Introduction to Computer Science via Scheme

Luke Hunsberger

© 2018 Luke Hunsberger
All rights reserved

Spring 2018
Contents

1 Introduction

I Non-Destructive Programming in Scheme

2 Scheme Expressions vs. Scheme Data
   2.1 Primitive Data Expressions
      2.1.1 Numbers
      2.1.2 Booleans
      2.1.3 The Empty List (or Null)
      2.1.4 The Void Datum
      2.1.5 Symbols
   2.2 String Expressions
   2.3 Summary

3 Evaluating Scheme Data
   3.1 Evaluating Numbers, Booleans, the Empty List, the void Datum, and Strings
   3.2 Evaluating Symbols
   3.3 Summary

4 Introduction to DrScheme
   4.1 Entering Expressions into the Interactions Window
   4.2 DrScheme’s Run Button
   4.3 Summary

5 Built-In Functions
   5.1 Built-in Functions for Arithmetic
   5.2 Contracts
   5.3 Built-in Functions for Integer Arithmetic
   5.4 The Built-in eval Function
   5.5 The Built-in Functions printf and void
      5.5.1 The printf Function
      5.5.2 The void Function
   5.6 Summary

6 Non-Empty Lists
   6.1 The Syntax and Semantics for Non-Empty Lists
   6.2 Evaluating Non-Empty Lists: the Default Rule
      6.2.1 A More Formal Description of the Default Rule
   6.3 Summary
# List of Figures

1.1 A function with no side effects (left) vs. a function with side effects (right) .................. 2
2.1 The universe of Scheme data, partitioned according to data type ................................. 8
4.1 A programmer’s interaction with DrScheme ................................................................. 16
4.2 Using the Run button to evaluate (the data denoted by) multiple expressions ................... 18
6.1 The evaluation of the list denoted by (+ 2 3) ............................................................. 28
6.2 DrScheme’s use of different colors to distinguish side-effect printing from output values .... 31
7.1 The side effect of define: inserting a new entry into the Global Environment .................. 35
9.1 Applying the function corresponding to (lambda (x) (* x x)) to the value 4 ................... 53
9.2 Applying the sphere-volume function to the value 2 .................................................... 54
12.1 DrScheme’s evaluation of (facty-v1 3) ............................................................. 79
13.1 Truth tables for AND, OR and NOT, expressed using truth values (above) and binary values (below) 87
14.1 DrScheme’s evaluation of (sum-to-n-acc 3 0) ............................................................ 106
15.1 Demonstrating that local environment entries have higher priority than Global Environment entries 120
16.1 The non-empty list, (3 4 6), as a single cons cell—with very particular contents ............ 169
16.2 An alternative depiction of the non-empty list, (3 4 6), as a chain of cons cells ............. 170
17.1 The environments for Example 17.1.2 before (left) and after (right) evaluating the set! expressions 184
17.2 The environments for Example 15.5.2 before (left) and after (right) evaluating the set! expression 186
18.1 A scattered chain of cons cells representing the list (32 7 #f xy w) .............................. 200
18.2 Accessing the elements of a vector by their numerical indices ................................. 200
19.1 Partitioning a vector of cards .................................................................................. 216
20.1 States and transitions in the game of dice-dice ......................................................... 222
A.1 A portion of a typical file system .............................................................................. 238
A.2 The organization of files and directories for your work in CMPU-101 ......................... 238
B.1 Using the cd command to navigate through a directory tree in the hunsberg account .... 246
Chapter 1

Introduction

Most kinds of communication are based on some kind of language, whether written, spoken, drawn or signed. To be used successfully, the syntax and semantics of a language must be (explicitly or implicitly) understood.

- The syntax rules of a language specify the legal words, expressions, statements or sentences of that language.
- The semantic rules of a language specify what the legal words, expressions, statements or sentences mean.

For example, the syntax rules of the English language tell us that person, tall, told, the, a, me and joke are legal words, and that “The tall person told me a joke” is a legal sentence, whereas pkrs, shrel and fidafid are not legal words, and “Person tall told a me the” is not a legal sentence. The semantic rules tell us what each of the words mean (e.g., what objects the nouns denote and what processes the verbs convey), as well as what the entire sentence means (e.g., that a particular tall person told me a joke). For another example, the syntax rules of French tell us that je, vais, au, tableau and noir are legal words, and that “Je vais au tableau noir” is a legal sentence; and the semantic rules tell us that this sentence means that I am going to the blackboard.

Just as people use so-called natural languages (e.g., English or French) to communicate with one another, people use programming languages to communicate with computers. Over the years, many programming languages have been introduced, having names such as Java, Scheme, Python, C, C++, Fortran, Lisp, Haskell, Basic, Algol, Javascript, and many others. Like any natural language, each programming language has an associated set of syntax rules that specify the legal expressions (or statements or sentences or programs) that can be used in that language, and a set of semantic rules that specify what the legal expressions mean.

• The meaning of a computer program includes the data denoted by expressions, the computations to be performed on that data, and any auxiliary actions to be done (e.g., printing information on the computer screen or changing the value of a variable stored in the computer’s memory).

For most computer programming languages, the constituents of the language, whether they are called expressions, statements or entire programs, usually comprise sequences of typewritten characters. For example, the following character sequences are legal building blocks of a Java program:

• `int x = 5;`
• `for (int i=0; i < 5; i++) System.out.println(i);`
• `public class Sample { }`

And the following character sequences are legal building blocks of a Scheme program:

• `(define x 5)`
• `(+ 2 3)`
• `(printf "Hi there...")`
The semantic rules of Java stipulate that the legal statement, `int x = 5;`, represents an instruction to the computer to create space for a variable named `x` whose value will, at least initially, be the integer five. Similarly, the semantic rules of Scheme stipulate that the legal expression, `(define x 5)`, when evaluated, should cause the computer to create a new variable named `x` whose value will be the integer five.

Although people can effectively communicate with one another using a natural language based on an informal, imprecise, intuitive understanding of its syntax and semantics, trying to program a computer based on an informal, imprecise, intuitive understanding of the syntax and semantics of a given programming language typically leads to trouble. Therefore, it is important to be explicit about the syntax and semantics of the programming language being used.

- Indeed, while programming, it is extremely important to have an accurate mental model of the computations you are effectively asking the computer to perform.

To enable us to enter the world of programming as quickly and painlessly as possible, it is helpful to use a programming language whose rules of syntax and semantics are relatively simple. Scheme is just such a language.

- Although Scheme has a relatively simple computational model (i.e., syntax and semantics), it is as computationally powerful as any programming language.

In contrast, the Java programming language has a much more complicated set of syntax rules, and a correspondingly complicated computational model—without any theoretical increase in computational power. Therefore, in this class, we begin with Scheme.

- The concepts you learn in this class will be helpful to you when learning any other programming language in the future.

In summary, to be effective, programmers need to have an accurate mental model of the operation of whatever computer they are programming. The complexity of their mental model depends in large part on the kind of programming language they are using. One of the significant advantages of the Scheme programming language is that it is based on a fairly simple computational model. Scheme’s computational model is based on the Lambda Calculus invented by the mathematician Alonzo Church in the 1930s, well before the advent of modern computers. Internalizing Scheme’s model of computation will make you an effective Scheme programmer in no time!

**Functions**

Scheme is an example of a functional programming language. The main thing that you, as a Scheme programmer, will do is design functions for solving problems. For our purposes, a function is something that takes zero or more inputs, and generates a single output, as illustrated on the lefthand side of Fig. 1.1. For example, you might define a Scheme function whose input is a scoresheet for some game, and whose output is the sum of the scores on that scoresheet.
In certain cases, we may also consider functions that generate side effects, as illustrated on the righthand side of Fig. 1.1. An example of a harmless, but very useful side effect is that of causing information to be displayed onscreen. For example, the above-mentioned function might not only compute the sum of the scores on a given scoresheet, but also have the side effect of displaying the contents of that scoresheet on a computer screen.

Functions that have either no side effects or only harmless side effects are called non-destructive.

As you will discover in Part I of this book (Non-Destructive Programming in Scheme) a wide variety of extremely useful computations can be performed by non-destructive functions. Furthermore, non-destructive functions tend to be very easy to write and debug (i.e., to find errors and fix them).

Nonetheless, as Part II (Destructive Programming in Scheme) reveals, there are also many areas where destructive functions (i.e., functions having destructive side effects) can be extremely useful. The most basic example of a destructive side effect is one that modifies the value assigned to a variable or to a slot within a data structure. For example, the above-mentioned function might not only compute the sum of the scores on the scoresheet, but also destructively modify the scoresheet by entering a new score into one of its slots. Although this kind of side effect may sound harmless, it can greatly complicate the task of writing and debugging functions. (For example, does the computed sum include the newly entered score?) Therefore, when we encounter destructive functions, starting with Chapter 17, we shall do so very carefully.

A Note about the Approach

This textbook takes a very careful, bottom-up approach to the computational model of Scheme. Each of the first several chapters introduces a small portion of the computational model, highlighting the syntax and semantics of each construct that is presented. Although this approach can seem slow at first, it leads to faster results in the long run because it helps to avoid many common pitfalls that can frustrate programmers who are relying on a casual understanding of the computational model being used.
Part I

Non-Destructive Programming in Scheme
Chapter 2

Scheme Expressions vs. Scheme Data

In our daily lives, we frequently use character sequences to denote both concrete and abstract data. For example, the character sequence *dog* can be used to denote a dog. Similarly, the character sequence *34* can be used to denote the number *thirty-four*. Of course, this book itself largely consists of a bunch of character sequences that denote all sorts of things. Well, in its physical form, it is a bunch of pages that are covered with ink marks. The ink marks represent characters that, in turn, form sequences of characters that denote other things. The point is: we are so used to using character sequences to denote (or represent) things that we tend to take it for granted. When programming computers, it is important to have a solid understanding of the legal character sequences, what they mean, and what computations they may lead to.

Any program in Scheme is a sequence of (usually typewritten) characters. The syntax rules of Scheme tell us which character sequences constitute legal Scheme programs.

The building blocks of a Scheme program are character sequences called *expressions*.

In other words, each Scheme program consists of one or more Scheme expressions. For example, as we’ll soon discover, *3*, *#t* and *()* are legal expressions in Scheme.

* In Scheme, each legal expression denotes a datum (i.e., a piece of data).
* The semantics of Scheme tells us which datum each legal expression denotes.

For example, in Scheme, the legal expressions *3*, *#t* and *()* respectively denote the following pieces of data: *the number three*, *the truth value true*, and *the empty list*.

As illustrated in Fig. 2.1, the universe of Scheme data is partitioned into *data types* having names such as: *numbers*, *booleans* (i.e., truth values), *symbols* and *functions*, among many others.

* Each datum belongs to one and only one data type.

For example, a Scheme datum might be a number or a symbol, but cannot be both. The rest of this chapter addresses expressions that denote some of the most commonly used types of Scheme data, beginning with *primitive data expressions*.

2.1 Primitive Data Expressions

A *primitive* datum is one that is atomic, in the sense that it is not composed of smaller parts that a Scheme program can access. Examples of primitive data in Scheme include *numbers*, *booleans* and the *empty list*. A *primitive data expression* is an expression that denotes a primitive datum.

2.1.1 Numbers

According to the syntax rules of Scheme, character sequences such as *3*, *-44*, *34.9* and *85/6* are legal Scheme expressions. According to the semantics of Scheme, these expressions respectively denote the numbers *three*, *negative forty-four*, *thirty-four point nine* and *eighty-five over six*. Continued...
negative forty-four, thirty-four point nine and eighty-five sixths. Each of these numbers is an example of a primitive Scheme datum.

For the purposes of this course, it is not necessary to explicitly write down the full set of syntax rules for numerical expressions in Scheme. We will only need the most basic sorts of numerical expressions in Scheme, most of whose rules are undoubtedly already familiar to you through whatever math classes you may have taken in years gone by.

Character sequences vs. the data they denote. It is extremely important to distinguish character sequences (e.g., 3) from the data they denote (e.g., the number three). To highlight this distinction, we use the following notation:

Character Sequence $\rightarrow$ Datum

For example, we can use this notation to describe the data denoted by the previously seen character sequences:

$$3 \rightarrow \text{the number three}$$

$$-44 \rightarrow \text{the number negative forty-four}$$

$$85/6 \rightarrow \text{the number eighty-five sixths}$$

In some cases, multiple Scheme expressions denote the same datum. For example, each of the following character sequences denotes the number zero in Scheme: 0, 000 and 000000, as indicated below.

$$0 \rightarrow \text{the number zero}$$

$$000 \rightarrow \text{the number zero}$$

$$000000 \rightarrow \text{the number zero}$$

As programmers, we only get to type the numerical expressions (i.e., character sequences); however, behind the scenes, the computer is working with the numbers (i.e., Scheme data) denoted by those expressions.

2.1.2 Booleans

According to the syntax rules of Scheme, the character sequences, #t and #f, are legal Scheme expressions. According to the semantics of Scheme, these expressions respectively denote the truth values true and false, as illustrated below:
Again, keep in mind the difference between the character sequences and the truth values they denote. The boolean data type consists solely of these two truth values (i.e., pieces of data). As programmers, we type the character sequences #t and #f; behind the scenes, the computer is working with the corresponding truth values.

2.1.3 The Empty List (or Null)

According to the syntax rules of Scheme, the character sequence, ( ), is a legal Scheme expression. According to the semantics of Scheme, it denotes the null datum, which is also called the empty list.

( ) → the empty list

(We’ll encounter non-empty lists later on.) The null data type includes only this one datum.

2.1.4 The Void Datum

Scheme includes a data type called void that contains only one datum, called the void datum. As will be seen later on (e.g., in Section 5.5), the void datum is used to represent “no value”. For example, a printing function, whose job is to print a bunch of information as a harmless side effect, will typically return the void datum as its output value. (Recall that there is a sharp distinction in Scheme between the output value of a function and any side effects it might have.)

* Although the void datum is a primitive datum, there is no corresponding primitive data expression that denotes the void datum.

Instead, the void datum is primarily used in cases where a function does not generate any meaningful output value.

2.1.5 Symbols

Another kind of primitive data in Scheme is a symbol. Symbols are typically used as names for things in a Scheme program. For example, each of the built-in functions in Scheme has a corresponding symbol that serves as its name. More generally, symbols can be used as names for any kind of Scheme data. That is, symbols can be used as variables in a Scheme program. For example, the symbol income might be used as a variable whose value is some amount of money.

To provide programmers with a great degree of flexibility when dealing with symbols, the syntax rules for symbol expressions in Scheme are very liberal. For example, miles-per-gallon, _LEGAL_SYMBOL_, *Legal-Symbol* and !even@me? are all legal expressions in Scheme that denote symbols. Because they are so flexible, it would be a bit tedious to explicitly write down all of the syntax rules for symbol expressions in Scheme. Fortunately, it is not necessary. For our purposes, the following general guidelines will suffice:

- Any sequence of letters, whether lower-case, upper-case, or a mixture of the two, is a legal symbol expression in Scheme. Examples include: hello, goodBye and gasMileage.
- Any character sequence consisting of letters and punctuation characters such as hyphens, asterisks, question marks and exclamation points is a legal symbol expression in Scheme. Examples include: new-world, gas-mileage, *CONSTANT*, _WIDTH_, roll-dice! and symbol?.
- Commonly used one-character expressions, such as *, +, – and /, also constitute legal symbol expressions in Scheme.

The semantics of Scheme specifies the datum denoted by each legal symbol expression. For example, the legal expression, hello, denotes the symbol hello; and the legal expression, *, denotes the asterisk symbol.

hello → the symbol hello
* → the asterisk symbol
Again, it is important to keep in mind the difference between the typewritten character sequences (e.g., `hello` and `bye-bye`) and the symbols (i.e., the Scheme data) that they denote (e.g., the symbol `hello`, and the symbol `bye-bye`). This distinction is hard to write down because we use symbols to denote character sequences, and we also use symbols to denote the symbols denoted by character sequences.) In addition, it is important to remember that symbols are primitive data; they do not have any parts that can be accessed by a Scheme program. For example, the symbol denoted by the expression `hello` does not have any parts; it is indivisible. It may help to think of it as a billiard ball with `hello` written on it.

### 2.2 String Expressions

This section introduces the `string` data type in Scheme. Unlike all of the data types discussed above, strings are non-primitive in Scheme: each string has parts, called characters, that can be accessed by a Scheme program. However, we will not be focusing on the non-primitive nature of strings in this book. In other words, although strings are non-primitive in Scheme, we will, in what follows, treat them as though they were primitive. Why, then, do we introduce them here? Because, as will be seen, Scheme’s very useful printing functions use strings!

* The `string` data type will not be a focus of this book (i.e., it will not play an important role in our understanding of Scheme’s computational model); instead, we will only use strings when we want our Scheme programs to print out useful information.

Syntactically, a string expression is a sequence of characters delimited by double-quotes. For example, "hi" and "Howdy!" are legal string expressions in Scheme. The semantics of Scheme stipulates that each string expression denotes a string datum (i.e., a non-primitive sequence of characters), as illustrated below.

\[
\begin{align*}
"hi" & \rightarrow \text{the string } "hi" \\
"Howdy!" & \rightarrow \text{the string } "Howdy!"
\end{align*}
\]

### 2.3 Summary

This chapter introduced the syntax and semantics for a variety of data types in Scheme, including: numbers, booleans, the empty list, the `void` datum, symbols, and strings. Examples of legal expressions that denote these kinds of data are given below.

- **Numbers:** 342, -81, 34/9, 21.832, etc.
- **Booleans:** #t, #f
- **The empty list:** ()
- **Symbols:** x, miles-per-gallon, dollarsPerGallon, *, +, /, etc.
- **Strings:** "hi", "Howdy!"

For each legal expression (i.e., piece of syntax), the semantics specifies the datum denoted by that expression. This book uses a single arrow (\(\rightarrow\)) to represent denotation. For example, the fact that the character sequence 34 denotes the number thirty-four is represented by: 34 \(\rightarrow\) the number thirty-four.

Although there are expressions that denote data belonging to most of the data types listed above, there is no legal expression in Scheme that denotes the `void` datum.

Of all the data types addressed above, only strings are non-primitive in Scheme; however, investigating the non-primitive aspects of strings shall not be a focus of this book. But have no fear: Chapter 6 will introduce non-empty lists, a non-primitive type of data that plays a central role in Scheme’s computational model.
Chapter 3

Evaluating Scheme Data

We have seen that a variety of character sequences (e.g., 34, xyz, () and #t) constitute legal expressions according to the syntax rules of Scheme. In addition, we’ve seen that each legal expression denotes a piece of data of a particular kind. For example, 34 denotes the number *thirty-four*, and xyz denotes the symbol *xyz*. The character sequences are expressions; the data they denote belong to the universe of Scheme data. As programmers, we type character sequences; the computer deals with the corresponding Scheme data.

This chapter addresses the one thing that a Scheme computer does—namely, it *evaluates* Scheme data. The following observations are important to keep in mind:

* Evaluation is done by the computer, not the programmer.
* Evaluation involves Scheme data, not expressions/character sequences.

Because evaluation is the one-and-only thing that a Scheme computer does, it is important to carefully describe it. The good news is that the process of evaluation can be described fairly succinctly for many kinds of Scheme data.

We begin by noting that evaluation is a *function*—in the mathematical sense (i.e., something that takes zero or more inputs, and generates a single output). In particular, the evaluation function takes one Scheme datum as its input, and generates another Scheme datum as its output, as illustrated below.

The result of applying the *evaluation* function depends on the type of data that it is applied to. Thus, in what follows, we describe what the evaluation function does for each kind of data we have seen so far.

* In most cases, the application of the evaluation function to a Scheme datum does not directly generate any side effects. However, there are some important exceptions that shall be highlighted as they are encountered—in Chapters 7, 18 and 19.
3.1 Evaluating Numbers, Booleans, the Empty List, the \textit{void} Datum, and Strings

The \textit{evaluation} function acts like the \textit{identity function} when applied to numbers, booleans, the \textit{empty list}, the \textit{void} datum, or strings, as illustrated below.

\begin{tabular}{|c|c|}
  \hline
  Input Datum & Output Datum \\
  \hline
  \textit{number two} & \textit{number two} \\
  \textit{boolean true} & \textit{boolean true} \\
  \hline
\end{tabular}

Since drawing all of these black boxes takes up so much space, from now on we’ll use a simpler, text-based notation to represent the application of the \textit{evaluation} function to some datum, as illustrated below.

\begin{tabular}{c}
  Input Datum $\implies$ Output Datum
\end{tabular}

The double arrow ($\implies$) is reserved solely for representing the application of the \textit{evaluation} function to some Scheme datum (called the input) to generate some, possibly quite different Scheme datum (called the output).

* Instead of saying that the \textit{evaluation} function generates the output datum when applied to a certain input datum, we may say that the output datum is the result of \textit{evaluating} the input datum (or that the input datum \textit{evaluates} to the output datum). Keep in mind that when we say such things, we are talking about the application of the one-and-only \textit{evaluation} function.

Here are some more examples illustrating the trivial behavior of the \textit{evaluation} function when applied to numbers, booleans, the \textit{empty list}, the \textit{void} datum, or strings:

- the number \textit{zero} $\implies$ the number \textit{zero}
- the boolean \textit{true} $\implies$ the boolean \textit{true}
- the \textit{empty list} $\implies$ the \textit{empty list}
- the \textit{void} datum $\implies$ the \textit{void} datum
- the string \textit{“hi there”} $\implies$ the string \textit{“hi there”}

Of course, if the \textit{evaluation} function acted like the identity function for \textit{every} kind of input, then it would not be very interesting. (It would just be the identity function.) The following section addresses one of the most important cases where the evaluation function does something a little more interesting.

3.2 Evaluating Symbols

In Scheme, symbols are frequently used as \textit{variables}. In math, variables frequently have values associated with them. For example, the variable \textit{x} may have the value \textit{3}. So it is with Scheme. For this reason, the evaluation of symbols is different from the evaluation of numbers, booleans, the \textit{empty list}, the \textit{void} datum, or strings. In particular, symbols typically do not evaluate to themselves; instead, they evaluate to the value associated with them. (Keep reading!)

In Scheme, every datum is evaluated with respect to an \textit{environment}. Now, for numbers, booleans, the empty list, the \textit{void} datum, and many other kinds of data in Scheme, it doesn’t matter which environment they get evaluated with respect to, because these kinds of data always evaluate to themselves. However, a given symbol \textit{x} might evaluate to the number \textit{three} in one environment, but to the boolean \textit{false} in another environment. How can this be? And why might it be useful? We’ll address the first of these questions below; the second will have to wait for a while.
Environments in Scheme. Each environment has a list of entries, where each entry pairs a symbol $s$ with its value $v$. To evaluate a symbol $s$ with respect to some environment, the evaluation function simply looks on the environment’s list of symbol/value pairs to see if there’s an entry that pairs $s$ with some value $v$. If there is, then the evaluation function reports that $s$ evaluates to $v$. (We’ll address what happens in cases where there isn’t an entry for $s$ in the relevant environment later on.) Therefore, if the environment $\mathcal{E}_0$ contains an entry that pairs the symbol $x$ with the number three, while the environment $\mathcal{E}_1$ contains an entry that pairs the symbol $x$ with the boolean $false$, then evaluating $x$ with respect to the environment $\mathcal{E}_0$ will yield the number three, while evaluating $x$ with respect to $\mathcal{E}_1$ will yield the boolean $false$.

Okay, that’s true enough. However, while there can be lots of different environments in Scheme, the focus of attention for the next several chapters shall be on the most important environment in Scheme: the Global Environment. The Global Environment is the environment that is used by default.

* When we are talking about evaluating some Scheme datum, unless we explicitly say something to the contrary, we shall assume that we are talking about evaluating that datum with respect to the Global Environment.

It may help to think of an environment as a room that has a list of symbol/value pairs tacked to one of its walls. When a symbol needs to be evaluated in that room/environment, the evaluation function simply fetches the symbol’s value from the relevant entry on that list.

Evaluating symbols in the Global Environment. Okay, so, if the Global Environment contains an entry that pairs the symbol $xyz$ with the number two, then the result of applying the evaluation function to the symbol $xyz$ will be the number two:

$$\text{the symbol } xyz \implies \text{the number two}$$

Importantly, the Scheme datum that is paired with a symbol in the Global Environment can be of any type. Thus, it might be that the boolean $true$ is paired with the symbol $pq$. Similarly, the empty list might be paired with the symbol $my\text{-}empty\text{-}list$, as illustrated below.

$$\text{the symbol } pq \implies \text{the boolean } true$$
$$\text{the symbol } my\text{-}empty\text{-}list \implies \text{the empty list}$$

Symbols can even evaluate to other symbols. For example, if the Global Environment contains an entry associating the symbol $bar$ with the symbol $foo$ (where $bar$ corresponds to the value), then the following would hold:

$$\text{the symbol } foo \implies \text{the symbol } bar$$

On the other hand, if a symbol does not have a corresponding entry in the Global Environment, then evaluating that symbol with respect to the Global Environment is undefined. A little later on, in Chapter 7, we’ll see how to insert entries into the Global Environment, thereby enabling us to create and use variables of our own.

* An environment is a context within which Scheme data get evaluated. However, an environment is not a Scheme datum. Thus, environments in Scheme are not available for direct inspection.

3.3 Summary

At the core of the Scheme computational model is the process of evaluation. Evaluation is a function that takes a Scheme datum as its input and generates a (possibly different) Scheme datum as its output. For each type of data, the semantics of Scheme specifies how instances of that data type are evaluated (i.e., what output is produced). Numbers, booleans, the empty list, the $void$ datum, and strings all evaluate to themselves (i.e., the evaluation function works like the identity function for instances of those data types). However, a symbol is evaluated differently: by looking for a corresponding entry in the Global Environment.

This book uses the double arrow ($\implies$) to represent the process of evaluation. For example, if the Global Environment contains an entry associating the symbol $x$ with the number $eighty\text{-}six$, this fact can be represented as follows:
It is important to remember that:

(1) each expression—which is a character sequence—denotes a Scheme datum; and

(2) each Scheme datum evaluates to a (possibly different) Scheme datum.

For example:

$$x \rightarrow \text{the symbol } \textit{x} \implies \text{the number } \textit{eighty-six}$$
Chapter 4

Introduction to DrScheme

This chapter introduces the piece of software known as DrScheme. This software simulates the operation of a computer that understands the Scheme programming language. It also enables us to interact with that simulated computer. In effect, we use DrScheme as an intermediary between us and that simulated computer. We interact with the simulated computer as follows:

- We type some character sequence into DrScheme’s Interactions Window (i.e., the lower window-pane in DrScheme’s window).
- DrScheme takes the datum denoted by that character sequence and feeds it into the evaluation function (i.e., DrScheme evaluates that datum), generating some output datum.
- DrScheme displays some typewritten text in the Interactions Window describing the output datum to us.

This process is illustrated in Fig. 4.1, where everything in the shaded box is carried out behind the scenes by DrScheme. Notice that our interaction with DrScheme is through the character sequences (i.e., expressions) we type into the Interactions Window; and the character sequences that DrScheme displays to us in response. We never get to “touch” the Scheme data denoted by our character sequences. (What would it mean to touch a number anyway?) For this reason, it is extremely important that we maintain an accurate mental model of what’s going on in that simulated world. In other words, we need to have an accurate understanding of Scheme’s computational model.

More formally, when we type a sequence of characters, $C_{\text{in}}$, into the Interactions Window, and then hit the Return (or Enter) key, DrScheme does the following:

1. It figures out which Scheme datum, $D_{\text{in}}$, is denoted by the character sequence $C_{\text{in}}$;
2. It feeds that Scheme datum as input to the evaluation function, which generates an output datum, $D_{\text{out}}$ (i.e., $D_{\text{in}}$ evaluates to $D_{\text{out}}$).
3. Finally, it displays some typewritten text, $C_{\text{out}}$, in the Interactions Window that describes the output datum, $D_{\text{out}}$.

This process is illustrated below.

---

1The DrScheme software is freely available from dscheme.org. For the purposes of this book, DrScheme and DrRacket, which is freely available from drracket.org, may be considered to be equivalent. Thus, DrRacket may be used in place of DrScheme, if desired.
Keep in mind that we only see the character sequences, \( C_{\text{in}} \) and \( C_{\text{out}} \); we do not see the Scheme data, \( D_{\text{in}} \) and \( D_{\text{out}} \). (What does a Scheme datum look like anyway?) We can more succinctly describe this process as follows:

\[
C_{\text{in}} \rightarrow \left[ D_{\text{in}} \rightarrow D_{\text{out}} \right] \rightarrow C_{\text{out}}
\]

where the single arrow (\( \rightarrow \)) represents the translation from character sequences to the denoted Scheme data (in either direction), the double arrow (\( \rightarrow\rightarrow \)) represents the application of the evaluation function, and the square brackets indicate that we don’t get to see the Scheme data, \( D_{\text{in}} \) and \( D_{\text{out}} \).

* When entering expressions into the Interactions Window, the datum \( D_{\text{in}} \) is evaluated with respect to the Global Environment.

### 4.1 Entering Expressions into the Interactions Window

We can use DrScheme to confirm some of the things discussed in previous chapters. In particular, we can enter character sequences (i.e., expressions) into the Interactions Window and then examine the results reported by DrScheme. In each case, we only get to see the character sequences we type in, and those reported back by DrScheme; we do not get to see the Scheme data that is manipulated by the Scheme computer.

