THROUGHOUT, use only the features of Lisp that are described in A Small Part of Common Lisp.

Functions that you define can assume that the arguments are the correct type (lists, numbers, or whatever is specified). Do not spend time making your functions check the types of the arguments.

1 Evaluating Expressions

1.1 Arithmetic and Simple Evaluation

Exercise 1.1.1 Evaluate:

(+ 2 3)
(+ (- 3 2) (* 4 6))
(* 20 (* 20 20))
(/ 4 2)
(/ (+ 1 2))
(/ (+ 2 3) (- 5 2))
(/ 4.5 2)

Exercise 1.1.2 Evaluate:

(+ 2 3)
(+ 2 3)
'(2 3 (+ 2 4))
(2 3 (+ 2 4))
'(2 3 ,(+ 2 4))

Exercise 1.1.3 Express in Lisp:

\[
(2 + 3) \times (4 - 5)
\]

\[
4 \times 5
\]

\[
\frac{15}{2}
\]

1.2 Binding Values to Symbols

Exercise 1.2.1 Evaluate these expressions in this order, as if they were typed into the computer in sequence, starting with a newly opened Lisp session. If an expression cannot be evaluated, say so.

(setq qwerty '(this is a list))
(setq x qwerty)
(setq y '(+ 2 3))
(eval y)
(eval (eval y))
(setq z'qwerty)
(setq w 'z)
(eval w)
(eval (eval w))
(eval (eval (eval w)))
(setq q '(setf z y))

(setq q '(setq z y))

(setq z (eval q))

(eval z)
Exercise 1.2.2 Evaluate these expressions in this order, as if they were typed into the computer in sequence, starting with a newly opened Lisp session.

(setf y '(try this))
(setf a 'setf)
(setf b (cons a '(y 300)))
b
eval b)
y

1.3 Types of S-Expressions

Exercise 1.3.1 Each of the following lines either is or is not an S-expression. If it is an S-expression, identify what type (e.g., symbol, list, etc.). If it is not an S-expression, say so.

this-is-one
cons
CONS
"C:\My Documents"
23.4
23/4
(blah blah)
(x "23" 23)
(this (and this (and this)))
this and (this and) this
sqrt(X)
(SQRT X)

1.4 List Manipulation

Exercise 1.4.1 Evaluate:

(first '(z y x w))
(rest '(z y x w))
(first '((a b c)))
(rest '((a b c)))
(cons 'a '(b c))
(cons '(a b) '(c d))
(cons (first '(a b c)) (rest '(a b c)))

Exercise 1.4.2 Suppose the variable x is bound to a 10-element list. Give an expression containing x whose value will be the 5th element of that list.

Exercise 1.4.3 Suppose x is bound to the complex list (((a b c) d) e). Give an expression containing x whose value will be c.

Exercise 1.4.4 Which of these is not a 3-element list?

(a b c)
((a b c) (10 20 30) (a b c))
((a b c) (10 20 30) a b c)

2 Defining Functions

2.1 Simple Arithmetic

Exercise 2.1.1 The formula for converting Fahrenheit to Celsius temperatures is:

\[ C = \frac{F - 32}{1.8} \]

Define a function named F-T0-C that performs this conversion, so that, for example:

(F-T0-C 68) \Rightarrow 20.

Exercise 2.1.2 Define a function named POLY that takes 4 arguments, a, b, c, and x, and computes \[ ax^2 + bx + c. \] For example:

(POLY 4 9 6 10) \Rightarrow 496.

Exercise 2.1.3 Define a function VOLBOX that takes 3 arguments, the length, width, and depth of a rectangular box (in feet), and computes the volume of the box (in cubic feet), so that, for example:

(VOLBOX 1.5 2 10) \Rightarrow 30

Exercise 2.1.4 The same, but this time name the function VOLBOXLIST and have the argument be a 3-element list of numbers giving the 3 dimensions of the box, like this:

(VOLBOXLIST '(1.5 2 10)) \Rightarrow 30
2.2 Logic and Conditionals

Exercise 2.2.1 Are expressions such as (NOT (NOT X)) useful? Is there any situation in which the value of (NOT (NOT X)) is different from the value of X?

Exercise 2.2.2 Define a function named MAX2 that gives the larger of its two arguments, which are numbers, like this:

\[
\begin{align*}
\text{(MAX2 35 20)} & \Rightarrow 35 \\
\text{(MAX2 (+ 30 5) (* 10 2))} & \Rightarrow 35
\end{align*}
\]

Exercise 2.2.3 Define a function named MAX3 that is like MAX2 except that it takes three arguments and gives the largest of the three numbers:

\[
\begin{align*}
\text{(MAX3 20 50 10)} & \Rightarrow 50 \\
\text{Hint: MAX3 can call MAX2.}
\end{align*}
\]

Exercise 2.2.4 Define a function named SAME-FIRST that takes 2 lists as arguments, and returns T if the two lists have the same first element, and NIL if they do not. Examples:

\[
\begin{align*}
\text{(SAME-FIRST '(A B C) '(A D E))} & \Rightarrow T \\
\text{(SAME-FIRST '(A B C) '(A))} & \Rightarrow T \\
\text{(SAME-FIRST '(A B C) '(B D E))} & \Rightarrow \text{NIL} \\
\text{(SAME-FIRST (REST '(A B C)) '(B D E))} & \Rightarrow T
\end{align*}
\]

