LISP PROGRAMMING

with Examples in SBCL

Edgar Granados
Eduardo Torres

Programming Languages

Spring, 2018

Contents

1 Introduction

LISP is the second oldest high-level programming language (FORTRAN being the first) which design began in 1958 by John McCarthy. The main idea behind the development of LISP was to compute with symbolic expressions rather than numbers, representing information using a list structure.

In order to understand some important features of modern LISP implementations, is important to understand the context in which the language was developed and what happened after the initial releases. The following subsections are subtracted from [1] and [2] unless otherwise noted.

1.1 LISP Prehistory

At the 1956 Dartmouth Summer Research Project on Artificial Intelligence, John McCarthy began developing the idea of an algebraic list-processing language, derived mainly from the FORTRAN idea of writing programs algebraically. This new language was to be written for the IBM 704 (which greatly influenced the language) since IBM was establishing a research facility at MIT and the company was developing a Marvin Minsky's idea of a program to prove theorems in plane geometry.

The design of the language was greatly influenced by the list processing *language* for FORTRAN (it was an extension to FORTRAN rather than a new language). Although it was useful for testing some ideas, some of them were never implemented, e.g. conditional expressions and recursive use of subroutines.

1.2 LISP Implementation

In Fall 1958 was the start of the implementation of LISP at MIT. Although two years had passed from the beginning of LISP, many decisions (e.g. garbage collection) were made during the implementation rather than during design. It has to be noted that many design decisions were made after the fact that many characters were not allowed, e.g. square brackets *[]*.

As LISP was intended to follow the *usual* mathematical laws, there was some debate on weather to include side effects or not (the so called impure or pure languages). The decision to include side effects was made after the fact that these make programs more efficient (this was a major concern since computer time was expensive). In spite of this, there are some *pure* LISP implementations (refer to [3] for more information).

Implementation of recursion made the description of computable functions neater compared to Turing Machines (The universal LISP function *eval* 2.3 is briefer and more comprehensible). In fact, LISP was described both as a programming language and a formalism for recursive theory [4].

As the focus of the language was list processing (hence the name: *LISt Processor*), two of the main functions to achieve this are known as *car* and *cdr* (see section 2.10.4 for details on its usage).

The names might seem weird at first but this is just some of the heritage from the IBM 704 which had a 36-bit word with two 15 bit parts: *address* and *decrement*. This (and some instructions of the architecture) made easy to implement the "Contents of the Address part of the Register number" and the "Contents of the Decrement part of the Register number".

One of the developers, S. R. Russell (who also invented one of the first video games) noticed that implementing the *eval* function meant having an interpreter. Therefore, LISP had an interpreter before having a compiler. This also speeds up the development of testing functions not longer meant *hand-compiling* them.

The first programmers manual was released in 1960 and in 1962 the LISP 1.5 programmers manual and the first compiler was released. LISP became popular on AI research, which meant that many groups implemented their own version of LISP, which eventually created non-compatible LISP programs.

1.3 Common Lisp

Since the release of LISP 1.5, there had been two major versions: BBN-Lisp and MacLisp, developed by Bolt, Beranek and Newman Inc. and the MAC project at MIT respectively. Each version was developed for different computers and had architecture-specific features. The BBN Lisp later transformed into Interlisp, which had better development tools and was better documented in contrast with MacLisp which the main feature was efficiency and implemented state of the art features e.g. arbitrary-precision arithmetic. As new and cheaper computers were available, there was a need to implement versions of LISP. The Carnegie-Mellon University SPICE (Scientific Personal Integrated Computing Environment) group began the development of SPICE Lisp trying to make it moderately portable, it later became CMUCL (Carnegie Mellon University Common Lisp).

As there was a special need for highly efficient LISP implementations, a project implementing a *LISP Machine* began at MIT which resulted in two MIT spin-offs: Lisp Machines, Inc, and Symbolics. These machines had LISP specific hardware features. It is important to highlight that Richard Stallman developed key features for these machines while working on *emacs* (which has parts developed in LISP) and he has suggested that the dispute between both companies led to the GNU OS [5].

In 1981 work began on what would be known as *Common Lisp*. The main objective was to integrate the best features each major LISP implementation had, thus a new specification was needed. *Retro-compatibility* was one of the main concerns as many systems were already running some LISP dialect. CL was to serve a *common* dialect for upward compatibility plus making portability easier as more diverse computers were being available. At this point, some features were decided not to be implemented: graphics, multiprocessing, and object-oriented programming.

The first manual of Common Lisp was published in 1984. The next year, there was a meeting in Boston to discuss the future of Common Lisp and one of the results was the beginning of the

ANSI standardization process of Common Lisp the following year. Although there were no major modifications to what was already published, there were some important additions such as *CLOS*, the Common Lisp Object System. After the process was completed, in 1994 the ANSI Common Lisp standard INCITS 226-1994 was finally released with the last revision being in 2008.

1.4 LISP after ANSI

The standardization of CL made it easier to implement compatible LISP interpreters and compilers. Although LISP is not one of the most popular languages today (according to [6], is ranked in position 21), LISP is still widely used in research fields, specifically in AI. Table 1 lists the current dialects of CL. The fact that both commercial and free software distributions exists reflects the importance and usage of CL in specific fields.