**Example 4.1.1: DrScheme’s Interactions Window**

The following interactions demonstrate that numbers, booleans, the empty list and strings all evaluate to themselves:

\[
\begin{align*}
> & 3 \\
 3 \\
> & \#t \\
\#t \\
> & () \\
() \\
> & "Howdy!" \\
"Howdy!"
\end{align*}
\]

In the Interactions Window, DrScheme uses the > character to prompt the user for input. Everything following the > character is typed by the programmer. The text on the next line is that generated by DrScheme in response. Thus, the above example shows four separate interactions.
In these simple examples, the character sequence displayed by DrScheme happens to be the same as that typed by the programmer. However, recall that, behind the scenes, DrScheme is doing quite a bit more than these examples suggest. In particular:

\[
\begin{align*}
3 & \rightarrow \{ \text{the number three} \implies \text{the number three} \} \rightarrow 3 \\
#t & \rightarrow \{ \text{the boolean true} \implies \text{the boolean true} \} \rightarrow #t \\
() & \rightarrow \{ \text{the empty list} \implies \text{the empty list} \} \rightarrow () \\
"Howdy!" & \rightarrow \{ \text{the string "Howdy!"} \implies \text{the string "Howdy!"} \} \rightarrow "Howdy!"
\end{align*}
\]

**Example 4.1.2**

The following interactions demonstrate that several different character sequences can be used to denote the number zero:

\[
> 0 \\
0 \\
> 000 \\
0 \\
> 00000 \\
0
\]

As this example illustrates, DrScheme need not use the same character sequence as the one we entered when reporting back that the result of evaluating the number zero is the number zero. Instead, DrScheme chooses the most compact character sequence.

For convenience, we may say that DrScheme is evaluating the expressions we type into the Interactions Window, when of course we mean that DrScheme is evaluating the data denoted by the expressions we type into the Interactions Window.

### 4.2 DrScheme’s Run Button

Although manually typing individual expressions into the Interactions Window and viewing DrScheme’s responses can be quite useful, it is often desirable to ask DrScheme to evaluate a large number of Scheme expressions. (Re-read the above note about “evaluating expressions.”) To avoid endless typing and re-typing (e.g., when fixing errors), the upper window-pane of DrScheme, called the Definitions Window, can be used to edit—and, if desired, save—any number of Scheme expressions. Afterward, clicking the Run button in DrScheme’s toolbar causes DrScheme to evaluate each of the expressions currently residing in the Definitions Window, one after the other, as if we had manually typed them into the Interactions Window, as illustrated in Fig. 4.2.

- When using the Run button, DrScheme only reports the results of evaluating the expressions from the Definitions Window.

More generally, the Definitions Window can be used to hold the contents of an entire Scheme program. In such cases, clicking the Run button would cause all of the expressions in that program to be evaluated, one implication being that any functions defined in that program could then be used.

### 4.3 Summary

The DrScheme software simulates a Scheme computer that we, as programmers, can interact with. We type expressions (i.e., character sequences) into the Interactions Window, and DrScheme responds by displaying some (possibly different) character sequence. However, something very important happens in between:
(1) the input character sequence $C_{in}$ denotes some Scheme datum $D_{in}$;

(2) DrScheme evaluates $D_{in}$, yielding some datum $D_{out}$; and

(3) DrScheme displays a character sequence $C_{out}$ that describes $D_{out}$ to us.

This process is concisely summarized by:

$$C_{in} \rightarrow [ D_{in} \implies D_{out} ] \rightarrow C_{out}$$

where the stuff between the square brackets is invisible to us. Since such important computations are happening behind the scenes, it is important that we, as programmers, have an accurate mental model of what Scheme is doing.

DrScheme’s Definitions Window can be used to hold multiple expressions. Clicking the Run button causes each of those expressions to be evaluated in turn, with the results reported in the Interactions Window.
Chapter 5

Built-In Functions

For convenience, Scheme includes a variety of built-in functions. Examples include the addition function, the multiplication function, and a printing function, among many others.

* Each built-in function is a Scheme datum that is primitive, like numbers and booleans, in the sense that they don’t have any parts that a Scheme program can access. Thus, a built-in function is a black box to us.

If you are wondering what character sequences in Scheme denote built-in functions, the answer may surprise you:

* There are no Scheme expressions that denote built-in Scheme functions.¹

This surprising fact leads to another question: How can a Scheme programmer make use of built-in functions if none of them are denoted by any Scheme expressions? The answer is indicated by the following observation.

* Although the Input Datum shown in Fig. 4.1 can never be a function, the Output Datum can be.

In particular, for each built-in function, there is an entry in the Global Environment that associates a particular symbol with that function. Therefore, evaluating that symbol generates the corresponding function as an output value. In other words, if the Input Datum from Fig. 4.1 is a symbol that serves as the name of a built-in function, then the corresponding Output Datum will be that function. That is: we gain access to a built-in function by evaluating the symbol that serves as its name.

The rest of this chapter introduces some of the most commonly used built-in functions.

5.1 Built-in Functions for Arithmetic

DrScheme provides a variety of built-in functions for doing basic arithmetic computations. For example, when DrScheme is first started up, the Global Environment is automatically populated with entries that ensure that each of the following evaluations holds:

the symbol +    \(\implies\) the addition function
the symbol -    \(\implies\) the subtraction function
the symbol *    \(\implies\) the multiplication function
the symbol /    \(\implies\) the division function

Thus, a Scheme programmer can refer to these built-in functions indirectly, by asking DrScheme to evaluate the corresponding symbols.

¹Indeed, there are no Scheme expressions that denote any kind of function, whether built-in or not!
Example 5.1.1: Accessing the built-in arithmetic functions

*That the abovementioned entries do indeed exist in the Global Environment can be confirmed by DrScheme, as illustrated below:*¢

```
> +  
#<procedure:+>
> -  
#<procedure:->
> *  
#<procedure:*>
> /  
#<procedure:/>
```

*The behind-the-scenes work involved in these interactions can be summarized as follows:*

- \( + \rightarrow \{ \text{the } + \text{ symbol} \rightarrow \text{the addition function} \} \rightarrow #<procedure:+> \)
- \( - \rightarrow \{ \text{the } - \text{ symbol} \rightarrow \text{the subtraction function} \} \rightarrow #<procedure:-> \)
- \( * \rightarrow \{ \text{the } * \text{ symbol} \rightarrow \text{the multiplication function} \} \rightarrow #<procedure:*> \)
- \( / \rightarrow \{ \text{the } / \text{ symbol} \rightarrow \text{the division function} \} \rightarrow #<procedure:/> \)

---

Notice that the character sequences reported by DrScheme need not be legal pieces of Scheme syntax. (Recall that there is no legal piece of Scheme syntax that denotes a primitive function.) Instead, a character sequence such as \( #<procedure:+> \) is DrScheme’s best attempt to describe to us the fact that the output datum is a function—namely, the function associated with the + symbol.

* Although we are required to type legal Scheme expressions into the Interactions Window, DrScheme is allowed to write whatever it wants when it seeks to describe the results of an evaluation.

5.2 Contracts

To be able to make proper use of a built-in function, it is important to know its name, the kinds of inputs it can be applied to, the order in which it expects its inputs, some sort of description of the output it is supposed to generate and, if applicable, any side effects it might have. This kind of information is typically gathered together into a contract, as illustrated by the following examples.

Example 5.2.1: Contracts for some built-in functions

*Here is a contract for the built-in addition function:*

<table>
<thead>
<tr>
<th>Name:</th>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs:</td>
<td>( x_1, x_2, \ldots, x_n; \text{any number of numerical inputs} )</td>
</tr>
<tr>
<td>Output:</td>
<td>The sum, ( x_1 + x_2 + \ldots + x_n )</td>
</tr>
<tr>
<td>Side Effects:</td>
<td>None</td>
</tr>
</tbody>
</table>

Notice that the contract describes what the output value should be, but it does not go into the underlying details about how that output value is actually computed. Similar remarks apply to the following contract for the built-in subtraction function:
* Since most functions encountered in this course will not have any side effects, we shall follow the convention that if a contract does not mention side effects, then the function can be assumed to not have any.

The rest of this chapter presents several other commonly used built-in functions. The next chapter will show how to apply functions to inputs (i.e., make them do something). Later chapters will show how to create functions of our own design and give them names by inserting appropriate entries into the Global Environment.

### 5.3 Built-in Functions for Integer Arithmetic

You may recall the process of doing integer division in grade school. For example, you may have been shown that 17 divided by 3 yields an answer of 5 with remainder 2. (The answer is often called the *quotient*—but I always had trouble remembering that.) DrScheme provides two built-in functions, called `quotient` and `remainder`, that together can be used to carry out integer division: `quotient` provides the answer; `remainder` provides the remainder. The contracts for these functions are given below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>quotient</td>
<td><code>numer</code>, <code>denom</code>, two integers</td>
<td>The (integer) answer that results from dividing <code>numer</code> by <code>denom</code>, ignoring any remainder.</td>
</tr>
<tr>
<td>remainder</td>
<td><code>numer</code>, <code>denom</code>, two integers</td>
<td>The (integer) remainder left over from dividing <code>numer</code> by <code>denom</code>.</td>
</tr>
</tbody>
</table>

Scheme also includes a built-in function, called `integer?`, for checking whether a given datum is an integer.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer?</td>
<td><code>datum</code>, anything</td>
<td>#t if <code>datum</code> is an integer; otherwise, #f</td>
</tr>
</tbody>
</table>

### 5.4 The Built-in `eval` Function

The evaluation function that is so important to the computational model of Scheme is itself provided as a built-in function. In particular, the Global Environment contains an entry that associates the `eval` symbol with the built-in evaluation function, as demonstrated by the following interaction:

```
> eval
#<procedure:eval>
```

Since it is a primitive, built-in function, we don’t get to see *how* the evaluation function operates; however, we have started to discover *what* the evaluation function does—at least for some kinds of Scheme data. Subsequent chapters will address what the evaluation function does for other kinds of Scheme data. Once we understand what the evaluation function does for each kind of Scheme data, we could think about writing down a contract for it.

* Like numbers, booleans, the empty list, the *void* datum, and strings, functions evaluate to themselves.

In other words, if you feed a Scheme function as input to the evaluation function, the output will be that same function. For example, the addition function evaluates to the addition function; the multiplication function evaluates to the multiplication function; and the evaluation function applied to itself yields itself (!), as summarized below.
22 Introduction to Computer Science via Scheme  © 2018 Luke Hunsberger  Spring 2018

The addition function \(\Rightarrow\) the addition function

The multiplication function \(\Rightarrow\) the multiplication function

The evaluation function \(\Rightarrow\) the evaluation function

A demonstration of functions evaluating to themselves will be given in the next chapter.

5.5 The Built-in Functions \texttt{printf} and \texttt{void}

Recall from Section 2.1.4 that Scheme includes a data type called \texttt{void} whose only datum is also called \texttt{void}. The purpose of the \texttt{void} datum is to represent “no value”. For example, a function whose main job is to do a bunch of side-effect printing might return the \texttt{void} datum as its output value, representing “no output value”. In such cases, DrScheme would display all of the side-effect printing, but would not display anything for the \texttt{void} output value. (Since \texttt{void} represents “no value”, DrScheme does not feel compelled to display anything for \texttt{void}.)

* If a function’s output is \texttt{void}, then we may say that the function does not generate any output value.

The built-in functions, \texttt{printf} and \texttt{void}, introduced below, are examples of functions whose output value is invariably the \texttt{void} datum.

5.5.1 The \texttt{printf} Function

Scheme provides a built-in \texttt{printf} function that can be used to display information in the Interactions Window.

* The display of textual information by the \texttt{printf} function is an example of a harmless side effect.

* The output value generated by the \texttt{printf} function is always the \texttt{void} datum.

Although the \texttt{printf} function has additional capabilities that won’t be explored until Chapter 10, the contract for the simplest use of the \texttt{printf} function, given below, will suffice for now.

\begin{verbatim}
Name: printf
Input: \texttt{str}, a string
Output: the \texttt{void} datum (i.e., “no value”)
Side Effect: displays the contents of the string \texttt{str} in the Interactions Window (without the double-quotes)
\end{verbatim}

5.5.2 The \texttt{void} Function

Recall from Section 5.5 that there is no legal Scheme expression that we can type into the Interactions Window that denotes the \texttt{void} datum. However, should you ever need to get your hands on the \texttt{void} datum, there is a built-in function, called \texttt{void}, that does nothing but generate the \texttt{void} datum as its output. Here is its contract:

\begin{verbatim}
Name: void
Inputs: Any number of inputs
Output: the \texttt{void} datum
Side Effects: none
\end{verbatim}

5.6 Summary

There are no Scheme expressions that denote functions! However, that is not a problem because there are Scheme expressions that denote Scheme data that evaluate to functions. (Denotation vs. evaluation.) In particular, the Global Environment comes pre-populated with entries that associate certain symbols with various built-in functions. For example, the \texttt{+} symbol is associated with the built-in \texttt{addition} function; and the \texttt{*} symbol is associated with the built-in \texttt{multiplication} function. As a result, we can effectively refer to the built-in functions by name, as illustrated below:
Note that DrScheme is not required to follow the rules of Scheme syntax when displaying information in the Interactions Window.

So that we may use the built-in functions properly, each function has an associated contract that specifies its name (a symbol), its inputs (how many and their types), its output, and any side effects it might have. The information found in the contracts for the built-in functions is available online, for example, using the HelpDesk feature of the DrScheme program. Later on, when we learn how to specify functions of our own design (cf. Chapter 9), we will include a contract for each new function.

The evaluation function itself is provided as a built-in function—it is the value associated with the eval symbol.

**Built-In Functions Introduced in this Chapter**

- **Basic Arithmetic:** +, -, *, /
- **Integer Arithmetic:** quotient, remainder, integer?
- **Evaluation Function:** eval
- **Basic Printing:** printf
- **Generating the void datum:** void
Chapter 6

Non-Empty Lists

Previously, we have seen many examples of primitive data: numbers, booleans, the empty list, the void datum, symbols, and primitive functions. Recall that each primitive datum is atomic in the sense that it has no parts that we, as Scheme programmers, can access. In contrast, strings are non-primitive data: they have parts, called characters, that are accessible to Scheme programmers. However, instead of exploring the non-primitive nature of strings, this chapter explores another kind of non-primitive data: *non-empty lists*.

As will soon be revealed, non-empty lists play a very important role in Scheme’s computational model.

A non-empty list is an ordered sequence of Scheme data. For example, a list might contain items such as the symbol $+$, the number *three*, and the boolean *true*. Other examples of non-empty lists are given below:

- a list containing the number *three* and the number *four*
- a list containing the $+$ symbol, the number *three*, and the number *four*
- a list containing: (1) the symbol *eval*, and (2) a subsidiary list containing the $+$ symbol, the number *three*, and the number *four*

The last example illustrates that a list can contain elements that are themselves lists.

A non-empty list is, by itself, a Scheme datum. It is a Scheme datum that happens to contain other Scheme data as its *elements*.

### 6.1 The Syntax and Semantics for Non-Empty Lists

Since a non-empty list is a Scheme datum, a natural question arises: what kinds of character sequences can the programmer use to denote non-empty lists (i.e., what are the syntax rules for non-empty lists)? We begin with sample character sequences that the programmer can use to denote the Scheme lists described above:

- $(3 \ 4)$ $\rightarrow$ a list containing the number *three* and the number *four*
- $(+ \ 3 \ 4)$ $\rightarrow$ a list containing the $+$ symbol, the number *three*, and the number *four*
- $(\text{eval} \ (+ \ 3 \ 4))$ $\rightarrow$ a list containing:
  - (1) the symbol *eval*, and
  - (2) a subsidiary list containing the $+$ symbol, the number *three*, and the number *four*

As these examples illustrate, if $E_1$, $E_2$, $\ldots$, $E_n$ are legal Scheme expressions (i.e., character sequences), then the character sequence

$$(E_1 \ E_2 \ \ldots \ E_n)$$

is a legal character sequence. (That’s the syntax!) Furthermore, that character sequence denotes a list containing the $n$ items denoted by $E_1, E_2, \ldots, E_n$. (That’s the semantics!) Thus, if
If each \( E_i \) is a Scheme expression that denotes a Scheme datum, \( D_i \), then the character sequence

\[
(E_1 \ E_2 \ \ldots \ E_n)
\]

is a legal character sequence that denotes a list \( D \) containing the \( n \) items \( D_1, D_2, \ldots, D_n \).

For example, the character sequences \(+, 3\) and \(4\) are legal Scheme expressions that respectively denote the \(+\) symbol, the number \(three\), and the number \(four\). Thus, the character sequence, \((+\ 3\ 4)\), is a legal Scheme expression that denotes a list containing the \(+\) symbol, the number \(three\), and the number \(four\). In this example, the expressions \(E_1, E_2\) and \(E_3\) are \(+, 3\) and \(4\), respectively; and the Scheme data \(D_1, D_2\) and \(D_3\) are the \(+\) symbol, the number \(three\), and the number \(four\).

Since \((+\ 3\ 4)\) denotes a list, if we type this character sequence into the Interactions Window, the Input Datum will be that list. (It may help to refer back to Fig. 4.1.) However, DrScheme will then evaluate that list—because DrScheme always evaluates the Input Datum to generate the Output Datum. Therefore, we need to talk about how non-empty lists are evaluated.

### 6.2 Evaluating Non-Empty Lists: the Default Rule

As already seen, the empty list evaluates to itself; however, the evaluation of a non-empty list is altogether different. This section presents the Default Rule for evaluating non-empty lists. Exceptions to the Default Rule—the so-called special forms—will be covered later on.

#### Example 6.2.1

We begin with some examples that confirm that something new is happening when DrScheme evaluates non-empty lists.

\[
> (+\ 2\ 3) \\
5 \\
> (*\ 3\ 4\ 5) \\
60 \\
> (+\ 2\ (*\ 3\ 10)) \\
32 \\
> (+\ 2\ (*\ 3\ (+\ 4\ 8\ 6))) \\
56
\]

In each of these examples, the expression entered by the programmer is a legal Scheme expression that denotes a Scheme list. (You should convince yourself of this.) In addition, the evaluation of each list appears to result in an arithmetic computation—in fact, the kind of arithmetic computations you’ve seen in math classes over the years. In each case, the list is being evaluated according to the Default Rule.

#### Example 6.2.2

Consider the expression \((+\ 2\ 3)\), which denotes a list containing three items: the \(+\) symbol, the number two, and the number three. The Default Rule for evaluating such lists has two steps. The first step is to evaluate each item in the list. Now, the \(+\) symbol evaluates to the built-in addition function because the Global Environment is guaranteed to contain an entry associating the \(+\) symbol with the addition function.
The remaining items in the list are numbers; thus, they trivially evaluate to themselves. The results of the first step are summarized below:

- The + symbol $\rightarrow$ the addition function
- The number two $\rightarrow$ the number two
- The number three $\rightarrow$ the number three

Okay, so after evaluating all of the items in the list, we have the addition function and two numbers. The second step in the Default Rule involves applying that function to the remaining items (i.e., feeding the remaining items as input into that function), as illustrated below:

The resulting output datum is what we take to be the result of evaluating the original non-empty list! Thus, the result of evaluating the list containing the + symbol, the number two, and the number three, is (not surprisingly perhaps) the number five, which DrScheme reports in the Interactions Window using the character sequence 5. Here’s a summary of this example:

$$(+ 2 3) \rightarrow \{ \text{list containing + symbol, number two, number three } \rightarrow \text{number five} \} \rightarrow 5$$

where the evaluation is explained by:

First Step of the Default Rule:

- $+ \text{ symbol } \rightarrow \text{addition function}$
- number two $\rightarrow$ number two
- number three $\rightarrow$ number three

Second Step of the Default Rule:

addition function applied to two and three yields output of five

The evaluation of this list is illustrated in Fig. 6.1.

Example 6.2.3

Although the Default Rule is not trivial, there are several advantages to it. First, it has only two steps, and they are always the same. Second, it can be used on arbitrarily complex lists without requiring any modifications. For example, recall the interaction:

```
> (+ 2 (* 3 10))
32
```

If we follow the rules we already know, we will see that nothing new is needed to explain this interaction. First, the character sequence $(+ 2 (* 3 10))$ is a legal Scheme expression that denotes a list. The denoted list contains three items: the + symbol, the number two, and a subsidiary list. The subsidiary list contains three items: the * symbol, the number three, and the number ten. (You should convince yourself
of all of this before proceeding.) Okay, so far so good: we have seen that our input expression denotes a particular list. That list, which happens to be a list of lists, shall be the Input Datum for the evaluation function.

To evaluate this list, we need to use the Default Rule. The first step of the Default Rule requires us to evaluate each item in the list:

\[
\begin{align*}
\text{the + symbol} & \quad \Rightarrow \quad \text{the addition function} \\
\text{the number two} & \quad \Rightarrow \quad \text{the number two} \\
\text{the subsidiary list} & \quad \Rightarrow \quad \text{oops!}
\end{align*}
\]

Before we can complete the first step of the Default Rule, we must evaluate the subsidiary list (i.e., the list containing the * symbol, the number three, and the number ten). Okay, so we pause for a moment, collect our thoughts, and then proceed.

To evaluate the subsidiary list, we need to use . . . the Default Rule! The first step of the Default Rule requires us to evaluate each item in the list:

\[
\begin{align*}
\text{the * symbol} & \quad \Rightarrow \quad \text{the multiplication function} \\
\text{the number three} & \quad \Rightarrow \quad \text{the number three} \\
\text{the number ten} & \quad \Rightarrow \quad \text{the number ten}
\end{align*}
\]

The second step of the Default Rule requires us to apply the first item (i.e., the function) to the rest of the items. In other words, we need to apply the multiplication function to the numbers three and ten. The result is the number thirty.

Now that we know that the subsidiary list evaluates to thirty, we can pick up from where we left off when evaluating the original list. The first step of the Default Rule (for evaluating the original list) requires us to evaluate each item in the list:

\[
\begin{align*}
\text{the + symbol} & \quad \Rightarrow \quad \text{the addition function} \\
\text{the number two} & \quad \Rightarrow \quad \text{the number two} \\
\text{the subsidiary list} & \quad \Rightarrow \quad \text{the number thirty}
\end{align*}
\]

The second step of the Default Rule then requires us to apply the first item (i.e., the addition function) to the rest of the items (i.e., the numbers two and thirty). The result is the number thirty-two. And that is the Output Datum that results from evaluating the original list! Phew! Of course, DrScheme reports this result using the character sequence 32.
6.2.1 A More Formal Description of the Default Rule

Consider a list \( L \) that contains \( n \) data items, \( D_1, D_2, \ldots, D_n \). The evaluation of the list \( L \) is derived as follows:

- First, evaluate each of the data items, \( D_1, D_2, \ldots, D_n \). The result will be \( n \) (possibly different) data items, \( K_1, K_2, \ldots, K_n \):
  \[
  D_1 \implies K_1 \\
  D_2 \implies K_2 \\
  \vdots \\
  D_n \implies K_n
  \]

- Now, for the Default Rule to work, \( K_1 \) must be a function. (If \( K_1 \) is some other kind of datum, then DrScheme will report an error.)

- The second step is to apply the function \( K_1 \) to the rest of the items, \( K_2, \ldots, K_n \). In other words, the items \( K_2, \ldots, K_n \) are fed as input to the function \( K_1 \). (If the function \( K_1 \) cannot accept that number of inputs, or if those items have the wrong data type, then DrScheme will report an error.) The resulting output will be some datum, \( P \).

- The evaluation of the list \( L \) is defined to be that datum \( P \) (i.e., \( L \implies P \)).

As indicated by the parenthetical comments, it is possible for some things to go wrong in the process of evaluating a non-empty list. For example, the function \( K_1 \) might expect a different number of inputs than are present in the rest of the original list. Or the attempt to evaluate one of the data \( D_i \) might be undefined. Or the application of the function \( K_1 \) to the inputs \( K_2, \ldots, K_n \) might be undefined because, for example, the function expects numbers and it gets something else. In any of these cases, the result is undefined and DrScheme would report an error. Thus, none of the following lists can be evaluated:

- a list containing the numbers one, two and three
- a list containing two instances of the empty list
- a list containing the + symbol, followed by the boolean true and the boolean false

It is important to understand that each of the above lists is a valid Scheme datum: each one is a list. It’s just that these lists cannot be evaluated.

**Example 6.2.4**

Here’s an example of the default case of evaluating a non-empty list where things work out. Let \( L \) be the list containing the following data:

- \( D_1 \): the + symbol
- \( D_2 \): the number one
- \( D_3 \): the number two
- \( D_4 \): the number three

These Scheme data evaluate to the following:

- \( K_1 \): the addition function
- \( K_2 \): the number one
- \( K_3 \): the number two
- \( K_4 \): the number three

Since the first of these, \( K_1 \), is in fact a function, it can be applied to the inputs \( K_2, K_3 \) and \( K_4 \) (i.e., the numbers one, two and three). This results in the output six, which is itself a Scheme datum. The number six is the result of evaluating the original list \( L \), as illustrated below.

\[
\texttt{> (+ 1 2 3)} \\
\texttt{6}
\]

Notice that because the addition function is a primitive function, its operation is invisible to us. We observe the inputs going in and the output coming out, but we do not get to see how the output is generated.
The Default Rule for evaluating non-empty lists is how function application is made available to the Scheme programmer. In particular, if you want to apply a given function to a bunch of inputs, you create an expression that denotes the appropriate list and feed it to DrScheme.

The Default Rule has two steps. The first step involves evaluating each item in the original list, resulting in a bunch of new items. The second step involves applying the first new item—which must be a function—to the rest of the new items—which are the inputs to that function. The output value obtained by applying that function to those inputs is taken to be the output of evaluating the original list.

Scheme is called a functional programming language because function application is the central part of the computational model of Scheme. And the Default Rule is how the programmer gets function application to happen.

At this point, you should be able to write arbitrarily complex expressions that, when fed to DrScheme, cause correspondingly complex arithmetic computations to happen. That’s pretty good. However, we’ll have much more fun when we can design our own functions to do whatever we want them to do. For that, we’ll need the `define` and `lambda` special forms, which shall be described in the next chapter.

---

**Example 6.2.5**

The fact that 17 divided by 3 yields an answer (i.e., quotient) of 5 with a remainder of 2 can be confirmed by applying the built-in `quotient` and `remainder` functions:

```scheme
> (quotient 17 3)
5
> (remainder 17 3)
2
```

---

**Example 6.2.6**

According to the contract for (the simplest use of) the built-in `printf` function (cf. Section 5.5.1), if the `printf` function is applied to a single input that is a string, then it will display the contents of that string—without the double-quotes—in the Interactions Window as side-effect printing, but the output value will be the void datum, as illustrated below:

```scheme
> (printf "hi there")
hi there
> (printf "this is a long string!")
this is a long string!
```

*Note that the textual information displayed by DrScheme in each case is side-effect printing, not a Scheme output value. More interesting uses of the `printf` function will be described in Chapter 10.*

---

* By default, DrScheme clearly distinguishes side-effect printing from Scheme output values by displaying side-effect printing in one color, and output values in another, as illustrated in Fig. 6.2.

---

**Example 6.2.7**

According to the contract for the built-in `void` function (cf. Section 5.5.2), the `void` function can be applied to any number of inputs, but invariably returns the void datum as its output, as illustrated below:

```scheme
> (void)
>
```
Note that DrScheme does not display anything (other than the prompt) in the Interactions Window when the output value is the void datum.

Example 6.2.8

We can use the Default Rule to explicitly apply the evaluation function to some inputs, as demonstrated below:

```
> (eval +)
#<procedure:+>
```

In this example, the list contains two items: the eval symbol and the + symbol. To evaluate this list using the Default Rule, we first evaluate each item in the list:

- eval symbol → the evaluation function
- + symbol → the addition function

The second step of the Default Rule requires us to apply the first item (i.e., the evaluation function) to the second item (i.e., the addition function). Since Scheme functions always evaluate to themselves, the result is simply the addition function. DrScheme reports this result to us, in effect, the function associated with the + symbol.

6.3 Summary

The evaluation of non-empty lists plays a critical role in Scheme’s computational model. By default, non-empty lists are evaluated using the Default Rule. The Default Rule has two steps:

1. evaluate each element of the non-empty list; and
2. apply the result of evaluating the first element to the results of evaluating all of the rest of the elements.

The result from Step Two is taken to be the result of evaluating the original non-empty list.

The Default Rule enables a Scheme programmer to apply a function to any desired inputs: just ask DrScheme to evaluate a list whose first element evaluates to the desired function, and the rest of whose elements evaluate to
the desired inputs, as illustrated below:

```scheme
> (+ 3 (* 4 10))
43
```

As this example demonstrates, the evaluation of a list containing other lists is handled quite naturally: during the first step, when each element of the list must be evaluated, any subsidiary lists are evaluated by . . . the Default Rule!

Later on, when you create functions of your own (cf. Chapter 9) you will give each new function a name (cf. Chapter 7). By doing so, you will then be able to apply your new function to whatever inputs you wish, courtesy of the Default Rule.

The evaluation of non-empty lists is only defined when the first element of the list evaluates to a function; and the rest of the elements evaluate to appropriate inputs for that function. Asking DrScheme to evaluate non-empty lists that do not meet these criteria typically results in an error. (The special forms introduced in Chapter 7 are exceptions to this.)
Chapter 7

Special Forms

In DrScheme, there is a special class of symbol expressions called *keywords*. Examples of keywords include: `and`, `cond`, `define`, `dotimes`, `if`, `lambda`, `let`, `or`, and `quote`. Each of these keywords is a legal Scheme expression that denotes a symbol. For example, `quote` denotes the *quote* symbol, and `lambda` denotes the *lambda* symbol. For expository convenience, we may refer to expressions such as `quote` and `lambda` as keyword expressions, and the corresponding symbols (i.e., the *quote* symbol and the *lambda* symbol) as keyword symbols. However, that is not the interesting thing about keywords. The interesting thing about keywords is this:

* When the first element of a non-empty list is a keyword symbol, then that list is a *special form*; and each kind of special form has its own special mode of evaluation.

