Common Lisp - The programmable programing language

Ben Dudson

The Lisp family

There are many different Lisp languages and varieties of Lisp including:

- **Common Lisp** - ANSI standard 1984 - 1994, multiple implementations
- **Racket** - Scheme descendant, active community
- **Clojure** - Runs in the Java Virtual Machine
- **Emacs lisp** - Extension language for Emacs
- **GNU Guile** - Extension language

Honorable mention:

- **Julia** – Numerical/scientific focus, lots of Lisp influence

What is (Common) Lisp?

Descendant of Lisp, developed ca. 1956 by John McCarthy (MIT, AI researcher) it is:

- Dynamic, compiled, strongly typed, multi-threaded, garbage collected, ...
- ANSI standardised, with several high-quality implementations

Many of the innovative features of Lisp have been adopted by other languages, so its use as a high-level language has been largely replaced by e.g. python.

Lisp still has a unique combination of features:

- Excellent interactive, incremental development
- One of the most flexible object systems available, CLOS and MOP
- A quite unusual error handling system, with conditions and restarts
- Powerful macro systems for defining new language features

Why learn Common Lisp?

1. Different kind of programming language: **symbolic**
   - Symbols are "first class" objects, like numbers. They can be created, manipulated, stored and evaluated
2. A minimal but flexible syntax
   - The core of the language is small, 7 - 25 "special" forms
   - Everything else is done by manipulating code into these forms
   - Gives you the power to define the language you want to use
   - Encourages programs made from many small pieces
3. Many different styles of programming have been implemented in Lisp:
   - Object oriented
   - Functional
   - Logic
   - Nondeterministic
   - ...

Some applications

- Lots of planning tools and expert systems
  - DART (US military)
  - ITS (Airline, now Google)
  - Cyc
  - NASA planning MARS pathfinder
- Onboard computers
  - NASA Deepspace 1 probe (1998-2001) patched with aid of lisp REPL
  - Roomba vacuum cleaner
- Computer algebra
  - Maxima, developed from Macsyma (Project MAC, 1968-1982)
  - Axiom (Scratchpad, IBM 1971)
- Quantum computing
  - Rigetti computing's Quantum Virtual Machine
  - D-wave systems

Development environment

- Online testing (today):
  - https://ideone.com/
  - Includes SBCL (native compiler) and CLISP (bytecode)
- SLIME: The most widely used environment in Emacs.
  - Includes documentation, interactive debugging, profiling, tracing, inspection, ...
  - `rainbow-delimiters` and `paredit` useful for braces
  - Basic support for mixing code, outputs, equations etc. using `org-mode`
- Jupyter Notebook: cl-jupyter
  - Relatively basic, but functional.
  - No real support for debugging, inspection, profiling etc.
  - Get the benefits of a notebook

See the Lisp Cookbook, https://lispcookbook.github.io

Through the looking glass (1/4)

Lets start with something simple but familiar

```
for i in range(10):
    if i % 2 == 0:
        print("{0} is even".format(i))
    else:
        print("{0} is odd".format(i))
```

```
0 is even
1 is odd
2 is even
...
```

Through the looking glass (2/4)

The conditional `if` is really a kind of function:

```
if( test, run-if-true, run-if-false )
```

. . . and so is the `for` loop:

```
for( variable, range, code-to-run )
```

. . . in fact (almost) everything is a kind of function!

Through the looking glass (3/4)

so we could write our code as

```
for( i, 10,
    if( i % 2 == 0,
        print("{0} is even".format(i)),
        print("{0} is odd".format(i))))
```

Whilst we're at it, both % and == are functions:

`%(number, divisor)` `==(left, right)`

```
for( i, 10,
    if( ==( %(i, 2), 0),
        print("{0} is even".format(i)),
        print("{0} is odd".format(i))))
```

Through the looking glass (4/4)

Now for the final leap... The function to call can be put inside the brackets

```
(for, i, 10,
    (if, (== (%, i, 2), 0),
        (print, "{0} is even".format(i)),
        (print, "{0} is odd".format(i))))
```

Then we tidy up the unnecessary commas and do some renaming:

```
(dotimes (i 10)
  (if (= (mod i 2) 0)
      (format t "~a is even~%" i)
      (format t "~a is odd~%" i)))
```

Voila! Common Lisp.

LISt Processing

In Lisp flow control, loops, and even function and class definitions, are all represented as a list

```
(function arg1 arg2 ...)
```

The first element is the function, followed by the arguments.

