

About this Document

This document was written to accompany an in-person Lisp tutorial. Therefore, the information on this document alone is not likely to be sufficient to get a good understanding of Lisp. However, the functions and operators that are listed here are a good starting point, so you may look up specifications and examples of usage for those functions in the Common Lisp HyperSpec (CLHS) to get an understanding of their usage (simple google “clhs [function/macro name]”).

Getting Started

Note on Common Lisp Implementations

We will be using SBCL since that is the de facto standard for open source CL work (and most of my research work). However, in this class, we won't be using many (if any) features that are specific to common lisp implementations so experience in any common lisp implementation will transfer well.

IDE/Editor

It is not important for this class to have an extensive IDE since we will be working on small coding projects. However, for your sake, it is important that you use an editor that has **syntax highlighting**, **lisp indentation**, and **parentheses highlighting**. The most important being parentheses highlighting. This will save a lot of time during programming since Lisp code can build up many parentheses that are semantically significant. Therefore, highlighting can greatly help with checking that the parentheses open and close in the correct places.

Recommended Environments:

- Emacs (with or without SLIME)

Lisp and emacs are closely related so there's a lot of support for Lisp in the emacs environment. SLIME is an extension to emacs that adds additional support. If you're someone who likes a lot of language-related functionality in the editor or plan to use Lisp after this class the time investment of installing and learning SLIME may be worthwhile.

- Vim

Though there's less support for Lisp than emacs, vim recognizes the syntax, indentation, and parentheses. Good choice if you're already comfortable with vim.

- Notepad++?

If you really don't like any of the other options (I admit those editors may have fairly steep learning curves), this will work since it supports syntax and parenthetical highlighting.

Interacting with the REPL

The REPL (read-eval-print-loop) allows you to run code dynamically (run one command at a time). Lisp is an interpreted language so it doesn't need to be compiled (though it can be). To start the REPL, from the department servers type the appropriate command `sbcl`. To exit type `(exit)` or `ctrl-d`.

To load/run a file (like your homework file) type `(load "filename")`.

Local Variables/Functions

Defining variables locally is important since the scoping of variables and functions in Lisp are global by default. The scopes for variables and functions stated with explicit operators.

Variables

Local variables are defined using *let* and *let**.

let evaluations are all performed in parallel, thus each definition must be independent of one another.

*let** evaluates each statement in order so that expressions can use variables defined previously.

Functions

Local functions are defined using *flet* and *labels*.

The syntax for these operators are analogous to *let* and *let**, extended to functions, so I recommend getting comfortable with those operators before trying these out.

flet evaluates the functions in parallel and the scopes for the definition are only in the body (after the function definitions).

labels has the function scopes encompassing the function definitions so that the local functions can call each other and themselves for recursive processes.

Functional Operations

There are some paradigms/operations in functional programming that are important for you to know when programming in a language like Lisp. Getting comfortable with these will make your life a lot easier since they simplify many problems.

Map (i.e. mapcar)

A map takes a function with one argument and a list, then performs the function on each element of the list and returns the result. This is a very common operation (if you think about problems in a functional mindset) which when combined with the next operation that we'll describe can simplify many complex problems into simple abstractions.

In Lisp, this is done with *mapcar*. (mapcar #'fn lst) => mapped lst. For example:

```
(mapcar #'not '(t nil nil t nil)) => (nil t t nil t)
```

The Common Lisp implementation of mapcar can also take multiple lists as arguments and take the elements of each together as separate arguments of the functions.

```
(mapcar #'+ '(1 2 3) '(4 5 6)) => '(5 7 9)
```

Reduce/Fold (i.e. reduce)

A reduce takes a function with two arguments and a list. It applies the function to the accumulated value (can specify an initial value or by default uses the first two elements for accumulation and next value) and the next value in the list and sets that as the new accumulated value. Returns accumulation at the end.

```
(reduce #'list '(1 2 3 4)) => (((1 2) 3) 4)
```

```
(reduce #'list '(1 2 3 4) :initial-value '()) => ((((((1) 2) 3) 4)
```

```
(reduce #'* '(1 2 3 4 5)) => 120
```

Filter (remove-if remove-if-not)

Takes a function with one parameter returning t or nil and a list. Returns the list with only elements that return t for the function.

```
(remove-if-not #'even-p '(1 2 3 4)) => '(1 3)
```

Conditionals

There is the classic *if* statement in Lisp, but it doesn't have support for layering *else ifs* for many levels as in most procedural languages. Alternatively, Lisp has the *cond* (conditional) which allows you to list a series of conditions followed by corresponding executions. For *if* statements without an else branch, Lisp has *when* and *unless*. *when* is equivalent to *if* without supporting an else branch and *unless* is equivalent to (*when (not ...)*).

