Common Lisp: A Brief Tutorial

August 2004

David R. Pierce
Stuart C. Shapiro

Table of Contents

1 Basics

1.1 Introduction

These notes on Common Lisp are adapted from

> Stuart C. Shapiro and David R. Pierce,
> *Lisp Programming for Graduate Students*, 2004,
> `http://www.cse.buffalo.edu/~shapiro/Courses/CSE202/Summer2004/`.

Since the notes rely heavily on Xanalys's *Common Lisp HyperSpec*,[1] they are best viewed in their hypertext form.

To facilitate simultaneous browsing of the notes while working in the Lisp interpreter, the notes are also available in Emacs Info. If you have followed the course Emacs setup instructions (see section "Emacs Setup" in *Emacs Setup*), type `C-h i` to start Info, and middle-click on '`Common Lisp`'. Otherwise, start Info using `C-u C-h i` and respond to the query for a file with '`/projects/drpierce/cse/467/info/lisp.info`'.

CSE uses Franz Inc.'s Allegro Common Lisp (ACL) implementation. To start ACL from the shell, simply execute the command '`mlisp`'. To start it within Emacs, see section "Running Lisp" in *Emacs Setup*. Once Lisp has started, it awaits your input expressions. To exit the Lisp interpreter, type '`(exit)`'.

1.2 Syntax

Common Lisp is an "expression-oriented" programming language. A Lisp program consists of expressions, or *forms*. A form by itself is a program, but most programs are made up of many forms.

There are three kinds of expressions:

Symbols Symbols are Lisp's variables. Symbol names may legally contain many characters, but are usually made up of letters and dashes — for example, `a-symbol`.

Lists Lists are delimited by parentheses. For example, `(f x y)` is a list containing the symbols `f`, `x`, and `y`. Lists are Lisp's function calls. Evaluating `(f x y)` calls the function named `f` on the arguments `x` and `y` (whatever they are).

Literals Literal expressions include numbers (e.g., `0.42e2`), strings (e.g., `"a string"`), characters (e.g., `#\c`), arrays (e.g., `#(1 2 3)`), etc.

1.3 Read-Eval-Print

Interaction with the Lisp interpreter takes place in a *read-eval-print* loop, which behaves as follows:

1. Read a form and construct a Lisp object from it.
2. Evaluate the input object to produce an output object.

[1] `http://www.lispworks.com/reference/HyperSpec/Front/index.htm`

3. Print the output object using some print representation.

A subtle, but far-reaching, consequence of the read-eval-print interpreter model is that all Lisp programs are in fact also Lisp data objects! For example, the “Hello World” program — `(print "Hello World")` — is a list containing a symbol and a string.

Another aspect of read-eval-print is that it allows the interpreter to have different reader/printer representations for the same object. When this is true, the Lisp printer simply chooses one of them; but the Lisp reader can always recognize any of them.

Next we will consider evaluation, the second step of the loop. We will leave a more detailed discussion of the reader and printer for later.

1.4 Evaluation

The read-eval-print loop *evaluates* input objects to produce output objects. Each different kind of expression is evaluated in a different way.

- When symbols are evaluated, Lisp treats the symbol as a program variable and looks up its value in its lexical scope.
- When lists are evaluated, Lisp expects to find a function name[2] (a symbol) as the first element of the list. The remaining elements of the list are evaluated to produce arguments, and the corresponding function is called with those arguments.
- Literals, also called *self-evaluating* objects, evaluate to... er... themselves. (Surprise!)

Examples:

```
;; read a symbol, evaluate as a variable
pi

;; oops
foo

;; read a list, evaluate as a function call
(+ 3 5 7 11 13)

;; oops
(1 2 3)

;; read a string literal, evaluate as itself
"hello world"
```

But if symbols and lists are always evaluated, how can we get a symbol or list object? The *special form*[3] `(quote` ***`expression`***`)` prevents *expression* from being evaluated.

```
;; a symbol
(quote pi)
```

[2] Or the name of a special operator or macro, but we'll ignore that for the moment.

[3] Special forms are like function calls, except that they don't necessarily evaluate all of their arguments. While we can define new functions in Lisp, there is no way to define new special operators.

```
;; same thing (an abbreviation)
'pi

;; ok this time
(quote foo)

;; some lists
(quote (1 2 3))
'(1 2 3)
'(a b c)
```

1.5 Comments

Comments in Lisp appear in two forms.

```
; this is a 1-line comment

#|
   this is a block comment
|#
```

When using semicolon comments, a conventional style is to use more semicolons for more important comments.

```
;;; a comment between function definitions for a section of code
;;; beginning in the first column of the text

    ;; a comment in the body of a function definition
    ;; indented at the same level as the code
    ...

                        ... ; a comment after a line of code
```

1.6 Running Programs

While developing programs, it is usually most convenient to work in the Lisp interpreter. However, you can also run Lisp programs on the command line. For example, suppose that 'hello.cl' contains the following program:

```
(princ "Hello World")
(princ #\newline)
```

To run this program, use the '-#!' option to mlisp.

```
mlisp -#! hello.cl
⊣ Hello World
```

2 Data Types

Before we consider Lisp's data types, there are a couple functions that are useful for looking at objects.

describe *object* [Function]

Describes *object*. .

type-of *object* [Function]

Returns the type of *object*.

2.1 Booleans

Notes:

- False is `nil`.
- Any non-`nil` object is true.
- The canonical true value is `t`.
- Actually `t` and `nil` are constant-valued symbols whose values are themselves.

if *test-form then-form* [*else-form*] [Special Operator]

The conditional expression. It evaluates *test-form*. If the test value is true, it evaluates *then-form* and uses that value as its result. Otherwise it evaluates *else-form* for its result. A missing *else-form* is equivalent to `nil`.

and *form** [Macro]

"Short-circuited" conjunction operator.[1] It evaluates each *form* from left to right. If any *form* is false, `and` immediately returns `nil`. Otherwise, `and` returns the value of the last *form*. If no *form*s are supplied, `and` returns `t`.

```
(and exp1 exp2) ≡ (if exp1 exp2 nil)
```

or *form** [Macro]

"Short-circuited" disjunction operator. It evaluates each *form* from left to right. If any *form* produces a true value, `or` immediately returns that value. Otherwise, `or` returns `nil`. If no *form*s are supplied, `or` returns `nil`.

```
(or exp1 exp2) ≡ (if exp1 exp1 exp2)
```

```
t
nil
(and t   42)
(and nil 42)
(or  t   42)
(or  nil 42)
```

[1] The specification says that `and` and `or` are *macros*. A macro is like a special operator, except that it is defined in Lisp code. We can define our own macros, but that is a fairly advanced topic.