Table 1: Current Dialects of CL [7]

Name	a.k.a	License	Platforms
AllegroCL		Comercial	GNU/Linux, Unix, Mac, Windows
Armed Bear CL	ABCL	GPL	GNU/Linux, Unix, Mac, Windows
CMU CL	CMUCL	Open	GNU/Linux, Unix, Mac
Clozure CL	OpenMCL	LLGPL	GNU/Linux, Unix, Mac
Corman CL		Comercial	Windows
Embedded CL	ECL	LGPL	GNU/Linux, Unix, Mac, Windows
GNU CL	GCL	LGCP, GPL	GNU/Linux, Unix, Mac, Windows
GNU CLISP	CLISP	GPL	GNU/Linux, Unix, Mac, Windows
LispWorks		Commercial	GNU/Linux, Unix, Mac, Windows
Scieneer CL	SCL	Commercial	GNU/Linux, Unix
Steel Banck CL	SBCL	BSD/Public	GNU/Linux, Unix, Mac

2 LISP - Basics

2.1 Introduction

The objectives of Common Lisp are basically achieved (as many programming languages) commonality, portability, consistency, expressiveness, compatibility, efficiency, and stability.
As a programming language, it is characterized by the following ideas:

- Computing with symbolic expressions rather than numbers.
- Representation of symbolic expressions and other information by list structure in the memory of a computer.
- Representation of information in external media mostly by multi-level lists and sometimes by S-expressions.
- A small set of selector and constructor operations expressed as functions.
- Composition of functions as a tool for forming more complex functions.
- The use of conditional expressions for getting branching into function definitions, the recursive use of conditional expressions as a sufficient tool for building computable functions.
- The use of λ-expressions for naming functions.
- The representation of LISP programs as LISP data.
- The conditional expression interpretation of Boolean connectives.
- The LISP function eval that serves both as a formal definition of the language and as an interpreter.
- And garbage collection as a means of handling the erasure problem.

Most of these ideas are going to be explained with details throughout these document.

In order to better explain all the complexities of the programming language, lets first begin with a brief-explanatory introduction to the language's functionality.

Basically, almost everything can be represented as a list, and it is denoted with round parenthesis.

A list may look like this:

```
1 (+ 1 4)
```

We will read this list as a function. It can be explained as interpreting the fist element in this list as an operand, and the arguments of that operand as the rest of the elements in the list, then each argument is evaluated and finally it is perform the operation defined by the operand, obtaining a result, i.e. first is read + and then is read 1 and 4 and you know that 1 is a number and values 1, 4 is a number that values 4 and the operation + means that you have to sum those numbers giving the result 5.

There are also different ways to do things in LISP; a special form is defined (just for the introduction purposes) as a different way of performing functionality without following the previous behavior, for example:

```
1 (setq x 3)
```

will assign the value of 3 to x, which is not a normal behavior of a function; here its arguments are not evaluated first one is used to bind a value to a variable,

```
1 (format t "hello-world")
```

will return NIL (the false/Null value in LISP) but as a side effect, it will print the string "hello-world" which is not a function behavior,

```
1 (if x (format t "yes") (format t "no"))
```

will first evaluate x and depending on the value of x it will evaluate the first or the second argument.

As we can see, in LISP there are more than just functions, it means that it is not a pure language in terms of the functional paradigm, but certainly, it opens a world of possibilities for the programmer.

Now, with this in mind, we will enter formally to the description of the language.

2.2 Syntax and Semantics

Here are shown the principal EBNF rules of LISP

⟨*s_expression*⟩ ::= ⟨*atomic_symbol*⟩
| '(' ⟨*s_expresion*⟩ '.' ⟨*s_expresion*⟩ ')'
| ⟨*list*⟩

⟨*list*⟩ ::= '(' ⟨*s_expresion*⟩ '*' ')'

⟨*atomic_symbol*⟩ ::= ⟨*letter_symbols*⟩ ⟨*atom_part*⟩

⟨*atom_part*⟩ ::= ⟨*empty*⟩
| ⟨*letter_symbols*⟩ ⟨*atom_part*⟩
| ⟨*number*⟩ ⟨*atom_part*⟩

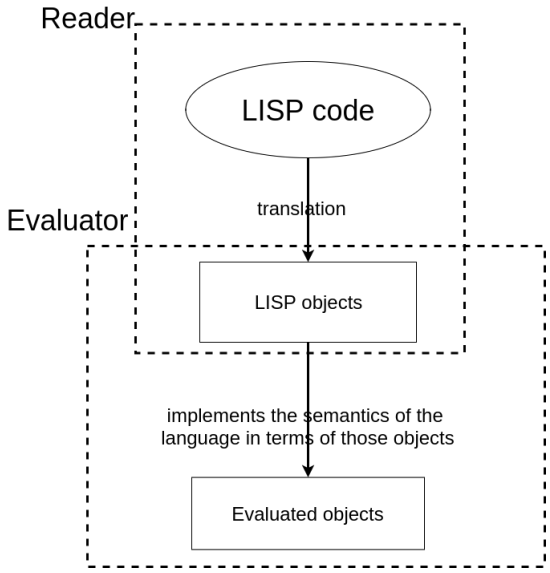
⟨*letter_symbols*⟩ ::= ‘a’ | ‘b’ | ‘. . .’ | ‘z’ | ‘*’ | ‘+’ | ‘. . .’ | ‘?’

⟨*number*⟩ ::= ‘1’ | ‘2’ | ‘. . .’ | ‘9’

The basic elements of LISP are s-expressions, lists, and atoms. Lists are delimited by parentheses and can contain any number of whitespace-separated elements. Atoms are everything else. The elements of lists are themselves s-expressions (in other words, atoms or nested lists).