- Since lists can be manipulated by Lisp code, code can also be manipulated (it is **homoiconic**)
- Before code is compiled, arbitrary lisp code (entire programs) can transform it
- This makes lisps unique in their ability to define Domain Specific Languages
- The language can be changed to fit the problems you want to solve

Everything is an expression

In Lisp everything is an expression which returns a value (though it may be NIL)

- In Python this is ok `(x + (y + 1))` but not `(x + (if is_true(): 1 else: 2))` because `if` is a statement not an expression
- In Lisp this would be `(+ x y 1)` and `(+ x (if (istrue) 1 2))`

For larger expressions we can define local variables and functions, putting together code in a very flexible way

```
(+ x (let ((y (random 10)))
       (format t "Chosen: ~a~%" y)
       (some-function y)))
```

(from https://practicaltypography.com/why-racket-why-lisp.html)

Get lisping!

Try evaluating the following expressions:

```
> (+ 1 2 3 4)
> (+ (* 2 3) (* 4 5))
> (list 2 3)
> (list (list 1 2) (list 3 4))
> (quote (1 2))
> (quote (+ 1 2 3 4))
> '(1 2)
```

Then:

1. Write an expression to make a nested list `(1 (2 (3 4)))`
2. Calculate

```
2 * sin(3.2) - 1 ; => -1.1167483
```

Solutions

1. Either

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

or

```
(quote (1 (2 (3 4))))
```

2.

```
(- (* 2 (sin 3.2)) 1)
```

which is easier to read if written:

```
(- (* 2
      (sin 3.2))
   1)
```

Functions

Functions can be defined using `defun`

```
(defun f (a x b)
  (+ b (* a x)))
```

or created without a name and passed around

```
(lambda (a x b)
  (+ b (* a x)))
```

so we can create a function and then apply it

```
(funcall (lambda (x) (+ 2 x)) 3)  ; => 5
```

Common Lisp is often compiled (e.g. SBCL)

```
(defun f (a x b)
  (+ b (* a x)))
```

To see the byte or assembly code:

```
(disassemble #'f)
```

Code can be optimised if given types:

```
(defun f (a x b)
  (declare (optimize (speed 3) (safety 0))
           (type single-float a x b))
  (+ b (* a x)))
```

Try disassembling again. . .

Functions of functions

Many of the Common Lisp standard functions take other functions as input e.g.

```
(mapcar (lambda (x) (+ 2 x))  '(1 2 3))  ; => (3 4 5)

(reduce #'+ '(1 2 3))  ; => 6

(sort '(1 2 3 4) #'>)  ; => (4 3 2 1)
```

Exercise 2: Functions

1. Define a function to square numbers e.g

```
(square 3) ; => 9
```

2. Calculate the sum of the squares of the numbers 1 to 9 (= 285)

You can use:

```
(defun range (n)
  (loop for i from 0 below n collecting i))
```

Solutions

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

```
(reduce #'+
        (mapcar #'square
                (range 10)))
```

or

```
(loop for i from 0 below 10 summing (* i i))
```

or

```
(defun sum-squares (numbers)
  (if numbers
      (+ (square (first numbers))
         (sum-squares (rest numbers)))
      0))
(sum-squares (range 10))
```

Common Lisp Object System (CLOS)

CLOS is quite different from the object system in Java/C++ or Python.

```
(defstruct circle
  radius)

(defparameter a (make-circle :radius 1.2))
```

Methods are defined outside classes, and can be specialised for particular types (multiple dispatch):

```
(defmethod area ((shape circle))
   (* pi (expt (circle-radius shape) 2)))

(area (make-circle :radius 2))
; => 12.566370614359172d0
```

Exercise 3: Areas of shapes

- Define a structure called `rectangle` with a length and a height, and an area method.
- Create a list of circles and rectangles:

```
(defparameter shapes
  (list
   (make-rectangle :length 0.5 :height 2.0)
   (make-circle :radius 3.1)
   (make-rectangle :length 4.2 :height 1.7)))
```

- Calculate the total area of all the shapes in the list (38.33070418890327)

Exercise 3: Solution part 1

```
(defstruct rectangle
  length
  height)

(defmethod area ((shape rectangle))
  (* (rectangle-length shape) (rectangle-height shape)))
```

Exercise 3: Solution part 2

Some possible solutions

```
(reduce #'+ (mapcar #'area shapes))  ;; Map shapes to areas, then sum
```

```
(defun sum-shapes (shapes)  ;; Recursive function
  (if shapes
      (+ (area (first shapes))
         (sum-shapes (rest shapes)))
      0.0))
(sum-shapes shapes)
```

```
(loop for s in shapes summing (area s))
```

```
(let ((sum 0.0))              ;; Accumulate the sum of the areas
  (dolist (s shapes)
    (incf sum (area s)))
  sum)
```

Macros

Macros are functions which transform code before it is compiled.