2.2 Numbers

Lisp has all the usual numerical functions, listed in *Common Lisp HyperSpec* section “Numbers”. An interesting aspect of Lisp is that it has rational and complex numeric types built in.

```
42
17/714 ; note different input/output printed representations
0.42e2 ; note different input/output printed representations
(* 1 2 3 4 5)
(/ 33 2)
(float (/ 33 2))
(max 1 5 2 4 3)
(cos (/ pi 4))
(gcd 2142 3066)
(= (* 3 14) 42.0 (/ 714 17))
(< 1/2 2/3 3/4)
(and (integerp 42)
     (rationalp 42) (rationalp 1/42)
     (realp 42) (realp 1/42) (realp 42.0))
```

Notes:

- Many Lisp functions can take any number of arguments; `*`, `max`, `<`, and `=` above are good examples.
- The letter ‘`p`’ at the end of a function name (e.g., `integerp`) stands for *predicate* to indicate a function that returns a boolean value.

Exercise: Write an expression to compute the average of 12 and 17.

2.3 Functions

Function objects are obtained using the `function` special form.

```
(function max)
#'max ; same thing (an abbreviation)
(describe #'max)
```

Functions are defined by `defun`.

```
(fboundp 'quadratic) ; not bound yet

(defun quadratic (a b c x)
  "Returns the value of the quadratic polynomial A*X^2 + B*X + C at X."
  (+ (* a x x) (* b x) c))

(fboundp 'quadratic) ; now bound
(function quadratic)
(quadratic 2 4 2 3)
```

Notes:

- Variables have lexical scope.

- Variables have no types. Objects have types. Variables can hold any type of object.
- The *body* of a function is a sequence of expressions. These expressions are evaluated in order, and the value of the last expression is returned by the function as its value.

Examples:

```
(defun factorial (n)
  "Returns the factorial of N."
  (if (<= n 0)
    1
    (* n (factorial (1- n)))))

(factorial 100)
```

Exercise: Define `(fibonacci n)` to compute the *n*th Fibonacci number (i.e., 1, 1, 2, 3, 5, 8, 13, ...). The Fibonacci numbers are defined by the relationship F[n] = F[n-1] + F[n-2], where F[1] = F[2] = 1.

2.4 Characters

Characters, like numbers, are self-evaluating objects. The syntax is `#\name`. Character functions can be found in *Common Lisp HyperSpec* section "Characters".

```
#\c
#\latin_small_letter_c ; note different printed representations
#\space

(characterp #\a)
(alpha-char-p #\a)
(char-upcase #\a)
(char-code #\a)
(code-char 98)
(char= #\a #\b)
(char< #\a #\b)
```

Exercise: Define `(char-1+ character)` to return the character whose code is one more than *character*'s.

2.5 Strings

Strings are also self-evaluating. They are delimited by double quotes. Backslash is the escape character for strings. String functions are listed in *Common Lisp HyperSpec* section "Strings". Since strings are also *sequences*,[2] additional useful functions can be found in *Common Lisp HyperSpec* section "Sequences".

```
"hello world"
"\" \\ \"" ; the string " \ "
```

[2] Sequences are one-dimensional ordered collections of objects. Lists and vectors are also sequences.

```
;; string functions
(string #\a)
(string #\\)
(length "\" \\ \"")
(char "\" \\ \"" 0)
(char "\" \\ \"" 1)
(char "\" \\ \"" 2)
(string-capitalize "hello world")
(string-trim "as" "sassafras")
(string= "hello world" "Hello World")
(string-equal "hello world" "Hello World")
(string< "hello" "world")
(string/= "world" "word")

;; sequence functions
(subseq "hello world" 4)
(subseq "hello world" 3 8)
(position #\space "hello world")
(position #\l "hello world")
(position #\l "hello world" :start 5)
(concatenate 'string "hello" "world")
```

Exercise: Define `(string-1+ `*`string`*`)` to construct a new string by adding 1 to the character code of each character in *string*. For example, `(string-1+ "a b c")` ⇒ `"b!c!d"`.

2.6 Symbols

Recall that symbols are evaluated as variables — that is, a symbol may have a value. If it does it is *bound*; otherwise it is *unbound*. Symbols also serve as function names, as we have already seen.

The syntax for symbols comprises any sequence of characters that can't be interpreted as a number or other literal. Backslash is the escape character, and the vertical bar '`|`' serves as escape brackets.

```
'hello
'hello\ world ; note different printed representations
(symbol-name 'hello\ world)
(boundp 'pi)
(boundp 'foo)
(describe '0.42e2)
(describe '0.42a2)
(describe '0.42\e2)
(describe '|0.42e2|)
```

Two symbols with the same name[3] are guaranteed to be the same object. When the Lisp reader (the first step of read-eval-print) reads a symbol, this is what it does:

[3] And interned in the same package.

1. Reads the characters you type, and constructs a string (the symbol's name).[4]
2. Looks up the symbol object by its name in a "catalog."
3. If it's not there, creates it and puts it there.

As a result, every time a symbol name is read, the same object is produced. The function `eq` tests for object identity.

```
(eq 'hello\ world '|hello world|)
(eq 'hello 'Hello)
```

The process of installing a symbol in the catalog is called *interning* it and a symbol that's been so installed is called an *interned* symbol. You can intern a symbol by calling the `intern` function with a string argument. Other symbol functions are listed in *Common Lisp HyperSpec* section "Symbols".

2.7 Packages

The "catalog" that a symbol is interned into is called a *package*. Packages are Lisp's mechanism for managing names: variable and function names are symbols, and their package is their "namespace." The reader uses the current package (the value of `*package*`) to look up symbol names. The current package when Lisp starts up is called '`common-lisp-user`'.

A package can *export* symbols for other packages to *use*. This gives the using package access to the variables and functions named by the exporting package's symbols. Consider the `describe` function:

```
(describe 'describe)
⊣
describe is a symbol.
  It is unbound.
  It is external in the common-lisp package and accessible in the
acl-socket, aclmop, antcw, clos-x-resources, cltl1, common-lisp-user,
common-windows, compiler, composer, cross-reference, debugger,
defsystem, excl, excl.scm, extended-io, foreign-functions, g6,
gprofiler, grapher, inspect, ipc, lep, lep-io, multiprocessing, net.uri,
profiler, quad-line, system, top-level, winx, and xlib packages.
  Its function binding is #<Function describe>
    The function takes arguments (x &optional stream)
  Its property list has these indicator/value pairs:
excl::dynamic-extent-arg-template  nil
```

See that `describe` is external in (exported from) the package `common-lisp`, and accessible in (used by) the `common-lisp-user` package, among others.