2.3 Read–Eval–Print Loop

The REPL is the *core* of LISP. It is defined two steps for the language processor, one that translates text into LISP objects and another that implements the semantics of the language in terms of those objects. The first step is called the Reader, and the second is called the Evaluator.



Each step defines one level of syntax. The reader defines how strings of characters can be translated into LISP objects called s-expressions. The evaluator then defines a syntax of Lisp forms that can be built out of s-expressions.

This split of the Reader and Evaluator allows the usage of s-expressions as an externalizable data format for data other than source code, using READ to read it and PRINT to print it.

Therefore, the syntax can be separated into two pieces: the syntax of s-expressions understood by the Reader and the syntax of LISP forms understood by the Evaluator.

In this document, we are going to refer a couple of times to these steps and the focus of action whether reading or evaluating.

There is a function defined in LISP that makes the exact thing that the Evaluator: it evaluates a function.

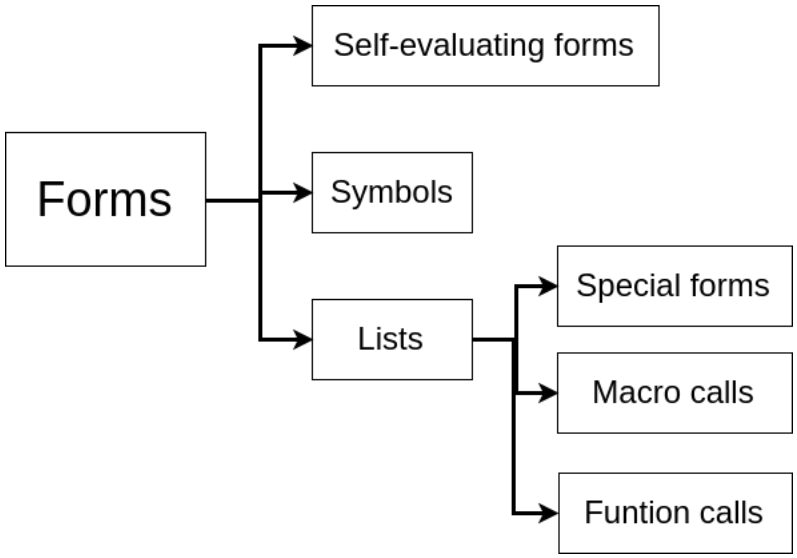
```
1 (eval e a)
```

Eval computes the value of a LISP expression e using the list of argument values to variables a.

2.4 S-expression Forms

The standard unit of interaction with a LISP implementation is the S-expression Form, which is simply a data object meant to be evaluated as a program to produce one or more values (which are also data objects).

Forms may be divided into three categories: self-evaluating forms, such as numbers; symbols, which stand for variables; and lists. The lists, in turn, may be divided into three categories: special forms, macro calls, and function calls. As the following chart expose:



We will describe each of them among other subjects that arise on the way.

2.5 Self-evaluating Forms

All numbers, characters, and strings are self-evaluating forms. When such an object is evaluated, that object is returned as the value of the form. The empty list (), which is also the false value NIL, is also a self-evaluating form: the value of NIL is NIL.

2.6 Variables

Symbols are used as names of variables in LISP programs. When a symbol is evaluated as a form, the value of the variable it names is produced.

In LISP variables are named places that can hold a value. It is dynamically typed in the sense that type errors are detected dynamically.

On the other hand, is a strongly typed language in the sense that all type errors will be detected.

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

f

In order to better understand the differences between the different kind of variables we will talk first about the Scope and Extent in Common Lisp and the relation with variable binding.

2.6.1 Scope and Extent

Scope refers to the delimitation of the interpretation of a particular name in a particular way within a block of code.
Extent refers to the interval of time during which references may occur.

In Common Lisp there are a few kinds of scope and extent:

- Lexical scope. Here references to the established entity can occur only within certain program portions that are lexically (that is, textually) contained within the establishing construct. Typically the construct will have a part designated the body, and the scope of all entities established will be (or include) the body. Example: the names of parameters to a function normally are lexically scoped.
- Indefinite scope. References may occur anywhere, in any program. Dynamic extent. References may occur at any time in the interval between the establishment of the entity and the explicit disestablishment of the entity. As a rule, the entity is disestablished when execution of the establishing construct completes or is otherwise terminated. Therefore entities with dynamic extent obey a stack-like discipline, paralleling the nested executions of their establishing constructs. Example: the with-open-file construct opens a connection to a file and creates a stream object to represent the connection. The stream object has indefinite extent, but the connection to the open file has dynamic extent: when control exits the with-open-file construct, either normally or abnormally, the stream is automatically closed.
- Indefinite extent. The entity continues to exist as long as the possibility of reference remains. (An implementation is free to destroy the entity if it can prove that reference to it is no longer possible. Garbage collection strategies implicitly employ such proofs.)

2.6.2 Binding

When an association of certain properties to a symbolic name, then a binding of a variable has been performed. There are two kinds of variables in Common Lisp, called lexical (or static) variables and special (or dynamic) variables. At any given time either or both kinds of the variable with the same name may have a current value. Which of the two kinds of the variable is referred to when a symbol is evaluated depends on the context of the evaluation.

A dynamically bound (special) variable can be referred to at any time from the time the binding is made until the time evaluation of the construct that binds the variable terminates. Therefore lexical binding of variables imposes a spatial limitation on occurrences of references (but no temporal limitation, for the binding, continues to exist as long as the possibility of reference remains). Conversely, dynamic binding of variables imposes a temporal limitation on occurrences of references (but no spatial limitation).