For example, each of the following expressions denotes a list that is a special form:

```scheme
(define x 3)
(quote (3 4 5))
(if condition then-clause else-clause)
(let ((x 4)) (+ x 8))
```

The important thing about special forms is that they are *not* evaluated according to the Default Rule introduced in Chapter 6. Instead, a special form is evaluated according to a special rule that is specific to the type of that special form—which is determined by the keyword symbol. Thus, there is one rule for evaluating `define` special forms, another rule for evaluating `quote` special forms, and so on. Importantly, each `define` special form is evaluated in the same way, just as each `quote` special form is evaluated in the same way. However, the rule for evaluating `define` special forms is very different from the rule for evaluating `quote` special forms.

Over the next several chapters, you will be introduced to about a dozen different kinds of special form. For each kind of special form, you will learn both the syntax and the semantics. The syntax of special forms is always in terms of a list whose first element is a keyword symbol; the rest of the list can be simple or complex, depending on the kind of special form. The semantics of a special form has two parts: (1) the list that is denoted by the special form expression, and (2) the special mode of evaluation for that kind of special form. As time goes on, you will use these special forms so often that their special modes of evaluation will become second nature to you. And, once you get the hang of it, learning the syntax and semantics for each new kind of special form will get easier and easier.

**Note.** In the Default Rule for evaluating non-empty lists, the first thing that happens is that each element of the list is evaluated, one after the other. In contrast, when evaluating a special form, which is also a non-empty list, some of the elements of that list may *not* be evaluated. Indeed, the *first* element of a special form (i.e., the keyword symbol) is *never* evaluated. (If DrScheme attempted to evaluate a keyword symbol, it would cause an error because the Global Environment typically does not contain entries corresponding to keyword symbols.)

The next sections introduce the `define` and `quote` special forms that you will use every day for the rest of your Scheme-programming life!
7.1 The define Special Form

The define special form is signaled by the define keyword.

7.1.1 The Syntax of the define Special Form

A define special form expression is any character sequence of the form

\[(define \ C_1 \ C_2)\]

where \(C_1\) is an expression denoting some Scheme symbol \(s\), and \(C_2\) can be any expression denoting any Scheme datum, \(e\), as illustrated below.

\[C_1 \rightarrow s \quad \text{and} \quad C_2 \rightarrow e\]

Therefore:

\[\text{(define } C_1 \ C_2) \rightarrow \text{List containing the define symbol, the } s \text{ symbol, and the datum } e\]

For example, \(\text{(define } x \ (\ + \ 3 \ 4))\) is a define special form expression that denotes a list containing:

(1) the define keyword symbol,
(2) the symbol \(x\), and
(3) the list denoted by \((+ \ 3 \ 4)\).

Some more examples of define special form expressions are given below.

\[
\begin{align*}
\text{(define addn-func } +) \\
\text{(define zero } 0) \\
\text{(define empty-list } ())
\end{align*}
\]

7.1.2 The Semantics of the define Special Form

Each special form denotes a list; the define special form is no exception. More interesting is what happens when a define special form is evaluated. The special rule for evaluating define special forms is illustrated below:

\[\text{(define } C_1 \ C_2) \rightarrow \text{[ List containing define, }s\text{ and }e\rightarrow ] \rightarrow \text{[ ]}\]

where the gray boxes are used to highlight the following fact:

\* The evaluation of a define special form does not generate any output value. (Well, technically, it generates the \textit{void} datum as its output. Recall from Section 2.1.4 that the \textit{void} datum is used to represent “no value”.)

Instead:

\* The purpose of the define special form is not to compute an output value, but to generate a very important \textit{side effect}—namely, to insert a new entry into the Global Environment.

DrScheme evaluates a define special form by taking the following steps, in order:

1. Insert a new entry, \(s \ [\text{void}]\), into the Global Environment, where \textit{void} is a temporary placeholder representing that there is not yet any value associated with the symbol \(s\).
2. Evaluate the datum \(e\), yielding some (usually different) datum \(E\): \(e \rightarrow E\).
3. Insert \(E\) as the value for \(s\) in the Global Environment: \(s \ [E]\).
Input Datum
List containing:
define symbol
The s symbol
The datum e

Input Datum
List containing:
define symbol
The s symbol
The datum e

Evaluation Function
no output!

Global Environment
symbol  value
i    i
s    E
i    i

side effect

(1) New entry inserted into Global Environment
(2) e ⇒ E
(3) E becomes value for s

Figure 7.1: The side effect of define: inserting a new entry into the Global Environment

This process, except for the part about the use of void as a temporary placeholder, is illustrated in Fig. 7.1.

The purpose of evaluating a define special form is its side effect: to create a new entry in the Global Environment. Since it does not generate any output value—or, rather, since it generates the void datum as its output—DrScheme does not display anything in the Interactions Window in response to define special forms, as illustrated below:

> (define x 6)
> (define y 3)
> (define z 34)
>

Of course, something has happened!

Example 7.1.1

Typing the character sequence, (define x (+ 1 2 3)), into the Interactions Window and hitting the Enter key would result in the number six being associated with the symbol x in the Global Environment, as illustrated below.

\[
x \rightarrow \text{the symbol } x \\
(+ 1 2 3) \rightarrow \text{a list containing the + symbol and the numbers one, two and three} \implies \text{the number six}
\]

Side Effect: New Global Environment Entry: the symbol x the number six

As noted above, DrScheme does not report any output value when evaluating a define special form. However, after evaluating it, subsequent attempts to evaluate the symbol x result in the value 6, as illustrated below:

> x
## Example 7.1.2: Confirming the semantics of `define`

The following admittedly unusual interactions confirm the semantics of the `define` special form.

```scheme
> (define w w)
> w
> (define w w)
> w
```

As described earlier, the following three steps are taken by DrScheme in evaluating the expression `(define w w)`: 

1. A new entry, `w` `void`, is inserted into the Global Environment.
2. The expression `w` is evaluated, yielding the value `void`: `w \rightarrow void`.
   (That’s what’s currently stored in the Global Environment as the value for `w`!)
3. That value (i.e., `void` is inserted as the value for `w` in the Global Environment.

Of course, in this case, the third step is redundant, since `void` is already there as the value for `w`.

Afterward, when we ask DrScheme to evaluate `w`, it does so, coming up with the answer `void`. However, since `void` is used to represent “no value”, DrScheme does not display anything! Instead, it just skips to the prompt, awaiting further instructions.

### Note

Since a keyword is a symbol, like any other Scheme symbol, you could use the `define` special form to assign some value to it in the Global Environment. However, this is a bad idea precisely because it would cause that symbol to lose its status as a keyword. Thereafter, you would not be able to use special forms relying on that keyword. This is something you might want to do once, just for fun. Afterward, you’ll want to hit DrScheme’s `Run` button to reset the Global Environment (i.e., to erase what you’ve done and thereby restore that symbol’s status as a keyword).
7.2 The quote Special Form

Recall that whenever we enter an expression into the Interactions Window, DrScheme invariably evaluates the corresponding Input Datum to generate an Output Datum. (You may wish to refer back to Fig. 4.1.) However, sometimes we are interested in data that cannot be evaluated (e.g., a list containing a bunch of Social Security numbers). Since attempting to evaluate such data would cause an error, and since DrScheme always performs an evaluation, we need some way of shielding data from DrScheme’s evaluation. That is the purpose of the quote special form.

7.2.1 The Syntax of the quote Special Form

The quote special form is indicated by the quote keyword. As a character sequence, it has the form

\[(\text{quote } C)\]

where \(C\) can be any legal Scheme expression. Below are listed several examples:

\[
\begin{align*}
(\text{quote } x) \\
(\text{quote } (1 \ 2 \ 3)) \\
(\text{quote } (\text{hi there } + \ \#t \ ())) \\
(\text{quote } (1 \ (2 \ (3))))
\end{align*}
\]

7.2.2 The Semantics of the quote Special Form

Each quote special form denotes a list. In particular, an expression of the form, \((\text{quote } C)\), denotes a list containing two items: the quote symbol and whatever \(C\) denotes. For example, the expression \((\text{quote } x)\) denotes a list containing the quote symbol and the symbol \(x\). Similarly, \((\text{quote } (1 \ 2 \ 3))\) denotes a list containing the quote symbol and a subsidiary list of numbers. More formally, if \(C\) denotes some datum, \(D\), then \((\text{quote } C)\) denotes a list containing the quote symbol and \(D\). Using the arrow notation, we can say:

If: \[C \rightarrow D\]  
Then: \[(\text{quote } C) \rightarrow \text{a list containing the quote symbol and } D\]

Evaluating quote special forms. The evaluation of a quote special form does not use the Default Rule for evaluating non-empty lists. Instead, quote special forms are evaluated using the following special rule:

\[\ast \text{ A list containing the quote symbol and } D \text{ evaluates to } \ldots \ D.\]

Notice that, according to this rule, neither the quote symbol nor the datum \(D\) are evaluated.\(^1\) Instead, \(D\) is the result of evaluating the two-element list. Indeed, the whole point of the quote special form is to shield \(D\) from evaluation.

Example 7.2.1

Each of the following is an example of a quote special form:

\[
\begin{align*}
> (\text{quote } x) \\
x \\
> (\text{quote } (1 \ 2 \ 3)) \\
(1 \ 2 \ 3) \\
> (\text{quote } (+ \ 2 \ 3))
\end{align*}
\]

\(^1\)In fact, the keyword symbol is never evaluated in a special form of any kind. The purpose of the keyword symbol is simply to indicate that the given list is a special form, thereby requiring a special mode of evaluation.
In the first example, `(quote x)` denotes a list containing the `quote` symbol and the symbol `x`. That list is the Input Datum. The result of evaluating that list is the symbol `x`—that is the Output Datum. Notice that the list is evaluated, but its second element is not. We can abbreviate this evaluation as follows:

\[(quote x) \rightarrow \{ \text{list with symbols quote and x} \} \Rightarrow \text{the symbol x} \rightarrow x\]

This is quite different from the Default Rule for evaluating non-empty lists. Well, that’s to be expected: the Default Rule was not used!

In the second example, `(quote (1 2 3))` denotes a list containing the `quote` symbol and a subsidiary three-element list. The result of evaluating that list is its second element (i.e., the subsidiary three-element list). Notice that the list containing the numbers one, two and three has not been evaluated. Indeed, any attempt to evaluate such a list would cause DrScheme to report an error since the first element of that list does not evaluate to a function. This example illustrates the use of a list as a container for data rather than something we’d like to have evaluated. The `quote` special form comes in handy for such cases.

In general, if \(C\) is an expression denoting some datum \(D\), then entering the expression, `(quote C)`, into DrScheme will cause the following to happen:

\[(quote C) \rightarrow \{ \text{list containing quote symbol and D} \} \Rightarrow D \rightarrow C'\]

Notice that the Input Datum is the two-element list that contains the `quote` symbol and the datum \(D\). The Output Datum is simply \(D\). Notice, too, that DrScheme may use a different character sequence, \(C'\), to describe \(D\) to us; however, \(C'\) must nonetheless denote \(D\). (An example of this will be given shortly.)

### Example 7.2.2

Notice the difference between the evaluations of \(x\) and `(quote x)` below:

\[
\begin{array}{c}
> (define x (+ 1 2 3)) \\
> x \\
6 \\
> (quote x) \\
> x
\end{array}
\]

### Example 7.2.3

Here, we use the `define` special form to create a variable named `my-list` whose value is a three-element list. Notice the use of the `quote` special form to shield the three-element list from evaluation.

\[
\begin{array}{c}
> (define my-list (quote (1 2 3))) \\
> my-list \\
(1 2 3)
\end{array}
\]

### 7.2.3 Alternate Syntax for `quote` Special Forms

Since `quote` special forms are used so frequently, there is an alternate syntax for them. In particular, if \(C\) is any Scheme expression denoting some datum \(D\), then the expressions, `(quote C)` and `
\(C\)`, denote the same two-element list—namely, a list containing the `quote` symbol and the datum \(D\):
The two character expressions are quite different, but both represent the same list! (Syntax vs. Semantics!)

Example 7.2.4

The expressions, ‘num and (quote num), each represent a list containing the quote symbol and the num symbol, as illustrated below:

```scheme
> (quote num)
num
> ’num
num
```

Although the abbreviation for quote special forms is useful, it requires care to remember that such expressions denote lists—and that those lists are evaluated using the special rule for the quote special form.

Example 7.2.5

The following examples demonstrate the equivalence between the two kinds of syntax for the quote special form.

```scheme
> (quote (quote x))
’x
> ’’x
’x
> (quote 000)
0
> ’000
0
```

In the first example, DrScheme has chosen a different character sequence for describing the Output Datum—in this case, a list containing the quote symbol and the x symbol. Similar remarks apply to the third and fourth examples, where the number zero has been shielded from evaluation, but DrScheme has chosen to report the result using a more compact character sequence.

7.3 Summary

This chapter introduced special forms. A special form is a non-empty list whose first element is one of Scheme’s special keyword symbols (e.g., define or quote). The keyword symbol determines the kind of special form (e.g., a define special form or a quote special form). Although they are non-empty lists, special forms are not evaluated by the Default Rule; instead, each kind of special form is evaluated by its own special rule: one rule for define special forms, one rule for quote special forms, and so on. The rules for evaluating special forms are very different from the Default Rule. For example, the first element of a special form is never evaluated. And, frequently, some or all of the other elements are not evaluated either. This chapter focused on the define and quote special forms.

* The define special form has no output value, but a very useful side effect: it inserts a new entry into the Global Environment.

* The quote special form is used to shield a datum from evaluation; it has no side effects.
The `define` special form enables us to use symbols as variables (i.e., names for pieces of data). Later on, when you create functions of your own design, you will typically use the `define` special form to give them names. In turn, this will enable you to apply your new functions to any desired inputs simply by asking DrScheme (and the Default Rule) to evaluate an appropriate non-empty list.

The `quote` special form is useful when treating symbols or non-empty lists as pieces of data, rather than using them as names of variables or vehicles for applying functions to inputs. For example, the Default Rule would have problems evaluating a list containing a bunch of student names, but the `quote` special form could be used to shield that list from evaluation, as illustrated below:

```scheme
> (quote (john paul george ringo))
(john paul george ringo)
> '(john paul george ringo)
(john paul george ringo)
```

**Special Forms Introduced in this Chapter**

- `define` For inserting a new entry in the Global Environment
- `quote` For shielding a Scheme datum from evaluation
Chapter 8

Predicates

A function whose output is always a boolean (i.e., true or false) is called a predicate. (This is just convenient terminology; there is no predicate type in Scheme.) This chapter describes some of the commonly used, built-in Scheme predicates and illustrates their use.

8.1 Type-Checker Predicates

Scheme includes a bunch of primitive data types, including: number, boolean, symbol, null and function. Scheme also includes non-primitive data types, including strings and non-empty lists. For each one of these data types, Scheme includes a primitive function called a type-checker predicate. When a type-checker predicate is applied to some Scheme datum, it outputs true if that datum belongs to the indicated data type; otherwise, it outputs false. Thus, the type-checker predicate associated with the number data type outputs true whenever the input belongs to the number data type. Similarly, the type-checker predicate associated with the list data type outputs true whenever the input datum belongs to the list data type. And so on.

For convenience, each of these type-checker predicates has an easy-to-remember name. In other words, for each type-checker predicate there is an entry in the Global Environment that links a particular symbol with that predicate. Thus, those symbols can be used to refer to the type-checker predicates. For example, the symbol number? evaluates to the type-checker predicate for the number data type; the symbol boolean? evaluates to the type-checker predicate for the boolean data type; and so on.

Example 8.1.1

The following Interactions Window session demonstrates the existence of some of the built-in type-checker predicates.

```
> number?
#<procedure:number?>
> symbol?
#<procedure:symbol?>
> boolean?
#<procedure:boolean?>
> list?
#<procedure:list?>
> null?
#<procedure:null?>
> procedure?
#<procedure:procedure?>
```

1The list data type is a compound data type that includes both non-empty lists and the empty list.
Notice that the symbols mirror the names of the corresponding data types, except that the symbol associated with the type-checker predicate for functions is `procedure?`, not `function?`.\footnote{This text uses the terms, `function` and `procedure`, interchangeably; however, the term `function` seems better suited given that Scheme is typically referred to as a functional programming language.}

Each type-checker predicate is a function that can be applied to a single input. That input can be any type of Scheme datum. A type-checker predicate returns `true` if that input datum is of the appropriate data type.

---

**Example 8.1.2**

Here’s a contract for the built-in `number?` type-checker predicate:

<table>
<thead>
<tr>
<th>Name:</th>
<th>number?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>(d), any Scheme datum</td>
</tr>
<tr>
<td>Output:</td>
<td><code>#t</code> if (d) is a number; otherwise, <code>#f</code></td>
</tr>
</tbody>
</table>

The contracts for the other type-checker predicates are similar.

---

**Example 8.1.3**

The following interactions illustrate the behavior of the type-checker predicates.

```scheme
> (number? 3)  #t
> (number? #t)  #f
> (boolean? #f)  #t
> (boolean? ’x)  #f
> (symbol? +)  #f
> (symbol? ’+)  #t
> (null? ())  #t
> (null? ’(+ 1 2))  #f
> (procedure? +)  #f
> (procedure? ’+)  #f
> (list? ’(+ 1 2))  #t
> (list? ())  #t
```
Each of these expressions denotes a non-empty list that is evaluated according to the Default Rule. In each case, the first element of the list is a symbol that evaluates to a function, which is then applied to whatever the second element evaluates to. Notice that the + symbol in (procedure? +) evaluates to the addition function, whereas the ’+ expression in (procedure? ’+) evaluates to the + symbol. Notice too that the list? type-checker predicate returns true for any list, whether empty or non-empty. Finally, recall that void is a built-in function whose output is the void datum. Thus, (void) evaluates to the void datum, whereas the symbol void evaluates to the built-in function.

### 8.2 Comparison Predicates

In addition to the primitive arithmetic functions for addition, subtraction, multiplication and division, Scheme includes several predicates for comparing numbers. Examples include the greater-than, less-than and equal predicates. To enable us to refer to such predicates, each is associated with a particular symbol in the Global Environment.

- > greater than
- >= greater than or equal to
- = equal to
- < less than
- <= less than or equal to

Each of these predicates, when applied to two numeric inputs, generates the expected boolean output, as illustrated below.

<table>
<thead>
<tr>
<th>Example 8.2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (&gt; 3 4)</td>
</tr>
<tr>
<td>#f</td>
</tr>
<tr>
<td>&gt; (&gt; 4 3)</td>
</tr>
<tr>
<td>#t</td>
</tr>
<tr>
<td>&gt; (&gt;= 4 3)</td>
</tr>
<tr>
<td>#t</td>
</tr>
<tr>
<td>&gt; (= 3 4)</td>
</tr>
<tr>
<td>#f</td>
</tr>
<tr>
<td>&gt; (= 3 3)</td>
</tr>
<tr>
<td>#t</td>
</tr>
</tbody>
</table>

---

2 In other contexts, these predicates are commonly called *relational operators*.

3 These predicates can also be applied to more than two inputs; however, we shall postpone discussion of such things until Chapter 16.
DrScheme also provides a comparison predicate called `eq?` that is more general than the `=` predicate. Whereas the `=` predicate only works on numerical input, the `eq?` predicate can be used to test the equality of inputs that can be any combination of numbers, booleans, symbols or the empty list. Here’s a contract for the `eq?` predicate.

<table>
<thead>
<tr>
<th>Name:</th>
<th><code>eq?</code></th>
</tr>
</thead>
</table>
| Inputs:  | \(d_1\), a number, boolean, symbol, or the empty list  \\
|          | \(d_2\), a number, boolean, symbol, or the empty list  |
| Output:  | \(\#t\) if \(d_1\) and \(d_2\) are the same; \(\#f\) otherwise. |

**Example 8.2.2**

*Here are some examples of the `eq?` predicate in action.*

\[
\begin{align*}
\texttt{> (eq? 3 3)} & \quad \texttt{#t} \\
\texttt{> (eq? 3 'x)} & \quad \texttt{#f} \\
\texttt{> (eq? 'x 'x)} & \quad \texttt{#t} \\
\texttt{> (eq? 'x #t)} & \quad \texttt{#f} \\
\texttt{> (eq? 'x ())} & \quad \texttt{#t} \\
\texttt{> (eq? () () )} & \quad \texttt{#t}
\end{align*}
\]

The `eq?` predicate is most frequently used to compare whether two symbols are the same. If you know that the inputs will be numbers, then you should use the `=` function. And if you know that the inputs will be booleans … stay tuned!

* The `eq?` function does not work well when comparing non-empty lists! More on that later!

### 8.3 Summary

This chapter introduced *predicates*—that is, functions that generate boolean output values. DrScheme provides a wide variety of built-in predicates. Each built-in predicate has a corresponding entry in the Global Environment so that it can be used by a Scheme programmer. For example, the built-in *less-than* predicate is the value associated with the `<` symbol in the Global Environment. By taking advantage of the Default Rule for evaluating non-empty lists, the *less-than* function can be applied to inputs, as demonstrated below:

\[
\begin{align*}
\texttt{> (< 3 4)} & \quad \texttt{#t} \\
\texttt{> (< (+ 2 3) (- 10 9))} & \quad \texttt{#f}
\end{align*}
\]

This chapter introduced two sets of built-in predicates: *type-checker predicates* and *comparison predicates*. Type-checker predicates simply check whether a given datum belongs to a specified data type. For example, the `number?` predicate checks whether its input is a number, and the `list?` predicate checks whether its input is a list, as demonstrated below:

\[
\begin{align*}
\texttt{> (number? 3)} & \quad \texttt{#t} \\
\texttt{> (number? '(a b c))} & \quad \texttt{#t}
\end{align*}
\]
The `list?` predicate works for any kind of list: empty or non-empty. The `null?` predicate works only for the empty list. The `procedure?` predicate works for functions. The comparison predicates include the standard functions for comparing numbers (e.g., `less-than` and `greater-than-or-equal-to`), as well as the more general `eq?` predicate that works on any combination of numbers, booleans, symbols, or the empty list.

**Built-in Functions Introduced in this Chapter**

**Type-checker Predicates:** `number?`, `symbol?`, `boolean?`, `list?`,

**Comparison Predicates:** `<`, `<=` `,`, `>=`, `>` (these work only on numbers).
`eq?` (this works on numbers, booleans, symbols or the empty list).
Chapter 9

Defining Functions

So far, what we know about Scheme is enough to enable us to use the Interactions Window like we would a glorified calculator. There are lots of built-in functions that we can apply to various kinds of input. Each built-in function has a more-or-less convenient name (i.e., for each built-in function there is an entry in the Global Environment that links a particular symbol to that function). However, the fun won’t really begin until we can design our own functions to do whatever we want them to do. This chapter describes how to do this in the Scheme programming language.

9.1 Defining Functions vs. Applying Them to Inputs

Example 9.1.1

In a math class, you might see a function defined using an equation such as

\[ f(x) = x^2 \]

In this case, the name of the function is \( f \), and we might casually describe it as the squaring function—because for each possible input value, \( x \), the corresponding output value is the square of \( x \) (i.e., \( x^2 \)). Notice that the mathematical definition, \( f(x) = x^2 \), gives a prescription for generating appropriate output values should \( f \) ever happen to be applied to any input values. In particular, the definition of \( f \) includes an input parameter, \( x \), that is used to refer to potential input values. In addition, the expression, \( x^2 \), on the righthand side of the equation indicates how to compute the corresponding output value for any given value of \( x \). (The expression on the righthand side is sometimes referred to as the body of the function.) For example, if we wanted to know the output value generated by \( f \) when given 3 as its input, we could get the answer by first substituting the value 3 for \( x \) in the expression, \( x^2 \), yielding 3\(^2\). Evaluating the expression, 3\(^2\), would then yield the desired output value, 9. Similarly, if we wanted to know the output value generated by \( f \) when given the input value 4, we would first substitute the value 4 for \( x \) in the expression, \( x^2 \), yielding 4\(^2\), which evaluates to 16.

Example 9.1.2

In the preceding example, the function \( f \) took a single input value. However, we can similarly define functions that take multiple inputs. For example, the function, \( g \), defined below, takes two inputs, represented by the input parameters \( w \) and \( h \):

\[ g(w, h) = wh \]

This function can be used to compute the area of a rectangle whose width is \( w \) and height is \( h \). To apply this function to the input values, 3 and 7, we first substitute 3 for \( w \), and 7 for \( h \) in the expression, \( wh \), yielding 3 \( \cdot \) 7. Evaluating this expression results in the desired output value, 21.
In general, the mathematical definition of a function specifies how to generate appropriate output values should it ever be applied to any input values. A function definition includes a list of input parameters and a body. Once a function has been defined, it can be applied to appropriate input values as follows. First, the desired input values are substituted for the appropriate input parameters in the body of the function. Next, the resulting expression is evaluated, thereby yielding the desired output value.

**Example 9.1.3**

The following defines a function, \( v \), that can be used to compute the volume of a cone:

\[
v(r, h) = \frac{1}{3} \pi r^2 h
\]

It has two input parameters, \( r \) and \( h \), that respectively represent the radius and height of the cone. To compute the volume of a cone of radius \( 3 \) and height \( 2 \), we apply the function \( v \) to the input values \( 3 \) and \( 2 \), as follows. First, we substitute the values \( 3 \) and \( 2 \) for \( r \) and \( h \), respectively, in the body, \( \frac{1}{3} \pi r^2 h \), yielding the expression, \( \frac{1}{3} \pi (3^2)(2) \). Evaluating this expression yields the desired output value, \( 6 \pi \).

### 9.2 The lambda Special Form

The Scheme programming language provides the lambda special form to enable us to specify functions of our own design.

* The use of the lambda symbol in a lambda special form is a tip of the cap to the fact that the underlying mathematical theory, originally developed in the 1930s, is called the Lambda Calculus.

Like any special form in Scheme, the lambda special form is a list whose first element is a keyword symbol—in this case, the symbol lambda. The second element in a lambda special form is used to specify the input parameter(s) for the function being defined. The rest of the elements in the lambda special form constitute the body of the function being defined. If you’re wondering where the name of the function is specified, recall that the define special form is used to assign names to things in Scheme. Furthermore, a single function could have several different names. Thus:

* The lambda special form specifies everything about a function except its name.

**Example 9.2.1: The Squaring Function in Scheme**

Recall the mathematical definition of the squaring function:

\[
f(x) = x^2
\]

This mathematical definition does three things:

- It specifies a single input parameter, \( x \), for the function being defined;
- It specifies a body, \( x^2 \), for the function being defined; and
- It specifies a name, \( f \), for the function being defined.

In Scheme, the first two jobs are handled by the lambda special form. For example, the following lambda expression can be used to specify a squaring function in Scheme:

\[
(\text{lambda} \ (\text{x}) \ (* \ \text{x} \text{x}))
\]

This lambda expression denotes a lambda special form (i.e., a Scheme list whose first element happens to be the lambda symbol). Like any special form, a lambda special form has its own, special rule for being evaluated. For now, suffice it to say that:
The evaluation of a lambda special form always results in a function. Thus, if the expression, `(lambda (x) (* x x))`, is typed into the Interactions Window, DrScheme will report that its evaluation yields a function, as illustrated below:

```
> (lambda (x) (* x x))
#<procedure>
```

Admittedly, the character sequence generated by DrScheme is not very descriptive. It simply says that the evaluation of the corresponding lambda special form has resulted in a function.

At this point, it is important to stress that the function has been created; however, it has not yet been applied to any inputs!

We can demonstrate that the function created above behaves like a squaring function by first giving it a name and then applying it to a variety of input values. The following Interactions Window session demonstrates how to name our function:

```
> (define square (lambda (x) (* x x)))
```

The `define` special form is used to create an entry in the Global Environment that associates the `square` symbol with the function specified by the lambda expression. Recall that when a `define` special form is evaluated, the given symbol—in this case, `square`—is not evaluated; however, the given expression—in this case, `(lambda (x) (* x x))`—is evaluated. Thus, the value associated with the `square` symbol is the function that results from evaluating the given `lambda` special form, as demonstrated below:

```
> square
#<procedure:square>
```

Once we have given a name to our function, we can then use it like any of the built-in functions, as demonstrated below:

```
> (square 3)
9
> (square 4)
16
> (square -8)
64
```

Each of the above expressions is evaluated using the Default Rule for evaluating non-empty lists. In each case, the `square` symbol evaluates to the function that we defined earlier, which is then applied to the desired input value.

Example 9.2.2

Incidentally, it is possible to define and apply a function without ever having given it a name, as the following Interactions Window session demonstrates:

```
> ((lambda (x) (* x x)) 4)
16
```

The Default Rule for evaluating non-empty lists is used to evaluate the above expression. In the process, each element of the list is evaluated. The first element of the list is the `lambda` special form, which
evaluates to the (unnamed) squaring function. The second element of the list evaluates to the number four. The result of applying that function to that input yields the desired output, sixteen. Later on, we shall encounter situations where it is convenient to use functions without bothering to name them.