An external (exported) symbol in another package can also be accessed by its "full" or *qualified* name — `package:name`. An internal (non-exported) symbol can also be accessed by doubling the colon — `package::name`. (It is usually a bad idea to access symbols in this way!)

[4] Some older Lisp implementations convert unescaped symbol names to upper case. In such an implementation, `'year2004` ⇒ `YEAR2004`, while `'|year2004|` ⇒ `year2004`.

```
(eq 'length 'common-lisp:length)
;; creates a new symbol in another package! (oops)
'common-lisp::helloworld
(eq 'common-lisp::helloworld 'common-lisp-user::helloworld)
```

One special package is called `keyword`. Any symbol beginning with '`:`' is a keyword symbol. Moreover, every keyword symbol is external and has a constant value, namely itself. This makes keyword symbols convenient to use for symbolic data. For example, many functions use keyword symbols to indicate that certain optional arguments are being provided.

```
(describe 'keyword::foo)
(eq :foo ':foo)
(position #\l "hello world" :start 5) ; a "keyword argument"
```

To build modular systems, one defines a package for related functions and variables, and exports the API symbols. We won't go into further detail about packages in this tutorial; instead you may read *Common Lisp HyperSpec* section "Packages".

2.8 Lists

As we know, lists are a fundamental data structure in Lisp. They are the syntax for function calls, as well as the origin of the language's name (LISt Processing).

Lists hold a sequence of elements, which may be (references to) any Lisp objects. They are created by `list`.

```
'()
'(1 2 3)
(list 1 2 3)
```

Notes:

- Notice that `'()` ⇒ `nil`. The symbol `nil` represents the empty list as well as false.
- Lists are compared using `equal` (see Section 2.10 [Equality Testing], page 14).

Exercise: Construct the list containing the two lists `(a b c)` and `(1 2 3)`.

List functions are documented in *Common Lisp HyperSpec* section "Conses".

```
(listp nil)
(listp '(1 2 3))
(endp nil)
(endp '(1 2 3))
(eq (list 1 2 3) (list 1 2 3))
(equal (list 1 2 3) (list 1 2 3))

(length '(1 2 3))
(first  '(1 2 3))
(second '(1 2 3))
(third  '(1 2 3))
(rest   '(1 2 3))
(member 2 '(1 2 3))
(nth    4 '(0 1 2 3 4 5 6 7 8 9))
(nthcdr 0 '(0 1 2 3 4 5 6 7 8 9))
```

```
(nthcdr 4 '(0 1 2 3 4 5 6 7 8 9))
(last     '(0 1 2 3 4 5 6 7 8 9))
(last     '(0 1 2 3 4 5 6 7 8 9) 4)
(butlast  '(0 1 2 3 4 5 6 7 8 9))
(butlast  '(0 1 2 3 4 5 6 7 8 9) 4)
(append '(1 2 3) '(4 5 6))
```

Since lists are also sequences, the functions in *Common Lisp HyperSpec* section "Sequences" apply to lists.

Exercise: Define `(count-symbol` ***`symbol list`***`)` to return the number of times *symbol* occurs in *list*. For example, `(count-symbol 'a '(a b r a c a d a b r a))` ⇒ 5.

The basic building block of a list is called a *cons*. A cons is an object containing two elements — called the *car* and the *cdr* (for historical reasons). The syntax of a cons is `(`*`car`*` . `*`cdr`*`)`. You can picture the cons `(1 . 2)` as:

```
    +---+---+
    | / | \ |
    +/--+--\+
car /       \ cdr
   /         \
  V           V
+---+       +---+
| 1 |       | 2 |
+---+       +---+
```

Conses are used to construct lists (aka *linked lists*) in a familiar way (e.g., `(1 2 3)`).

```
+---+---+    +---+---+    +---+---+
| | | --+--->| | | --+--->| | | --+---> nil
+-|-+---+    +-|-+---+    +-|-+---+
  |            |            |
  |            |            |
  V            V            V
+---+        +---+        +---+
| 1 |        | 2 |        | 3 |
+---+        +---+        +---+
```

When we use conses as lists, we usually refer to the *car* and *cdr* as the *first* and *rest*, or *head* and *tail*. A list whose final *cdr* is not `nil` is called a *dotted list* — for example, `(1 2 . 3)`. A *proper list* has `nil` as its final *cdr*. The function `cons` creates cons cells. Since lists are linked cons cells, it follows that `cons` is also the function to add elements to the front of a list. The functions `car` and `cdr` access the elements of a cons, so they are equivalent to `first` and `rest`.

```
(cons 1 2)
(cons 1 nil)
'(1 . nil)
(cons 1 '(2 3))
(consp '(1 . 2))
(car   '(1 . 2))
(cdr   '(1 . 2))
```

```
(first '(1 . 2))
(rest  '(1 . 2))
```

Exercise: Implement `append`: Define `(my-append` ***`list1 list2`***`)` to append *list1* to *list2*.

2.9 Objects

This section introduces the Common Lisp Object System (CLOS). More detail about CLOS can be found in *Common Lisp HyperSpec* section "Objects".

2.9.1 Classes

Classes are defined by `defclass`. The definition consists of the class name, list of superclasses, list of slots, and other options. Documentation is found in *Common Lisp HyperSpec* section "Classes".

```
(defclass animal ()
  ((name :type string :reader name :initarg :name)
   (weight :type integer :accessor weight :initarg :weight)
   (covering :type symbol :reader covering :initarg covering)))

(defclass mammal (animal)
  ((covering :initform 'hair)))

(defclass bird (animal)
  ((covering :initform 'feathers)))

(defclass penguin (bird) nil)
```

Slot options for `defclass` include the following:

`:documentation`
: A documentation string.

`:allocation`
: Either `:instance`, meaning that this is a slot local to each instance, or `:class`, meaning that this slot is shared among instances.

`:initarg`
: A symbol used like a keyword in `make-instance` to provide the value of this slot.

`:initform`
: A form, evaluated when each instance is created, to give the value for this slot.

`:reader`
: A symbol which is made the name of a method used to get the value of this slot for each instance.

`:writer`
: A symbol which is made the name of a method used to set the value of this slot for each instance. For example, if `set-slot` is the symbol, it is used by evaluating `(set-slot value instance)`.

`:accessor`
: A symbol which is made the name of a method used to get or set the value of this slot for each instance.

`:type`
: The permitted type of values of this slot.

Instances are created by `make-instance`. Slots are accessed by their reader or accessor methods.

```
(setf willy (make-instance ’penguin :name "Willy" :weight 20))
(class-of willy)
(typep willy ’penguin)
(name willy)
(covering willy)
```

(The ‘`setf`’ used in this example is an assignment statement, assigning the variable `willy` the value resulting from `make-instance`. See Section 3.3 [Assignment], page 21.)

