and has experimental support for running on Windows AMD64. SBCL does not use an interpreter by default; all expressions are compiled to native code unless the user switches the interpreter on. The SBCL compiler generates fast native code according to a previous version of The Computer Language Benchmarks Game.[26]

Ufasoft Common Lisp
port of CLISP for windows platform with core written in C++.

Other implementations

Austin Kyoto Common Lisp
an evolution of Kyoto Common Lisp by Bill Schelter

Butterfly Common Lisp
an implementation written in Scheme for the BBN Butterfly multi-processor computer[27][28]

CLICC
a Common Lisp to C compiler[29]

CLOE
Common Lisp for PCs by Symbolics

Codemist Common Lisp
used for the commercial version of the computer algebra system Axiom[30][31]

ExperCommon Lisp
an early implementation for the Apple Macintosh by ExperTelligence

Golden Common Lisp
an implementation for the PC by GoldHill Inc.[32][33]

Ibuki Common Lisp
: a commercialized version of Kyoto Common Lisp

Kyoto Common Lisp
: the first Common Lisp compiler that used C as a target language. GCL, ECL and MKCL originate from this Common Lisp implementation.

L
: a small version of Common Lisp for embedded systems developed by IS Robotics, now iRobot[34]

Lisp Machines (from Symbolics, TI[35][36] and Xerox[37])
: provided implementations of Common Lisp in addition to their native Lisp dialect (Lisp Machine Lisp or Interlisp). CLOS was also available. Symbolics provides an enhanced version Common Lisp.[38][39][40]

Procyon Common Lisp
: an implementation for Windows and Mac OS, used by Franz for their Windows port of Allegro CL

Star Sapphire Common LISP
: an implementation for the PC

SubL
: a variant of Common Lisp used for the implementation of the Cyc knowledge-based system[41]

Top Level Common Lisp
: an early implementation for concurrent execution[42]

WCL
: a shared library implementation[43][44]

Vax Common Lisp
: Digital Equipment Corporation's implementation that ran on VAX systems running VMS or ULTRIX

XLISP
: an implementation written by David Betz[45]

Applications

Common Lisp is used to develop research applications (often in Artificial Intelligence), for rapid development of prototypes or for deployed applications.

Common Lisp is used in many commercial applications, including the Yahoo! Store web-commerce site, which originally involved Paul Graham and was later rewritten in C++ and Perl.[46] Other notable examples include:

- ACT-R, a cognitive architecture used in a large number of research projects.
- Authorizer's Assistant,[47][48] a large rule-based system used by American Express, analyzing credit requests.
- Cyc, a long running project a to create a knowledge-based system that provides a huge amount of common sense knowledge
- Gensym G2, a real-time expert system and business rules engine[49]
- Genworks GDL (http://www.genworks.com), based on the open-source Gendl kernel.
- The development environment for the Jak and Daxter video game series, developed by Naughty Dog.
- ITA Software's low fare search engine, used by travel websites such as Orbitz and Kayak.com and airlines such as American Airlines, Continental Airlines and US Airways.
- Mirai, a 3d graphics suite. It was used to animate the face of Gollum in the movie Lord of the Rings: The Two Towers.

- Prototype Verification System (PVS), a mechanized environment for formal specification and verification.
- PWGL is a sophisticated visual programming environment based on Common Lisp, used in Computer assisted composition and sound synthesis.[50]
- Piano, a complete aircraft analysis suite, written in Common Lisp,[51] used by companies like Boeing, Airbus, Northrop Grumman.[52]
- Grammarly, an English-language writing-enhancement platform, has its core grammar engine written in Common Lisp[53]
- The Dynamic Analysis and Replanning Tool (DART), which is said to alone have paid back during the years from 1991 to 1995 for all thirty years of DARPA investments in AI research.
- NASA's (Jet Propulsion Lab's) "Remote Agent", an award-winning[54] Common Lisp[55] program for autopiloting the Deep Space One spaceship.
- SigLab, a Common Lisp platform for signal processing used in missile defense, built by Raytheon[51]
- NASA's Mars Pathfinder Mission Planning System[56]
- SPIKE, a scheduling system for earth or space based observatories and satellites, notably the Hubble Space Telescope.,[57] written in Common Lisp[58]
- Common Lisp has been used for prototyping the garbage collector of Microsoft's .NET Common Language Runtime[59]
- The original version of Reddit, though the developers later switched to Python due to the lack of libraries for Common Lisp, according to an official blog post by Reddit co-founder Steve Huffman.[60]