The value a special variable has when there are currently no bindings of that variable is called the global value of the (special) variable. A global value can be given to a variable only by assignment because a value given by binding is by definition not global.

It is possible for a special variable to have no value at all, in which case it is said to be unbound. By default, every global variable is unbound unless and until explicitly assigned a value, except for those global variables defined in this book or by the implementation already to have values when the Lisp system is first started. It is also possible to establish a binding of a special variable and then cause that binding to be valueless by using the function makunbound. In this situation the variable is also said to be "unbound," although this is a misnomer; precisely speaking, it is bound but valueless. It is an error to refer to a variable that is unbound.

The general rule is that if the symbol occurs textually within a program code that creates a binding for a variable of the same name, then the reference is to the variable specified by the binding; if no such program construct textually contains the reference, then it is taken to refer to the special variable of that name.

As a simple example, consider this program:

```
1 (defun foo (x y) (* x (+ y 1)))
```

there is a single name, x, used to refer to the first parameter of the procedure whenever it is invoked; however, a new binding is established on every invocation. A binding is a particular parameter instance. The value of a reference to the name x depends not only on the scope within which it occurs (the one in the body of foo in the example occurs in the scope of the function definition's parameters) but also on the particular binding or instance involved.

Each time a function is called, Lisp creates new bindings to hold the arguments passed by the function's caller. A binding is the runtime manifestation of a variable. A single variable can have many different bindings during a run of the program. A single variable can even have multiple bindings at the same time; parameters to a recursive function, for example, are rebound for each call to the function.

As with all Common Lisp variables, function parameters hold object references. Thus, you can assign a new value to a function parameter within the body of the function, and it will not affect the bindings created for another call to the same function. But if the object passed to a function is mutable and you change it in the function, the changes will be visible to the caller since both the caller and the caller will be referencing the same object.

2.6.3 Rules of scope and extent

The important scope and extent rules in Common Lisp follow:

- Variable bindings normally have lexical scope and indefinite extent.
- Variable bindings for which there is a dynamic-extent declaration also have lexical scope and indefinite extent, but objects that are the values of such bindings may have dynamic extent. (The declaration is the programmer's guarantee that the program will behave correctly even if certain of the data objects have only dynamic extent rather than the usual indefinite extent.)
- Bindings of variable names to symbol macros by symbol-macrolet have lexical scope and indefinite extent.
- Variable bindings that are declared to be special have dynamic scope (indefinite scope and dynamic extent).
- Bindings of function names established, for example, by flet and labels have lexical scope and indefinite extent.
- Bindings of function names for which there is a dynamic-extent declaration also have lexical scope and indefinite extent, but function objects that are the values of such bindings may have dynamic extent.
- Bindings of function names to macros as established by macrolet have lexical scope and indefinite extent.
- Named constants such as nil and pi to have an indefinite scope and indefinite extent.

2.7 Special Forms

Another form that introduces new variables is the LET special operator. The skeleton of a LET form looks like this:

```
1 (let (variable*)
2   body-form*)
```

where each variable is a variable initialization form. Each initialization form is either a list containing a variable name and an initial value form or–as a shorthand for initializing the variable to NIL. The following LET form, for example, binds the three variables x, y, and z with initial values 10, 20, and NIL:

```
1 (let ((x 10) (y 20) z)
2   ...)
```

When the LET form is evaluated, all the initial value forms are first evaluated. The new bindings are created and initialized to the appropriate initial values before the body forms are executed. Within the body of the LET, the variable names refer to the newly created bindings. After the LET, the names refer to whatever, if anything, they referred to before the LET.

The value of the last expression in the body is returned as the value of the LET expression.

The scope of function parameters and LET variables is delimited by the form that introduces the variable. The LET form is called the binding form.

2.8 Macros

Macros are a way of rewriting pieces of code. Macros are used to make programs efficient since function calls are avoided. When a macro call is encounter, LISP does the following things:

1. Calls the *macroexpansion* function with the unevaluated arguments.
2. Evaluates the expanded macro in place of the original macro call.

It is important to note that macros must be defined before they are called because the macros are expanded at compile time.

Example 2 shows how a simple macro named *when-macro* is defined, notice the back quote (‘) in the second line and the commas. In LISP, most of what may appear to be a function are in fact a macro. For example, *cond* is actually a bunch of nested *IFs* as seen in example **??** when used the function *macroexpand*. There is no restriction for macros using other macros, or even for macros to produce other macros but it can easily become complicated to deal with nested backquotes. In CL there are macros to perform iteration control similar to other languages: loop and do list.

Example 1: Macro definition example

```
1 (defmacro when-macro (test body)
2         `(if ,test
3              ,body
4               nil ))
```

Example 2: Macroexpand example

```
1 * (macroexpand
2     '(cond ((eql '+ operator) (+ var arg1 arg2))
3            ((eql '- operator) (- var arg1 arg2))
4            ((eql '* operator) (* var arg1 arg2))
5            (t (/ var arg1 arg2))
6            ))
7 (IF (EQL '+ OPERATOR)
8     (+ VAR ARG1 ARG2)
9     (IF (EQL '- OPERATOR)
10        (- VAR ARG1 ARG2)
11        (IF (EQL '* OPERATOR)
12            (* VAR ARG1 ARG2)
13            (THE T (/ VAR ARG1 ARG2)))))
14 T
```

2.9 Funtions

Function Calls

If a list is to be evaluated as a form and the first element is not a symbol that names a special form or macro, then the list is assumed to be a function call. The first element of the list is taken to name a function. Any and all remaining elements of the list are forms to be evaluated; one value is obtained from each form, and these values become the arguments to the function.