Exercise: Define a new class `fish`, which is a kind of `animal`, covered by `scales`.

Exercise: Create an instance, a 0.1 pound fish named “Nemo”.

All of the Lisp types we are already familiar with are built-in classes. The root of the class hierarchy is the type `t`. Here is a partial class hierarchy for the built-in classes.

```
t
   number
      real
         rational
            ratio
            integer
         float
            single-float
            double-float
      complex
   character
   sequence
      vector
         bit-vector
         string
      list
         null
         cons
   array
      vector
   symbol
      null
   function
   hash-table
   standard-object (the default superclass of instances)
```

2.9.2 Methods

CLOS employs *generic functions*, which are conceptually slightly different from member methods in languages like Java. A generic function is a function that dispatches to different *methods* based on the types of the arguments. Generic functions are covered in *Common Lisp HyperSpec* section “Generic Functions and Methods”.

```
(defgeneric moves-by ((a animal))
  (:documentation "Returns the mode of mobility of animal A.")
  (:method ((a mammal))
    "Mammals walk."
    'walking)
  (:method ((a bird))
    "Birds fly."
    'flying))

(setf tweety (make-instance 'bird :name "Tweety" :weight 0.1))
(moves-by tweety)
(moves-by willy) ;; oops

(defmethod moves-by ((a penguin))
  "Penguins swim (not fly)."
  'swimming)
(moves-by willy)
```

Methods can be defined using either `defgeneric` or `defmethod`.

Exercise: Define classes to represent geometric figures — circles, triangles, and rectangles.

Exercise: Define a generic function `area` to compute the area of a geometric figure.

2.9.3 Multiple Inheritance

CLOS allows a class to have multiple superclasses. This of course complicates the inheritance conceptually (see *Common Lisp HyperSpec* section “Determining the Class Precedence List”), so it should be used carefully.

```
(defclass carnivore (animal) nil)
(defclass herbivore (animal) nil)

(defgeneric eats-p ((predator animal) (prey animal))
  (:documentation "Returns true if PREDATOR will eat PREY.")
  (:method ((pred herbivore) (prey animal))
    nil)
  (:method ((pred carnivore) (prey herbivore))
    (< (weight prey) (weight pred)))
  (:method ((pred carnivore) (prey carnivore))
    (< (weight prey) (* 0.5 (weight pred)))))

(defclass cat (mammal carnivore) nil)
```

```
(defclass canary (bird herbivore) nil)

(setf sylvester (make-instance 'cat :name "Sylvester" :weight 10))
(setf tweety (make-instance 'canary :name "Tweety" :weight 0.1))

(eats-p sylvester tweety)
(eats-p tweety    sylvester)
(eats-p tweety    tweety)
(eats-p sylvester sylvester)
```

Exercise: What builtin classes have multiple superclasses?

2.9.4 Predefined Methods

CLOS predefines a few methods that are useful to specialize.

print-object *object stream* [Function]

Writes the printed representation of *object* to *stream*. The implementation dependent method for `standard-object` usually prints the class of the object. It is often overridden to print additional useful features of the object.

initialize-instance *instance* &rest *initargs* &key &allow-other-keys [Function]

Initializes a newly created instance. The predefined method initializes instance slots using the *initargs* and the `:initform` slot options. This primary method should not be overridden, but it is often convenient to define `:after` methods to perform additional initialization after the slots have been set up.

Examples:

```
(defmethod print-object ((a animal) stream)
  (print-unreadable-object (a :type t)
    (princ (name a) stream)))
```

2.10 Equality Testing

`eq`	Checks object identity. This is appropriate also for symbols, since symbols with the same name are the same object.
`eql`	Checks equality of characters and numbers of the same type. Additionally, if two objects are `eq`, they are also `eql`.
`equal`	Checks equality of strings and lists (recursively). Additionally, if two objects are `eql`, they are also `equal`.
`=`	Checks numerical equality.

2.11 Data Conversion

The function `coerce` converts data between compatible types.

```
(coerce 1/42 'float)
(coerce "Hello World" 'list)
```

3 Control Structures

3.1 Functional Programming

3.1.1 Functions are Objects

We already know quite a bit about functions — at least, named functions.

- Named functions are defined by `defun` forms.
- Functions are called by evaluating a list form whose first element is the function name — `(`*`function-name argument`*` ...)`.
- The form `(function `*`function-name`*`)` can be used to obtain a function object for a name; `#'function-name` is an abbreviation for `(function function-name)`.

What can we do with function objects in Lisp?

- Functions can be bound to variables, passed as arguments, and stored in data structures, just like other Lisp objects. Functions with these abilities are often referred to as "first class functions".
- Functions can be called on arguments *argument1 ... argumentn* using the form `(funcall `*`function argument1 ... argumentn`*`)`.
- Functions can be called on arguments *argument1 ... argumentn* and a list of additional arguments *arguments* using the form `(apply `*`function argument1 ... argumentn arguments`*`)`.

Examples:

```
(member '(a c) '((a b) (a c) (b c)) :test #'equal)

(setf z '(5 3 7 2))

(funcall
  (if (oddp (length z))
    #'first
    #'second)
  z)

(apply
  (if (< (first z) (second z))
    #'+
    #'-)
  z)

(defun funcall-nth (n functions arguments)
  "Calls the Nth function on the Nth argument."
  (funcall (nth n functions) (nth n arguments)))
```

```
(funcall-nth 1
  (list #'1+ #'string-upcase #'second)
  (list 42   "hello world"   '(a b c)))
```

Exercise: Define a function `(my-find-if predicate list)` that returns the first element of *list* which satisfies *predicate*, or `nil` if no element is satisfactory.

3.1.2 Anonymous Functions

Since function objects can be used so flexibly in Lisp, it makes sense that we should be able to create a function without having to define a named function; that is, use a throwaway function instead of a permanent `defun`. That's what `lambda` is for. A "lambda expression" can be used in place of a function name to obtain a function.

```
(function (lambda (x) (+ x 1)))
((lambda (x) (+ x 1)) 42)
(funcall #'(lambda (x) (+ x 1)) 42)
```

Note that

`((lambda` *`params`* `.` *`body`*`) .` *`args`*`)`

is equivalent to

`(funcall #'(lambda` *`params`* `.` *`body`*`) .` *`args`*`).`

In fact, the `function` form isn't really necessary, because `lambda` is set up so that

`(lambda` *`params`* `.` *`body`*`)` $\equiv$ `#'(lambda` *`params`* `.` *`body`*`).`

Lambda functions are actually closures, which means that they comprise not only their code, but also their lexical environment. So they "remember" variable bindings established at the time of their creation.

```
(defun make-adder (delta)
  (lambda (x) (+ x delta)))

(setf f (make-adder 13))
(funcall f 42)
(funcall (make-adder 11) (funcall (make-adder 22) 33))
```

Exercise: Define a function `(compose` *`f g`*`)` that composes functions *f* and *g*. Assume that *f* composed with *g* is defined as `(`*`f`* `o` *`g`*`)(`*`x`*`) =` *`f`*`(`*`g`*`(`*`x`*`))`. Try `(funcall (compose #'char-upcase #'code-char) 100)`.