There also exist open-source applications written in Common Lisp, such as:

- ACL2, a full-featured automated theorem prover for an applicative variant of Common Lisp.
- Axiom, a sophisticated computer algebra system.
- Maxima, a sophisticated computer algebra system, based on Macsyma.
- OpenMusic is an object-oriented visual programming environment based on Common Lisp, used in Computer assisted composition.
- Pgloader (https://pgloader.io/), a data loader for PostgreSQL, which was re-written from Python to Common Lisp.[61]
- Stumpwm, a tiling, keyboard driven X11 Window Manager written entirely in Common Lisp.

See also

- *Common Lisp the Language*
- *On Lisp*
- *Practical Common Lisp*

References

1. Quoted from cover of cited standard. ANSI INCITS 226-1994 [S2008], for sale on standard's document page (https://webstore.ansi.org/Standards/INCITS/INCITS2261994S2008) Archived (https://web.archive.org/web/20200927024609/https://webstore.ansi.org/Standards/INCITS/INCITS2261994S2008) September 27, 2020, at the Wayback Machine.
2. "CLHS: About the Common Lisp HyperSpec (TM)" (http://www.lispworks.com/documentation/HyperSpec/Front/Help.htm#Authorship). *lispworks.com*.

3. "CLHS: Section 1.1.2" (http://www.lispworks.com/documentation/HyperSpec/Body/01_ab.htm). *lispworks.com*.
4. "Common Lisp Implementations: A Survey" (https://web.archive.org/web/20120421181340/http://common-lisp.net/~dlw/LispSurvey.html). Archived from the original (http://common-lisp.net/~dlw/LispSurvey.html) on April 21, 2012. Retrieved December 22, 2007.
5. "Old LISP programs still run in Common Lisp" (http://www.informatimago.com/develop/lisp/com/informatimago/small-cl-pgms/wang.html). Retrieved May 13, 2015.
6. "Roots of "Yu-Shiang Lisp", Mail from Jon L White, 1982" (https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/doc/history/cl.txt). *cmu.edu*.
7. "Mail Index" (http://cl-su-ai.lisp.se/maillist.html). *cl-su-ai.lisp.se*.
8. Knee-jerk Anti-LOOPism and other E-mail Phenomena: Oral, Written, and Electronic Patterns in Computer-Mediated Communication, JoAnne Yates and Wanda J. Orlikowski., 1993 (http://ccs.mit.edu/papers/CCSWP150.html) Archived (https://web.archive.org/web/20120808032049/http://ccs.mit.edu/papers/CCSWP150.html) August 8, 2012, at the Wayback Machine
9. Jr, Steele; L, Guy (August 15, 1982). *An overview of COMMON LISP*. Lfp '82. ACM. pp. 98–107. doi:10.1145/800068.802140 (https://doi.org/10.1145%2F800068.802140). ISBN 9780897910828. S2CID 14517358 (https://api.semanticscholar.org/CorpusID:14517358).
10. Reddy, Abhishek (August 22, 2008). "Features of Common Lisp" (http://random-state.net/features-of-common-lisp.html).
11. "Unicode support" (http://www.cliki.net/Unicode%20Support). *The Common Lisp Wiki*. Retrieved August 21, 2008.
12. Richard P. Gabriel; Kent M. Pitman (June 1988). "Technical Issues of Separation in Function Cells and Value Cells" (http://www.nhplace.com/kent/Papers/Technical-Issues.html). *Lisp and Symbolic Computation*. **1** (1): 81–101. doi:10.1007/bf01806178 (https://doi.org/10.1007%2Fbf01806178). S2CID 26716515 (https://api.semanticscholar.org/CorpusID:26716515).
13. "Common Lisp Hyperspec: Section 3.1.7" (http://www.lispworks.com/documentation/HyperSpec/Body/03_ag.htm).
14. "Common Lisp Hyperspec: Function FLOOR" (http://www.lispworks.com/documentation/HyperSpec/Body/f_floorc.htm).
15. "Common Lisp Hyperspec: Accessor GETHASH" (http://www.lispworks.com/documentation/HyperSpec/Body/f_gethas.htm).
16. "Let Over Lambda" (http://letoverlambda.com/index.cl/guest/chap2.html). *letoverlambda.com*.
17. Peter Seibel (April 7, 2005). *Practical Common Lisp* (http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html). Apress. ISBN 978-1-59059-239-7.
18. "Design Patterns in Dynamic Programming" (http://norvig.com/design-patterns/ppframe.htm). *norvig.com*.
19. Common-Lisp.net (http://common-lisp.net/)
20. Common Lisp Open Code Collection (http://clocc.sourceforge.net/)
21. "32.6. Quickstarting delivery with CLISP" (http://clisp.cons.org/impnotes/quickstart.html#quickstart-unix). *clisp.cons.org*.
22. "Armed Bear Common Lisp" (http://common-lisp.net/project/armedbear/).
23. "Corman Lisp sources are now available" (http://lispblog.xach.com/post/107215169193/corman-lisp-sources-are-now-available).
24. "History and Copyright" (http://sbcl.sourceforge.net/history.html). *Steel Bank Common Lisp*.
25. "Platform Table" (http://www.sbcl.org/platform-table.html). *Steel Bank Common Lisp*.
26. "Which programs are fastest? – Computer Language Benchmarks Game" (https://web.archive.org/web/20130520184339/http://benchmarksgame.alioth.debian.org/u32q/benchmark.php?test=all&lang=all). May 20, 2013. Archived from the original (http://benchmarksgame.alioth.debian.org/u32q/benchmark.php?test=all&lang=all) on May 20, 2013.