The function is then applied to the arguments. The functional computation normally produces a value, but it may instead call for a non-local exit. A function that does return may produce no value or several values. If and when the function returns, whatever value it returns become the values of the function-call form.

For example, consider the evaluation of the form (+ 3 (* 4 5)). The symbol + names the addition function, not a special form or macro. Therefore the two forms 3 and (* 4 5) are evaluated to produce arguments. The form 3 evaluates to 3, and the form (* 4 5) is a function call (to the multiplication function). Therefore the forms 4 and 5 are evaluated, producing arguments 4 and 5 for the multiplication. The multiplication function calculates the number 20 and returns it. The values 3 and 20 are then given as arguments to the addition function, which calculates and returns the number 23. Therefore we say (+ 3 (* 4 5)) => 23.

There are two ways to indicate a function to be used in a function-call form. One is to use a symbol that names the function. This use of symbols to name functions is completely independent of their use in naming special and lexical variables. The other way is to use a lambda-expression, which is a list whose first element is the symbol lambda.

A lambda-expression is a list with the following syntax:

```
1 (lambda lambda-list . body)
```

The first element must be the symbol lambda. The second element must be a list. It is called the lambda-list and specifies names for the parameters of the function. When the function denoted by the lambda-expression is applied to arguments, the arguments are matched with the parameters specified by the lambda-list. The body may then refer to the arguments by using the parameter names. The body consists of any number of forms (possibly zero). These forms are evaluated in sequence, and the results of the last form only are returned as the results of the application (the value nil is returned if there are zero forms in the body). The complete syntax of a lambda-expression is:

```
1 (lambda ({var}*
2         [&optional {var | (var [initform [svar]])}*]
3         [&rest var]
4         [&key {var | ({var | (keyword var)} [initform [svar]])}*
5               [&allow-other-keys]]
6         [&aux {var | (var [initform])}*)]
7   [[{declaration}* | documentation-string]]
```

```
8     {form}*)
```

Each element of a lambda-list is either a parameter specifier or a lambda-list keyword; lambda-list keywords begin with .

A lambda-list has five parts, any or all of which may be empty:

Specifiers for the required parameters. These are all the parameter specifiers up to the first lambda-list keyword; if there is no such lambda-list keyword, then all the specifiers are for required parameters. Specifiers for optional parameters. If the lambda-list keyword optional is present, the optional parameter specifiers are those following the lambda-list keyword optional up to the next lambda-list keyword or the end of the list. A specifier for a rest parameter. The lambda-list keyword rest, if present, must be followed by a single rest parameter specifier, which in turn must be followed by another lambda-list keyword or the end of the lambda-list. Specifiers for keyword parameters. If the lambda-list keyword key is present, all specifiers up to the next lambda-list keyword or the end of the list are keyword parameter specifiers. The keyword parameter specifiers may optionally be followed by the lambda-list keyword allow-other-keys. Specifiers for aux variables. These are not really parameters. If the lambda-list keyword key is present, all specifiers after it are auxiliary variable specifiers.

Functions are defined using the DEFUN macro. The basic skeleton of a DEFUN looks like this:

```
1 (defun name (parameter*)
2   "Optional documentation string."
3   body-form*)
```

Observe the next funtion-call:

```
1 (defun hello-world () (format t "hello, world"))
```

Let's analyze the each part of these funtion. Its name is hello-world, its parameter list is empty so it takes no arguments, it has no documentation string, and its body consists of one expression.

```
1 (format t "hello, world")
```

Parameter list The basic purpose of a parameter list is to declare the variables that will receive the arguments passed to the function. When a parameter list is a simple list of variable names, the parameters are called required parameters. When a function is called, it must be supplied with one argument for every required parameter. Each parameter is bound to the corresponding argument. If a function is called with too few or too many arguments, Lisp will signal an error.

2.10 Data Types

Common Lisp provides a variety of types of data objects. It is important to note that in LISP it is data objects that are typed, not variables. Any variable can have any LISP object as its value.

A data type is a set of LISP objects. Many LISP objects belong to more than one such set, and so it doesn't always make sense to ask what is the type of an object; instead, one usually asks

only whether an object belongs to a given type.

Example with type-of and typep (The predicate typep may be used to ask whether an object belongs to a given type, and the function type-of returns a type to which a given object belongs).

The following categories of Common LISP objects are of particular interest: numbers, characters, symbols, lists, arrays, structures, and functions.

2.10.1 Numbers

Numbers are provided in various forms and representations. Common Lisp provides a true integer data type: any integer, positive or negative, has in principle a representation as a Common LISP data object, subject only to total memory limitations (rather than machine word width). A true rational data type is provided: the quotient of two integers, if not an integer, is a ratio. Floating-point numbers of various ranges and precisions are also provided, as well as Complex numbers.

Several kinds of numbers are defined in Common LISP. They are divided into integers; ratios; floating-point numbers, with names provided for up to four different floating-point representations; and complex numbers.