Example 9.2.3

The following Interactions Window session demonstrates how to define, name, and apply functions analogous to the functions, \( g(w, h) = wh \) and \( v(r, h) = \frac{1}{3}\pi r^2 h \), seen earlier:

```scheme
> (define rect-area (lambda (w h) (* w h)))
> (rect-area 2 3)
6
> (rect-area 3 8)
24
> (define cone-volume (lambda (r h) (* 1/3 3.14159 r r h)))
> (cone-volume 3 2)
18.849539999999998
> (cone-volume 10 1)
104.71966666666665
```

In the cone function, \( 3.14159 \) is used as an approximation of \( \pi \), and the expression, \( (* 1/3 3.14159 r r h) \), takes advantage of the fact that the built-in multiplication function can be applied to any number of input values.

9.3 The Syntax and Semantics of Lambda Expressions

This section presents the syntax and semantics of lambda expressions. Initially, it restricts attention to those in which the body consists of a single expression; later, it addresses those in which the body consists of multiple expressions.

9.3.1 The Syntax of a Lambda Expression

A lambda expression has the following syntax:

\[
\text{(lambda } (C_1 \ C_2 \ldots \ C_n) \ B)\]

where:

- each \( C_i \) is a character sequence denoting some Scheme symbol, \( s_i \);  
- the symbols, \( s_1, s_2, \ldots, s_n \), are distinct (i.e., there are no duplicates); and  
- \( B \) is a character sequence denoting a Scheme datum, \( D \), of any kind.

Thus, \( C_1, C_2, \ldots, C_n \) specify \( n \) distinct input parameters for the lambda expression, and \( B \) specifies the body of the lambda expression.

Example 9.3.1

The following are examples of well-formed lambda expressions:

- (lambda () 44)
- (lambda (x) (* x x))
• \( (\text{lambda} \ (w \ h) \ (* \ w \ h)) \)
• \( (\text{lambda} \ (r \ h) \ (* \ 1/3 \ 3.14159 \ r \ r \ h)) \)
• \( (\text{lambda} \ (x \ y \ z) \ (* \ x \ (- \ y \ z))) \)

For the last expression, \((x \ y \ z)\) specifies the parameter list and \((* \ x \ (- \ y \ z))\) specifies the body.

---

**Example 9.3.2**

In contrast, the following are examples of malformed lambda expressions:

• \( (\text{lambda} \ (x \ y) \ (* \ x \ y)) \)
• \( (\text{lambda} \ (x \ 10) \ (* \ x \ 10)) \)
• \( (\text{lambda} \ x) \)

---

### 9.3.2 The Semantics of a Lambda Expression

The semantics of a lambda expression stipulates the Scheme datum that the lambda expression denotes, as well as how that Scheme datum is evaluated. As suggested by the preceding examples, a lambda expression invariably denotes a list—called a lambda special form—and the evaluation of that list invariably results in a Scheme function. The semantics of the lambda expression also includes a description of the subsequent behavior of that function should it ever be applied to any input(s).

The list denoted by a lambda special form. Assuming that

• each \( C_i \) denotes a Scheme symbol, \( s_i \);
• the symbols, \( s_1, s_2, \ldots, s_n \), are distinct; and
• \( B \) denotes some Scheme datum \( D \),

then a lambda expression of the form

\[
(\text{lambda} \ (C_1 \ C_2 \ \ldots \ C_n) \ B)
\]

denotes a Scheme list whose elements are as follows:

• the lambda symbol;
• a list containing \( n \) distinct symbols, \( s_1, s_2, \ldots, s_n \); and
• the Scheme datum, \( D \)

This list is referred to as a lambda special form.

**Note.** By now, you should be getting used to the fact that a piece of syntax, such as \((\text{lambda} \ (x) \ (* \ x \ x))\),

denotes a Scheme datum—in this case, a Scheme list containing the lambda symbol and two subsidiary lists. Although it is important to be able to distinguish expressions from the Scheme data they denote, doing so can get quite tedious in chapter after chapter. Therefore, for the sake of expository convenience, the rest of this book shall frequently blur this distinction. Thus, we may talk of the list, \((1 \ 2 \ 3)\), even though we really mean the list denoted by the expression \((1 \ 2 \ 3)\). Similarly, we may say that the expression \((\text{lambda} \ (x) \ (* \ x \ x))\) evaluates to a function, when we really mean that the list denoted by the expression \((\text{lambda} \ (x) \ (* \ x \ x))\) evaluates to a function.
The Evaluation of a $\text{lambda}$ Special Form

$\star$ The most important thing to know about the evaluation of a $\text{lambda}$ special form is that the result is invariably a function; however, the evaluation of a $\text{lambda}$ special form only creates the function; it does not apply it to any input(s).

For convenience, we shall refer to such functions as $\text{lambda}$ functions. Thus, a $\text{lambda}$ function is a function that resulted from having evaluated a $\text{lambda}$ special form.

Although evaluating a $\text{lambda}$ special form only creates the corresponding function, it is necessary to describe what that function would do if it ever were applied to input values. That is the subject of the next section.

9.3.3 Applying a $\text{lambda}$ Function to Input Values

Up to this point, the only environment that we have considered has been the Global Environment. However, when a $\text{lambda}$ function is applied to inputs, the expressions in the function’s body are evaluated with respect to an automatically-created local environment. As will be seen, the relationship between the Global Environment and the new local environment is one of inclusion: the local environment can be thought of as a smaller room that sits inside the Global Environment.

$\star$ For the purposes of this chapter, it is assumed that the $\text{lambda}$ function was created by evaluating its $\text{lambda}$ special form with respect to the Global Environment, as has been the case in all of the preceding examples.

---

Example 9.3.3: Applying the Squaring Function

Consider the expression, $(\text{lambda} \ (x) \ (* \ x \ x))$. As noted above, it evaluates to a Scheme function. When this lambda function is applied to some input value, say $4$, the following things happen:

- A local environment is created that contains a single entry in which the symbol $x$ has the value $4$.
- The expression, $(* \ x \ x)$, which constitutes the body of the function, is evaluated with respect to the newly created local environment. This means that: (1) any occurrence of the symbol $x$ is evaluated using the entry for $x$ in the local environment, ignoring any entry for $x$ that might exist in the Global Environment; and (2) all other symbols are evaluated with respect to the Global Environment. The evaluation of $(* \ x \ x)$ therefore yields the result $16$, because $x$ evaluates to $4$ in the local environment, and $*$ evaluates to the built-in multiplication function in the Global Environment.
- That value, $16$, is taken to be the output value that results from applying the $\text{lambda}$ function to the input value $4$.

This process is illustrated in Fig. 9.1.

---

Example 9.3.4: Computing the Volume of a Sphere

You may recall that the volume of a sphere of radius, $r$, is given by the function $f(r) = \frac{4}{3} \pi r^3$. Thus, for example, the volume of a sphere of radius $1$ is $\frac{4}{3} \pi$; and the volume of a sphere of radius $2$ is $\frac{32}{3} \pi$.

The following Interactions Window session first creates a global variable, $\pi$, to hold the value $3.14159$. It then defines a function, named $\text{sphere-volume}$. Finally, it applies this function to some sample input values.

```scheme
> (define pi 3.14159)
> (define sphere-volume (lambda (r) (* 4/3 pi r r r)))
> (sphere-volume 1)
```
Consider the evaluation of the expression, (sphere-volume 2). It involves the following steps:

- First, a local environment is set up containing a single entry in which the symbol \( r \) has the value 2.

- Next, the expression, \((\frac{4}{3} \pi r r r)\), which constitutes the body of the function, is evaluated with respect to that local environment. In the process, the \( \ast \) symbol evaluates to the built-in multiplication function, \( \frac{4}{3} \) evaluates to itself, the symbol \( \pi \) evaluates to 3.14159, and the symbol \( r \) evaluates to 2. Applying the multiplication function to the values \( \frac{4}{3}, 3.14159, 2, 2 \) and 2 yields the result: 33.51029333333333.

- Finally, the value 33.51029333333333 is reported as the output value generated by applying the sphere-volume function to the input value 2.

Notice that in the second step, the value for \( r \) came from the local environment, whereas the values for \( \ast \) and \( \pi \) came from the Global Environment.

* When evaluating a symbol such as \( r \) or \( \pi \) with respect to a local environment, if the symbol has an entry in the local environment, that entry is used; otherwise, the symbol’s value is derived from the Global Environment.

The evaluation of (sphere-volume 2) is illustrated in Fig. 9.2.

The following Interactions Window session (continuing from the one given above) illustrates that the existence of a global variable named \( r \) has no effect on the local variable that also happens to be named
* mult. func.
+ pi
3.14159

---

**Figure 9.2: Applying the sphere-volume function to the value 2**

---

In contrast, changing the value of the global variable, \texttt{pi}, has disastrous effects! (That is one of many reasons why the use of global variables should be very carefully restricted!)

```scheme
> (define r 55)
> (sphere-volume 1)
4.188786666666666
> (sphere-volume 2)
33.51029333333333
> (define pi 100)          ← Yikes!!
> (sphere-volume 1)         ← Yikes!!
400/3
```

---

**Example 9.3.5: More Complex Input Expressions**

So far, the examples have involved simple input expressions such as 1 or 2. This example demonstrates that complex input expressions can be handled without requiring any new evaluation tools. Consider the following Interactions Window session:

```scheme
> (define square (lambda (x) (* x x)))
> (square (+ 2 3))
25
> (square (- 8 5))
9
```
> (square (square 10))
10000

The evaluation of the first expression simply defines a squaring function, as seen in previous examples. The evaluation of the expression, (square (+ 2 3)), is done according to the Default Rule for evaluating non-empty lists. In particular:

- The square symbol evaluates to the squaring function;
- The expression, (+ 2 3), evaluates to 5;
- The squaring function is applied to the input value 5, generating the output value 25.

Similar remarks apply to the evaluation of (square (- 8 5)) and (square (square 10)). In each case, the input expressions, no matter how complex, are evaluated first to generate the corresponding input values. For example, the evaluation of (square (square 10)) involves the following steps:

- The square symbol evaluates to the squaring function;
- The expression, (square 10), evaluates to 100;
- The squaring function is applied to 100, yielding the output value, 10000.

Notice that the evaluation of the input expression, (square 10), itself required using the Default Rule for evaluating non-empty lists. In particular:

- The square symbol evaluates to the squaring function;
- The expression, 10, evaluates to 10; and
- The squaring function is applied to 10, yielding the output value 100.

---

Example 9.3.6

Here’s an example of a function that takes more than one input (i.e., parameter).

> (define discriminant
  (lambda (a b c)
    (- (* b b) (* 4 a c))))
> (discriminant 1 2 -4)
20
> (discriminant 1 0 -3)
12

Notice that the syntax of Scheme allows expressions to occupy multiple lines. This is quite useful when writing longer expressions. DrScheme automatically indents sub-expressions to make longer expressions easier to read. Hitting the tab key will automatically cause the current line to snap to the appropriate amount of indentation.

---

Differences Between Mathematical Notation and Lambda Notation

Recall that in a math class, you might define a function using an equation such as \( f(x) = x^2 \). Later on, you might apply that function to various inputs, using expressions such as \( f(3) = 9 \) or \( f(5) = 25 \).

In Scheme, we can use a lambda special form to define a function without giving it a name. For example, we might evaluate `(lambda (x) (* x x))` to create a squaring function. However, we cannot replace the
parameter \( x \) in that lambda expression by arbitrary expressions. For example, \((\text{lambda } (3) (* 3 3))\) is malformed in Scheme. (Recall the rules of syntax for lambda expressions.) But we can see a similarity to the common mathematical notation for applying functions to inputs as follows.

**Example 9.3.7: lambda functions vs. mathematical functions**

```scheme
> (define f (lambda (x) (* x x)))
> (f 3)
  9
> (f (+ 2 3))
  25
> (f (f 10))
  10000
```

The corresponding mathematical equations/expressions would be:

\[
\begin{align*}
  f(x) &= x^2 \\
  f(3) &= 9 \\
  f(2 + 3) &= 25 \\
  f(f(10)) &= 10000
\end{align*}
\]

**Example 9.3.8: A Lambda Expression with a Bigger Body**

The following illustrates that a lambda expression can have more than one expression in its body.

```scheme
> (define useless-function
  (lambda (input)
    input
    (* input input)
    (* input input input)
    input
    ()))
> (useless-function 35)
  ()
> (useless-function 888)
  ()
```

In this case, the body of the function includes five expressions (i.e., everything after the parameter list).

* The semantics of Scheme stipulates that when a lambda function having multiple expressions in its body is subsequently applied to input(s), the expressions in the body are evaluated sequentially, one after the other.

* Furthermore, the value of the last expression in the body is taken to be the output value for the function.

Thus, in the above example, each of the expressions in the body is evaluated in turn, and the value of the last expression (i.e., \(()\)) serves as the output value. This function is kind of silly since the results of evaluating the first four expressions in its body are thrown away.

* The only way that intermediate expressions in the body of a function could have any impact is if they caused side effects.
Up to this point, the only function that we have seen that has side effects is the built-in `printf` function. It displays the contents of a string in the Interactions Window. This is a harmless side effect that can be very useful.

### 9.4 Summary

This chapter introduced the `lambda` special form whose purpose is to enable a Scheme programmer to specify functions. A `lambda` special form includes:

1. the `lambda` symbol;
2. a list of input parameters; and
3. one or more expressions constituting the body of the function.

The result of evaluating a `lambda` special form is always a function. For example, the result of evaluating `(lambda (x) (* x x))` is a function whose sole input parameter is `x`, and whose body is `(* x x)`.

In Scheme, the following are distinct:

- The function that is generated by evaluating a `lambda` special form;
- Any name(s) that might be given to that function; and
- The process of applying that function to input(s).

The `define` special form is used to give names to things, including functions. For example, the following expression associates the squaring function with the name `square`.

```
(define square
  (lambda (x)
    (* x x)))
```

The application of this function to an input is handled by the evaluation of an expression such as `(square 10)`, which is carried out by the Default Rule for evaluating non-empty lists.

The application of a lambda function involves the creation of a local environment that contains one entry for each input parameter. The input values to which the function is being applied become the values associated with the corresponding input parameters in the local environment. For example, when applying the squaring function to the input value 10, the input parameter `x` receives the value 10 in the local environment. Next, each expression in the body of the function is evaluated with respect to that local environment. In particular, any symbol `s` that must be evaluated is evaluated by looking first for a corresponding entry in the local environment; if no entry for `s` is found there, then the Global Environment is checked. In other words, the local environment has higher priority when evaluating symbols in the body of a lambda function. Thus, when evaluating `(* x x)` in the body of the squaring function, `x` evaluates to 10, courtesy of the local environment, whereas `*` evaluates to the built-in multiplication function courtesy of the Global Environment. Finally, the output obtained by evaluating the last expression in the body of the function is taken to be the result of applying the function to the given input(s). Thus, the output 100, obtained by evaluating `(* x x)`, is taken to be the output value for the application of the squaring function to the input value 10.

The parameter lists in a `lambda` special form may specify zero or more parameters, each represented by a Scheme symbol. And the body of a `lambda` special form may include one or more expressions. However, it is only reasonable to include more than one expression in the body of a function if the evaluation of those expressions cause some side effects.

### Special Forms Introduced in this Chapter

- `lambda` Used to specify functions of our own design.
Chapter 10

Some practicalities

This chapter introduces the following practicalities:

- Further capabilities of the built-in `printf` function. This function, which takes a `string` as one of its inputs, can be used to display nicely formatted information in DrScheme’s Interactions Window. Its functionality is similar to that of the `format/print` functions found in many programming languages.

- The built-in `load` function. This function causes the Scheme expressions in a specified file to be evaluated as though they had been manually typed into the Interactions Window. As such, these expressions are evaluated with respect to the Global Environment. In this way, a library of useful Scheme definitions can be incorporated into your own program quite easily. The name of the file is specified by a `string`.

- Comments. A comment is a piece of syntax that DrScheme completely ignores. Comments are used by programmers to help clarify—for people—what the program/code is supposed to do.

10.1 More Fun with the Built-in `printf` Function

Recall from Section 5.5 that the built-in `printf` function can be applied to strings to generate side-effect printing in the Interactions Window, as illustrated below.

```
> (printf "you are amazing!")
you are amazing!
```

Unlike the input string "you are amazing!", which is a Scheme datum, the text displayed in the Interactions Window is not a Scheme datum; instead, it is merely something that happens on the side. The output value generated by the `printf` function is the `void` datum.

**Escape sequences.** In the above example, the `printf` function effectively copied the contents of the input string into the Interactions Window verbatim. However, the `printf` function sometimes deviates from this simple behavior. In particular, as the `printf` function walks through the input string, it reacts to a few special character sequences in special ways. For example, it reacts to the character sequence, `\%`, by moving to a new line in the Interactions Window (i.e., it interprets `\%` as a newline character). It also interprets `\n` as a newline character. In addition, whenever it encounters the character sequence, `\A`, in the input string, the `printf` function treats it as a place-holder for a piece of data to be displayed, as discussed in Example 10.1.1 below. Because the character sequences `\%`, `\n`, and `\A` are not interpreted literally, but involve the `printf` function escaping from a literal interpretation, they are frequently called escape sequences. (And the characters `\` and `\` that introduce escape sequences are sometimes called escape characters.) Although the `printf` function can deal with a variety of other escape sequences, these are the only ones that we’ll need for this course. Their use enables the `printf` function to generate nicely formatted text in the Interactions Window. For this reason, the input string is frequently called a `format string`—which explains the `f` in `printf`.

In summary, the `printf` function causes the contents of the format string (i.e., its first input) to be displayed verbatim in the Interactions Window, except that:
• the quotation marks are omitted;
• each instance of \" or \n is interpreted as a newline character and, thus, causes subsequent text to be displayed on the next line in the Interactions Window; and
• each instance of the escape sequence, \A, is replaced by a character sequence representing the value of the corresponding input expression.

Notice that if the format string contains $n$ instances of \A, then there must be $n$ input expressions following the format string, as follows:

\[
\text{(printf format-string expr\textsubscript{1} ... expr\textsubscript{n})}
\]

### Example 10.1.1: Formatted printing with printf

The following Interactions Window session illustrates the use of the escape sequences \%, \n and \A by the printf function.

```scheme
> (printf "Hi there!\nBye there!")
Hi there!
Bye there!
> (printf "Oh, I get it!\%This sentence begins on a new line!"")
Oh, I get it!
This sentence begins on a new line!
> (printf "First thing: \A, second thing: \A\% (+ 2 3) (+ 6 7))
First thing: 5, second thing: 42
> (printf "Line One!\% Line Two!!\% Line Three!!!\%"")
Line One!
   Line Two!!
      Line Three!!!
> (printf "First ===> \A, Second ===> \A, Third ===> \A\% (+ 4 2) (- 9 6.3) (* 4 100))
First ===> 6, Second ===> 2.7, Third ===> 400
> (printf "A symbol: \A, a string: \A, a boolean: \A\% "
   ‘I-am-a-symbol
   "I am a String!"
   (> 4 2))
A symbol: I-am-a-symbol, a string: I am a String!, a boolean: #t
```

The last of the above interactions illustrates a peculiarity of the printf function: when displaying instances of the string data type, it does not display the double-quotes.

### Example 10.1.2: The printf function and the void datum

The following interaction demonstrates that the printf function generates the void datum as its output:

```scheme
> (void? (printf "hi\n"))
hi
#t
```

In this example, the Default Rule for evaluating non-empty lists is used to evaluate the expression, (void? (printf "hi\n")) First, each element of the list is evaluated:

- the void? symbol evaluates to the built-in void? function; and
• (printf "hi\n") evaluates to the void datum—while causing hi to be displayed in the Interactions Window as a side effect.

Next, the void datum is fed as input into the void? type-checker predicate, resulting in the output value #t. Thus, hi is side-effect printing, while #t is the output value.

Although DrScheme does not normally display the void datum, we can force it to do so, as follows:

```scheme
> (printf "Show us void: ˜A˜% (void))
Show us void: #<void>
```

However, keep in mind that #<void> is not legal Scheme syntax. If you enter #<void> into the Interactions Window, you’ll get a red error message!

### Putting Multiple Expressions in the Body of a Lambda Function

Recall that the body of a lambda function may contain multiple expressions. When such a function is called, each of the expressions in the body is evaluated in turn. However, it is only the value of the last expression in the body that determines the output value for the function call. Since the output values of earlier expressions are ignored, it only makes sense to include multiple expressions in the body of a function if some of those expressions generate side effects. The following example considers a function whose body contains expressions that generate side-effect printing.

#### Example 10.1.3

The following lambda function, called verbose-func, contains multiple expressions in its body. When the verbose-func is called, each expression in its body is evaluated. The first four expressions cause the built-in printf function to be called, thereby generating several lines of side-effect printing in the Interactions Window. However, it is the evaluation of the last expression in the function’s body that generates an output value for the function call.

```scheme
> (define verbose-func
  (lambda (a b)
    (printf "Hi. This is verbose-func!˜%")
    (printf "The value of the first input is: ˜A˜%" a)
    (printf "The value of the second input is: ˜A˜%" b)
    (printf "Their product is:˜%")
    (* a b)))
> (verbose-func 3 4)
Hi. This is verbose-func!
The value of the first input is: 3
The value of the second input is: 4
Their product is:
12
>
```

In this case, the output value of the function call is twelve, which DrScheme displays in one color; the previous four lines of text are just side-effect printing, which DrScheme displays in a different color.

* Part 1 of this book explores how much can be accomplished without using side effects. Therefore, most of the functions we write will include only a single expression in the body. However, we will sometimes use the printf function to generate useful side-effect printing in the Interactions Window.
Example 10.1.4: Defining a useful tester function

The printf function can be used to define a tester function that will greatly facilitate the testing of whatever Scheme function we happen to be creating. The tester function can also be used to test our understanding of how arbitrary Scheme data get evaluated.

```
(define tester
  (lambda (datum)
    (printf "A ==> " datum)
    (eval datum)))
```

The tester function takes any Scheme datum as its input. As a side effect, it prints out a representation of that datum in the Interactions Window. For its output value, it simply evaluates the input datum. The following Interactions Window session demonstrates its use.

```
> (tester '(+ 1 2))
(+ 1 2) ==> 3
> (tester (+ 1 2))
3 ==> 3
> (tester '+)
+ ==> #<primitive:+>
> (tester +)
#<primitive:+> ==> #<primitive:+>
```

These examples demonstrate that the tester function is most useful when the quote special form is used to shield the desired input expression from evaluation. For example, notice the difference between the evaluations of (tester '(+ 1 2)) and (tester (+ 1 2)). In the first case, (+ 1 2) is shielded from evaluation by the quote special form; thus, the list (+ 1 2) is fed as input to the tester function. That is why (+ 1 2) is printed out in the Interactions Window before the arrow. After that side-effect printing, the eval function is then used to explicitly evaluate the list (+ 1 2), generating the output value 3. Since the formatting string given to printf does not include a newline character, the side-effect printing and the output value are both displayed on the same line.

* The tester function is one of the rare cases where the built-in eval function is explicitly invoked.

10.2 The Built-in load Function

Scheme includes a built-in load function that causes all of the Scheme expressions in a specified file to be evaluated in an Interactions Window session. Here’s the contract:

<table>
<thead>
<tr>
<th>Name:</th>
<th>load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>filename, a string</td>
</tr>
<tr>
<td>Output:</td>
<td>None</td>
</tr>
<tr>
<td>Side Effect:</td>
<td>Evaluates all of the Scheme expressions in the file named filename.</td>
</tr>
</tbody>
</table>

Example 10.2.1

Suppose the file "test.txt" contains the following expressions:

```
(define tester
  (lambda (datum)
```
Then the following Interactions Window session could ensue:

```
> x
BUG! reference to undefined identifier: x
> tester
BUG! reference to undefined identifier: tester
> (load "test.txt")
Loading test.txt!!
> x
34
> (tester 'x)
x ==> 34
```

Notice that the first attempts to evaluate tester and x generated errors because there were not yet any entries for these symbols in the Global Environment. However, after loading the file test.txt, subsequent attempts to evaluate x and to use tester succeed.

This example demonstrates that useful function definitions can be conveniently stored in a file, to be loaded whenever needed.

* The Run button on DrScheme’s toolbar is similar to the load function, except that it causes the Scheme expressions currently residing in the Definitions Window to be evaluated within a fresh Interactions Window session.

## 10.3 Comments

In Scheme, the semi-colon character is used to initiate comments. The text that constitutes a comment is ignored by DrScheme, as illustrated by the following example.

```
(define tester
  (lambda (datum)
    ;; Print (the value of) DATUM -- without a newline character
    (printf "A ==> " datum)
    ;; Then explicitly evaluate (the value of) DATUM
    (eval datum)))

;; Sample TESTER expressions
;; -----------------------------
(tester '(+ 2 3))
(tester (+ 2 3))
```

Evaluating the above code in the Interactions Window would have the same result as evaluating the following, uncommented code:
(define tester
  (lambda (datum)
    (printf "A ==> " datum)
    (eval datum)))
(tester '(+ 2 3))
(tester (+ 2 3))

The purpose of comments is to make a Scheme program easier for people to understand. DrScheme ignores the comments completely.

Contracts in Scheme programs. One of the most important uses of comments is to enable a Scheme program to include an explicit contract for each function it defines. The following example illustrates the format for contracts that will be used for the rest of the course.

---

**Example 10.3.2: A contract for the squaring function**

The following comment block constitutes a contract for the squaring function seen in Example 9.2.1.

;; SQUARE
;; ------------------------------------------
;; INPUT: X, a number
;; OUTPUT: The value X*X (i.e., X squared)

My personal convention is to use upper-case letters for the names of the function and its inputs, while the actual Scheme code uses lower-case letters.

* Aside from this difference, the names of the function and its inputs in the contract should match the corresponding names in the actual function definition.

By convention, if a function does not generate any side effects, then the contract need not mention side effects.

---

**Example 10.3.3: A contract for the tester function**

The following code fragment includes a contract for the tester function followed by the actual function definition. Note that a blank line should separate the contract from the function definition.

;; TESTER
;; ----------------------------------------------------------
;; INPUT: DATUM, any Scheme datum
;; OUTPUT: The result of evaluating (the value of) DATUM
;; SIDE EFFECT: Displays (the value of) DATUM *before*
;; evaluating it

(define tester
  (lambda (datum)
    ;; Display (the value of) DATUM
    (printf "A ==> " datum)
    ;; Evaluate (the value of) DATUM
    (eval datum)))
To avoid being overly cumbersome, contracts may intentionally blur the distinction between the names of input parameters—which are symbols—and their values—which can be anything.

Example 10.3.4: Revised contract for tester

Instead of (correctly) saying that the tester function displays (the value of) datum before evaluating (the value of) datum, a typical contract might say that the tester function displays datum before evaluating it. (Even though the symbol datum is not what is displayed by tester!) In effect, the contract is using the symbol datum to refer to its value in the local environment, much as a person uses the name Barack Obama to refer to the 44th president of the United States. Of course, you should never let the true distinction between a symbol and its value stray too far from conscious awareness!

;; TESTER
;; -------------------------------------------------------
;; INPUT:  DATUM, any Scheme datum
;; OUTPUT: The result of evaluating DATUM
;; SIDE EFFECT: Displays DATUM *before* evaluating it

(define tester
  (lambda (datum)
    ;; Display DATUM
    (printf "A ==> " datum)
    ;; Evaluate DATUM
    (eval datum)))

10.4 Summary

This chapter introduced the string data type, the built-in printf function, the built-in load function, and comments.

Almost any character sequence that begins and ends with double quotes denotes a string datum in Scheme. (The exceptions (e.g., "hi") involve escape sequences (e.g., ") that effectively capture the final double quote. They need not concern us.) For example, "the brown dog\n" and "i am a fox" both denote strings in Scheme.

The built-in printf function has the useful side effect of displaying text in the Interactions Window. The printf function takes a string—sometimes called a formatting string—as its first input. That string may include escape sequences such as \%, \n and A that are interpreted in special ways by the printf function. In particular, the printf function interprets each character of the formatting string literally, except that \% and \n are interpreted as newline characters, and A is interpreted as a placeholder for a piece of data. For each occurrence of A in the formatting string, there must be a corresponding additional input to printf. Thus, if the formatting string includes n occurrences of A, then there must be n additional inputs to printf after the formatting string, as illustrated below:

> (printf "One: \A, Two: \A, Three: \A\%" 1 2 (+ 1 2))
One: 1, Two: 2, Three: 3

Notice that the double quotes from the formatting string are not displayed in the Interactions Window.

The tester function was defined to use printf to display a datum before evaluation, and then to explicitly use the built-in eval function to evaluate that datum. When using the tester function, input expressions are typically quoted to shield them from evaluation by the Default Rule, as illustrated below:

> (tester '(+ 1 2))
(+ 1 2) ==> 3
The built-in `load` function can be used to load the contents of a file automatically, instead of having to manually type its contents directly into the Interactions Window. The input to the `load` function is a string representing the name of the file. For example, if `myfile.txt` contains a bunch of function definitions, then the expression `(load "myfile.txt")` would cause those function definitions to be evaluated by DrScheme just as though they had been manually typed into the Interactions Window. Those functions could then be used during the remainder of the Interactions Window session. DrScheme’s `Run` button is similar, except that it loads the expressions currently residing in the Definitions Window into the Interactions Window.

Finally, the semi-colon is a character that is used to introduce comments in Scheme. In particular, any sequence of characters that starts with a semi-colon and continues to the end of the line is completely ignored by DrScheme. An effective programmer uses concise comments to explain what their code is (supposed to be) doing. One important use of comments is to provide a contract for each function that is defined in a given program.