27. "Package: lang/lisp/impl/bbn/" (https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/impl/bbn/0.html). *cs.cmu.edu*.
28. "Recent Developments in Butterfly Lisp, 1987, AAAI Proceedings" (http://www.aaai.org/Papers/AAAI/1987/AAAI87-001.pdf) (PDF). *aaai.org*.
29. Burkart, O.; Goerigk, W.; Knutzen, H. (June 22, 1992). "CLICC: A New Approach to the Compilation of Common Lisp Programs to C". CiteSeerX 10.1.1.38.1282 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.1282).
30. "codemist.co.uk" (http://lisp.codemist.co.uk). *lisp.codemist.co.uk*.
31. Axiom, the 30 year horizon, page 43 (http://www.axiom-developer.org/axiom-website/bookvol4.pdf)
32. "Golden Common Lisp Developer" (http://www.goldhill-inc.com/developer.html). *goldhill-inc.com*.
33. Golden Common LISP: A Hands-On Approach, David J. Steele, June 2000 by Addison Wesley Publishing Company
34. Brooks, Rodney A.; al., et (June 22, 1995). "L – A Common Lisp for Embedded Systems". CiteSeerX 10.1.1.2.1953 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.1953).
35. TI Explorer Programming Concepts (http://bitsavers.trailing-edge.com/pdf/ti/explorer/2549830-0001A_PgmgConcepts.pdf)
36. TI Explorer Lisp Reference (http://bitsavers.trailing-edge.com/pdf/ti/explorer/2243201-0001_LispRef.pdf)
37. Medley Lisp Release Notes (http://bitsavers.trailing-edge.com/pdf/xerox/interlisp-d/198809_Medley_1.0/400006_Lisp_Release_Notes_Medley_Release_1.0_Sep88.pdf)
38. "Symbolics Common Lisp Dictionary" (http://bitsavers.trailing-edge.com/pdf/symbolics/software/open_genera/Symbolics_Common_Lisp_Dictionary.pdf) (PDF). *trailing-edge.com*.
39. "Symbolics Common Lisp Language Concepts" (http://bitsavers.trailing-edge.com/pdf/symbolics/software/open_genera/Symbolics_Common_Lisp_Language_Concepts.pdf) (PDF). *trailing-edge.com*.
40. "Symbolics Common Lisp Programming Constructs" (http://bitsavers.trailing-edge.com/pdf/symbolics/software/open_genera/Symbolics_Common_Lisp_Programming_Constructs.pdf) (PDF). *trailing-edge.com*.
41. "SubL Reference – Cycorp" (http://www.cyc.com/documentation/subl-reference/). *cyc.com*.
42. "Top Level Inc. – Software Preservation Group" (http://www.softwarepreservation.org/projects/LISP/toplevel). *softwarepreservation.org*.
43. WCL: Delivering efficient Common Lisp applications under Unix , Proceedings of the 1992 ACM conference on LISP and functional programming (http://dl.acm.org/citation.cfm?id=141560), Pages 260–269
44. "commonlisp.net :: WCL" (https://web.archive.org/web/20160405012115/http://pgc.com/commonlisp/). *pgc.com*. Archived from the original (http://pgc.com/commonlisp/) on April 5, 2016. Retrieved March 25, 2016.
45. "Package: lang/lisp/impl/xlisp/" (https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/impl/xlisp/0.html). *cs.cmu.edu*.
46. "Beating the Averages" (http://www.paulgraham.com/avg.html). *paulgraham.com*.
47. "Authorizer's Assistant" (http://www.aaai.org/Papers/IAAI/1989/IAAI89-031.pdf) (PDF). *aaai.org*.
48. American Express Authorizer's Assistant (http://www.prenhall.com/divisions/bp/app/alter/student/useful/ch9amex.html) Archived (https://web.archive.org/web/20091212141726/http://www.prenhall.com/divisions/bp/app/alter/student/useful/ch9amex.html) December 12, 2009, at the Wayback Machine
49. Real-time Application Development (http://www.gensym.com/platforms/g2-standard/) Archived (https://web.archive.org/web/20160802230335/http://www.gensym.com/platforms/g2-standard/) August 2, 2016, at the Wayback Machine. Gensym. Retrieved August 16, 2016.