3.1.3 Mapping

A very common use of a function object is calling it on each element of a list. This is referred to as *mapping*. Common Lisp includes several mapping functions.

```
(mapcar #'(lambda (s) (string-capitalize (string s)))
        '(fee fie foe fum))

(maplist #'reverse '(a b c d e))

(mapcar #'(lambda (s n) (make-list n :initial-element s))
'(a b c d e) '(5 2 3 7 11))

(mapcan #'(lambda (s n) (make-list n :initial-element s))
'(a b c d e) '(5 2 3 7 11))

(mapcon #'reverse '(a b c d e))
```

General sequences also support mapping. (Recall that sequences are one-dimensional collections, i.e., either vectors or lists.)

```
(map 'list #'(lambda (c) (position c "0123456789ABCDEF"))
     "2BAD4BEEF")

(map 'string #'(lambda (a b) (if (char< a b) a b))
     "Buckaroo" "Rawhide")
```

Here are examples of other useful functions on sequences. Many of these take functions as arguments.

```
(count-if #'oddp '(2 11 10 13 4 11 14 14 15) :end 5)

(setf x "Buckaroo Banzai")
(sort x #'(lambda (c d)
    (let ((m (char-code c)) (n (char-code d)))
      (if (oddp m)
                (if (oddp n) (< m n) t)
        (if (oddp n) nil (< m n))))))
;; note that SORT does destructive modification
x

(find-if
 #'(lambda (c) (= (* (first c) (first c)) (second c)))
 '((1 3) (3 5) (5 7) (7 9) (2 4) (4 6) (6 8)))

(position-if
 #'(lambda (c) (= (* (first c) (first c)) (second c)))
 '((1 3) (3 5) (5 7) (7 9) (2 4) (4 6) (6 8)))

(reduce #'expt '(3 3 2 2))
(reduce #'list '(a b c d e))
(reduce #'list '(a b c d e) :initial-value 'z)
(reduce #'list '(a b c d e) :from-end t)
(reduce #'append '((a b) (c d) (e f g) (h) (i j k)))
```

3.1.4 Lambda Lists

We have noted in passing that some functions take different numbers of arguments, or take arguments in strange ways. This section explains what's really going on.

A *lambda list* is a list of parameters for a function. They are described in *Common Lisp HyperSpec* section "Ordinary Lambda Lists". So far we have only used "normal" (or *required*) parameters in our functions, but there are five kinds of parameters a function might have. The syntax of a lambda list is as follows:

```
(var*
 [&optional {var | (var [init-form [supplied-p]])}*]
 [&rest var]
 [&key {var | ({var | (keyword-name var)} [init-form [supplied-p]])}*
      [&allow-other-keys]]
 [&aux {var | (var [init-form])}*])
```

Required Parameters

Required parameters are the normal formal parameters you are used to. There must be one actual argument for each required parameter, and the required parameters are bound to the values of the actual arguments in left-to-right order.

Optional Parameters

If there are more actual arguments than required parameters, the extras are bound to the optional parameters in left-to-right order. If there are extra optional parameters, they are bound to the value of their *init-form*, if it is present, else to `nil`. If *supplied-p* is present and there is an actual argument for the optional parameters, it is bound to `t`, otherwise it is bound to `nil`.

Rest Parameters

Using required and optional parameters, a Lisp function is restricted to a maximum number of actual arguments. But a `&rest` parameter *var* is bound to a list of the values of all the actual arguments that follow the required and optional arguments, however many.

Keyword Parameters

Keyword parameters are optional, but unlike `&optional` parameters, their arguments may appear in any order, and any one may be supplied or omitted independently of the others. A keyword parameter *var* is used in the body of the function as you would expect, but in the function call, a keyword argument is specified by preceding it with a symbol in the keyword package whose name is the same as the name of the var, that is, `:var`.

Aux Parameters

Aux parameters are essentially just local variables with initializations.

It is unusual (and a bit difficult) to use all five kinds of parameters together. More typical is to use required parameters with `&optional` and/or `&rest`, or to use required parameters with keyword parameters.

Examples:

```
(defun extract (list &optional index &rest indices)
  "Returns the elements of LIST at intervals specified by the indices.
The first INDEX gives the offset of the first element from the beginning
of the LIST.  Each remaining index in indices gives the offset of the
next element from the previous element.  For example,
(extract '(0 1 2 3 4 5 6 7 8 9) 2 3 2 1) => (2 5 7 8)"
  ;; keep going until either list or indices exhausted
  (if (and list index)
    ;; check if first element is one we want
    (if (zerop index)
      (cons (first list)
            ;; continue with next index from indices, if available
            (if indices
              (apply #'extract
                     (rest list) (1- (first indices)) (rest indices))
              nil))
      (apply #'extract (rest list) (1- index) indices))
    nil))

(defun group (list &key (size 1) (skip 0))
  "Returns a list of groups of elements of LIST.
Each group has SIZE elements and is separated from the elements
of the previous group by SKIP elements of LIST."
  (if (and list (<= size (length list)))
    (cons (subseq list 0 size)
          (group (subseq list (+ size skip)) :size size :skip skip))
    nil))
(group '(0 1 2 3 4 5 6 7 8 9))
(group '(0 1 2 3 4 5 6 7 8 9) :size 3 :skip 1)
(group '(0 1 2 3 4 5 6 7 8 9) :skip 2 :size 2)
```

3.2 Variables

3.2.1 Global Variables

Global variables in Lisp programs conventionally have names that begin and end with '`*`' (e.g., '`*package*`'). As always, the number of global variables in a program should be minimized to avoid confusion. Here are the forms that declare global variables:

defconstant *name initial-value* [*documentation*] [Macro]

Declares a global constant.

defparameter *name initial-value* [*documentation*] [Macro]

Declares a global variable.

defvar *name* [*initial-value* [*documentation*]] [Macro]

Like `defparameter`, declares a global variable. Unlike `defparameter`, does not assign the *initial-value* to the variable if it already has a value. This is primarily useful for customization variables, where a user may set these values before loading the system, and the defaults will only take effect when another value is lacking.