**Built-in Functions Introduced in this Chapter**

- `printf` To do side-effect printing in the Interactions Window
- `load` To load the contents of a file
Chapter 11

Conditional Expressions I

This chapter introduces conditional expressions. A conditional expression is a compound expression whose evaluation depends on the evaluation of one or more subsidiary expressions, called conditions. A condition is any expression that evaluates to #t or #f (e.g., “It is raining” or “x > y”). More generally, a condition can be any evaluable expression, with the understanding that any value other than #f will be interpreted as boolean true; only #f counts as boolean false.

Scheme provides several special forms to facilitate the writing of conditional expressions. This chapter focuses on the if special form. From the programmer’s perspective, a single if special form can be used to, in effect, make a binary decision, as in: “If x > y, evaluate x² - y²; otherwise, evaluate y² - x².” In addition, multiple if special forms can be strung together to, in effect, make an n-ary decision (i.e., a decision to select one from among n choices), as in: “If the grade is at least 90, give an A; otherwise, if the grade is at least 80, give a B; otherwise, if the grade is at least 70, ....”

* The evaluation of the if special form is lazy in the sense that only the computations needed to ascertain the final value are actually performed.

This chapter also introduces the when special form, which is useful in cases where an else expression is not needed.

11.1 The if Special Form

We begin by introducing the if special form under the assumption that its condition evaluates to an actual boolean value (i.e., #t or #f). Afterward, we will relax that assumption.

The syntax of an if special form is as follows:

(if condExpr thenExpr elseExpr)

where:

- condExpr is a condition (i.e., an expression that evaluates to #t or #f); and
- thenExpr and elseExpr are any Scheme expressions.

Example 11.1.1

The following expressions are examples of the if special form:

(if (> 2 4) (* 8 2) (* 6 5))

(if (> 4 2) ’then ’else)
The semantics of the if special form stipulates that it is evaluated as follows.

- First, the condition, condExpr, is evaluated.
- If condExpr evaluates to #t, then thenExpr is evaluated—and the value of the if special form is whatever thenExpr evaluates to.
- On the other hand (i.e., if condExpr evaluates to #f), then elseExpr is evaluated—and the value of the if special form is whatever elseExpr evaluates to.

Notice that the condition, condExpr, is always evaluated; however, after that, one and only one of the remaining expressions, thenExpr or elseExpr, is evaluated. We say that the evaluation of the if special form is lazy, in the sense that it only evaluates the expressions needed to compute the value of the entire if expression. For example, if the condition evaluates to #t, then the value of the else expression is not needed and, thus, it is not computed. This kind of selective evaluation is available to special forms because each special form specifies its own mode of evaluation; however, this kind of selective evaluation is not available to the Default Rule, which always evaluates every item in a non-empty list.

Example 11.1.2

The following interactions demonstrate the evaluation of the if special forms seen earlier.

```scheme
> (if (> 2 4) (* 8 2) (* 6 5))
30
> (if (> 4 2) 'then 'else)
then
> (if #f "then" "else")
"else"
```

In the first expression, the condition, (> 2 4), evaluates to #f. Thus, the else expression, (* 6 5), is evaluated. Its value, 30, is the value of the entire if expression.

In the second expression, the condition, (> 4 2), evaluates to #t. Thus, the then expression, 'then, is evaluated. Its value, then, is the value of the entire if expression.

In the third expression, the condition, #f, evaluates to #f. Thus, the else expression, "else", is evaluated. Its value, "else", is the value of the entire if expression. (Recall that strings evaluate to themselves.)

Although the preceding examples illustrate the semantics of the if special form, they are kind of silly because in each case the condition has a determined value and, therefore, the entire if expression seems unnecessary. That is true. However, as the following example demonstrates, an if expression that appears in the body of a function can involve conditions that depend on the values of one or more input parameters—and those values are not known until the function is applied to inputs.

Example 11.1.3: Using an if expression in the body of a function

Below, a function, how-big, is defined. If given a number less than 10, its output is the symbol, small; otherwise, its output is the symbol, big.

```scheme
;; HOW-BIG
;; ------------------------------------
;; INPUT: NUM, a number
;; OUTPUT: The symbol SMALL, if NUM is less than 10;
```
Otherwise, the symbol BIG.

(define how-big
  (lambda (num)
    (if (< num 10)
        'small
        'big))))

The following interactions demonstrate its behavior:

> (how-big 5)
small
> (how-big 102)
big

Notice that the result of evaluating the condition, (< num 10), depends on the value of num in the local environment, which is not known at the time the function is specified by the programmer; instead, the value of num is known only when the function how-big is eventually applied to some input.

* The values of the input parameters for a function cannot be known when the programmer is writing the body of the function. Therefore, if the programmer wants the function to do different things for different inputs, the if special form can be quite useful.

---

In-Class Problem 11.1.1

Define a function, called sign, that satisfies the following contract.

;;; SIGN
;;; ---------------------------------------------
;;; INPUT: X, a number
;;; OUTPUT: 1, if X > 0; 0, if X = 0; -1, if X < 0

Here are some examples of the desired behavior:

> (sign 3)
1
> (sign 0)
0
> (sign -4.2)
-1

Hint: Start by defining a function that outputs 1 if x > 0, and 0 in all other cases.

---

The non-strict version of the if special form. In the strict version of the if special form, the condition must be an expression that evaluates to a boolean (i.e., either #t or #f). In the non-strict version, the condition can be any Scheme expression, as illustrated below.
**Example 11.1.4**

The following are legal instances of the `if` special form:

```
(if 72 "yup" "nope")
(if "condie" "yup" "nope")
(if (* 3 4) 'hello 'goodbye)
```

The semantics of the non-strict version of the `if` special form is governed by the following rule:

* When interpreting the value of the condition, anything other than `#f` counts as boolean `true` (i.e., `#f` is the only Scheme datum that counts as boolean `false`).

**Example 11.1.5**

The following Interactions Window session demonstrates the evaluation of the non-strict `if` expressions seen earlier.

```
> (if 72 "yup" "nope")
"yup"
> (if "condie" "yup" "nope")
"yup"
> (if (* 3 4) 'hello 'goodbye)
hello
```

*In each case, the condition being tested evaluates to a non-boolean value. Since `#f` is the only thing that counts as boolean false, the conditions in these examples all count as boolean true. Thus, in each case, the then expression is evaluated—and the value of the then expression is the value of the entire `if` expression.*

### 11.2 Simplifying Conditional Expressions

Conditional expressions can be used in many ways to enable Scheme functions to make finely tuned decisions amongst any number of cases. Although conditional expressions stated in English can guide your programming efforts, they can sometimes lead to solutions that are more complex than they need to be. That’s okay! Once your function is working, you can focus attention on how to simplify the expressions it uses. In addition, as you gain more practice, the simpler expressions may come to mind sooner in the programming process.

At first, we restrict attention to expressions that evaluate to boolean values—that is, either `#t` or `#f`. Afterward, we consider expressions that may evaluate to any type of Scheme data, but subject to the interpretation that anything other than `#f` counts as `true`, while only `#f` counts as `false`.

**Definition 11.1: Equivalent boolean conditions**

Suppose that `boolOne` and `boolTwo` are two boolean conditions (i.e., expressions that evaluate to booleans no matter what environment they are evaluated in). The expressions, `boolOne` and `boolTwo` are called equivalent if, whenever they are evaluated with respect to the same environment, the resulting boolean values are the same. In other words, `boolOne` evaluates to `#t` if and only if `boolTwo` evaluates to `#t`. 
Example 11.2.1: Simplifying if expressions involving boolean conditions

According to the above definition, the expression, \((\text{if} \ (\gt \ x \ y) \ \#t \ \#f)\), is equivalent to the simpler expression, \((\gt \ x \ y)\). The following interactions demonstrate the equivalence in two different environments: one where \(x > y\), and one where \(x < y\).

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (define x 32)</td>
<td>Setting up an environment where (x &gt; y)</td>
</tr>
<tr>
<td>&gt; (define y 4)</td>
<td></td>
</tr>
<tr>
<td>&gt; (if (&gt; x y) #t #f)</td>
<td>#t</td>
</tr>
<tr>
<td>&gt; (&gt; x y)</td>
<td>#t</td>
</tr>
<tr>
<td>&gt; (define x 32)</td>
<td>Setting up an environment where (x &lt; y)</td>
</tr>
<tr>
<td>&gt; (define y 1000)</td>
<td></td>
</tr>
<tr>
<td>&gt; (if (&gt; x y) #t #f)</td>
<td>#f</td>
</tr>
<tr>
<td>&gt; (&gt; x y)</td>
<td>#f</td>
</tr>
</tbody>
</table>

More generally, if \(\text{boolCond}\) is any boolean condition, then the following simplification yields an equivalent expression:

\[
(\text{if} \ \text{boolCond} \ \#t \ \#f) \sim \text{boolCond}
\]

So, if you ever find yourself writing an if expression whose then and else clauses are \#t and \#f, respectively, consider making the above simplification.

Next, we consider the same simplification, but applied to conditions whose evaluations do not necessarily yield boolean values. In such cases, the simplification yields equivalent expressions—as long as we consider anything other than \#f to count as true, and \#f to be the only thing that counts as false.

Example 11.2.2: Simplifying if expressions: non-strict truth values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (if ’happy #t #f)</td>
<td>’happy ⇒ happy, which counts as true</td>
</tr>
<tr>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>&gt; ’happy</td>
<td>happy</td>
</tr>
<tr>
<td>happy</td>
<td></td>
</tr>
<tr>
<td>&gt; (if #f #t #f)</td>
<td>The condition #f ⇒ #f, which counts as false</td>
</tr>
<tr>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>&gt; #f</td>
<td>#f</td>
</tr>
</tbody>
</table>

In the first example, the condition ’happy evaluates to the symbol happy, which counts as true. Therefore, the if expression evaluates to #t, which is the result of evaluating its then expression. In the second example, the expression ’happy evaluates to the symbol happy, which counts as true. Thus, the expressions, (if ’happy #t #f) and ’happy, are equivalent in the sense that they both evaluate to things that count as true. The third and fourth lines demonstrate the case where the condition of an if expression evaluates to #f.
In-Class Problem 11.2.1

Define a function, called convert-to-boolean, that takes any Scheme datum as its input. It should return #t as its output if the input is anything that counts as true; otherwise, it should return #f, as illustrated below.

> (convert-to-boolean #t)
#t
> (convert-to-boolean (+ 3 2))
#t
> (convert-to-boolean #f)
#f

*Hint: Use an unsimplified conditional expression!*

11.3 The when Special Form

In certain programming circumstances, you may want an if special form that does not need an else case. The when special form is provided to handle such circumstances. In its simplest form, the when special form has the following syntax:

(when condExpr thenExpr)

Such an expression is evaluated as follows. First, the condition, condExpr, is evaluated. If it evaluates to #t (or something that counts as true), then thenExpr is evaluated, and its value is the value for the entire when expression. However, if condExpr evaluates to #f, then thenExpr is skipped, and the value of the entire when expression is void.

Example 11.3.1

The following interactions demonstrate the semantics of the simplest use of the when special form.

> (when #t 3)
3
> (when (> 3 2) (* 4 5))
20
> (when (> 2 3) (* 4 5))
> (void? (when #f 3))
#t

Like the if special form, the when special form is most useful when used within the body of a function.

Example 11.3.2

Consider the following version of the how-big function that takes an extra input, verbose?. When verbose? is true, the function prints out some information about the inputs; otherwise, it doesn’t print out anything.

;;; HOW-BIG-V2
;;; ------------------------------------------------------
;;; INPUTS: NUM, a number
;;; VERBOSE?, a boolean
;;; OUTPUT: A symbol, either SMALL or BIG, depending on
;;; whether NUM < 10
;;; SIDE EFFECT: When VERBOSE? is true, it prints out
;;; information about the inputs.

(define how-big-v2
  (lambda (num verbose?)
    ;; Do some side-effect printing?
    (when verbose?
      (printf "Inside HOW-BIG-V2 with NUM = \"A\" and VERBOSE? = \"A\"\%
               num verbose?))
    ;; Output value
    (if (< num 10)
      'small
      'big)))

Here are some examples of its behavior.

> (how-big-v2 3 #t)
Inside HOW-BIG-V2 with NUM = 3 and VERBOSE? = #t
small
> (how-big-v2 3 #f)
small
> (how-big-v2 15 #t)
Inside HOW-BIG-V2 with NUM = 15 and VERBOSE? = #t
big
> (how-big-v2 15 #f)
big

* Because the when special form can evaluate to void (e.g., when its condition evaluates to #f), when should not be used to generate output values. Instead, like in the preceding example, when should only be used to generate helpful side effects (e.g., side-effect printing).

Later on, in Part II of the book, when we discuss destructive programming, we will see additional uses of the when special form.

More general version of the when special form. Because the when special form never includes an else expression, it can include multiple then expressions in its body. In this way, the body of a when expression is similar to the body of a lambda function. In general, the when special form has the following syntax:

(when condExpr
  expr1
  expr2
  ...
  exprn)

The semantics of the when special form stipulates that it is evaluated as follows. First, the expression, condExpr, is evaluated. If it evaluates to something that counts as true, then the expressions, expr1, ..., exprn, are evaluated in turn, and the value of the last expression serves as the value of the entire when expression. However, if condExpr evaluates to #f, then the subsidiary expressions, expr1, ..., exprn, are skipped, and the entire when expression simply evaluates to void.


### In-Class Problem 11.3.1

Modify the `how-big-v2` function so that it includes a `when` expression that has multiple `printf` expressions in its body.

---

#### 11.4 Summary

This chapter introduced the `if` special form for making binary decisions. When evaluating conditions, the `if` special form accommodates *non-strict* truth values. In particular, anything other than `#f` counts as true. Equivalently, only `#f` counts as false.

An `if` special form has the form, `(if condExpr thenExpr elseExpr)`. An `if` special form is evaluated as follows. First, the condition, `condExpr`, is evaluated. If it evaluates to `#t` (or something that counts as true), then the `then` expression, `thenExpr`, is evaluated, and its value is taken to be the value of the entire `if` expression. However, if the condition evaluates to `#f`, then the `else` expression, `elseExpr`, is evaluated, and its value is taken to be the value of the entire `if` expression. Thus, either the `then` expression or the `else` expression is evaluated, but never both.

An `if` expression whose `then` expression is `#t`, and whose `else` expression is `#f` can be simplified. For example, `(if (> x y) #t #f)` is equivalent to `(> x y)`.

A `when` expression is useful in cases where no `else` expression is needed. For example, a `when` expression can be used to generate side-effect printing in certain cases, but not others. Because a `when` expression can evaluate to `void`, it should *not* be used to generate an output value!

---

#### Special Forms Introduced in this Chapter

- **if**  For making binary decisions
- **when**  For cases where an `else` expression is not needed
Chapter 12

Recursion I

This chapter introduces recursive functions. Defining recursive functions in Scheme requires no new computational constructs (e.g., no new special forms or built-in functions) beyond those seen in the preceding chapters; instead, we combine existing constructs in a new way. In many cases, recursive functions can provide compact and elegant solutions to interesting computational problems.

We begin by recalling that the evaluation of a non-empty list according to the Default Rule typically involves the application of a function to zero or more inputs. For convenience, we make the following definition.

**Definition 12.1: Function-call expression**

Suppose that \( expr \) is a Scheme expression that denotes a non-empty list, \( L \), whose evaluation is governed by the Default Rule. Then we say that \( expr \) is a function-call expression. Furthermore, suppose that \( f \) is the function that results from evaluating the first element of the list \( L \). Then we say that \( expr \) calls \( f \).

Thus, for example, the expression, \((+ 2 3)\), is a function-call expression that calls the built-in *addition* function. Similarly, \((\text{symbol? } 'x)\) is a function-call expression that calls the built-in *symbol?* function. In contrast, the expressions, \(\text{(define myVar 3)}\) and \(\text{(lambda (x) (* x x))}\), denote special forms and, thus, are not function-call expressions.

**Definition 12.2: Recursive function**

A function, \( f \), is said to be recursive if its body contains a function-call expression that calls \( f \).

At first glance, this might seem like a crazy idea—after all, a function calling itself sounds like the kind of circularity that might lead to infinite loops. However, this dreaded form of circularity is generally quite easy to avoid, as follows.

* A recursive function typically includes a conditional expression that tests some stopping condition (or base case). If the stopping condition evaluates to (something that counts as) *true*, then no recursive function call is made. Not only that, in cases where the recursive function call *is* made, it typically involves applying the function to different inputs.

Thus, as will be amply demonstrated, a typical sequence of recursive function calls is less like a circle that forever loops back on itself, and more like a spiral that converges to some stopping point.

### 12.1 Defining Recursive Functions in Scheme

In Scheme, the typical characteristics of the definition of a recursive function, \( f \), are:
• a `define` special form that effectively gives a name to $f$;
• a conditional expression (in the body) that distinguishes the base case from the recursive case; and
• a function-call expression (in the body) that typically involves applying $f$ to other inputs.

---

**Example 12.1.1: The factorial function**

The factorial function, $f(n) = n!$, is sometimes casually defined as follows:

$$ f(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 $$

This definition is casual because the dot-dot-dot is not precisely defined. We can give a more precise, recursive definition of the factorial function, as follows:

**Base Case** ($n = 1$): \[ 1! = 1 \quad (i.e., f(1) = 1) \]

**Recursive Case** ($n > 1$): \[ n! = n \cdot (n-1)! \quad (i.e., f(n) = n \cdot f(n-1)) \]

According to this definition, the following equalities hold:

- $4! = 4 \cdot 3!$
- $3! = 3 \cdot 2!$
- $2! = 2 \cdot 1!$
- $1! = 1$

Putting all of this information together yields:

$$ 4! = 4 \cdot 3! = 4 \cdot (3 \cdot 2!) = 4 \cdot (3 \cdot (2 \cdot 1!)) = 4 \cdot (3 \cdot (2 \cdot 1)) = 24. $$

---

**Example 12.1.2: The factorial function in Scheme**

The following Scheme expression defines a recursive function, `facty-v1`, whose definition is based on the above insights. (The function is called, `facty-v1`, because it is the first version of the factorial function we will look at.)

```scheme
;;; FACTY-V1
;;; -------------------------------------------------------
;;; INPUT: N, a positive integer
;;; OUTPUT: The factorial of N (i.e., N*(N-1)*...*3*2*1)

(define facty-v1
  (lambda (n)
    (if (= n 1)
        ;; Base Case: N = 1
        1
        ;; Recursive Case: N > 1
        (* n (facty-v1 (- n 1))))))
```

Notice that the `define` special form effectively gives the name, `facty-v1`, to the function defined by the `lambda` special form. Notice, too, that the body of this function includes a conditional expression that distinguishes the base case (i.e., when $n = 1$) from the recursive case (i.e., when $n > 1$). Finally, notice that the body includes a function-call expression that calls `facty-v1`. (We’ll have more to say about this!)
Okay, so what happens when the above expression is evaluated? Well, the expression is a define special form. So, the symbol, facty-v1, is not evaluated. Only the third element of the define special form (i.e., the lambda expression) is evaluated. Like any lambda expression, the one above evaluates to a function. However:

* It is important to remember that evaluating the above lambda expression only creates a function. It does not call the function. Thus, the expressions in the body of the lambda expression are not evaluated—yet!

The reason this is important is that when the lambda expression is evaluated, the Global Environment does not yet associate any value with the symbol, facty-v1. Recalling Section 7.1, the order of events in the evaluation of this define special form is:

1. an entry for facty-v1 in the Global Environment is created with a temporary value: void;
2. the lambda expression is evaluated, which yields a function; and
3. that function is entered into the Global Environment as the value associated with facty-v1.

Thus, during Step 2, any attempt to evaluate an expression of the form (facty-v1 ... ) would cause an error because facty-v1 would evaluate to void. However, after the lambda expression has been evaluated (to a function), and that function has been inserted as the value for facty-v1 in the Global Environment, then expressions such as (facty-v1 3) can be successfully evaluated, as shown below.

Next, let’s observe that facty-v1 appears to correctly compute the factorial of its input:

```
> (facty-v1 1)
1
> (facty-v1 2)
2
> (facty-v1 3)
6
> (facty-v1 4)
24
```

**Evaluating** (facty-v1 3). Consider DrScheme’s evaluation of the expression, (facty-v1 3). This is a function-call expression whose evaluation is governed by the Default Rule. Thus, the symbol facty-v1 and the number 3 must both be evaluated. The symbol facty-v1 evaluates to the function we just defined; and 3 evaluates to itself. Next, the facty-v1 function is applied to the input 3.

The application of the facty-v1 function to the input 3 is depicted at the top of Fig. 12.1. First, a local environment is created with an entry associating the input parameter n with the value 3. Next, the expression in the body of the facty-v1 function, shown below, is evaluated with respect to that local environment.¹

```
(if (= n 1)
  ;; Base Case: N = 1
  1
  ;; Recursive Case: N > 1
  (* n (facty-v1 (- n 1)))
```

Since the value of n is 3 in the local environment, the condition (= n 1) evaluates to #f. Thus, the then expression, 1, is skipped, and the else expression, (* n (facty-v1 (- n 1))), is evaluated—according to the Default Rule. The * symbol evaluates to the multiplication function, n evaluates to 3, and (facty-v1 (- n 1)) evaluates to ... Gosh, we need a new paragraph!
The subsidiary expression, \( (\text{facty-v1} (- n 1)) \), is evaluated according to the Default Rule. First, the \( \text{facty-v1} \) symbol evaluates to the \( \text{facty-v1} \) function; and \( (- n 1) \) evaluates to 2 (since \( n \) has the value 3). Next, the \( \text{facty-v1} \) function must be applied to the input value 2, as depicted in the second box in Fig. 12.1.

* Notice that the evaluation of the expression, \( (* (\text{facty-v1} (- n 1))) \), in the top function-call box cannot continue until the subsidiary expression, \( (\text{facty-v1} (- n 1)) \), has been evaluated. However, this value cannot be known until the output value for the second function-call box has been generated! In other words, the evaluation of the expression in the top box must be suspended, pending the outcome of the second box.

The application of the \( \text{facty-v1} \) function to the value 2, depicted in the second function-call box in the figure, is similar to the application of the \( \text{facty-v1} \) function to 3 in the top box, except that the local environment in the second box associates the input parameter, \( n \), with the value 2.

* Crucially, the local environments in separate function-call boxes do not cause a conflict! They can’t see one another. Neither knows that the other even exists! Thus, although the two input parameters are both called \( n \), they are quite distinct!

Thus, the evaluation of the body of the function in the second box proceeds in the environment where \( n \) has the value 2. Thus, the base case is skipped and the expression, \( (* n (\text{facty-v1} (- n 1))) \), is evaluated. This leads to yet another recursive function call—this time the application of the \( \text{facty-v1} \) function to the input value 1, as illustrated in the third box in Fig. 12.1.

* As in the preceding case, the evaluation of the expression, \( (* n (\text{facty-v1} (- n 1))) \), in the second box cannot continue until the output value for the third box has been generated. In other words, the evalution of the expression in the second box must be suspended, pending the outcome of the third box.

The application of the \( \text{facty-v1} \) function to the value 1 begins by creating a local environment entry that associates the input parameter \( n \) with the value 1. (Again, this is a new input parameter, distinct from the other \( n \)s.) Next, the if expression in the body of the function is evaluated. This time, however, the condition \( (= n 1) \) evaluates to \#t; thus, the base case expression, 1, is evaluated, yielding the output value, 1, for the application of the \( \text{facty-v1} \) function to the value 1 (i.e., the output value for the third box).

This output value, 1, is the value of the expression, \( (\text{facty-v1} (- n 1)) \), that was being evaluated in the middle function-call box (where \( n \) has the value 2). Now that the value of \( (\text{facty-v1} (- n 1)) \) is in hand, the evaluation of the expression, \( (* n (\text{facty-v1} (- n 1))) \), in the middle box can continue. To wit, the multiplication function is applied to 2 and 1, yielding the output value 2 for the middle function-call box.

This output value, 2, is the value of the expression, \( (\text{facty-v1} (- n 1)) \), that was being evaluated in the top function-call box (where \( n \) has the value 3). Now that the value of \( (\text{facty-v1} (- n 1)) \) is in hand, the evaluation of the expression, \( (* n (\text{facty-v1} (- n 1))) \), in the top box can continue. To wit, the multiplication function is applied to 3 and 2, yielding the output value 6 for the top function-call box.

Phew!

\*To decrease clutter, only a portion of the body is shown in each function-call box in the figure.

The above example illustrates many of the features that are frequently found in recursive functions.

- The body of the function contains a conditional expression that enables a stopping condition—commonly called a base case—to be recognized. If that stopping condition evaluates to \#t (or any non-strict true), then no more recursive function calls are made.
Figure 12.1: DrScheme's evaluation of (facty-v1 3)
• The body of the function contains an expression that involves a recursive call to that same function—but with different input(s). It is crucial that the inputs to the recursive function call be different in some way; otherwise, that recursive function call would lead to another identical recursive function call, and so on, *ad infinitum*. Because the inputs to the recursive function call are different in some way, the recursive function call is not circular; instead, the sequence of recursive function calls is more like a spiral that eventually stops when the base case is arrived at.

• Although the same function is being applied each time a recursive function call is made, it is being applied to different inputs. As a result, the body of the function is being evaluated with respect to a different local environment. In this way, the evaluation of the same function body is affected by the value of the input parameter(s). This helps to avoid circularity and infinite loops.

**In-Class Problem 12.1.1**

Define a version of the `facty-v1` function, called `facty-verbose`, that includes an extra input, `verbose?`. Whenever `verbose?` is true, the function should do some side-effect printing before generating its output value, as illustrated below:

```plaintext
> (facty-verbose 3 #f)
6
> (facty-verbose 3 #t)
Inside facty-verbose with N = 3
Inside facty-verbose with N = 2
Inside facty-verbose with N = 1
6
```

*Hint:* Use the `when` special form to decide whether to do any side-effect printing, as was done in Example 11.3.2.

**In-Class Problem 12.1.2**

Define a function, called `sum-to-n`, that satisfies the following contract:

```plaintext
;; SUM-TO-N
;; -----------------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: The sum of all the integers from 0 to N
;; Example: (sum-to-n 4) = 4 + 3 + 2 + 1 + 0 = 10
```

### 12.2 Summary

This chapter introduced recursive functions, using the `factorial` function as a running example. The body of a recursive function typically includes a conditional expression that distinguishes a stopping case (the *base case*) from the recursive case. The recursive case involves a recursive function call. For example, when applying the `factorial` function to some number `n`, the recursive function call entails calling that same `factorial` function on `n - 1`. The inputs to successive recursive function calls must eventually hit the base case; otherwise, the recursion will go on forever.
Chapter 13

Conditional Expressions II

The if special form that was introduced in Chapter 11 is quite convenient for making binary decisions. And, for many recursive functions, binary decisions are sufficient. However, using if to make \( n \)-ary decisions (i.e., decisions among \( n \) choices) requires stringing together multiple if expressions, which can get quite cumbersome. Because programmers often want to make \( n \)-ary decisions, Scheme provides the cond special form, which has a simpler syntax for conditional expressions associated with \( n \)-ary decisions.

On another front, the conditions in a conditional expression can be simple or complicated. For example, compare “\( x > y \)” and “\((x > y) \) or \((x^2 < y^3)\) and \((x + y < 10)\)”.

13.1 The cond Special Form

Often times, it is useful to nest one conditional expression inside another. For example, the else expression for an if expression might itself be another if expression. Although useful, the nesting of if expressions can get quite complicated. Thus, Scheme provides the cond special form as a convenient short-cut.

---

Example 13.1.1: Nested if expressions

Consider the following letter-grade function, implemented using a sequence of nested if special forms to distinguish four cases.

```scheme
;;; LETTER-GRADE
;;; ----------------------------------------------------------
;;; INPUT: NUM, a number between 0 and 100
;;; OUTPUT: One of the symbols, A, B, C or D, corresponding
;;; to the standard 90/80/70 cutoffs for letter grades.

(define letter-grade
  (lambda (num)
    (if (>= num 90)
        'A
        (if (>= num 80)
            'B
            (if (>= num 70)
                'C
                'D)))))
```

1 As will be discussed below, in Scheme, and and or are provided as special forms, whereas not is provided as a built-in function. The generic term operator is used here to include and, or and not, regardless of how they are implemented in Scheme.

81
The following interactions illustrate its behavior:

```
> (letter-grade 86)
B
> (letter-grade 95)
A
> (letter-grade 43)
D
```

The body of this function consists of a single if expression. The reason it looks so complicated is that the
else expression for that if expression is another if expression. (Here’s where the automatic indenting of
DrScheme really helps.) That if expression is itself quite complicated because its else expression is yet
another if expression. Consider the evaluation of the expression, (letter-grade 86). The input to the function is 86;
thus, the input parameter, num, has the value 86. Since the body of the function consists of a single if expression,
that if expression must be evaluated. Since num has the value 86, the condition, (>= num 90), evaluates to #f. Thus, DrScheme
skips the then expression and, instead, evaluates the else expression.

The else expression is another if expression. So, DrScheme evaluates its condition, (>= num 80). Since num has
the value 86, the condition evaluates to #t. Thus, DrScheme evaluates the then expression, ’B. Since ’B evaluates to B, the
output value for the inner if expression is B. Since the inner if expression is the else expression for the outer if expression,
its value, B, also serves as the value of the outer if expression. Furthermore, since the outer if expression is the only
expression in the body of the function, its value, B, also serves as the output value for the original expression, (letter-grade 86),
as shown in the Interactions Window.