50. PWGL – Home (http://www2.siba.fi/PWGL/). . Retrieved July 17, 2013.
51. "Aerospace – Common Lisp" (http://lisp-lang.org/success/aero/). *lisp-lang.org*.
52. [1] (http://www.lissys.demon.co.uk/users.html) Piano Users, retrieved from manufacturer page.
53. [2] (https://tech.grammarly.com/blog/running-lisp-in-production) Grammarly.com, Running Lisp in Production
54. "Remote Agent" (https://ti.arc.nasa.gov/tech/asr/groups/planning-and-scheduling/remote-agent/). *ti.arc.nasa.gov*.
55. http://www.flownet.com/gat/jpl-lisp.html
56. "Franz Inc Customer Applications: NASA" (https://franz.com/success/customer_apps/scheduling/nasa.lhtml). *franz.com*.
57. Spike Planning and Scheduling System (http://www.stsci.edu/resources/software_hardware/spike/). Stsci.edu. Retrieved July 17, 2013.
58. "Franz Inc Customer Applications: Space Telescope Institute" (https://franz.com/success/customer_apps/scheduling/sti.lhtml). *franz.com*.
59. "How It All Started…AKA the Birth of the CLR" (https://blogs.msdn.microsoft.com/patrick_dussud/2006/11/21/how-it-all-startedaka-the-birth-of-the-clr/). *microsoft.com*.
60. Huffman, Steve. "on lisp" (https://web.archive.org/web/20180517155308/https://redditblog.com/2005/12/05/on-lisp/). *Upvoted*. Archived from the original (https://redditblog.com/2005/12/05/on-lisp/) on May 17, 2018. Retrieved May 11, 2019.
61. https://tapoueh.org/blog/2014/05/why-is-pgloader-so-much-faster/

Bibliography

A chronological list of books published (or about to be published) about Common Lisp (the language) or about programming with Common Lisp (especially AI programming).

- Guy L. Steele: *Common Lisp the Language, 1st Edition*, Digital Press, 1984, ISBN 0-932376-41-X
- Rodney Allen Brooks: *Programming in Common Lisp*, John Wiley and Sons Inc, 1985, ISBN 0-471-81888-7
- Richard P. Gabriel: *Performance and Evaluation of Lisp Systems*, The MIT Press, 1985, ISBN 0-262-57193-5, PDF (http://www.dreamsongs.com/Files/Timrep.pdf)
- Robert Wilensky: *Common LISPcraft*, W.W. Norton & Co., 1986, ISBN 0-393-95544-3
- Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott, James R. Meehan: *Artificial Intelligence Programming, 2nd Edition*, Lawrence Erlbaum, 1987, ISBN 0-89859-609-2
- Wendy L. Milner: *Common Lisp: A Tutorial*, Prentice Hall, 1987, ISBN 0-13-152844-0
- Deborah G. Tatar: *A Programmer's Guide to Common Lisp*, Longman Higher Education, 1987, ISBN 0-13-728940-5
- Taiichi Yuasa, Masami Hagiya: *Introduction to Common Lisp*, Elsevier Ltd, 1987, ISBN 0-12-774860-1
- Christian Queinnec, Jerome Chailloux: *Lisp Evolution and Standardization*, Ios Pr Inc., 1988, ISBN 90-5199-008-1
- Taiichi Yuasa, Richard Weyhrauch, Yasuko Kitajima: *Common Lisp Drill*, Academic Press Inc, 1988, ISBN 0-12-774861-X
- Wade L. Hennessey: *Common Lisp*, McGraw-Hill Inc., 1989, ISBN 0-07-028177-7
- Tony Hasemer, John Dominque: *Common Lisp Programming for Artificial Intelligence*, Addison-Wesley Educational Publishers Inc, 1989, ISBN 0-201-17579-7
- Sonya E. Keene: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, 1989, ISBN 0-201-17589-4
- David Jay Steele: *Golden Common Lisp: A Hands-On Approach*, Addison Wesley, 1989, ISBN 0-201-41653-0