3.2.2 Local Variables

Local variables are introduced by `let`. For example, to avoid the repeated computation of the first elements in `my-merge1` below,

```
(defun my-merge1 (list1 list2 &key (test #'<))
  "Merges the elements of LIST1 and LIST2 together in order.
Assumes both LIST1 and LIST2 are already in order.
Comparison is performed by calling TEST on X1 and X2,
where X1 is from LIST1 and X2 is from LIST2."
  (if (and list1 list2)
    (if (funcall test (first list1) (first list2))
      (cons (first list1) (my-merge1 (rest list1) list2 :test test))
      (cons (first list2) (my-merge1 list1 (rest list2) :test test)))
    (or list1 list2)))
```

we introduce local variables *x1* and *x2*.

```
(defun my-merge (list1 list2 &key (test #'<))
  "Merges the elements of LIST1 and LIST2 together in order.
Assumes both LIST1 and LIST2 are already in order.
Comparison is performed by calling TEST on X1 and X2,
where X1 is from LIST1 and X2 is from LIST2."
  (if (and list1 list2)
    (let ((x1 (first list1))
          (x2 (first list2)))
      (if (funcall test x1 x2)
        (cons x1 (my-merge (rest list1) list2 :test test))
        (cons x2 (my-merge list1 (rest list2) :test test))))
    (or list1 list2)))
```

The general form of a let expression is:

```
(let ((v1 e1)
      (v2 e2)
      ...
      (vn en))
  expression
  ...)
```

The variables *v1* through *vn* are bound to the results of the expressions *e1* through *en*. These bindings have effect throughout the body *expression*s. As usual, the result of the `let` expression is the result of the last body expression.

Notes:

- Let bindings are lexically scoped.

```
(let ((x 1))
  (list
    (let ((x 2))
      x)
    (let ((x 3))
      x)))
```

- Let bindings are performed in parallel.

  ```
  (let ((x 3))
    (let ((x (1+ x))
          (y (1+ x)))
      (list x y)))
  ```

- A variant of `let` called `let*` performs the bindings sequentially.

  ```
  (let ((x 3))
    (let* ((x (1+ x))
           (y (1+ x)))
      (list x y)))
  ```

Exercise: Finish implementing mergesort by defining `(mergesort` *`list`* `&key` *`test`*`)`. If the list has zero or one elements, return it; otherwise split the list in half, mergesort each half, and merge them. Use `let` to define any local variables you need (e.g., the midpoint of the list). (Hint: Use `(floor` *`n`* `2)` to get the closest integer to $n/2$.)

3.3 Assignment

Assignment in Lisp is handled by `setf`, and is much more general than in other languages.

setf {*place value*}* [Macro]

Assigns to each *place* its corresponding *value*, in sequential order. Returns the result of the last assignment (i.e., the last *value*).

psetf {*place value*}* [Macro]

Assigns to each *place* its corresponding *value*, in parallel. Returns `nil`.

A *place* is a *generalized reference* — a form that corresponds to a location that can hold a value. (See *Common Lisp HyperSpec* section “Overview of Places and Generalized References”.) Here are some examples of places and assignment.

```
;; simple variables
(defparameter *answer* 42)
(setf *answer* 'no)

;; list element accessors
(setf z '(a b c))
(setf (first z) 1)
(setf (rest z) '(2 3))

;; object slot accessors
(setf (weight tweety) 0.2) ; tweety starts pumping iron
```

Lisp style is conventionally more functional than imperative, so Lisp programmers tend to minimize their use of `setf`.

Throughout this document, we have used `setf` to assign variables during interactions with the Lisp interpreter. This has the effect of introducing a global variable without declaring it. This is acceptable practice for interacting with interpreter. However, when you write more formal programs (e.g., the sort you save in a source file), always declare your global variables. And again, try to minimize your use of global variables.

In addition to explicit assignment using `setf`, there are some useful macros that implicitly perform assignments.

```
;; (incf place &optional (delta 1)) ≡ (setf place (+ place delta))
(setf x 42)
(incf x)
x
(decf x 3)
x

;; (pop place) ≡
;; (let ((x (car place)))
;;   (setf place (cdr place))
;;   x)

;; (push item place) ≡ (setf place (cons item place))
(setf z '(a b c))
(pop z)
z
(push 'c z)
z
```

3.4 Conditionals

As we know, the basic conditional expression, `if`, is a two-branch conditional. There are also single-branch conditionals `when` and `unless`, and a multi-branch conditional `cond`.

when *test expression** [Macro]

Evaluates *test*. If the test result is true, evaluates the *expressions* in order, returning the result of the last one. Otherwise returns `nil`.

unless *test expression** [Macro]

Evaluates *test*. If the test result is false, evaluates the *expressions* in order, returning the result of the last one. Otherwise returns `nil`.

cond *branch** [Macro]

The form of the multi-branch conditional is:

```
(cond
 (expression11 expression12 ...)
 (expression21 expression22 ...)
 ...
 (expressionn1 expressionn2 ...))
```

The *expressioni1* are evaluated starting at $i = 1$ until one of them evaluates to a non-null value. If so, the rest of the expressions in that group (if any) are evaluated, and the value of the last one evaluated becomes the value of the `cond`. If all of the *expressioni1* evaluate to `nil`, then the value of the cond is `nil`. As is often the case, the value of a Lisp expression is the value of the last subexpression evaluated under its control.

Most frequently, cond is thought of as a multi-branch `if`:

```
(cond
 (test1 expression1 ...)
 (test2 expression2 ...)
 ...
 (testn expressionn ...))
```

The last test may be `t`, when it is to be considered the default clause.

Examples:

```
(defun my-elt (list index)
  "Returns the INDEXth element of LIST, or nil if there isn't one."
  (cond
   ((endp list)
    nil)
   ((zerop index)
    (first list))
   (t
    (elt (rest list) (1- index)))))
```

Exercise: Try defining `(my-position` *`elt list`* `&key` *`test`*`)` to return the index of *elt* in *list* using `cond`. Use the *test* keyword argument to compare *elt* to elements of *list*; the default should be `#'eql`.

3.5 Loops

Common Lisp has one iteration macro — `loop` — that subsumes the kinds of iteration constructs found in most languages (e.g., while, for, foreach).

The form of a `loop` is quite simple: it is a series of *loop clauses*.

```
(loop
  clause1
  clause2
  ...)
```

However, there are some 25 different kinds of loop clauses, so it takes a while to appreciate all the things that `loop` can do. Instead of an exhaustive list, we'll just look at some of the most commonly used clauses.

1. Numerical iteration: `for` *`var`* `from` *`start`* `[to` *`end`*`] [by` *`incr`*`]`

   ```
   (loop for i from 99 downto 66 by 3
     do (print i))
   ```

   Alternatives to the '`to`' (or '`upto`') limit are '`downto`', '`below`', and '`above`'.