Evaluating Lisp/Function

There are three main functions for evaluating lisp functions or code: *eval*, *funcall*, and *apply*.

eval interprets its argument as lisp code and runs it.

```
(e.g. (eval '(+ 1 2 3)) => 6)
```

funcall takes a function followed by its arguments (one-by-one) as arguments

```
(e.g. (funcall #'+ 1 2 3) => 6)
```

apply similar to funcall but takes the arguments as a list. (There are other subtleties, but likely not important for your work here).

```
(e.g. (apply #'+ '(1 2 3)) => 6)
```

Type Tests

Types can be checked with a variety of type predicates. A full(?) list can be found https://www.gnu.org/software/emacs/manual/html_node/elisp/Type-Predicates.html. They are generally suffixed with “p” to indicate a type predicate, and sometimes have negated varieties that a prefixed with “n” (short for not).

The most common ones are:

- atom - is it an atom (i.e. not a non-empty list)
- listp - is it a list (including nil)
- consp - is it a cons cell (i.e. listp without nil)
- null - is it null (i.e. nil)
- symbolp - atom that isn't an integer, string, or other special type
- numberp - is it a number
- integerp
- floatp
- rationalp

Equality

Lisp has several different equality predicates. They vary on how closely they follow the computer representation of the value vs the human semantic value. For details see <https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node74.html>. In most cases the most intuitive equality predicate is *equal*.

Equality predicates in increasing complexity

- eq* - only the same instances
- eql* - compares integer/character semantics
- equal* - compares strings and lists semantically
- equalp* - compares between number types (integer and float) as others

Lisp Weirdness

Quotes

Quoted symbols will be interpreted as symbols directly, otherwise will be interpreted as some value. If it's a number or a string, this will feel normal. But if put a symbol somewhere without a quote, Lisp will try to interpret it as a variable, a parameter, or a function.

Pipes

Symbols are by default uppercase only (case-insensitive). Therefore, to specify case-sensitive symbols, use pipes (e.g. '|Hi There|). As you can see, it also allows spaces and other such characters that normally are not interpreted as a part of a symbol.

Characters

Lisp specifies characters with #\ . So to specify a character, it must be prefixed as such. Otherwise, it will be interpreted directly as a symbol.

Testing

I highly recommend looking at the link on my site (cs.rochester.edu/~gkim21/cs291) to the Practical Common Lisp guide to testing in Lisp. It will help you structure your tests transparently and informatively.

Debugging

Having trouble printing big lists on a single line?

Try turning off the pretty printer: `(setf *print-pretty* nil)`.

Still not working? Try increasing the margin: `(setf *print-right-margin* 10000)`

When a program crashes:

- check the backtrace with: `backtrace`
- go up and down the stack with: `up/down`
- check the local variables: `list-locals` (you may need to first go down the stack to get to the code you wrote)

Practice Problems

If you have not used Lisp before, the homework problems are likely too difficult of a starting point for getting started in Lisp. I recommend trying to solve the following problems using recursion and higher-order functions (`map/reduce/filter`) to get comfortable with Lisp before starting the homework problems.

- Count list length without *length* predicate recursively
- Index a list recursively [`'(a b c) => '((0 a) (1 b) (2 c))`]
- Implement `map`
- Implement `reduce`
- All elements in list are true (not nil)
- Drop nth element in a list
- Depth-first search of a tree without non-terminal labels. The tree is represented as a list structure. Each non-terminal node is a list and a terminal node is a symbol.
Ex. `'((a b) c (d (e f)))` -- (perhaps return the pre-ordering of elements to check it works)

Note from Gene:

If you've tried these problems and want to see other plausible solutions I came up with, contact me by email (with your solutions attached) and I can send you what I've tried. I'm making it hard for you guys to get the solutions so that you actually try to solve them on your own before seeing solutions!