- Some languages have mechanisms for doing this, but usually use a different syntax e.g. C preprocessor, C++ templates
- To generate code a separate tool is often needed
- Lisp macros are lisp functions, and can run arbitrary lisp code (including other macros).

e.g. All the looping constructs `loop`, `dolist`, `dotimes`, `...` are macros. Try

```
(macroexpand '(loop for s in shapes summing (area s)))
```

Example

For quick tests, it would be nice if we could write something like

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

(example (square 3) => 9)
```

and if `(square 3)` returned something different get a printed error like

```
Expected (square 3) => 9
   but got 3
```

Example macro version 1

Just write the code we want in a backtick ` and insert bits of code where we need them by using , (or ,@ to splice in lists)

```
(defmacro example (code arrow result)
  `(if (equalp ,code ,result)
       t
       (format t "Expected ~a => ~a~%    but got ~a"
               ',code ,result ,code)))
```

Example macro version 2

Avoids evaluating expressions more than once

```
(defmacro example (code arrow result)
  `(let ((code-result ,code)
         (expected ,result))
     (if (equalp code-result expected)
         t
         (format t "Expected ~a => ~a~%     but got ~a"
                   ',code expected code-result))))
```

Example macro version 3

Doesn't accidentally capture symbols

```
(defmacro example (code arrow result)
  (declare (ignore arrow))
  (let ((code-result (gensym))
        (expected (gensym)))
    `(let ((,code-result ,code)
           (,expected ,result))
       (if (equalp ,code-result ,expected)
           t
           (format t "Expected ~a => ~a~%     but got ~a"
                   ',code ,expected ,code-result)))))
```

Note: Not really needed here, but necessary in general. Racket and other lisps have a different approach to "hygenic" macros

Exercise 4: Macros

Backquote examples:

```
> `(a (+ 1 2) c)
> `(a ,(+ 1 2) c)
> `(a (list 1 2) c)
> `(a ,(list 1 2) c)
> `(a ,@(list 1 2) c)
```

- Define a macro `my-and` that takes two expressions and evaluates the first. If the first evaluates to `nil`, return `nil`. Otherwise evaluate the second expression and return its result.
- Write a macro which evaluates an expression a random number of times between 0 and 10

```
(random-times (format t "hello~%"))
```

(Hint: See `dotimes` earlier)

Solution part 1

```
(defmacro my-and (a b)
  `(if ,a
       ,b
       nil))
```

or

```
(defmacro my-and (a b)
  `(when ,a
         ,b))
```

Solution part 2

```
(defmacro random-times (expr)
  `(dotimes (i (random 10))
     ,expr))
```

or using a gensym to avoid introducing variable `i`:

```
(defmacro random-times (expr)
  (let ((sym (gensym)))
    `(dotimes (,sym (random 10))
       ,expr)))
```

Reader macros

Some dialects (including Common Lisp and Racket) have reader macros.

- Transform input text as it is read, turning it into lisp expressions
- The transformation code can be arbitrary lisp functions, macros etc.

Some example uses:

- Infix notation

```
#i(result[i j] += A[i k] * B[k j])
```

- List comprehensions

```
{i j || i <- '(1 2 3 4 5 6 7 8) j <- '(A B)}    ;=> ((1 A) (2 B))
```

- Reading JSON

```
[{"foo": 1}, "bar", {"baz": [2, 3]}]
```

- Creating your own language!

Further reading

Getting started

- Practical Common Lisp: http://www.gigamonkeys.com/book/
- Lisp in small pieces: http://lisp.plasticki.com
- Debugging Common Lisp: http://malisper.me/debugging-lisp-part-1-recompilation/
- Common Lisp Cookbook https://lispcookbook.github.io/

Getting started (other lisps)

- The Racket Guide: http://docs.racket-lang.org/guide/index.html
- Structure and Interpretation of Computer Programs http://mitpress.mit.edu/sicp/

Other links

- Common Lisp HyperSpec. Google “CLHS function” for reference page
- European Lisp Symposium with slides and videos about current projects