2. List iteration: `for` *`var`* `{in|on}` *`list`* `[by` *`step-fun`*`]`

   ```
   (loop for x in '(a b c d e)
     do (print x))
   ```

```
(loop for x on '(a b c d e)
  do (print x))
```

3. General iteration: `for` *`var`* `=` *`init-expr`* `[then` *`update-exrp`*`]`

   ```
   (loop
     for x from 0 below 10
     for y = (+ (* 3 x x) (* 2 x) 1)
     do (print (list x y)))

   (let ((z '(4 2 0 1 3)))
     (loop repeat 5
       for prev = 0 then next
       for next = (nth prev z)
       do (print next)))
   ```

4. Value accumulation: `{collect|append}` *`expr`* `[into` *`var`*`]`

   ```
   (loop for r on '(a b c d e)
     collect (length r)
     append r)
   ```

   Value accumulation clauses construct either a list value or a numeric value: The '`collect`' and '`append`' clauses construct a list value; '`count`', '`sum`', '`minimize`', and '`maximize`' construct a numeric value. Either way, the value is returned by the loop.

5. Initial-final: `{initially|finally}` *`expr*`*

   ```
   (loop
     initially (princ "testing")
     repeat 10 do
     (sleep 0.5)
     (princ #\.)
     finally (princ "done"))
   ```

   Another way to return a value from the loop is using '`finally`'.

   ```
   (defun my-expt (base exponent)
     (loop repeat (1+ exponent)
       for x = 1 then (* x base)
       finally (return x)))
   ```

6. Unconditional execution: `do` *`expr*`*

   ```
   (loop repeat 10 do (print "ha"))
   ```

7. Conditional execution:

   ```
   if test
     selectable-clause {and selectable-clause}*
   [else
     selectable-clause {and selectable-clause}*]
   [end]
   ```

   A *selectable-clause* is a value accumulation clause, or unconditional or conditional execution clause.

   ```
   (loop for x in '(1 (2 3) 4 (5 6) 7 8)
     if (listp x)
   ```

```
          sum (apply #'* x)
        else
          sum x)
```

8. Termination test: `{while|until}` ***`test`***

   Another termination test clause, which we have already seen, is `repeat` ***`number`***.

   ```
   (defun user-likes-lisp-p ()
     (loop initially (princ "Do you like Lisp? ")
       for x = (read)
       until (member x '(y n))
       do (princ "Please type either `y' or `n'. ")
       finally (return (eq x 'y))))
   ```

Exercise: Rewrite the factorial function using `loop`.

Exercise: Define a function `(fibonacci-table` *`n`*`)` to return a list of the Fibonacci numbers F[1] ... F[*n*].

Exercise: Implement `assoc` using `loop`. Specifically, define a function `(my-assoc item assoc-list &key test)` that returns the mapping whose key is *item*, or `nil` if there isn't one. An *association list* is a list of conses `(`*`key`*` . `*`value`*`)`. For example,

```
(my-assoc 'b '((a . 1) (b . 2) (c . 3)) ⇒ (b . 2).
```

4 Input/Output

4.1 Streams

I/O in Lisp is based on streams. A stream is a source or destination for characters or bytes. For example, streams can be directed to or from files, strings, or the terminal. Output functions (e.g., `print` and `format`) and input functions (e.g., `read`) normally take stream arguments; although frequently the stream argument is optional. Several streams are available when Lisp starts up, including `*standard-input*` and `*standard-output*`. If the session is interactive, both of these are the same as `*terminal-io*`.

The basic output functions for streams are `write-char` and `write-line`. The basic input functions are `read-char` and `read-line`.

File streams are created by the `open` function. However, it is more convenient to use the `with-open-file` form, which ensures that the file is closed regardless of whether control leaves normally or abnormally.

```
(with-open-file (output-stream "/tmp/drpierce.txt" ; put your name here
                 :direction :output)
  (write-line "I like Lisp" output-stream))

(with-open-file (input-stream "/tmp/drpierce.txt" :direction :input)
  (read-line input-stream))

(with-open-file (output-stream "/tmp/drpierce.txt"
                 :direction :output
                 :if-exists :supersede)
  (write-line "1. Lisp" output-stream))

(with-open-file (output-stream "/tmp/drpierce.txt"
                 :direction :output
                 :if-exists :append)
  (write-line "2. Prolog" output-stream)
  (write-line "3. Java" output-stream)
  (write-line "4. C" output-stream))

;; read lines until eof
(with-open-file (input-stream "/tmp/drpierce.txt" :direction :input)
  (loop for line = (read-line input-stream nil nil)
    while line
    collect line))
```

Similarly, a string stream is usually manipulated using `with-output-to-string` and `with-input-from-string`.

```
(with-output-to-string (output-stream)
  (loop for c in '(#\L #\i #\s #\p)
    do (write-char c output-stream)))
```

```
(with-input-from-string (input-stream "1 2 3 4 5 6 7 8 9")
  (loop repeat 10 collect (read-char input-stream)))
```

Although the basic I/O functions are available, you will normally invoke the higher-level capabilities of the Lisp printer and the Lisp reader. We discuss the printer (see Section 4.2 [The Printer], page 27) and reader (see Section 4.3 [The Reader], page 29) in the following sections.

Streams are closed using `close`. Other stream functions include `streamp`, `open-stream-p`, `listen`, `peek-char`, `clear-input`, `finish-output`.

4.2 The Printer

The standard entry point into the printer is the function `write`; and `prin1`, `princ`, `print`, and `pprint` are wrappers for certains settings of `write`. The optional output stream argument of each of these functions defaults to standard output. Another useful set of print functions is `write-to-string`, `prin1-to-string`, and `princ-to-string`.

```
(setf z
  '("animal"
    ("mammal"
     ("feline" ("lion") ("tiger") ("kitty"))
     ("ursine" ("polar bear") ("teddy bear"))
     ("rodent" ("squirrel") ("bunny") ("beaver")))
    ("bird" ("canary") ("pigeon"))
    ("reptile" ("turtle") ("snake"))))
(prin1 z) ; same as (write z :escape t)
(princ z) ; same as (write z :escape nil :readably nil)
(write z :escape nil :pretty t :right-margin 40)
(write-to-string z :escape nil :pretty nil)
```

A more sophisticated and flexible aspect of the printer is the `format` function – `(format` ***`destination control-string argument`***`...)`. This function consults the *control-string* to determine how to format the remaining arguments (if any) and transfers the output to *destination*.