Example 13.1.2: Using cond instead of nested if

Below, an equivalent function, called letter-grade-v2, is defined that uses a cond expression instead of the nested if
expressions seen above. This cond expression serves the same purpose as the nested if expressions.

```
;;  LETTER GRADE V2
;;  -----------------------------------------------
;;  INPUT:  NUM, a number between 0 and 100
;;  OUTPUT: One of the symbols, A, B, C or D, corresponding
;;          to the standard 90/80/70 cutoffs for letter grades.

(define letter-grade-v2
  (lambda (num)
    (cond
      ;; Case 1: Got an A
      ((>= num 90)
        'A)
      ;; Case 2: Got a B
      ((>= num 80)
        'B)
      ;; Case 3: Got a C
      ((>= num 70)
        'C)
    )))
```

Notice that, as a matter of syntax, each case of the cond expression is represented by a subsidiary list and, since the first element of that subsidiary list is (almost always) a list, the beginning of each case of a cond (except the last one) is typically signalled by two left parentheses. For example, the first case is represented by the list ((>= num 90) 'A). The first element of that list, (>= num 90), is the condition for that case; the second element, 'A, is the answer expression for that case. By convention, to make the code readable, the answer expression should always be placed on the line following its condition, even if both expressions are short.

That the above function is equivalent to letter-grade is demonstrated below:

```
> (letter-grade-v2 93)
A
> (letter-grade-v2 82)
B
> (letter-grade-v2 74)
C
> (letter-grade-v2 61)
D
```

Consider the evaluation of the expression, (letter-grade-v2 74). In this case, the input parameter num has the value 74. The above cond expression is evaluated as follows. First, the first condition, (>= num 90), is evaluated. Since num is 74, this condition evaluates to #f; thus, the first case is skipped. Next, the second condition, (>= num 80), is evaluated. Since num is 74, this condition also evaluates to #f and, thus, the second case is skipped. Next, the third condition, (>= num 70), is evaluated. Since num is 74, this condition evaluates to #t. As a result, the answer expression for this case (i.e., 'C) is evaluated. Furthermore, the value of this expression (i.e., C) is taken to be the value of the entire cond expression. Since the third condition evaluated to #t, the fourth case was ignored.

For the expression, (letter-grade-v2 61), the first three conditions all evaluate to #f. However, the fourth condition, #t, evaluates to #t. Thus, the value of the entire cond expression is D in this case, because 'D evaluates to D.

* In most programming circumstances, the last condition in a cond expression should be #t. This ensures that at least one of the conditions in the cond will evaluate to #t. As an alternative, the last condition in a cond can be the else keyword symbol, which serves the same purpose as #t.

* If all of the conditions in a cond evaluate to #f, then the entire cond expression will evaluate to void, something that is typically not desirable. Indeed, a cond that expression that evaluates to void typically signals that the programmer forgot about a possible case.

The cond special form, more generally. More generally, the syntax of a cond special form looks like this:

```
(cond
  (cond1
    expr1)
  (cond2
    expr2)
  ...
  (condn
    exprn)
)
```
where:
- each \( \text{cond}_i \) is a (strict or non-strict) condition;
- (in most circumstances) the last condition, \( \text{cond}_n \), is either \#t or else; and
- each \( \text{expr}_i \), called an answer expression, can be any Scheme expression.

The value of a \text{cond} expression is determined as follows:
- Each condition, \( \text{cond}_i \), is evaluated in turn until one is found that evaluates to \#t—or something that counts as true.
- The value of the \text{cond} expression is the value of the corresponding answer expression, \( \text{expr}_i \).
- If every condition evaluates to \#f, then the entire \text{cond} expression evaluates to \text{void}.

Like the \text{if}, \text{and} and \text{or} special forms, the evaluation of the \text{cond} special form is lazy. In other words, DrScheme evaluates only those subsidiary expressions that are needed to determine the final value of the \text{cond} special form. In particular, if the condition, \( \text{cond}_i \), evaluates to true, then no subsequent conditions will be evaluated. In addition, only one expression, \( \text{expr}_i \), is evaluated; all others are ignored.

**Example 13.1.3**

*If all of the conditions of a \text{cond} expression evaluate to \#f, then the \text{cond} expression itself will evaluate to \text{void}, as illustrated below.*

```
> (cond)           ← No cases in this \text{cond}
> (cond
  (\text{>} 2 3)
  'hi)
  (\text{=} 3 5)
  'bye))
> (\text{void?} (cond))  ← The \text{void?} predicate confirms that (cond) evaluates to \text{void}
#t
> (\text{void?} (cond
  (\text{>} 2 3)
  'hi)
  (\text{=} 3 5)
  'bye)))
#t
```

*In many programming circumstances, when a \text{cond} expression evaluates to \text{void}, it means that the programmer forgot about the existence of one or more cases. That is why it is strongly recommended that the last case of a \text{cond} use \text{else} or \#t as its condition, thereby enabling it to serve as a “catch-all” (or default) case, ensuring that no cases are missed. (Exceptions will be discussed below.)*

**The \text{cond} special form, even more generally!** Recall that the body of a \text{lambda} expression can include multiple subsidiary expressions. The semantics of Scheme stipulates that the expressions in the body are evaluated sequentially, and that the value of the last expression serves as the output value for the function. Recall, too, that the expressions before the last one would be meaningless unless they have side effects (e.g., printing information to the Interactions Window).

In a \text{cond} expression, each condition, \( \text{cond}_i \) can be followed by multiple subsidiary expressions. Typically, having multiple expressions for a single condition only makes sense if the expressions before the last one have side effects. As with the body of a \text{lambda} expression, it is the value of the last subsidiary expression in the selected case that serves as the value of the entire \text{cond} expression.
Example 13.1.4

Below, a function, cond-effects, is defined whose body contains a cond special form in which each condition has multiple subsidiary expressions associated with it. Notice how comments are used to make the code easier on the eyes.

```
(define cond-effects
  (lambda (num)
    (cond
      ;; Case 1: Got an A
      ((>= num 90)
       (printf "Oh my gosh! You did great!!!"%"
       'A)
      ;; Case 2: Got a B
      ((>= num 80)
       (printf "Well, you know, a B is pretty good!!"%"
       (printf "Nothing to be ashamed of at all!!"%"
       'B)
      ;; Case 3: Got a C
      ((>= num 70)
       (printf "The student handbook says a C is average!"%"
       (printf "Thus, your grade, "A, is average!"%"
       num)
       'C)
      ;; Case 4: Something else
      (else
       (printf "Hmmm... Hard to find much positive to say here."
       (printf "Maybe there’s been a mistake..."
       (printf "But until we find it, your grade stands...
       'D))))

The behavior of this function is illustrated below:

> (cond-effects 94)
Oh my gosh! You did great!!!
A
> (cond-effects 86)
Well, you know, a B is pretty good!!
Nothing to be ashamed of at all!!
B
> (cond-effects 75)
The student handbook says a C is average!
Thus, your grade, 75, is average!
C
> (cond-effects 41)
Hmmm... Hard to find much positive to say here.
Maybe there’s been a mistake...
But until we find it, your grade stands...
D
```

In each case, the conditions were evaluated sequentially until one was found that evaluated to #t. The subsidiary expressions associated with that condition were then evaluated sequentially, and the value of the last subsidiary expression was given as the value of the entire cond expression. (Although DrScheme reports the output value in a different color, it is hard to see the differences in color in a black-and-white transcript of an Interactions Window session.)
For example, the expression, (cond-effects 86), was evaluated as follows. First, the condition, (>= num 90), was evaluated. Since it evaluated to #f, the second condition, (>= num 80), was evaluated. This one evaluated to #t. Thus, the associated subsidiary expressions were evaluated in turn. The value of the last subsidiary expression was B. Thus, B was returned as the output value for the entire cond expression. Notice that only the subsidiary expressions associated with the condition, (>= num 80), were evaluated. The subsidiary expressions associated with the other conditions were ignored. The remaining conditions (i.e., (>= num 70) and #t) were also ignored.

You should walk through the evaluation of the other sample expressions (e.g., (cond-effects 94) and (cond-effects 41)) to make sure that you understand what DrScheme is doing.

A note about when. Recall that, in general, the syntax of a when special form is as follows.

\[
\text{(when } \text{condExpr}
\begin{align*}
\text{expr}_1 \\
\text{expr}_2 \\
\vdots \\
\text{expr}_n
\end{align*}
\text{)}
\]

This is equivalent to the following cond expression, which has exactly one case.

\[
\text{(cond}
\begin{align*}
\text{(when } \text{condExpr}
\begin{align*}
\text{expr}_1 \\
\text{expr}_2 \\
\vdots \\
\text{expr}_n
\end{align*}
\text{)}
\end{align*}
\text{)}
\]

For each expression, when or cond, if the condition evaluates to something that counts as true, then each of the expressions, \text{expr}_i, is evaluated in turn, and the value of the last expression, \text{expr}_n, is taken to be the value of the entire when or cond expression. However, if the condition evaluates to #f, then the value of the entire expression is void.

13.2 Boolean Operators: \textit{AND}, \textit{OR} and \textit{NOT}

This section introduces the boolean operators, \textit{AND}, \textit{OR} and \textit{NOT}. Mathematically, these operators are functions that take boolean inputs and generate boolean outputs. In Scheme, for efficiency reasons, the first two are provided as special forms—\textit{and} and \textit{or}—while \textit{not} is provided as a built-in function. Here, the generic term \textit{operator} is used to include all three, regardless of how they are implemented in Scheme.

Since there are only two possibilities for the value of a boolean input—either \textit{true} or \textit{false}—boolean operators are frequently defined using truth tables. A truth table simply lists the possible inputs together with the corresponding output. Since \textit{NOT} takes only one input, there are two rows in its truth table. However, \textit{AND} and \textit{OR} each take two boolean inputs, so there are four rows in their truth tables. The top row of Fig. 13.1 shows the truth tables for \textit{AND}, \textit{OR} and \textit{NOT}. These truth tables indicate that:

- the \textit{AND} operator takes two boolean inputs, and outputs \textit{true} if \textit{both} of its inputs are \textit{true};
- the \textit{OR} operator takes two boolean inputs, and outputs \textit{true} if \textit{at least one} of its inputs is \textit{true}; and
- the \textit{NOT} operator takes a single boolean input, and outputs the opposite boolean value.

The bottom row of the figure presents the same information using the binary values 0 and 1, instead of \textit{true} and \textit{false}, respectively.
Figure 13.1: Truth tables for AND, OR and NOT, expressed using truth values (above) and binary values (below)

13.2.1 The Built-in not Function

The Global Environment associates the not symbol with a built-in function. When given a boolean input, the not function returns the opposite boolean value, as illustrated below.

```
> (not #t)
#f
> (not #f)
#t
```

However, the not function also accepts any other kind of Scheme datum as input. It, too, observes the rule that anything other than #f counts as boolean true, as demonstrated below.

```
> (not 'symbol)
#f
> (not (+ 2 3))
#f
> (not ())
#f
> (not "string")
#f
> (not #f)
#t
```

In the first four examples, the non-boolean input is interpreted as boolean true; thus, the output is #f. In the last example, the input is boolean false; hence, the output is boolean true.

The following contract summarizes the behavior of not.

```
;; NOT (built-in)
;; -----------------------------------------------
;; INPUT: DATUM, any Scheme datum
;; OUTPUT: If DATUM is #f, then the output is #t
;; Otherwise the output is #f.
;; Note: Any datum other than #f is interpreted as boolean true.
```
In-Class Problem 13.2.1

Define a function, called my-not, that exhibits the same behavior as the not function described above. Implement it using the if special form.

13.2.2 The and Special Form

In the simplest case, the syntax of the and special form looks like this:

```
(and boolOne boolTwo)
```

where `boolOne` and `boolTwo` are any Scheme expressions that evaluate to booleans. If `boolOne` and `boolTwo` both evaluate to `#t`, then the and special form itself evaluates to `#t`. If either or both evaluate to `#f`, then the and special form evaluates to `#f`.

Example 13.2.2

The following Interactions Window session demonstrates the behavior of and:

```
> (and #t #t)
#t
> (and (> 3 2) (< 5 9))
#t
> (and #t #f)
#f
> (and (> 3 2) (= 5 9))
#f
> (and #f #t)
#f
> (and (> 2 5) #t)
#f
> (and #f #f)
#f
> (and (> 2 5) (= 9 91))
#f
```

Although and could have been provided as a built-in function, Scheme provides it as a special form. To see why, suppose `myBigBadFunc` is a function that takes a really long time to compute its output value. Now consider the expression, `(and (= 9 21) (myBigBadFunc 32))`. Since the first boolean expression, `(= 9 21)`, evaluates to `#f`, the value of the entire and expression must be `#f`. Thus, there is no reason to waste time computing the value of `(myBigBadFunc 32)`. If and were provided as a built-in function, there would be no way to avoid such useless computations. (Recall that the Default Rule for evaluating non-empty lists starts by evaluating all of the entries in a given list.) Thus, Scheme provides and as a special form. The evaluation rule for the and special form is lazy in that it stipulates that only the expressions needed to ascertain the answer are actually evaluated. In particular, if the first boolean expression evaluates to `#f`, then the second boolean expression is not evaluated—because its value does not affect the value of the entire and expression.

The non-strict version of the and special form. The and special form also accepts non-boolean input expressions. Like the not function, it treats any non-boolean expression as though it were boolean true (i.e., anything other than `#f` is interpreted as boolean true). The only catch is that the non-strict version of the and special form may not generate strictly boolean output values! However, as long as we interpret non-boolean output values as though they were boolean `true`, all will be well.
Example 13.2.3

The following Interactions Window session demonstrates the behavior of `and` with non-strict truth values:

```scheme
> (and 3 4) ← the output is 4, which counts as true
4
> (and (* 3 4) (* 8 8)) ← the output is 64, which counts as true
64
> (and (* 3 4) (= 9 7)) ← the output is boolean false
#f
```

This behavior of the `and` special form is easy to explain. The only way that the value of an `and` expression can be `true` is if both input expressions evaluate to `true`—or something that counts as true. In such cases, the value of the `and` expression is simply the value of the last input expression. On the other hand, the only way that an `and` expression can evaluate to boolean `false` is if at least one of the input expressions evaluates to `#f` (i.e., the only thing that counts as `false`).

In-Class Problem 13.2.2

Define a function, called `my-and`, satisfies the following contract:

```scheme
;; MY-AND
;; -----------------------------------------
;; INPUTS: D1, D2, any Scheme data
;; OUTPUT: #t (or something that counts as true) if both D1
;; and D2 are #t (or something that counts as true);
;; #f otherwise (i.e., if D1 or D2 is false)
```

Implement this function using the `if` special form; do not use the `and` special form.

* Because `my-and` is the name of a function, not a special form, an expression such as
  `(my-and (+ 2 3) (* 5 6))` will be evaluated by the Default Rule. Therefore, both
  `(+ 2 3)` and `(* 5 6)` will necessarily be evaluated—in this case, `my-and` would be applied
to the inputs 5 and 30, not the lists `(+ 2 3)` and `(* 5 6)`.

More than two input expressions for the `and` special form. The `and` special form, like many of the built-in arithmetic functions, can take more than two input expressions. In such cases, the value of the `and` expression is true (or something that counts as true) if and only if all of the input expressions evaluate to true (or something that counts as true), as demonstrated below.

Example 13.2.4

```scheme
> (and #t #t #t #t) #t
> (and #t #t #f #t #t) #f
> (and (> 3 2) (= 9 9) (<= 5 20)) #t
> (and 1 2 3 4 5) ← the output value is 5, which counts as true
5
> (and 1 2 #f 4 5)
```
Notice that if the input expressions are strict (i.e., expressions that evaluate to booleans), then the `and` expression will evaluate to a boolean. However, if one or more of the input expressions is non-boolean, then the `and` expression might evaluate to a non-boolean value.

### 13.2.3 The `or` Special Form

The `or` special form is very similar to the `and` special form. The key difference is that an `or` special form evaluates to boolean `true` (or something that counts as true) if and only if at least one of the input expressions evaluates to boolean `true` (or something that counts as true). The behavior of the `or` special form is illustrated below.

**Example 13.2.5**

```
> (or #f #f #f #f)
#f
> (or #f #f #t #f)  ← at least one input evaluates to #t
#t
> (or #t #t #t #t)  ← ditto!
#t
> (or (= 9 8) (> 7 9) (<= 4 2))  ← each input evaluates to #f...
#f
> (or #f #f 3 #f #f 5)  ← 3 "counts as" true
3
```

In the first four examples, all of the input expressions evaluate to actual booleans; thus, the `or` expression itself evaluates to an actual boolean. In the last example, one of the input expressions, 3, is not an actual boolean—although it counts as true. In this case, the value of the `or` expression is 3, which counts as true.

### In-Class Problem 13.2.3

Define a function that satisfies the following contract:

```
;;  IN-CLASS?
;;  -------------------------------------------------------------
;;  INPUTS: DAY, a symbol, one of MON, TUE, ..., SUN
;;           AM-OR-PM, a symbol, one of AM or PM
;;  OUTPUT: #t if we have a lecture or lab scheduled during
;;           that portion of the day; #f otherwise.
```

For the purposes of this exercise, assume that our class holds lectures on Tuesday and Thursday mornings, and labs on Friday afternoons.

**Hint:** Use the built-in `eq?` function to test whether two symbols are equal (e.g., the symbol `mon` and the value of the input parameter `day`, or the symbol `am` and the value of the input parameter `am-or-pm`).
In-Class Problem 13.2.4

Recall that times in the 24-hour military clock involve hours that range from 0 to 23. For example, 00:00 corresponds to midnight; 08:23 is sometime in the morning; 12:00 corresponds to noon; and 15:39 is sometime in the afternoon. For this problem, you will focus on the number of hours, and the time of day (e.g., AM, PM, NOON or MIDNIGHT). In particular, define a function that satisfies the following contract:

\[
\text{CIVIL-TO-MIL-HOURS} \\
\begin{align*}
\text{;; INPUTS:} & \text{ CIVIL-HOURS, an integer from 1 to 12, inclusive} \\
\text{;; TIME-OF-DAY, a symbol, one of AM, PM, NOON or MIDNIGHT} \\
\text{;; OUTPUT:} & \text{ An integer from 0 to 23, inclusive, representing the corresponding number of hours in military notation.}
\end{align*}
\]

Here are a few examples of the desired behavior:

\[
> (\text{civil-to-mil-hours} 3 \text{'am})
3
\]
\[
> (\text{civil-to-mil-hours} 3 \text{'pm})
15
\]
\[
> (\text{civil-to-mil-hours} 12 \text{'midnight})
0
\]

Hints: Use \text{eq?} to test whether two symbols are equal (e.g., am and the value of the input \text{time-of-day}). Use the = function to test whether two numbers are the same (e.g., 3 and the value of civil-hours).

13.3 Defining Predicates using Boolean Operators

When defining predicates (i.e., functions that output boolean values), it is often possible to write the body of the predicate using only the boolean operators, \text{and}, \text{or} and \text{not}, instead of the conditional expressions, \text{if} or \text{cond}. Often times, the solutions using the boolean operators can be quite elegant, matching the structure of how we might think about the solutions in English. The examples below contrast the two approaches to defining a predicate.

Example 13.3.1: Defining a predicate using conditional expressions

The CMPU-101 Cafe is open from 11:30 p.m. on Wednesdays thru 9:15 a.m. on Fridays. The goal of this example is to define a function, called \text{cafe-open?}, that satisfies the following contract:

\[
\text{CAFE-OPEN?} \\
\begin{align*}
\text{;; INPUTS:} & \text{ DAY, a symbol, one of SUN, MON, TUE, ..., FRI, SAT} \\
\text{;; AM-OR-PM, a symbol, either AM or PM} \\
\text{;; HOUR, an integer from 1 to 12, inclusive} \\
\text{;; MINUTES, an integer from 0 to 59, inclusive} \\
\text{;; OUTPUT:} & \text{ #t, if the inputs specify a time at which the CAFE CMPU-101 is open; #f, otherwise.}
\end{align*}
\]

* For this problem, we will ignore the issue of midnight vs. noon. In other words, we won’t deal with inputs for which hour = 12 and minutes = 0. However, we will deal with inputs such as: hour = 12, minutes = 25, and am-or-pm = am (i.e., 12:25 a.m.).

Here are some examples of the desired behavior of this function:
> (cafe-open? 'tue 'am 10 30) #f
> (cafe-open? 'wed 'am 11 45) #f
> (cafe-open? 'wed 'pm 11 45) #t
> (cafe-open? 'thu 'am 12 15) #t

For this version of the cafe-open? predicate, we’ll use a cond special form, where the first case will handle inputs representing a time after 11:30 p.m. on Wednesday night; the second case will deal with Thursdays; and so on.

(define cafe-open?
  (lambda (day am-or-pm hour minutes)
    (cond
      ;; Case 1: Open after 11:30 pm on Wednesdays
      ((and (eq? day 'wed)
            (eq? am-or-pm 'pm)
            (= hour 11)
            (> minutes 30))
       #t)
      ;; Case 2: Open all day on Thursdays
      ((eq? day 'thu)
       #t)
      ;; Case 3: Open Friday mornings *before* 9
      ;; (including times such as 12:25 a.m.)
      ((and (eq? day 'fri)
            (eq? time-of-day 'am)
            (or (< hour 9) (= hour 12)))
       #t)
      ;; Case 4: Open Friday mornings between 9 and 9:15
      ((and (eq? day 'fri)
            (eq? time-of-day 'am)
            (= hour 9)
            (<= minutes 15))
       #t)
      ;; Case 5: Closed at all other times
      (else #f)))))

Note that the cases in this cond expression can be built up incrementally. For example, we could have started with just case 1 and the else case. When those were working, we could’ve inserted case 2, testing to make sure the new case was working before inserting case 3, and so on, until all cases were working.

Example 13.3.2: Defining a predicate using boolean operators

This example illustrates that a predicate such as cafe-open? can be written using the boolean operators, and, or and not, instead of the conditional expressions, cond or if. When approaching the definition of a predicate in this way, the following advice can be very helpful:

* The body of the predicate should specify the conditions under which the predicate will output #t (or
something that counts as true).

In the preceding example, each of the cases 1 through 4 of the cond expression represented one range of times when the cafe is open. We might think about it this way: the cafe is open if case 1 holds, or case 2 holds, or case 3 holds, or case 4 holds. This observation leads to the following solution, which we'll call, cafe-open?-alt:

```
(define cafe-open?-alt
  (lambda (day am-or-pm hour minutes)
    ;; The following expression specifies the conditions under
    ;; which this function will output #t (or something that
    ;; counts as true):
    (or;; Case 1: Wednesday after 11:30 p.m.
      (and (eq? day 'wed)
        (eq? am-or-pm 'pm)
        (= hour 11)
        (> minutes 30))
    ;; Case 2: Anytime Thursday
    (eq? day 'thu)
    ;; Case 3: Friday before 9 a.m.
    (and (eq? day 'fri)
      (eq? time-of-day 'am)
      (or (< hour 9) (= hour 12)))
    ;; Case 4: Friday between 9 and 9:15 a.m.
    (and (eq? day 'fri)
      (eq? time-of-day 'am)
      (= hour 9)
      (<= minutes 15)))))
```

Notice that there is no need to provide anything resembling an else condition. If the expression in the body evaluates to #t: fine, the cafe is open; if it evaluates to #f, then the cafe is closed.

### 13.4 Simplifying Conditional and Boolean Expressions

Just as an expression of the form (if condExpr #t #f) is equivalent to the simpler expression condExpr, an expression of the form (if condExpr #f #t) is equivalent to the simpler expression (not condExpr), as illustrated below.

```
> (if 'sad #f #t)
#f
> (not 'sad)
#f
```

Using the notation introduced in Section 11.2, we may write:

```
(if condExpr #f #t)  ~>  (not condExpr)
```

Of course, if condExpr does not evaluate to a boolean, then the equivalence requires us to accept that anything other than #f counts as true.

Next, we consider (possibly non-strict) conditions involving the and and or special forms. In particular, it is never necessary to embed one and expression directly inside another and expression, and it is never necessary to embed one or expression directly inside another or expression. For example:

- For any (possibly non-strict) expressions, $e_1, e_2$ and $e_3$, the following simplifications yield equivalent expressions:
13.5 Summary

This chapter introduced the cond special form for making n-ary decisions; and the boolean operators and, or and not that can be combined to form complex boolean expressions.

* Like the if special form, the and, or and cond special forms, as well as the built-in not function, all accommodate non-strict truth values (i.e., where anything other than #f counts as true).

The and special form can take any number of arguments. It evaluates to (non-strict) true if and only if all of its arguments evaluate to (non-strict) true. Similarly, the or special form can take any number of arguments, and evaluates to (non-strict) true if and only if at least one of its arguments evaluates to (non-strict) true. Evaluation of the and special form is lazy in that if any argument evaluates to #f, none of the remaining arguments are evaluated, because the value of the entire and expression must be #f. Similarly, evaluation of the or special form is lazy in that if any argument evaluates to (non-strict) true, then none of the remaining arguments are evaluated, because the value of the entire or expression must be true.

The cond special form facilitates making decisions among any number of cases. Each case in a cond expression is represented by a subsidiary list whose first element represents the condition to be tested, and the rest of whose expressions form the body of that case. A cond expression is evaluated by considering each case, in turn, until one is found whose condition evaluates to (non-strict) true. At that point, the expressions in the body of that case are evaluated; and the value of the last expression in the body of that case is taken to be the value of the entire cond expression.

* If the condition for a given case evaluates to #f, the expressions in the body of that case are ignored.

* If the i^{th} case is the first case whose condition evaluates to (non-strict) true, then the expressions in the body of that case are evaluated; and all subsequent cases are ignored.

Although a cond expression involving n cases can often be re-written using n – 1 nested if expressions, the syntax of the cond expression is simpler, especially for large n. However, a cond expression can also be more general than a chain of nested if expressions because the body of each case of a cond expression can include multiple expressions, just as the body of a lambda expression can include multiple expressions. In contrast, the then and else expressions in an if special form can only consist of a single expression each. In addition, the syntax of cond expressions make them more amenable to inserting helpful comments.

* To ensure that some case of a cond is selected, the condition for the last case—sometimes called the default or catch-all case—should always be either #t or else.

This chapter also demonstrated that predicates can be defined using the boolean operators, and, or and not, instead of the conditional expressions, if or cond. When using this approach, the expression in the body of the predicate should specify the conditions under which the predicate should output #t (or something that counts as true). And finally, this chapter exhibited some common ways of simplifying certain conditional and boolean expressions:

\[
\begin{align*}
\text{and} & \quad \text{and} & \quad \text{and} \\
(\text{and} \; e_1 \; (\text{and} \; e_2 \; e_3)) & \quad \rightarrow & \quad (\text{and} \; e_1 \; e_2 \; e_3) \\
(\text{or} \; e_1 \; (\text{or} \; e_2 \; e_3)) & \quad \rightarrow & \quad (\text{or} \; e_1 \; e_2 \; e_3)
\end{align*}
\]

Since and and or can each take any number of inputs, there are many other examples of this kind of simplification. However:

* Be careful about cases where an and expression is directly embedded within an or expression, or vice-versa. These sorts of expressions do not simplify as readily.

For example, De Morgan’s Laws stipulate that the following equivalences hold, but we can’t really call them simplifications:

\[
\begin{align*}
\text{not} & \quad \text{not} & \quad \text{not} \\
(\text{not} \; (\text{and} \; e_1 \; e_2)) & \quad \rightarrow & \quad (\text{or} \; (\text{not} \; e_1) \; (\text{not} \; e_2)) \\
(\text{not} \; (\text{or} \; e_1 \; e_2)) & \quad \rightarrow & \quad (\text{and} \; (\text{not} \; e_1) \; (\text{not} \; e_2))
\end{align*}
\]
(if someExpr #t #f)  \rightarrow  someExpr
(if someExpr #f #t)  \rightarrow  (not someExpr)
(and e_1 (and e_2 e_3))  \rightarrow  (and e_1 e_2 e_3)
(or e_1 (or e_2 e_3))  \rightarrow  (or e_1 e_2 e_3)

Special Forms Introduced in this Chapter

and  Evaluates to true if all its inputs evaluate to true
or  Evaluates to true if at least one of its inputs evaluates to true
cond  For making decisions among any number of choices

Built-in Functions Introduced in this Chapter

not  Toggles boolean values
Chapter 14

Recursion II

This chapter continues the exploration of recursion begun in Chapter 12.

14.1 Recalling the Factorial Function

We begin by considering an equivalent version of the facty-v1 function, called facty-v2, that uses the \texttt{cond} special form instead of the \texttt{if} special form.

\begin{example}

\begin{verbatim}
;; FACTY-V2
;; -----------------------------------------------
;; INPUT: N, a positive integer
;; OUTPUT: The factorial of N (i.e., N*(N-1)*...*3*2*1)

(define facty-v2
  (lambda (n)
    (cond
      ;; Base Case: n = 1
      ((= n 1) 1)
      ;; Recursive Case: n > 1
      (#t (* n (facty-v2 (- n 1))))))

Notice how the comments clearly distinguish the base case from the recursive case. And notice that the answer expression, 1, in the base case is written on a separate line, even though it is quite short. Following the convention of putting the answer expression for each case on the line following the corresponding condition makes life easier for people looking at your code! Finally, notice how indentation helps to distinguish the cases of the \texttt{cond}.