Number data type	Description	Observations
Integers	The integer data type is intended to represent mathematical integers. Common LISP imposes no limit on the magnitude of an integer; storage is automatically allocated as necessary to represent large integers.	• 0 Zero • -0 This always means the same as 0 • 1024. two to the tenth power • 15511210043330985984000000. 25!
Ratios	A ratio is a number representing the mathematical rational numbers $\mathbb{Q}$. Its representation as an integer if its value is integral, and otherwise as the ratio of two integers, the numerator and denominator, whose gcd = 1, and of which the denominator is positive. A ratio is notated with / as a separator.	• 2/3 This is in canonical form • 4/6 A non-canonical form for the same number • -30517578125/32768 This is $(-5/2)^{15}$ • 10/5 The canonical form for this is 2
Floating-point numbers	There are short, long and intermediate floating-point numbers each one of them depending on the size of the representation of fixed precision provided bu an implementation. The intermediate between short and long formats are divided by single and double (types single-float and double-float).	• 0.0 Fp zero in default format • 0E0 Also fp zero in default format • 0.0s0 A fp zero in short format • 3.141592653589793238 4d0 A double-format approximation to π • 6.02E+23 Avogadro's number, in default format • 602E+21 Also Avogadro's number, in default format • 3.010299957f-1 in single format • -0.000000001s9 in short format
Complex numbers	Complex numbers (type complex) are represented by $c = ai + b$ where $i^2 = -1$ and a, b are non-complex numbers (integer, ratio, or floating-point number), requiring both parts of the same type. Complex numbers may be notated by writing the characters #C followed by a list of the real and imaginary parts.	• #C(3.0s1 2.0s-1) Real and imaginary parts are short format • #C(5 -3) A Gaussian integer • #C(5/3 7.0) Will be converted internally to #C(1.66666 7.0) • #C(0 1) The imaginary unit, i

2.10.2 Characters

Characters are represented as data objects of type character.

A character object can be notated by writing \followed by the character itself. For example, \g means the character object for a lowercase g. This works well enough for printing characters. Non-printing characters have names, and can be notated by writing \and then the name; for example, \Space (or \SPACE or \space or \sPaCE) means the space character. Other semistandard names (that implementations must use if the character set has the appropriate characters) are Tab, Page, Rubout, Linefeed, Return, and Backspace.

2.10.3 Strings

Strings are a composite data type, namely, a one-dimensional array of characters. Strings also have their own literal syntax and a library of functions for performing string-specific operations.

Literal strings are written enclosed in double quotes. It can be include any character supported by the character set in a literal string except double quote (") and backslash (\). They can be included if you escape them with a backslash. In fact, backslash always escapes the next character, whatever it is, though this it is not necessary for any character except for " and itself.

Table 2: Literal Strings

Literal	Contents	Comment
"foobar"	foobar	Plain string.
"foo\"bar"	foo"bar	The backslash escapes quote.
"foo\\bar"	foo\bar	The first backslash escapes second backslash.
"f̈oobar\""	"foobar"	The backslashes escape quotes.
"foo	foobar	The backslash "escapes" b

The type string is therefore a subtype of the type vector.

A string can be written as the sequence of characters contained in the string, preceded and followed by a " (double quote) character. Any " or character in the sequence must additionally have a character before it.

For example:

```
"Foo"                          ;A string with three characters in it
""                             ;An empty string
"\"APL\\360?\" he cried."      ;A string with twenty characters
"|x| = |-x|"                   ;A ten-character string
```

Notice that any vertical bar — in a string need not be preceded by a Ṡimilarly, any double quote in the name of a symbol written using vertical-bar notation need not be preceded by a Ṫhe double-quote and vertical-bar notations are similar but distinct: double quotes indicate a character

string containing the sequence of characters, whereas vertical bars indicate a symbol whose name is the contained sequence of characters.

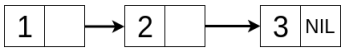
The characters contained by the double quotes, taken from left to right, occupy locations within the string with increasing indices. The leftmost character is string element number 0, the next one is element number 1, the next one is element number 2, and so on.

Note that the function prin1 will print any character vector (not just a simple one) using this syntax, but the function read will always construct a simple string when it reads this syntax.

2.10.4 Lists and Conses

A cons is a record structure containing two components called the car and the cdr. Conses are used primarily to represent lists.

A list is recursively defined to be either the empty list or a cons whose cdr component is a list. A list is, therefore, a chain of conses linked by their cdr components and terminated by NIL, the empty list. The car components of the conses are called the elements of the list. For each element of the list, there is a cons. The empty list has no elements.



A list is notated by writing the elements of the list in order, separated by blank space (space, tab, or return characters) and surrounded by parentheses.

```
(a b c)                 ;A list of three symbols
(2.0s0 (a 1) #\*)       ;A list of three things: a short floating-point
                        ; number, another list, and a character object
```

The empty list nil therefore can be written as (), because it is a list with no elements.

2.10.5 Dotted Lists

A dotted list is one whose last cons does not have nil for its cdr, rather some other data object (which is also not a cons, or the first-mentioned cons would not be the last cons of the list). Such a list is called "dotted" because of the special notation used for it: the elements of the list are written between parentheses as before, but after the last element and before the right parenthesis are written a dot (surrounded by blank space) and then the cdr of the last cons. As a special case, a single cons is notated by writing the car and the cdr between parentheses and separated by a space-surrounded dot.

```
(a . 4)             ;A cons whose car is a symbol
                    ; and whose cdr is an integer
(a b c . d)         ;A dotted list with three elements whose last cons
```

```
4                    ; has the symbol d in its cdr
```

2.10.6 Arrays

An array is an object with components arranged. In general, these components may be any Lisp data objects.

The number of dimensions of an array is called its rank. The total number of elements in the array is the product of all the dimensions.