If the destination is:	then the output:
a stream	appears on that stream
`t`	appears on the standard output
`nil`	is returned as a string

The control string consists of simple text, with embedded format control directives. Some of the simpler, more commonly used directives are summarized below.

~W	format as if by write; any kind of object; obey every printer control variable
~S	format as if by prin1; any kind of object; "standard" format
~A	format as if by princ; any kind of object; human readable ("asthetic") format
~D	(or ~B, ~O, ~X) decimal (or binary, octal, hex) integer format

~F (or ~E, ~G, ~$) fixed-format (or exponential, general, monetary) floating-point format

~{*control-string*~}
: format a list; repeatedly uses *control-string* to format elements of the list until the list is exhausted

~% print a newline

~& print a newline unless already at the beginning of the line

~~ print a (single) tilde

~* ignore the corresponding argument

~*newline* ignore the newline and any following whitespace (allows long format control strings to be split across multiple lines)

Many format control directives accept "arguments" — additional numbers or special characters between the '~' and the directive character. For example, a common argument allowed by many directives is a column width. See the documentation for individual directives for details about their arguments.

```
;; format an invoice
(loop for (code desc quant price) in
  '((42 "House" 1 110e3) (333 "Car" 2 15000.99) (7 "Candy bar" 12 1/4))
  do (format t "~3,'0D ~10A ~3D @ $~10,2,,,'*F~%"
               code desc quant price))

;; format an invoice again, one-liner
(format t "~:{~3,'0D ~10A ~3D @ $~10,2,,,'*F~%~}"
  '((42 "House" 1 110e3) (333 "Car" 2 15000.99) (7 "Candy bar" 12 1/4)))

;; comma-separated list
(loop for i from 1 to 4 do
  (format t "~{~A~^, ~}~%" (subseq '(1 2 3 4) 0 i)))

;; comma-separated list again, but cleverer
;; (using things I didn't mention above :-)
(loop for i from 1 to 4 do
  (format t "~{~A~#[~; and ~:;, ~]~}~%" (subseq '(1 2 3 4) 0 i)))
```

Exercise: Remember that CLOS objects have "unreadable" printed representations? Let's fix that now. Define a function (or generic function) `(print-geom` ***figure*** `&optional` ***stream***`)` to print a suitable representation for geometric figures (circles, triangles, rectangles). "Suitable" means a representation that will make it easy for you to define a `(read-geom &optional` ***stream***`)` function to reconstruct a geometric object. For example, `(circle :radius 42)` might be a good representation for circles (particularly if `:radius` is the initarg for the radius slot). The *stream* should default to `*standard-output*`.

4.3 The Reader

The standard entry point into the reader is the function `read`. The function `read-from-string` is also often convenient.

```
(with-input-from-string (input-stream "(a b c)")
  (read input-stream))

(with-input-from-string (input-stream "5 (a b) 12.3 #\\c \"foo\" t")
  (loop repeat (read input-stream)
    do (describe (read input-stream))))
```

Exercise: Define `(read-geom &optional` *`stream`*`)` function to reconstruct a geometric object from the output produced by `print-geom`. The *stream* should default to `*standard-input*`. (Hint: if you used a printed representation like `(circle :radius 42)`, then you should be able to reconstruct the circle using `(apply #'make-instance` *`r`*`)`, where *r* is that list representation.

5 Mutable Data Structures

5.1 Arrays

Notes:

- The syntax for arrays is `#nA(rows)`, where *n* is the rank (or dimensionality) of the array.
- Elements of an array are accessed by `aref`.
- Arrays can also be created using `make-array`.

Examples:

```
;; a 2-dimensional array
(setf a
  #2A((0 1 2)
      (1 2 3)
      (2 3 4)))
(aref a 1 1)
(setf (aref a 1 1) 42)
a
;; a 3-dimensional array
;; 2 rows, 2 columns, 3 subcolumns
(setf b
  (make-array '(2 2 3) :initial-contents
    '(((a b c) (d e f))
      ((g h i) (j k l)))))
(aref b 1 0 2)
b
```

5.2 Hash Tables

Notes:

- Hash tables are created by `(make-hash-table &key test)`.
- The *test* may be `eq`, `eql`, `equal`, or `equalp`. The default is `eql`.
- Elements are accessed using `(gethash key hash-table &optional default-value)`.

Examples:

```
(setf h (make-hash-table))
(setf (gethash 'a h) 1)
(setf (gethash 'b h) 2)
(gethash 'a h)
```

Appendix A Exercise Solutions

This appendix contains solutions to the exercises organized by sections.

Section 2.2 [Numbers], page 5

```
(/ (+ 12 17) 2)
(float (/ (+ 12 17) 2))
```

Section 2.3 [Functions], page 5

```
(defun fibonacci (n)
  "Returns the Nth Fibonacci number."
  (if (<= n 2)
    1
    (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))

(fibonacci 10)
```

Section 2.4 [Characters], page 6

```
(defun char-1+ (character)
  "Returns the character whose code is one more than CHARACTER's."
  (code-char (1+ (char-code character))))
```

Section 2.5 [Strings], page 6

```
(defun string-1+ (string)
  "Returns the string formed by adding 1 to each char-code of STRING."
  (if (zerop (length string))
    string
    (concatenate 'string
      (string (char-1+ (char string 0)))
      (string-1+ (subseq string 1)))))
```

Section 2.8 [Lists], page 9

```
'((a b c) (1 2 3))
(list '(a b c) '(1 2 3))

(defun count-symbol (symbol list)
  "Returns the number of occurrences of SYMBOL in LIST."
  (if (endp list)
    0
    (+ (if (eq symbol (first list)) 1 0)
       (count-symbol symbol (rest list)))))
```

```
(defun my-append (list1 list2)
  "Appends LIST1 and LIST2."
  (if (endp list1)
    list2
    (cons (first list1) (append (rest list1) list2))))
```

Section 2.9.1 [Classes], page 11

```
(defclass fish (animal)
  ((covering :initform 'scales)))

(make-instance 'fish :name "Nemo" :weight 0.1)
```

Section 2.9.2 [Methods], page 13

```
(defclass circle ()
  ((radius :accessor radius :initarg :radius)))

(defclass rectangle ()
  ((width :accessor width :initarg :width)
   (height :accessor height :initarg :height)))

(defclass triangle ()
  ((base :accessor base :initarg :base)
   (height :accessor height :initarg :height)))

(defmethod area ((c circle))
  (* pi (radius c) (radius c)))

(defmethod area ((r rectangle))
  (* (width r) (height r)))

(defmethod area ((x triangle))
  (* 0.5 (base x) (height x)))
```

Section 2.9.3 [Multiple Inheritance], page 13

From the diagram in Section 2.9.1 [Classes], page 11, `vector` and `null`.