Like facty-v1, this function appears to correctly compute the factorial of its input:

> (facty-v2 1)
1
> (facty-v2 2)
2
> (facty-v2 3)
6
\end{verbatim}
\end{example}
Example 14.1.2

Finally, we can define another equivalent version of the factorial function, this one called facty. This function differs only in that it contains some printf expressions that will help us to trace what happens when an expression such as (facty 3) is evaluated.

;;; FACTY
;;; -----------------------------------------------
;;; INPUT: N, a positive integer
;;; OUTPUT: The factorial of N (i.e., N*(N-1)*...*3*2*1)
;;; SIDE EFFECT: Displays base-case vs. recursive-case information for each function call.

(define facty
  (lambda (n)
    (cond
     ;; Base Case: n = 1
     (= n 1)
     (printf "Base Case (n = 1)" n)
     1)
     ;; Recursive Case: n > 1
     (#t
      (printf "Recursive Case (n = ~A)" n)
       (* n (facty (- n 1)))))))

Notice that the printf expressions do not affect the output of the function; they only cause some useful side-effect printing to occur. The following interactions demonstrate the desired behavior.

> (facty 3)
Recursive Case (n = 3)
Recursive Case (n = 2)
Base Case (n = 1)
6

Notice how the side-effect printing helps to demonstrate that the evaluation of (facty 3) mirrors the evaluation of (facty-v1 3) seen previously in Fig. 12.1.

Example 14.1.3: Summing Squares

Consider the function, \( g(n) = 1^2 + 2^2 + 3^2 + \ldots + n^2 \). Notice that \( g(n) \) sums the squares of the integers between 1 and \( n \), inclusive. Furthermore, for any \( n > 1 \), notice that the sum of the first \( n \) squares is the same as the sum of the first \( n - 1 \) squares plus \( n^2 \). Therefore, we can define \( g \) recursively, as follows:

\[
\begin{align*}
\text{Base Case (n = 1):} & \quad g(1) = 1 \\
\text{Recursive Case (n > 1):} & \quad g(n) = g(n - 1) + n^2
\end{align*}
\]

Notice that \( g(1) = 1 \), \( g(2) = 1^2 + 2^2 = 5 \), \( g(3) = 1^2 + 2^2 + 3^2 = 14 \), and so on.
In Scheme, we can define a function, called `sum-squares`, that computes the sum of the squares from 1 to its input value `n`, as follows:

```scheme
;; SUM-SQUARES
;; -----------------------------------------
;; INPUT: N, a positive integer
;; OUTPUT: The sum \( 1 \times 1 + 2 \times 2 + \ldots + N \times N \)
(define sum-squares
 (lambda (n)
   (cond
    ;; Base Case: n = 1
    ((= n 1) 1)
    ;; Recursive Case: n > 1
    (#t (+ (sum-squares (- n 1)) (* n n))))))
```

We can test this function in the Interactions Window, as follows:

```
> (sum-squares 1)
1
> (sum-squares 2)
5
> (sum-squares 3)
14
> (sum-squares 4)
30
```

### 14.2 Tail Recursion

Typically, the evaluation of a recursive function-call expression leads to a sequence of recursive function calls. For example, evaluating the expression, `(facty 5)`, effectively requires evaluating `(facty 4)`, `(facty 3)`, `(facty 2)` and `(facty 1)`. Similarly, evaluating `(facty 100)` would involve a sequence of nearly one hundred recursive function calls. For functions such as `facty`, the evaluation of each recursive function call is suspended pending the evaluation of all of the subsidiary function calls. Keeping track of all of these suspended evaluations requires storing relevant information somewhere in the computer's memory. Thus, if the value of `n` gets large enough, DrScheme's evaluation of `(facty n)` would eventually cause problems. In particular, at some point, the operating system would refuse to grant DrScheme more memory to hold the needed information.

If this kind of memory-usage problem were characteristic of all recursive functions, it might severely limit their usefulness. However, if the body of the recursive function is defined in a certain way, the memory-usage problem ceases to be a problem. In particular, if the recursive function is tail recursive—which shall be defined below—then DrScheme can, in effect, re-use a single block of memory, over and over again, as it evaluates all of the recursive function calls in a given sequence, instead of requiring a separate block of memory for each recursive function call. In effect, for a tail-recursive function, DrScheme can use a single function-call box to process an entire sequence of recursive function calls, instead of using a separate function-call box for each function call.

This section describes tail-recursive functions and shows how DrScheme can avoid the memory-usage problems associated with non-tail-recursive functions. We begin with an example of a tail-recursive function.
### Example 14.2.1: Printing Dashes

Consider the `print-n-dashes` function, defined below:

```scheme
;; PRINT-N-DASHES
;; -----------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: None
;; SIDE EFFECT: Prints N dashes in the Interactions Window

(define print-n-dashes
  (lambda (n)
    (cond
      ;; Base Case: n <= 0
      ((<= n 0) (newline))
      ;; Recursive Case: n > 0
      (#t
       ;; Print one dash
       (printf "-")
       ;; The recursive func call prints the rest of the dashes
       (print-n-dashes (- n 1))))))
```

This function does not generate any output value; instead, it has the side effect of displaying a row of \( n \) dashes in the Interactions Window, as illustrated below.

```
> (print-n-dashes 5)
-----
> (print-n-dashes 12)
----------
```

Consider the evaluation of the expression, `(print-n-dashes 5)`. According to the Default Rule for evaluating non-empty lists, evaluating this list requires applying the `print-n-dashes` function to the input value 5. Thus, a function-call box must be set up with a local environment containing an entry for the input parameter, \( n \), whose value is 5. Next, the body of the function is evaluated. Since \( n \) has the value 5 in this function-call box, we are in the recursive case. Thus, the two printing expressions must be evaluated in turn. Recall, too, that the value of the last expression will be the output for this function call. Evaluating the first expression, `(printf "-")`, causes a single dash to be displayed in the Interactions Window. Evaluating the second expression, `(print-n-dashes (- n 1))`, requires making a recursive function call.

At this point, we would normally require a new function-call box to process the recursive application of `print-n-dashes` to the value 4. However, we make the following crucial observation:

* When the value of the recursive function-call expression, `(print-n-dashes (- n 1))`, is known, it will be the output value for the original expression, `(print-n-dashes 5)`. **Thus**, we don’t really need the information in the first function-call box anymore. As a result, we can simply re-use the function-call box for the second function call.

Thus, instead of creating a new function-call box for the application of `print-n-dashes` to the value 4, DrScheme simply re-uses the function-call box it already has at hand. **This will require DrScheme to erase the value 5 for the local parameter \( n \) and replace it with the value 4, and then proceed to evaluate the body of the function with respect to this new local environment.**

* You may object that DrScheme is engaged in destructive programming. **And you are right! However, that does not have any bearing on the non-destructiveness of the `print-n-dashes` function.**
The semantics of Scheme stipulates that each recursive function call gets a new function-call box. Thus, according to the semantics of Scheme, the print-n-dashes function is non-destructive. However, DrScheme is privately re-using a single block of memory, using destructive techniques to perform a sequence of computations that are equivalent to those it would have performed if it were using the non-destructive techniques. Because DrScheme’s use of destructive computation is equivalent to the desired non-destructive computation, this is a safe use of destructive computing. Notice, too, that our hands are clean! We are writing non-destructive functions!

To reiterate: From a theoretical viewpoint, the evaluation of tail-recursive function calls is no different from the evaluation of non-tail-recursive function calls: neither is destructive. However, the DrScheme software makes efficient use of memory when evaluating tail-recursive function calls. At a very low-level, this can be construed as destructive; however, our Scheme programs are nonetheless non-destructive!

If I were to ask you to draw a sequence of function-call boxes for all of the expressions, (print-n-dashes 5), (print-n-dashes 4), ..., (print-n-dashes 0), you would probably get tired—especially when you realized that you would lose no information by simply re-using a single function-call box for processing the entire sequence of recursive function calls. That’s all that DrScheme is doing when it processes a tail-recursive function call.

The print-n-dashes function is an example of a tail-recursive function. But what exactly do we mean by the term, tail recursive?

**Definition 14.1: Tail-recursive function**

Suppose that $f$ is a function, $\mathcal{B}$ is its body, and $\text{expr}$ is a recursive function-call expression somewhere within $\mathcal{B}$. We say that $\text{expr}$ is a tail-recursive function-call expression within $\mathcal{B}$ if, whenever evaluating $\mathcal{B}$ requires evaluating $\text{expr}$, it is necessarily the case that the last step in evaluating $\mathcal{B}$ is the evaluation of $\text{expr}$ and, thus, the value of $\mathcal{B}$ is identical to the value of $\text{expr}$. If every recursive function-call expression in the body of $f$ is tail-recursive, then $f$ is called a tail-recursive function.

Okay, the above definition is correct and completely general, but it may be a little hard to process. The following example considers a less general, but quite common case of a tail-recursive function—one that exhibits the characteristic features, and covers the print-n-dashes from Example 14.2.

**Example 14.2.2**

Suppose that $\text{rec-func}$ is a recursive function whose body $\mathcal{B}$ consists of a single cond expression. Suppose further that this cond has only two cases: a base case and a recursive case. The only way that $\text{rec-func}$ can be tail recursive is if, as shown below, the recursive function-call expression, ($\text{rec-func } ...$), is the last (i.e., tail) expression within the recursive case.

```scheme
(define rec-func
  (lambda (...)
    (cond
      ;; Base Case
      (... ...
      )
      ;; Recursive Case
      (... ...
      (rec-func ...)...)```


The recursive function-call expression must not be a subsidiary expression within some larger expression within the recursive case; it must be the entirety of the last (i.e., tail) expression. If that is the case, then whenever the recursive case applies, the value for the entire cond expression will be the result of evaluating the recursive function call. (It is precisely this feature that enables DrScheme to recycle the function-call box as described earlier.) Hence, according to Defn. 14.2, this function is tail recursive; as is the print-n-dashes function from Example 14.2.

In contrast, consider the definition of the facty function, seen earlier:

```
(define facty
  (lambda (n)
    (cond
      ;; Base Case:  n = 1
      (eqv? n 1) 1)
      ;; Recursive Case:  n > 1
      (else (* n (facty (- n 1)))))))
```

Notice that the last expression in the recursive case of the cond is (* n (facty (- n 1))). This expression includes the recursive function-call expression, (facty (- n 1)), as a subsidiary expression. This means that the value of the recursive function-call expression is not simply returned as the output value of the parent function-call box. Instead, when the value of the recursive function-call expression is known, some additional computation—in this case, multiplying by n—has to be performed in order to generate the desired output value. For this reason, DrScheme must keep track of the contents of the original function call-box while it processes the recursive function call. Thus, DrScheme must create a separate function call-box for the recursive function call. Thus, DrScheme cannot use the memory-saving trick described for tail-recursive functions. The problem? The function, facty, is not tail recursive.

This is actually the facty-v2 function, but the same points apply to all versions of the facty function seen earlier.

---

**In-Class Problem 14.2.1**

Define a function that satisfies the following contract:

```
;; PRINT-FUNC-VALS
;; -----------------------------------------------------------------
;; INPUTS:  FUNC, a function that expects a single numerical input
;;          FROM, a starting input
;;          TO, an ending input
;; OUTPUT:  None
;; SIDE EFFECT:  Prints the values of FUNC when applied to
;;               the successive inputs from FROM to TO.
```

Tail-recursive functions like print-n-dashes do not generate interesting output values; instead, their primary purpose is to display information in the Interactions Window as a side effect. Functions that generate interesting output values can also be tail recursive; however, they typically require one or more additional input parameters. Frequently, those additional input parameters are called *accumulators* because they are used to incrementally accumulate values of interest. Section 14.3 addresses accumulator-based tail-recursive functions.
14.3 Accumulators

In the factorial example, seen earlier, each recursive function call generated an output value that represented a solution to a simpler problem. For example, the evaluation of \((\text{facty} \ 4)\) (i.e., \(4!\)) resulted in the recursive function calls, \((\text{facty} \ 3), (\text{facty} \ 2)\) and \((\text{facty} \ 1)\), whose values were \(3!, 2!\) and \(1!\), respectively. This section explores a slightly different way of organizing recursive computations using accumulators.

An accumulator is nothing more than an input parameter that is used, in effect, to incrementally accumulate the result of a desired computation.

As each recursive function call is made, the value of the accumulator gets closer and closer to the desired output value, until finally, when the base case is reached, the accumulator holds the desired answer. Accumulator-based recursive functions are typically tail recursive. This section explores the use of accumulators in tail-recursive functions.

---

**Example 14.3.1: Computing sums of the form, \(0 + 1 + 2 + \ldots + n\) without accumulators**

We begin with a non-tail-recursive function, \(\text{sum-to-n}\):

```scheme
;;; SUM-TO-N
;;; ------------------------------------------------
;;; INPUT: N, number (non-negative integer)
;;; OUTPUT: The value of the sum \(0 + 1 + 2 + \ldots + n\)
;;; NOTE: This function is NOT tail recursive and does
;;; NOT have any accumulators!

(define sum-to-n
  (lambda (n)
    (cond
      ;; Base Case: n = 0
      (=(n 0)
        (printf "Base Case \(n=0\)\n"
          0)
      ;; Recursive Case: n > 0
      (#t
        (printf "Recursive Case \(n=\) \n"
          (+ n (sum-to-n (- n 1))))))))

As in prior examples, the printf expressions serve only to display information about the recursive function calls; they do not affect the output value, as illustrated below:

> (sum-to-n 3) ;; compute \(0 + 1 + 2 + 3\)
Recursive Case \(n=3\) ...
Recursive Case \(n=2\) ...
Recursive Case \(n=1\) ...
Base Case \(n=0\)
6

Notice that the evaluation of \((\text{sum-to-n} \ 3)\) involved a sequence of function calls—namely: \((\text{sum-to-n} \ 3), (\text{sum-to-n} \ 2), (\text{sum-to-n} \ 1)\) and \((\text{sum-to-n} \ 0)\).```
Example 14.3.2: Computing sums of the form, $0+1+2+\ldots+n$ with an accumulator

Below, we define a function, `sum-to-n-acc`, that solves the same problem using an extra input parameter, called an accumulator. The accumulator is like a basket that starts out empty, but incrementally accumulates stuff; when the base case is reached, the accumulator (i.e., the basket) holds the desired answer. Once again, the `printf` expressions serve only to display useful information; they do not affect the output value.

```scheme
;;; SUM-TO-N-ACC
;;; ----------------------------------------------
;;; INPUTS: N, a non-negative integer
;;; ACC, a number (an accumulator)
;;; OUTPUT: When called with ACC=0, the output is the value
;;; 0 + 1 + 2 + \ldots + N.
;;; More generally, the output is the value of
;;; ACC + 0 + 1 + 2 + \ldots + N.
(define sum-to-n-acc
  (lambda (n acc)
    (cond
      ;; Base Case: n = 0
      (= n 0)
        (printf "Base Case (n=0, acc=\~A)\" acc)
      ;; Return the accumulator!
      acc)
      ;; Recursive Case: n > 0
      (#t
        (printf "Recursive Case (n=\~A, acc=\~A)\" n acc)
      ;; Make recursive function call with updated inputs
      (sum-to-n-acc (- n 1) (+ acc n))))))
```

Since the function, `sum-to-n-acc`, includes an extra input parameter, we need to supply the values for both `n` and `acc` when calling this function. Thus, to compute the sum, $0+1+2+3$, using this function, we would evaluate the expression, `(sum-to-n-acc 3 0)`. Notice that the initial accumulator has a value of 0, which is akin to our basket being initially empty. Here’s what the evaluation of `(sum-to-n-acc 3 0)` looks like in the Interactions Window:

```
> (sum-to-n-acc 3 0)
Recursive Case (n=3, acc=0)
Recursive Case (n=2, acc=3)
Recursive Case (n=1, acc=5)
Base Case (n=0, acc=6)
6
```

First off, notice that we see a similar sequence of function calls, where the value of `n` goes from 3 down to 0. However, the value of the accumulator goes from 0—its initial value—up to 6—the desired answer. Notice that the recursive function call, in the body of the function, looks like this:

```scheme
(sum-to-n-acc (- n 1) (+ acc n))
```

Thus, the value of the accumulator for the recursive function call is the original value of the accumulator plus `n`. In other words, our basket has accumulated `n`. However:

* This is not destructive programming! We are not changing the values of any variables! Each function call has its own local environment that includes its own input parameters, called `n` and `acc`. 
Fig. 14.1 illustrates the sequence of recursive function calls generated by DrScheme’s evaluation of (sum-to-n-acc 3 0). Notice that each function-call box has its own input parameters, called n and acc, that are distinct from all the other parameters with the same names in the other function-call boxes.

Although the basket metaphor sounds destructive; it’s not. Instead of a single basket, think of multiple baskets. Each recursive function call involves taking the contents of the old basket (i.e., accumulator) plus some other stuff (i.e., n) and putting the result into a new basket (i.e., accumulator).

Notice that sum-to-n-acc is tail recursive, since the value of the recursive function-call expression, by itself, constitutes the last expression in the recursive case. Thus, the value of the recursive function-call expression is returned as the output value of the original function call. Thus, DrScheme can do its memory-saving trick on this tail-recursive function.

Some of the key characteristics of tail recursion are evident in the figure:

- When the base case is reached, the accumulator holds the desired answer—in this case, 6—for the original computation.
- The output of each of the recursive function calls is the same. In this case, each function call outputs the value 6.

Example 14.3.3: Factorial Revisited

Here is a tail-recursive version of the factorial function, called facty-acc:

```scheme
;;; FACTY-ACC
;; ;----------------------------------------------------
;;; INPUTS: N, a positive integer
;;; ACC, a number
;;; OUTPUT: When called with ACC=1 the output is N! (i.e., the factorial of N).
;;; More generally, the output is: ACC * N!
(define facty-acc
 (lambda (n acc)
   (cond
    ;; Base Case: n = 1
    ((= n 1)
     (printf "Base Case (n=1, acc=\n" acc)
     ;; Return the accumulator!
     acc)
    ;; Recursive Case: n > 1
    (else
     (printf "Recursive Case (n=\n" n acc)
     ;; Recursive function call (tail-recursive)
     (facty-acc (- n 1) (* n acc))))))
```

An expression of the form, (facty-acc n 1), will evaluate to the factorial of n. In other words, the initial value of the accumulator must be 1 (i.e., the multiplicative identity) for this function to achieve its desired result.

Notice that the function, facty-acc, is tail recursive, as evidenced by the fact that the recursive function-call expression, (facty-acc (- n 1) (* n acc)), by itself constitutes the last expression in the recursive case. It is not a subsidiary expression within some larger expression. Thus, the value of the recursive function-call expression is the output value for the original function call-box.\(^a\)
Figure 14.1: DrScheme’s evaluation of \((\text{sum-to-n-acc} \ 3 \ 0)\)
For `facty-acc`, the current accumulator, `acc`, is multiplied by `n` to generate the value of the accumulator for the recursive function call. Since `facty-acc` involves multiplying the current accumulator to generate the value of the next accumulator, the appropriate initial value for the accumulator is 1. Thus, to use `facty-acc` to compute `4!`, we would evaluate an expression such as `(facty-acc 4 1)`, as illustrated below:

```
> (facty-acc 4 1)
Recursive Case (n=4, acc=1)
Recursive Case (n=3, acc=4)
Recursive Case (n=2, acc=12)
Base Case (n=1, acc=24)
24
```

Remember that each function call-box includes its own local environment that contains two parameters, `n` and `acc`. The parameters in each call-box may have the same names as the parameters in the other call-boxes; however they are quite distinct. Thus, there are four distinct parameters named `n`, having the values 4, 3, 2 and 1. Similarly, there are four separate parameters named `acc`, having the values 1, 4, 12 and 24. Notice that by the time the base case is reached, in the final function call, the accumulator, `acc`, has the desired value 24.

Incidentally, the following description of the output value for the function, `facty-acc`, is more general, in that it allows the accumulator to have values other than 1:

* The output value for `(facty-acc n acc)` is equal to the factorial of `n` multiplied by `acc`.

Notice that if `acc` equals 1, then the output value is indeed `n!`. However, if `acc` is something other than 1, then the value is `n! * acc`.

---

Example 14.3.4: Summing squares: $1^2 + 2^2 + \ldots + n^2$

Here’s a tail-recursive function for computing the sums of squares from 1 to `n`:

```scheme
;;; SUM-SQUARES-ACC
;;; -----------------------------------------------
;;; INPUTS: N, a non-negative integer
;;; ACC, a number (accumulator)
;;; OUTPUT: If the accumulator is 0, then the output
;;; is equal to the sum 0*0 + 1*1 + 2*2 + ... + N*N.
;;; More generally, the output is the sum:
;;; ACC + 0*0 + 1*1 + 2*2 + ... + N*N.
(define sum-squares-acc
  (lambda (n acc)
    (cond
      ;; Base Case: n <= 0
      ((<= n 0)
       (printf "Base Case: n=˜A, acc=˜A˜%" n acc)
       ;; Return the accumulator!
       acc)
      ;; Recursive Case: n > 0
      (#t
       ;; Recursive Case: n > 0
       ;; Recursively call the function with n-1 and acc+n^2
       (+ (sum-squares-acc (- n 1) (+ acc (* n n))) acc))))
```

*In contrast, the non-tail-recursive function, `facty`, seen earlier, included the recursive function-call expression, `(facty (- n 1))`, within the larger expression, `(* n (facty (- n 1)))`. 
(printf "Recursive Case: n=\textasciitilde A, acc=\textasciitilde A\textasciitilde\%" n acc)
(sum-squares-acc (- n 1) (+ acc (* n n))))

Notice that the function is clearly tail recursive, since the recursive function-call expression, by itself, is the last expression in the recursive case. (It is not a subsidiary expression within some larger computation.) Notice, too, that the accumulator is initially 0. Finally, notice that the value of the accumulator for the recursive function call is the original accumulator plus \(n^2\). In other words, each recursive function call involves accumulating a squared term.

Here’s the result of evaluating the expression, \((\text{sum-squares-acc } 3 \text{ 0})\), in the Interactions Window:

\[
\begin{align*}
> (\text{sum-squares-acc } 3 \text{ 0}) & \quad \leftarrow 3^2 + 2^2 + 1^2 + 0^2 = 14 \\
\text{Recursive Case: n=3, acc=0} \\
\text{Recursive Case: n=2, acc=9} \\
\text{Recursive Case: n=1, acc=13} \\
\text{Base Case: n=0, acc=14} \\
& 14
\end{align*}
\]

Notice that by the time the base case is reached, the accumulator holds the desired answer—in this case, 14—for the original computation. You should convince yourself that 14 is the output value for each of the recursive function calls shown above.

Although the function returns the desired output value when the accumulator is 0, the following is a more general characterization of this function’s behavior:

> * An expression of the form, \((\text{sum-squares-acc } n \text{ acc})\), evaluates to \(0^2 + 1^2 + \ldots + n^2 + \text{acc}\).

For example, when \(n = 2\) and \(\text{acc} = 9\), the result is \(0^2 + 1^2 + 2^2 + 9\) (i.e., 14). Similarly, when \(n = 0\) and \(\text{acc} = 14\), the result is \(0^2 + 14\) (i.e., 14).

---

**Example 14.3.5: Approximating \(\pi\)**

Mathematicians tell us that the value of \(\pi\) can be approximated using sums of the form shown below:

\[
\pi \approx 4 \cdot \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots \pm \frac{1}{n} \right)
\]

where \(n\) is some positive odd number. Furthermore, as the value of \(n\) increases, the approximation becomes better and better. This example defines a function, called \(\text{approx-pi-acc}\), that processes terms in the above sum from left to right, using several inputs to keep track of relevant information along the way. On successive recursive function calls, the input \(\text{from}\) will be 1, then 3, then 5, etc. It will be used to identify the particular term in the sum that is currently being processed. The input \(\text{sign}\) will alternate between 1 and -1 and, thus, keeps track of the sign of the current term. The input \(\text{n}\) will not change on successive recursive function calls. It is used as a fixed upper bound that indicates the last term in the sum. And the input \(\text{acc}\) is used to accumulate the desired sum. Here is the contract:

```scheme
;; APPROX-PI-ACC
;; ---------------------------
;; INPUTS: FROM, a positive odd number that specifies the term
;; that is currently being processed
;; SIGN, either +1 or -1, the sign of the term currently
;; being processed
;; N, a positive odd number that specifies the last term
;; in the sum
;; ACC, an accumulator
```
OUTPUT: When called with FROM=1, SIGN=1, and ACC=0, computes
the following estimate of the value of PI:
4 * (1 - 1/3 + 1^5/5 - 1/7 + ... (+/-) 1/n)

(define approx-pi-acc
  (lambda (from sign n acc)
    (cond
      ;; Base Case: FROM > N (i.e., we’ve gone too far!)
      ;; Multiply the accumulator by 4:
      ((> from n)
        (* 4 acc))
      ;; Recursive Case: FROM <= N
      (else
       ;; Tail-recursive function call with adjusted inputs
       (approx-pi-acc
        (+ from 2.0) ;; increment by 2
        (* sign -1) ;; alternate between 1 and -1
        n ;; fixed upper bound
        (+ acc (/ sign from)) ;; accumulate current term
       ))))))

Notice how the accumulator is multiplied by 4 in the base case. In addition, from is incremented by 2.0 to ensure that the computations are done using floating-point numbers instead of fractions. (To see the difference, try testing the function with from incremented by 2 instead of 2.0.) Here are some examples of its use:

> (approx-pi-acc 1 1 3 0) ;; = 4*(1 - 1/3)
 2.666666666666667
> (approx-pi-acc 1 1 5 0) ;; = 4*(1 - 1/3 + 1/5)
 3.466666666666667
> (approx-pi-acc 1 1 101 0) ;; = 4*(1 - 1/3 + ... + 1/101)
 3.1611986129870506
> (approx-pi-acc 1 1 10001 0) ;; = 4*(1 - 1/3 + ... + 1/10001)
 3.1417926135957908
> (approx-pi-acc 1 1 100001 0) ;; = 4*(1 - 1/3 + ... + 1/100001)
 3.1415946535856922

Notice how big the input n must be to get even modestly accurate approximations of π.

Example 14.3.6: Approximating e

Mathematicians tell us that the number e is well approximated by sums of the form

\[ 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!} \]

In particular, as the value of n gets larger, the sum gets closer and closer to the value of e. Below, we define a function, approx-e-acc, that involves several input parameters that can be construed as accumulators. (Sometimes accumulators accumulate really interesting stuff; sometimes they accumulate boring stuff.) For this function:

- the input parameter n, which indicates the last term in the sum, will stay the same across all recursive function calls;
• the input parameter indy will take on the values, 0, 1, 2, ..., n, on successive recursive function calls, and will be used to identify the current term;

• the input parameter curr-denom (i.e., current denominator) will accumulate the factorials that comprise the various denominators that appear in the sum (i.e., 1, 1, 2, 6, 24, ..., n!); and

• the input parameter acc will accumulate the desired sum; it will take on the values 1, 2, 2.5, 2.6666666666666666, ....

;; APPROX-E-ACC
;; -----------------------------------------------
;; INPUTS: N, non-negative integer (indicates last term)
;;        INDY, non-negative integer (indicates current term)
;;        CURR-DENOM, positive integer (current denominator)
;;        ACC, accumulates desired sum
;; OUTPUT: When called with INDY=0, CURR-DENOM=1, and ACC=0,
;;        the output is the following approximation of e:
;;        \[1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{N!}\]

(define approx-e-acc
  (lambda (n indy curr-denom acc)
    ;; Print out the values of the input parameters first...
    (printf "n=\(\text{~A}\), indy=\(\text{~A}\), curr-denom=\(\text{~A}\), acc=\(\text{~A}\)\%"
            n indy curr-denom acc)
    (cond
      ;; Base Case: INDY > N (we’re done!)
      ((> indy n)
       ;; Return the accumulator!
       acc)
      ;; Recursive Case: INDY <= N
      (#t
       ;; Tail-recursive function call with adjusted inputs
       (approx-e-acc
        n  ;; n doesn’t change
        (+ 1 indy) ;; increment indy
        (* (+ 1 indy) curr-denom) ;; update current denom
        (+ acc (/ 1.0 curr-denom))))))))

To get the desired results, the various input parameters must be properly initialized. In particular, the initial values for indy, curr-denom and acc must be 0, 1 and 0, respectively. Thus, the sum

\[1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!}\]

can be computed by evaluating (approx-e-acc 4 0 1 0), as illustrated below:

> (approx-e-acc 4 0 1 0)
n=4, indy=0, curr-denom=1, acc=0
n=4, indy=1, curr-denom=1, acc=1.0
n=4, indy=2, curr-denom=2, acc=2.0
n=4, indy=3, curr-denom=6, acc=2.5
n=4, indy=4, curr-denom=24, acc=2.6666666666666665
n=4, indy=5, curr-denom=120, acc=2.708333333333333
2.708333333333333
Notice that the \( n \) parameter stays fixed at 4 across all the recursive function calls. This parameter is used only to signal the end of the sum. The parameter \( \text{indy} \) takes on the integer values from 0 to 5. It identifies the current term. Thus, when \( \text{indy} \) is greater than \( n \), no further accumulation of terms is necessary. The parameter \( \text{curr-denom} \) represents the denominator of the term currently being worked on; thus, it takes on the values of the successive factorials: 0!, 1!, 2!, 3!, ... Notice that these factorials are not computed from scratch each time; instead, the value of \( \text{curr-denom} \) is simply multiplied by \((+ \text{indy} 1)\) to generate the next factorial. Finally, the parameter \( \text{acc} \) accumulates the desired sum. By the time the base case is reached (i.e., when \( \text{indy} > n \)), the accumulator holds the desired answer. Thus, the accumulator is simply returned as the output value for this function.