An implementation of Common Lisp may impose a limit on the rank of an array, but this limit may not be smaller than 7. Therefore, any Common Lisp program may assume the use of arrays of rank 7 or less. (A program may determine the actual limit on array ranks for a given implementation by examining the constant array-rank-limit.)

It is permissible for a dimension to be zero. In this case, the array has no elements, and any attempt to access an element is in error. However, other properties of the array, such as the dimensions themselves, may be used. If the rank is zero, then there are no dimensions.A zero-rank array, therefore, has a single element.

An array element is specified by a sequence of indices. The length of the sequence must equal the rank of the array. Each index must be a non-negative integer strictly less than the corresponding array dimension. Array indexing is therefore zero-origin.

As an example, suppose that the variable foo names a 3-by-5 array. Then the first index may be 0, 1, or 2, and the second index may be 0, 1, 2, 3, or 4. One may refer to array elements using the function aref; for example, (aref foo 2 1) refers to the element (2, 1) of the array. Note that aref takes a variable number of arguments: an array and as many indices as the array has dimensions. A zero-rank array has no dimensions, and therefore aref would take such an array and no indices, and return the sole element of the array.

In general, arrays can be multidimensional, can share their contents with other array objects, and can have their size altered dynamically (either enlarging or shrinking) after creation.

Multidimensional arrays store their components in row-major order; that is, internally a multidimensional array is stored as a one-dimensional array, with the multidimensional index sets ordered lexicographically, last index varying fastest. This is important in two situations: (1) when arrays with different dimensions share their contents, and (2) when accessing very large arrays in a virtual-memory implementation. (The first situation is a matter of semantics; the second, a matter of efficiency.)

2.10.7 Vectors

One-dimensional arrays are called vectors in Common Lisp and constitute the type vector (which is, therefore, a subtype of the array). Vectors and lists are collectively considered to be sequences. They differ in that any component of a one-dimensional array can be accessed in constant time,

whereas the average component access time for a list is linear in the length of the list; on the other hand, adding a new element to the front of a list takes constant time, whereas the same operation on an array takes time linear in the length of the array.

A general vector (a one-dimensional array) can be notated by notating the components in order, separated by whitespace and surrounded by (and). For example:

```
#(a b c)                        ;A vector of length 3
#()                             ;An empty vector
#(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)
                                ;A vector containing the primes below 50
```

Implementations may provide specific specialized representations of arrays for efficiency in the case where all the components are of the same specialized (typically numeric) type. All implementations provide specialized arrays for the cases when the components are characters (or rather, a special subset of the characters); the one-dimensional instances of this specialization are called strings. All implementations are also required to provide specialized arrays of bits, that is, arrays of type (array bit); the one-dimensional instances of this specialization are called bit-vectors.

2.10.8 Bit vectors

A bit-vector can be written as the sequence of bits contained in the string, preceded by *; any delimiter character, such as whitespace, will terminate the bit-vector syntax. For example:

```
#*10110         ;A five-bit bit-vector; bit 0 is a 1
#*              ;An empty bit-vector
```

The bits notated following the $*$, taken from left to right, occupy locations within the bit-vector with increasing indices. The leftmost notated bit is bit-vector element number 0, the next one is element number 1, and so on.

The function *prin*1 will print any bit-vector (not just a simple one) using this syntax, but the function read will always construct a simple bit-vector when it reads this syntax.

2.10.9 Other data types

- Hash tables
  Provide an efficient way of mapping any Lisp object (a key) to an associated object.
- Readtables
  Are used to control the built-in expression parser read.
- Packages
  Are collections of symbols that serve as name spaces. The parser recognizes symbols by looking up character sequences in the current package.
- Streams
  Represent sources or sinks of data, typically characters or bytes. They are used to perform

I/O, as well as for internal purposes such as parsing strings.

- Random-states

  Are data structures used to encapsulate the state of the built-in random-number generator.

2.10.10 Structures

Structures are instances of user-defined data types that have a fixed number of named components. Structures are declared using the *defstruct* construct; *defstruct* automatically defines access and constructor functions for the new data type.

Different structures may print out in different ways; the definition of a structure type may specify a print procedure to use for objects of that type. The default notation for structures is

```
#S(structure-name
        slot-name-1  slot-value-1
        slot-name-2  slot-value-2
                          ...)
```

where *S* indicates structure syntax, structure-name is the name (a symbol) of the structure type, each slot-name is the name (also a symbol) of a component, and each corresponding slot-value is the representation of the Lisp object in that slot.

2.11 LABELS and FLET

The special form *LABELS* introduces a new lexical block which temporarily defines new functions. The new defined functions are only visible to the functions inside *LABELS*. Although this might seem weird at first, it is useful to define auxiliary-recursive functions (if *foo* is the primary function, auxiliary recursive functions are usually named *foo-1*,...,*foo-n*) that are only visible to the primary one. A similar form *FLET* where new functions are defined but the names of the local functions cannot be referenced by the bodies of those functions. In example 3 is shown how *labels* is used while example 4 shows the use of flet.