- David S. Touretzky: *Common Lisp: A Gentle Introduction to Symbolic Computation*, Benjamin-Cummings, 1989, ISBN 0-8053-0492-4. Web/PDF (https://www.cs.cmu.edu/~dst/LispBook/) Dover reprint (2013) ISBN 978-0486498201
- Christopher K. Riesbeck, Roger C. Schank: *Inside Case-Based Reasoning*, Lawrence Erlbaum, 1989, ISBN 0-89859-767-6
- Patrick Winston, Berthold Horn: *Lisp, 3rd Edition*, Addison-Wesley, 1989, ISBN 0-201-08319-1, Web (http://people.csail.mit.edu/phw/Books/LISPBACK.HTML)
- Gerard Gazdar, Chris Mellish: *Natural Language Processing in LISP: An Introduction to Computational Linguistics*, Addison-Wesley Longman Publishing Co., 1990, ISBN 0-201-17825-7
- Patrick R. Harrison: *Common Lisp and Artificial Intelligence*, Prentice Hall PTR, 1990, ISBN 0-13-155243-0
- Timothy Koschmann: *The Common Lisp Companion*, John Wiley & Sons, 1990, ISBN 0-471-50308-8
- W. Richard Stark: *LISP, Lore, and Logic*, Springer Verlag New York Inc., 1990, ISBN 978-0-387-97072-1, PDF (https://link.springer.com/book/10.1007/978-1-4613-8931-6)
- Molly M. Miller, Eric Benson: *Lisp Style & Design*, Digital Press, 1990, ISBN 1-55558-044-0
- Guy L. Steele: *Common Lisp the Language, 2nd Edition*, Digital Press, 1990, ISBN 1-55558-041-6, Web (https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html)
- Robin Jones, Clive Maynard, Ian Stewart: *The Art of Lisp Programming*, Springer Verlag New York Inc., 1990, ISBN 978-3-540-19568-9, PDF (https://link.springer.com/book/10.1007/978-1-4471-1719-3)
- Steven L. Tanimoto: *The Elements of Artificial Intelligence Using Common Lisp*, Computer Science Press, 1990, ISBN 0-7167-8230-8
- Peter Lee: *Topics in Advanced Language Implementation*, The MIT Press, 1991, ISBN 0-262-12151-4
- John H. Riley: *A Common Lisp Workbook*, Prentice Hall, 1991, ISBN 0-13-155797-1
- Peter Norvig: *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann, 1991, ISBN 1-55860-191-0, Web (http://norvig.com/paip.html)
- Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow: *The Art of the Metaobject Protocol*, The MIT Press, 1991, ISBN 0-262-61074-4
- Jo A. Lawless, Molly M. Miller: *Understanding CLOS: The Common Lisp Object System*, Digital Press, 1991, ISBN 0-13-717232-X
- Mark Watson: *Common Lisp Modules: Artificial Intelligence in the Era of Neural Networks and Chaos Theory*, Springer Verlag New York Inc., 1991, ISBN 0-387-97614-0, PDF (https://link.springer.com/book/10.1007/978-1-4612-3186-8)
- James L. Noyes: *Artificial Intelligence with Common Lisp: Fundamentals of Symbolic and Numeric Processing*, Jones & Bartlett Pub, 1992, ISBN 0-669-19473-5
- Stuart C. Shapiro: *COMMON LISP: An Interactive Approach*, Computer Science Press, 1992, ISBN 0-7167-8218-9, Web/PDF (https://archive.is/19981205110924/http://www.cse.buffalo.edu/pub/WWW/faculty/shapiro/Commonlisp/)
- Kenneth D. Forbus, Johan de Kleer: *Building Problem Solvers*, The MIT Press, 1993, ISBN 0-262-06157-0
- Andreas Paepcke: *Object-Oriented Programming: The CLOS Perspective*, The MIT Press, 1993, ISBN 0-262-16136-2
- Paul Graham: *On Lisp*, Prentice Hall, 1993, ISBN 0-13-030552-9, Web/PDF (http://www.paulgraham.com/onlisp.html)
- Paul Graham: *ANSI Common Lisp*, Prentice Hall, 1995, ISBN 0-13-370875-6
- Otto Mayer: *Programmieren in Common Lisp*, German, Spektrum Akademischer Verlag, 1995, ISBN 3-86025-710-2
- Stephen Slade: *Object-Oriented Common Lisp*, Prentice Hall, 1997, ISBN 0-13-605940-6
- Richard P. Gabriel: *Patterns of Software: Tales from the Software Community*, Oxford University Press, 1998, ISBN 0-19-512123-6, PDF (http://www.dreamsongs.com/Files/PatternsOfSoftware.pdf)