Exercise 2.2.5 Define a function named ZERO-OF-3 that takes 3 arguments, which can be S-expressions of any type, and returns T if at least one of the arguments is the number 0, and NIL otherwise. Examples:

\[
\begin{align*}
\text{(ZERO-OF-3 'A 0 '(b c))}\Rightarrow T \\
\text{(ZERO-OF-3 30 40 0)} & \Rightarrow T \\
\text{(ZERO-OF-3 1 2 3)} & \Rightarrow \text{NIL} \\
\text{(ZERO-OF-3 'A 'B 'C)} & \Rightarrow \text{NIL}
\end{align*}
\]

Exercise 2.2.6 Define a function named ALL-ZERO-3 that takes 3 arguments, which can be S-expressions of any type, and returns T if all three of them are the number 0, and NIL otherwise.

3 List Handling

3.1 Without Recursion

Exercise 3.1.1 Define a function named E5 whose argument is a list with at least 5 elements, which returns the 5th element, like this:

\[
\text{(E5 '(a b c d e f g))}\Rightarrow e
\]

Exercise 3.1.2 Define a function named SWAP-FIRST-TWO that takes one argument, which is a list, and creates a list like it but with the first two elements swapped, thus:

\[
\text{(SWAP-FIRST-TWO '(A B C D E))}\Rightarrow (B A C D E)
\]

This should work for lists of any length ≥ 2.

Exercise 3.1.3 Define a function named IA (standing for “infix arithmetic”) which takes a 3-element list, swaps the first 2 elements, and then evaluates the list. This enables you to do things like this:

\[
\text{(IA '(2 + 3))}\Rightarrow 5
\]

Note: The spaces are necessary; do not write (2+3) or the like.

Because it is not recursive, this version of IA can’t do this:

\[
\text{(IA '((2 + 3) * (4 + 5))) (ERROR)}
\]

You may, however, want to think about how to make it do so.

3.2 With Recursion

Exercise 3.2.1 Define a function called TRIPNUM that takes a list of numbers and returns a similar list in which all the numbers have been tripled:

\[
\text{(TRIPNUM '(10 100 6))}\Rightarrow (30 300 18)
\]
Exercise 3.2.2 Define a function called EVEN-COUNT that takes as its argument any list, and returns T if the list has an even number of elements, and NIL otherwise:

(EVEN-COUNT '(A B C D)) ⇒ T
(EVEN-COUNT '(A B C D E)) ⇒ NIL

Hint: Do not count the elements. Just take them off 2 at a time. If the list has an even number of elements, you will end up with nil; otherwise you will end up with a one-element list.

Exercise 3.2.3 Define a function called CONTAINS-LIST that takes as its argument a list, and returns T if any element of that list is a list, or NIL otherwise:

(CONTAINS-LIST '(A B (C D) E)) ⇒ T
(CONTAINS-LIST '(A B C D E)) ⇒ NIL

Exercise 3.2.4 Define a function called MAX-LIST that takes as its argument a list of numbers and returns the largest number:

(MAX-LIST '(5 4 9 4 6)) ⇒ 9

Exercise 3.2.5 Define a function called DUP-ELEMENT that takes 2 arguments, an element and a number ≥ 1, and returns a list containing that element repeated that number of times, like this:

(DUP-ELEMENT 'A 5) ⇒ (A A A A A)
(DUP-ELEMENT '(A B) 3) ⇒ ((A B) (A B) (A B))

Hint: Here you are not looking for the end of the list; you are looking for the number to be ≤ 1. To get started, consider what the function should do if the number is 1, and then what it should do if the number is larger.

Exercise 3.2.6 Define a function named RIA that is the recursive version of the function you defined in Exercise 3.1.3. That is, it should be able to handle not only (2 + 3) but also expressions like (2 + (3 * 4)), ((2 + 3) - (3 + 1)), and so on, thus:

(RIA '((2 + 3) - (3 + 1))) ⇒ 1

Note: The spaces are necessary; do not write (2+3) or the like.

Here’s how RIA should work. The argument of RIA is always either a number or a 3-element list. If it’s a number, return it unchanged. If it’s a 3-element list, create a new list as you did before, except that you use RIA to process the first and third elements so that they, too, get swapped around (as do their own first and third elements if they are lists, and so on).

To show how RIA works, you may want to try it out without the final EVAL step, so you can see the rearranged expression rather than its value.

4 Higher-Order Programming

4.1 Functions that Define Functions

Exercise 4.1.1 Define a function MAKE-MULTIPLIER that takes 2 arguments, a symbol and a number, and defines a new function whose name is that symbol and whose effect is to multiply numbers by that number.

(To do its job, MAKE-MULTIPLIER will have to create a DEFUN and then evaluate it.)

Example:

(MAKE-MULTIPLIER 'KILOGRAMS-TO-POUNDS 2.206) ⇒ KILOGRAMS-TO-POUNDS

Then you can do this:

(KILOGRAMS-TO-POUNDS 100) ⇒ 220.6

Another example:

(MAKE-MULTIPLIER 'CM-TO-INCH (/ 1 2.54)) ⇒ CM-TO-INCH

Then you can do this:

(CM-TO-INCH 50) ⇒ 19.68504

— END —