Example 3: Labels

```
;;; Code without using labels
;; (defun element-at (li at)
;;      (element-at-1 li at 0)
;;      )
;; (defun element-at-1 (li at i)
;;      (if (= at i)
;;          (car li)
;;          (element-at-1 (cdr li) at (+ i 1) )
;;      )
;; )
;; One could do:
```

```
;; (element-at-1 '(a b c d) 9 3) ;this is not desired

;;; Code using labels
(defun element-at (list at)
    (labels ((element-at-1 (list at i)
                (if (= at i)
                    (car list)
                    (element-at-1 (cdr list) at (+ i 1) )
                 )
             ))
             (element-at-1 list at 0) )
    )

(element-at '(a b c d e) 1)
;; is not possible to call element-at-1
```

Example 4: flet

```
(flet ((element-at (l i)
        (if (listp l)
            (element-at l i)
            l )))
        (element-at '(LISP flet example) 2)
;;          (element-at 9 9 )
      )
```

3 Steel Bank Common LISP

The name, Steel Bank Common LISP is a reference to the CMU founders: Andrew Carnegie was in the steel industry while Andrew Mellon was a banker. SBCL began as a forked branch from the CMUCL dialect. The main reason was to get rid of dependencies on old CMUCL libraries (bootstrapping). In theory, SBCL could be built from any CL system. IT has an interesting building process since most of SBCL is coded in LISP: more than 75% of the lines are LISP code with only about 17% of C code and assembly, most of which deals with OS's and architecture stuff. The garbage collector is only built in C because debuggers, e.g. *gdb*, work better with C than with LISP. In [8] there is an extensive explanation on the SBCL compilation and the compiler itself.

Since the initial fork, there have been some changes in the system but not on the language itself. Although the ANSI standard can be bought, there is a free version of the specifications available at [9] called CL Hyperspec. SBCL is a mostly-conforming implementation of the ANSI Common Lisp standard, the following are some of the features different or not specified in the standard extracted from the SBCL manual [10].

3.1 Installation

SBCL can be easily installed using package managers such as *apt*, *dnf* or *homebrew*. Just search for the SBCL package (usually named *SBCL*). If a Windows installation is necessary refer to the SBCL download page: `http://www.sbcl.org/platform-table.html` (SBCL can run on Windows but not all its features are available).

To run SBCL, in a terminal just type *SBCL*.

3.2 Extensions

SBCL implements the system-definition tool Another System Definition Facility (ASDF) which is what *make* is for C. Was originally built as a third-party application but SBCL integrated it.

3.3 Declarations

Declarations are assertions: the compiler doesn't believe type declarations, they are considered assertions that should be checked.

3.4 Interpreter/Compiler

On SBCL everything goes through the compiler, the *eval* function is implemented by calling the native code compiler. Although it has an interpreter it is only called internally and the user can

call it if necessary. The interpreter isn't safer or more debuggable than the compiler (in other LISP implementations, the interpreter gives more information).

3.5 Executables

Stand-alone executables generation is possible. In fact, SBCL can generate an executable of itself meaning that a deployed program can call the compile and load functions.

3.6 scripting

Although not recommended [11], SBCL can be used as a scripting language with the following line at the top of the file:

```
#!/usr/local/bin/sbcl --script
```

3.7 Debugger

When using SBCL, the debugger is invoked if it encounters an error during the REPL. As seen in example 5, the debugger offers the user alternatives to try to solve the problem. The user can input the desired command or its number. If options 1 or 2 of the example 5 are used, SBCL will ask for an input. There are debugger commands that can help in the debugger process.

Example 5: Debugger example

```
* (+ 4 a)

debugger invoked on a UNBOUND-VARIABLE in thread
#<THREAD "main thread" RUNNING {1001948083}>:
  The variable A is unbound.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [CONTINUE    ] Retry using A.
  1: [USE-VALUE   ] Use specified value.
  2: [STORE-VALUE] Set specified value and use it.
  3: [ABORT       ] Exit debugger, returning to top level.

(SB-INT:SIMPLE-EVAL-IN-LEXENV A #<NULL-LEXENV>)
0]
```

3.8 Tail Recursion

Because in LISP recursion is meant to be used, tail recursion optimization is essential for efficient programs. SBCL's compiler is "properly tail recursive", which means that the stack frame is deallocated at the time of the call rather than after the call returns. Since this can cause the program to be difficult to debug, there is an option to disable this.

3.9 Threading

Threading is supported by SBCL by the SB-THREAD package which maps onto the host OS concept of threads. With threading, a program can take advantage of hardware multiprocessing although there is no control over the scheduler.

References

[1] J. McCarthy, "History of lisp," in *History of programming languages I.* ACM, 1978, pp. 173–185.

[2] G. L. Steele Jr, "An overview of common lisp," in *Proceedings of the 1982 ACM symposium on LISP and functional programming.* ACM, 1982, pp. 98–107.

[3] N. Pippenger, "Pure versus impure lisp," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 2, pp. 223–238, 1997.

[4] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[5] R. Stallman. (2002, oct) My lisp experiences and the development of gnu emacs. [Online]. Available: https://www.gnu.org/gnu/rms-lisp.en.html

[6] T. T. S. Q. Company. (2018, apr) Tiobe index. [Online]. Available: https://www.tiobe.com/tiobe-index/

[7] C. L. I. A. Survey. (2010, feb) Daniel weinreb. [Online]. Available: common-lisp.net/~dlw/LispSurvey.html

[8] C. Rhodes, "Sbcl: A sanely-bootstrappable common lisp," in *Self-Sustaining Systems.* Springer, 2008, pp. 74–86.

[9] K. Pitman *et al.*, "Common lisp hyperspec," 2012.

[10] C. S. Rhodes *et al.*, "Steel bank common lisp user manual," *preparation. 3This quality is part of SBCL's raison d'être.*

[11] F.-R. Rideau, "Asdf 3, or why lisp is now an acceptable scripting language," in *7 th European Lisp Symposium*, 2014, p. 12.