- Taiichi Yuasa, Hiroshi G. Okuno: *Advanced Lisp Technology*, CRC, 2002, ISBN 0-415-29819-9
- David B. Lamkins: *Successful Lisp: How to Understand and Use Common Lisp*, bookfix.com, 2004. ISBN 3-937526-00-5, Web (https://web.archive.org/web/20100106112300/http://www.psg.com/~dlamkins/sl/contents.html)
- Peter Seibel: *Practical Common Lisp*, Apress, 2005. ISBN 1-59059-239-5, Web (http://www.gigamonkeys.com/book/)
- Doug Hoyte: *Let Over Lambda*, Lulu.com, 2008, ISBN 1-4357-1275-7, Web (http://letoverlambda.com/)
- George F. Luger, William A. Stubblefield: *AI Algorithms, Data Structures, and Idioms in Prolog, Lisp and Java*, Addison Wesley, 2008, ISBN 0-13-607047-7, PDF (https://web.archive.org/web/20110707202029/http://wps.aw.com/wps/media/objects/5771/5909832/PDF/Luger_0136070477_1.pdf)
- Conrad Barski: *Land of Lisp: Learn to program in Lisp, one game at a time!*, No Starch Press, 2010, ISBN 1-59327-200-6, Web (http://www.lisperati.com/landoflisp/)
- Pavel Penev: *Lisp Web Tales*, Leanpub, 2013, Web (https://leanpub.com/lispwebtales)
- Edmund Weitz: *Common Lisp Recipes*, Apress, 2015, ISBN 978-1-484211-77-9, Web (http://www.apress.com/9781484211779)
- Patrick M. Krusenotto: *Funktionale Programmierung und Metaprogrammierung, Interaktiv in Common Lisp*, Springer Fachmedien Wiesbaden 2016, ISBN 978-3-658-13743-4, Web (https://link.springer.com/book/10.1007%2F978-3-658-13744-1)

External links

- Quicklisp (https://www.quicklisp.org/beta/) - A very popular and high quality library manager for Common Lisp
- The Awesome CL (https://github.com/CodyReichert/awesome-cl) list, a curated list of Common Lisp frameworks and libraries.
- The Common Lisp Cookbook (https://lispcookbook.github.io/cl-cookbook/), a collaborative project.
- The CLiki (http://www.cliki.net/), a Wiki for free and open-source Common Lisp systems running on Unix-like systems.
- One of the main repositories for free Common Lisp for software is Common-Lisp.net (http://www.common-lisp.net/).
- lisp-lang.org (http://lisp-lang.org/) has documentation and a showcase of success stories.
- An overview of the history of Common Lisp: "History" (http://www.lispworks.com/documentation/HyperSpec/Body/01_ab.htm). *Common Lisp HyperSpec*.
- Common Lisp Quick Reference (http://clqr.boundp.org) – a compact overview of the Common Lisp standard language.
- Planet Lisp (http://planet.lisp.org) Articles about Common Lisp.
- Quickdocs (http://quickdocs.org/) summarizes documentation and dependency information for many Quicklisp projects.