If the \printf expression is commented out, then the function can be used to compute a very close approximation of \( e \) without a lot of excess printing, as demonstrated below:

\[
\text{> (approx-e-acc 20 0 1 0)} \\
2.7182818284590455
\]

14.4 Wrapper Functions

One annoying characteristic of accumulator-based functions is that the accumulators need to be given appropriate initial values to ensure the desired results. Fortunately, this problem is easily overcome by providing wrapper functions. In this setting, a wrapper function is designed to properly initialize any accumulators so that the user of an accumulator-based function need not remember the appropriate values. This section gives wrapper functions for some of the accumulator-based functions seen earlier.

Example 14.4.1: A wrapper for \texttt{facty-acc}

The following defines a wrapper function, \texttt{facty-wr}, for the accumulator-based function, \texttt{facty-acc}, defined earlier. Notice that the wrapper function simply calls \texttt{facty-acc} with the accumulator appropriate initialized to 1. It is called a wrapper function because it hides the use of the accumulator-based helper function, and because the wrapper function doesn’t do that much: it lets the helper function do the heavy lifting!

\[
\begin{array}{l}
\text{;; FACTY-WR} \\
\text{;; ---------------------------------------} \\
\text{;; INPUT: N, a non-negative integer} \\
\text{;; OUTPUT: The factorial of N (i.e., N!)} \\
\end{array}
\]

\begin{verbatim}
(define facty-wr
  (lambda (n)
    ;; Call the accumulator-based helper function with ACC=1
    (facty-acc n 1)))
\end{verbatim}

The following Interactions Window session demonstrates how the wrapper function shields the user from the accumulator. In fact, the user of \texttt{facty-wr} may not even be aware that an accumulator is being used at all.

\[
\begin{array}{l}
\text{> (facty-wr 3)} \\
6 \\
\text{> (facty-wr 4)} \\
24 \\
\text{> (facty-wr 5)} \\
120 \\
\end{array}
\]
Example 14.4.2: A wrapper for \texttt{approx-pi-acc}

The function, \texttt{approx-pi-acc}, from Example 14.3.5, uses several inputs to keep track of relevant parts of the computation over the course of all of the recursive function calls. The following wrapper function, \texttt{approx-pi}, shields the user from having to know the appropriate initial values for these additional inputs:

\begin{verbatim}
(define approx-pi-wr
  (lambda (n)
    (approx-pi-acc 1 1 n 0)))
\end{verbatim}

Here are some examples of its use:

\begin{verbatim}
> (approx-pi-wr 5)
3.466666666666667
> (approx-pi-wr 10001)
3.1417926135957908
\end{verbatim}

Example 14.4.3: A wrapper for \texttt{approx-e-acc}

The function, \texttt{approx-e-acc}, from Example 14.3.6, involved several accumulators. The following wrapper function, \texttt{approx-e-wr}, shields the user from having to know the appropriate initial values for these accumulators:

\begin{verbatim}
(define approx-e-wr
  (lambda (n)
    (approx-e-acc n 0 1 0)))
\end{verbatim}

Here's what it looks like in the Interactions Window:

\begin{verbatim}
> (approx-e-wr 4)
2.7083333333333333
> (approx-e-wr 5)
2.7166666666666663
> (approx-e-wr 6)
2.7180555555555554
> (approx-e-wr 100)
2.7182818284590455
\end{verbatim}

Notice that the user of \texttt{approx-e-wr} may not even be aware that accumulators are being used!
Example 14.4.4: A wrapper function for input validation

The facty-v1 function defined in Example 12.1.2 presumes that its input will be a positive integer. If it is applied to any other kind of input, bad things can happen. For example, if it is applied to a negative number, the facty-v1 function will go into an infinite loop, each recursive call moving further away from the base case. And if it is applied to a non-numeric input, it will generate an error (e.g., because the built-in = function cannot be applied to non-numeric input). To avoid these sorts of problems, we can provide a wrapper function for facty-v1 that checks whether the input is valid before applying facty-v1 to it. Here is its contract and definition, followed by some examples of its use. (The wrapper function makes use of the built-in integer? function, seen previously in Section 5.3, whose output is #t if and only if its input is an integer.)

;; FACTY-V1.WRAPPER
;; -------------------------------------------------------
;; INPUT: DATUM, anything
;; OUTPUT: If DATUM is a positive integer, the output
;; is the factorial of DATUM; otherwise, the
;; output is void.
;; SIDE EFFECT: If DATUM is not a positive integer, it
;; prints out an error message

(define facty-v1-wrapper
  (lambda (n)
    (cond
      ;; Good case: N is a positive integer
      ((and (integer? n)
            (> n 0))
        (facty-v1 n))
      ;; Bad case: N is something else
      (else
        (printf "ERROR: Input must be a positive integer!\n")))))

> (facty-v1-wrapper 5)
120
> (facty-v1-wrapper -3)
ERROR: Input must be a positive integer!
> (facty-v1-wrapper 4.32)
ERROR: Input must be a positive integer!
> (facty-v1-wrapper 'xyz)
ERROR: Input must be a positive integer!

Although the process of input validation could be taken care of in the facty-v1 function itself, that would not be a good idea because it would occur on every recursive function call! It is much better to do the input validation once, in the wrapper function.

* In general, a wrapper function is a function that takes care of basic, one-time tasks, while letting some other function do most of the work.

Thus, a wrapper function can shield a user from having to know the appropriate initial values of accumulator inputs, or it could take care of input validation, or it could print out some useful information, or ... There are lots of things that a wrapper function could do!
14.5 Summary

A *recursive* function is any function $f$ whose body contains an expression that involves a call to $f$. The body of a recursive function also typically contains a conditional expression that distinguishes one or more *base cases* from one or more *recursive cases*. Evaluating a recursive function call typically involves evaluating a chain of recursive function calls that eventually terminate in a base case. To avoid circularity, the recursive cases typically involve applying $f$ to different inputs. For example, consider the `facty` function:

```scheme
(define facty
  (lambda (n)
    (cond
      ;; Base Case: N <= 1
      ((<= n 1) 1)
      ;; Recursive Case: N > 1
      (else (* n (facty (- n 1)))))))
```

The `cond` special form is used to distinguish the base case from the recursive case. The recursive case involves applying `facty` not to $n$, but to $(- n 1)$. As a result, the chain of recursive function calls will eventually involve applying `facty` to 1, at which point the recursion stops.

The above function `facty` is not *tail recursive* since the recursive function call, $(\text{facty } (- n 1))$, is embedded within a larger expression, $(\ast n (\text{facty } (- n 1)))$. The evaluation of the larger expression is suspended while waiting for $(\text{facty } (- n 1))$ to be evaluated. After $(\text{facty } (- n 1))$ is evaluated, the evaluation of the larger expression can be completed. For this reason, the *function-call boxes* for all of the recursive function calls must be maintained in the computer’s memory simultaneously until the last one completes.

In general, non-tail-recursive functions can require a large amount of memory.

Recursive solutions to computational problems often become apparent when considering concrete examples. For example, if we seek a function $g(n)$ that computes the sum of the squares from 1 to $n$, inclusive, it is not hard to see that $g(5) = g(4) + 5^2$, as demonstrated below.

$$
g(5) = 1^2 + 2^2 + 3^2 + 4^2 + 5^2
= (1^2 + 2^2 + 3^2 + 4^2) + 5^2
= g(4) + 5^2
$$

In turn, this suggests that $g(n) = g(n-1) + n^2$ for each $n > 1$, which leads to the following solution in Scheme:

```scheme
(define sum-squares
  (lambda (n)
    (cond
      ;; Base Case: N <= 1
      ((<= n 1) 1)
      ;; Recursive Case: N > 1
      (else (+ (sum-squares (- n 1)) (* n n))))))
```

A *tail-recursive* function call is a recursive function call whose evaluation, if it is needed, is necessarily the last (i.e., tail) step in the evaluation of the body of the parent function. For example, the following function is tail recursive.

```scheme
(define print-n-dashes
  (lambda (n))
(cond
  ;; Base Case: N <= 0
  ((< n 0)
    (newline))
  ;; Recursive Case: N > 0
  (else
    (printf "-")
    (print-n-dashes (- n 1)))))

Notice that, if the recursive case is followed, the last expression in that case, (print-n-dashes (- n 1)), will generate the output value for this function—without any subsequent computation. In general, when DrScheme encounters a tail-recursive function call, the function-call box for the original function call is no longer needed. Therefore, it can be recycled, to be used for the recursive function call. As a result, instead of using a large number of function-call boxes for a chain of recursive function calls, DrScheme can use just one function-call box over and over again. This can result in a tremendous reduction in memory usage, which makes defining tail-recursive functions well worth the effort.

Because tail-recursive function calls must be the last expression to be evaluated, the output value obtained by a tail-recursive function call cannot be subject to further computation (e.g., given as input to some other function). Therefore, computations in tail-recursive functions are typically organized a bit differently—in most cases, by computing the inputs that are fed into the recursive function call, as illustrated below.

(define facty-acc
  (lambda (n acc)
    (cond
      ;; Base Case: N <= 1
      ((< n 1)
        acc)
      ;; Recursive Case: N > 1
      (else
        (facty-acc (- n 1) (* n acc))))))

Instead of taking the answer returned by the recursive function call and multiplying it by n, this solution uses an extra input, called an accumulator, to accumulate the desired answer. The main computations involve determining the values to be fed to the recursive function call—in this case, (- n 1) and (* n acc). In the base case, the accumulator is returned as the output value, since it has, by that time, accumulated the desired answer.

Because tail-recursive functions often require extra inputs (e.g., accumulators), it is frequently desirable to provide wrapper functions that take care of the annoying job of giving appropriate values to the extra inputs. For example, a wrapper function for the facty-acc function would take care of calling facty-acc with an initial value of 1 for acc.

### Built-in Functions Introduced in this Chapter

- **even?:** Returns #t if its input is an even number
- **odd?:** Returns #t if its input is an odd number
- **sin:** Returns the sine of its input
- **log:** Returns the natural logarithm of its input
Chapter 15

Local Variables, Local Environments

Recall that, in Scheme, every expression is evaluated with respect to some environment. Up to this point, most of the expressions we have encountered have been evaluated with respect to the Global Environment. For example, expressions entered into the Interactions Window are evaluated with respect to the Global Environment, as are expressions from the Definitions Window when the Run button is pressed. Now, when evaluating a symbol with respect to the Global Environment, there is only one place to look for that symbol’s value: in the Global Environment. However, we have also seen that when DrScheme applies a lambda function to one or more inputs, a local environment is automatically created within a function-call box to house variable/value entries for the function’s input parameters. Assuming that the evaluation of the lambda expression that created the lambda function was done with respect to the Global Environment, the new local environment is considered to be nested inside the Global Environment. During the application of the function to inputs, each expression in the body of that lambda function is evaluated with respect to the local environment. As a result, whenever any symbol s needs to be evaluated, DrScheme looks first in the local environment. If it contains an entry for s, then the corresponding value is used as the value for s; otherwise, DrScheme looks in the Global Environment.

This chapter introduces the let special form—along with its more general variants, let* and letrec. The purpose of a let special form is to create a new local environment that is populated with local variables, just like the local environments that exist within function-call boxes. When a let special form is evaluated with respect to some parent environment $E$, the new local environment $E'$ that it creates is nested inside the parent environment $E$ (i.e., $E' \subseteq E$). Each of the expressions in the body of the let special form is evaluated with respect to that new local environment $E'$. As a result, when any symbol $s$ needs to be evaluated, DrScheme gives priority to the new environment $E'$. The result of evaluating the entire let special form is simply the value of the last expression in its body. Once a let expression has been evaluated, its local environment typically vanishes.\(^1\)

The other special forms introduced in this chapter are variants of let that have extra capabilities. The let* special form can do everything that a let can do, plus a little bit more; and a letrec special form can do everything that a let* can do, plus a little bit more. Thus, the let special form is the most basic of the three.

15.1 The let Special Form

The purpose of the let special form is to set up a local environment populated with local variables that provides a temporary context for the evaluation of the expressions in the body of the let. A let special form is often used to store the value of some lengthy computation in a local variable, after which that value can be accessed as many times as needed without having to re-do the lengthy computation over and over again. For example, suppose it takes a year to compute some desired numerical value. You wouldn’t want to have to re-do that year-long computation each time you wanted to print out that value. It would be much more efficient to store the computed value in a local variable and then refer to that stored value as often as desired. Furthermore, it is not desirable to overpopulate the Global Environment with values that may only be needed for a brief time. It is preferable to create local variables to store values for only as long as they are needed.

\(^1\)There are some exceptions whereby a local environment can outlast the evaluation of the body, but a discussion of these exceptions would take us too far afield.
15.1.1 The Syntax of the let Special Form

The syntax of the let special form is as follows:

```
(let ((var₁ val₁)
       (var₂ val₂)
       ...
       (varₙ valₙ))
  expr₁
  expr₂
  ...
  exprₖ)
```

where:
- \( var₁, \ldots, varₙ \) are character sequences representing \( n \) distinct Scheme symbols, where \( n \geq 0 \);
- \( val₁, \ldots, valₙ \) are \( n \) Scheme expressions of any kind; and
- \( expr₁, \ldots, exprₖ \) are \( k \) Scheme expressions of any kind, where \( k \geq 1 \).

The expressions, \( expr₁, \ldots, exprₖ \), constitute the body of the let expression.

* Notice that a let can include zero or more \( var/val \) pairs; however, the body of a let must include at least one expression.

<table>
<thead>
<tr>
<th>Example 15.1.1: Some legal let expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following expressions are all legal let expressions:</td>
</tr>
<tr>
<td>(let () #t)</td>
</tr>
</tbody>
</table>
| (let ((x (+ 2 3)))
  (* x x)) |
| (let ((x (+ 2 3))
  (y 3)
  (z (* 2 z)))
  (printf "x: \"A, y: \"A, z: \"A\"%" x y z)
  (+ x y z)) |

* The first let expression includes no \( var/val \) pairs, as indicated by the empty list. Its body consists of the single expression, \#t. The second let expression includes a single \( var/val \) pair: \((x (+ 2 3))\). Its body consists of the single expression, \((* x x)\). The third let expression includes three \( var/val \) pairs: \((x (+ 2 3)), (y 3)\) and \((z (* 2 z))\). Its body consists of two expressions: a printf expression and \((+ x y z)\). |

15.1.2 The Semantics of the let Special Form

Like any special form expression, a let special form expression denotes a list. The more interesting part of the semantics of a let special form is how it is evaluated. When a let expression of the form

```
(let ((var₁ val₁)
       (var₂ val₂)
       ...
       (varₙ valₙ))
```

is evaluated with respect to some environment $\mathcal{E}$, the following steps are taken.

- First, the expressions, $val_1, \ldots, val_n$, are evaluated with respect to the environment $\mathcal{E}$.
- Second, a local environment $\mathcal{E}'$ is created containing $n$ entries—one for each of the $var/val$ pairs in the $let$ expression. In particular, each symbol $var_i$ is associated with the result of evaluating the corresponding $val_i$ expression. This new environment is nested inside $\mathcal{E}$. In other words: $\mathcal{E}' \subseteq \mathcal{E}$.
- Third, the expressions, $expr_1, \ldots, expr_k$, in the body of the $let$ special form are evaluated, in turn, with respect to the newly created local environment, $\mathcal{E}'$. Therefore, in the process of evaluating these expressions, if any symbol $var_i$ ever needs to be evaluated, its value is drawn from the new environment $\mathcal{E}'$. Any other symbols are evaluated with respect to the parent environment $\mathcal{E}$.
- The value of the last expression, $expr_k$, is the value of the entire $let$ expression.

**Example 15.1.2: Evaluating $let$ expressions**

The following Interactions Window session demonstrates the evaluation of the sample $let$ expressions seen earlier.

```scheme
> (let () #t)
#t
> (let ((x (+ 2 3)))
  (* x x))
25
> (let ((x (+ 2 3))
  (y 3)
  (z (* 2 z)))
  (printf "x: ~A, y: ~A, z: ~A\%" x y z)
  (+ x y z))
  x: 5, y: 3, z: 4
x: 5, y: 3, z: 4
12
```

In the first expression, the local environment contains no entries. Thus, when the body of the $let$ is evaluated, the result is the same as if it were evaluated outside the $let$. In particular, the expression, $#t$, evaluates to $#t$, which is reported as the value of the entire $let$ expression. Since the purpose of a $let$ expression is to set up a local environment, it is rare to see a $let$ expression that contains no $var/val$ pairs.

In the second expression, the local environment contains a single entry that associates the value 5 with the symbol $x$. Notice the plethora of parentheses required to represent a list containing a single entry that is itself a list! Furthermore, the second entry in that subsidiary list is itself a list! The body of the $let$ consists of the single expression, $(* x x)$, which evaluates to 25 in this context. Notice that 25 is reported as the value of the entire $let$ expression.

In the third expression, the local environment contains three entries: one associating the value 5 with $x$, one associating the value 3 with $y$, and one associating the value 4 with $z$. The body contains two expressions. The $printf$ expression causes information to be displayed in the Interactions Window; the expression $(+ x y z)$ is then evaluated, resulting in the value 12, which is reported as the value for the entire $let$ expression.
Example 15.1.3: Local vs. Global

The following Interactions Window session demonstrates that local environment entries are given priority over Global Environment entries when evaluating expressions in the body of a let special form.

```
> (define x 1000)
> (define y 100)
> (define z 10)
> (+ x y z)
1110
> (let ((x 3)
         (y 4))
    (+ x y z))
17
```

The first three expressions use the define special form to create three global variables, named x, y and z. The last expression uses a let to create a local environment containing two local variables, named x and y. When the single expression in the body of the let is evaluated, the values for x and y are drawn from the local environment, whereas the values for + and z are drawn from the Global Environment. The entries for x and y in the Global Environment play no role in the evaluation of the expression (+ x y z) in the body of the this let expression, as illustrated in Fig. 15.1.
15.2 Flipping Coins and Tossing Dice

The following example introduces a destructive built-in function called `random` that has many uses, one of which is to demonstrate the need for the `let` special form. I know... this part of the book is supposed to only deal with non-destructive functions. But, this one exception is too much fun to postpone any further.

**Example 15.2.1: The built-in random function**

Scheme includes a built-in function called `random` that can be used to generate pseudo-random numbers. Unlike all of the functions that we have seen so far in this book, the `random` function has the unusual property that successive applications of it to the same input can generate different output values! This can happen because the computations it performs to generate its output depend on the values of secret global variables that it destructively modifies. Yep, it’s a destructive function! Despite being destructive, it is introduced here for three reasons: (1) it is fun; (2) it can be quite useful when programming games; and (3) it provides a nice demonstration of the need for the `let` special form (cf. Example 15.2.3, below).a

The `random` function satisfies the following contract.

```scheme
;; RANDOM
;; -------------------------------
;; INPUT: N, a positive integer
;; OUTPUT: A pseudo-random number drawn from the set
;;         {0, 1, 2, ..., N-1}
;; SIDE EFFECTS: Destructively modifies some secret global
;;               variables that enable it to (possibly) generate a different
;;               output the next time it is called---even if it is called
;;               with the same input!
```

Here are some examples demonstrating its behavior:

```scheme
> (random 2)    ← output will be 0 or 1
  0
> (random 2)    ← output will be in \{0, 1, 2, 3, 4, 5\}
  1
> (random 2)    0
> (random 6)    4
> (random 6)    3
> (random 6)    5
```

In general, when called with an input \(n\), the `random` function returns one of the \(n\) numbers in the set \(\{0, 1, 2, \ldots, n-1\}\).

---

*aThere’s an entire field of Computer Science that deals with so-called randomized algorithms (i.e., algorithms whose computations depend on pseudo-random generators). Randomized algorithms can often be surprisingly efficient.*

---

**Example 15.2.2: Flipping coins and tossing dice**

When the `random` function is called with 2 as its input, the output is one of two possible values: 0 or 1. And when called with 6 as its input, the output is one of six possible values: 0, 1, 2, 3, 4 or 5. Thus, the `random` function can be used to simulate the flipping of a coin or the tossing of a six-sided die, as
demonstrated by the flip-coin and toss-die functions, defined below.

;; FLIP-COIN
;; ------------------------------------------------------
;; INPUTS: None
;; OUTPUT: A symbol, either H or T, chosen randomly

(define flip-coin
  (lambda ()
    (if (= (random 2) 0)
      'H
      'T)))

;; TOSS-DIE
;; ----------------------------------------------------------
;; INPUTS: None
;; OUTPUT: A randomly chosen number, one of: 1, 2, 3, 4, 5, 6

(define toss-die
  (lambda ()
    ;; Since (RANDOM 6) generates a number in {0, 1, 2, 3, 4, 5},
    ;; we must add one to generate a number in {1, 2, 3, 4, 5, 6}.
    (+ 1 (random 6))))

Here are some examples of their use:

> (flip-coin)
H
> (flip-coin)
T
> (flip-coin)
H
> (toss-die)
3
> (toss-die)
1
> (toss-die)
6

* One of the most reliable features of non-destructive programming is that no matter how many times you apply a given function \( f \) to the same inputs, you will always get the same output. In other words, non-destructive functions are truly functions, in the mathematical sense. Such functions are sometimes called pure functions. In contrast, a function such as random, which has the potential to generate a different output every time it is called on the same input, is sometimes called an impure function.

* The preceding example demonstrates that a function such as toss-die, which makes use of an impure function such as random, can itself become impure. In other words, the impurity of random can infect the otherwise pure function that calls it.

* Because impure functions can be difficult to debug (i.e., find errors and fix them), introducing impure functions into a program should be done with great care! A good rule of thumb is: Do as much as you can with pure (non-destructive) functions; only introduce impure (destructive) functions when they are absolutely necessary—or, as in this chapter, when they are fun!
Example 15.2.3: Using let to store a randomly generated value

The `toss-die` function is fine, but suppose that you toss a die and want to do several things with the result (e.g., print out the value, print out the square of the value, and so on). The following attempt does not work:

```scheme
> (printf "My toss: \"A\"%" (toss-die))
3
> (printf "The square of my toss: \"A\"%" (* (toss-die) (toss-die)))
10
```

Why? Because each time DrScheme evaluates `(toss-die)`, it may generate a different value. To get the desired behavior, you need some way of storing the value of a single toss, so that you may then refer to it as often as you like. In short, you need a `let` special form, as illustrated below:

```scheme
> (let ((toss (toss-die)))
  (printf "My toss: \"A\"%" toss)
  (printf "The square of my toss: \"A\"%" (* toss toss))
My toss: 4
The square of my toss: 16
64
> toss
ERROR: reference to undefined identifier: toss
```

In this example, the `let` special form creates a local variable named `toss` whose value is the result of randomly tossing a six-sided die. The expressions in the body of the `let` can then refer to `toss`—and thereby gain access to that stored value—as many times as needed. However, the local environment only exists while the `let` special form is being evaluated. Once the evaluation of the `let` is completed, its local environment evaporates. It is for this reason that any later attempt to evaluate `toss` will cause DrScheme to report an error, as shown above. (This example assumes that there is no entry for `toss` in the Global Environment.)

15.3 Nested let Expressions and Nested Environments

When a `let` special form is evaluated with respect to the Global Environment, it creates a local environment, $E_1$, that is nested inside the Global Environment. For convenience, we can represent this by writing $E_1 \subset E_0$. Any expression in the body of that `let` may refer to any variable in that local environment $E_1$, as well as any variable in the Global Environment $E_0$, with the proviso that the local environment has priority. For this reason, the following `let` expression evaluates to 25 (i.e., 5 times 5) because the value of the symbol $x$ is fetched from the local environment, not the Global Environment.

```scheme
> (define x 100)
> (let ((x 5))
  (* x x))
25
```

Note that the value of the `asterisk` symbol is fetched from the Global Environment, because there is no entry in the local environment for that symbol. Note, too, that there is no way that an expression in the body of this `let` could refer to the global variable $x$, because the existence of the local variable $x$ effectively blocks access to the globally defined $x$.

Continuing in this way, a `let` nested inside another `let` (i.e., a `let` expression that appears in the body of another `let`) creates a new local environment, $E_2$, where $E_2 \subset E_1 \subset E_0$. Thus, any expression in the body of
that \textit{let} is evaluated with respect to the environment $E_2$, which implies that $E_2$ has the highest priority, $E_1$ has the next highest priority and, as always, the Global Environment $E_0$ has the lowest priority. The following example demonstrates that this is the case.

\begin{example}
\begin{verbatim}
> (define x 100)
> (let ((x 5))
  (let ((x (* x x)))
    (printf "x: ˜A˜%" x))
  (printf "x: ˜A˜%" x))
  x: 25
  x: 5
> x
100
\end{verbatim}
\end{example}

The \textit{let} special form creates a local variable $x$, in the environment $E_1$, whose value is 5. In the body of that \textit{let}, the next \textit{let} creates a different local variable, in the environment $E_2$, that also happens to be called $x$. In the environment $E_2$, the value of $x$ is 25 (i.e., 5 times 5). Note that its value is computed before the creation of the environment $E_2$; its value is computed with respect to the environment $E_1$. The first \texttt{printf} expression is evaluated with respect to the innermost environment $E_2$, where $x$ has the value 25. Since there is no entry for the \texttt{printf} symbol in $E_2$ or $E_1$, its value is obtained from the Global Environment $E_0$. The second \texttt{printf} expression is evaluated with respect to the environment $E_1$, where $x$ has the value 5.

It is important to point out that since the environment $E_2$ defines a local variable named $x$, then any expression evaluated with respect to that environment cannot access any other variable named $x$ that might exist in any of the parent environments (e.g., the variable named $x$ in the environment $E_1$, or the variable named $x$ in the Global Environment). In the same way, if some local environment has a variable named list, then any expression being evaluated with respect to that local environment cannot access the built-in \texttt{list} function, because the local variable named list would have priority over the globally defined \texttt{list} function.

In general, a \textit{let} expression that is evaluated with respect to some parent environment $E$ creates a new local environment $E'$ that is nested inside $E$ (i.e., $E' \subseteq E$). To evaluate a symbol $s$ with respect to the new environment $E'$, involves the following \textit{recursive} process:

(Base Case) If there is an entry in the environment $E'$ that pairs $s$ with a value $v$, then $s$ evaluates to $v$ in $E'$.

(Recursive Case) Otherwise, the value for $s$ is obtained by evaluating $s$ in the parent environment $E$.

Note that this process is recursive because if the parent environment does not have an entry for $s$, then $s$ will have be be evaluated with respect to its parent environment, and so on, until, eventually, an ancestor environment is reached that has an entry for $s$. Note that if this process goes all the way to the Global Environment without finding any entry for $s$ in any environment along the way (including the Global Environment), then the evaluating $s$ in the environment $E'$ is undefined.

We can describe this process as follows. Since each environment other than the Global Environment is nested inside its parent environment, each environment $E_n$ determines a chain of ancestor environments of the form, $E_n \sqsubset E_{n-1} \sqsubset \ldots \sqsubset E_2 \sqsubset E_1 \sqsubset E_0$, where $E_0$ is the Global Environment. When a symbol is being evaluated with respect to the environment $E_n$, the environment $E_n$ has the highest priority and the Global Environment has the lowest priority. When evaluating a symbol $s$ in the environment $E_n$, the environments are checked, in order, from $E_n$ to $E_0$, until one is found that has an entry for $s$. The value for $s$ in that entry will be the result of evaluating $s$ in $E_n$.

The same considerations apply to the local environment that is automatically created when a \texttt{lambda} function is applied to inputs. The main thing to remember is:
* If a lambda function \( f \) was created by evaluating a lambda special form with respect to an environment \( \mathcal{E} \), then the local environment \( \mathcal{E}' \) that is automatically created when \( f \) is applied to inputs is nested inside \( \mathcal{E} \):
\[
\mathcal{E}' \subset \mathcal{E}.
\]
The following example illustrates that this is the case.

**Example 15.3.2**

\[
\begin{align*}
> &\quad \text{(let \((x 5)\))} \\
&\quad \quad \text{(let \((f \text{ (lambda \((n)\)\)\)}} \\
&\quad\quad\quad \text{(printf "Inside function body: x = \text{"A"\%" x)\)}} \\
&\quad\quad\quad \text{(* x n)))}) \\
&\quad \quad \text{(let \((x 10)\))} \\
&\quad \quad \text{(f 12))}) \\
\text{Inside function body: x = 5} \\
60
\end{align*}
\]

The first \text{let} creates a local environment \( \mathcal{E}_1 \) in which \( x \) has the value 5. The second \text{let} expression creates a local environment \( \mathcal{E}_2 \) in which a local variable \( f \) has a function as its value. However, according to the semantics for \text{let} expressions, recall that the value for \( f \) is obtained by evaluating the lambda special form with respect to the parent environment \( \mathcal{E}_1 \). Since \( \mathcal{E}_1 \) is the environment within which this lambda function was created, any time this function is applied to inputs, the expressions in its body will be evaluated in a local environment \( \mathcal{E}_f \) that is nested inside \( \mathcal{E}_1 \): \( \mathcal{E}_f \subset \mathcal{E}_1 \). For this reason, even though the next \text{let} expression creates a local environment \( \mathcal{E}_3 \) in which \( x \) has the value 10, the evaluation of the expression \((f 12)\) in the environment \( \mathcal{E}_3 \) leads to the application of the above lambda function to the input 12 which, in turn, leads to evaluating the body of that function with respect to the environment \( \mathcal{E}_f \subset \mathcal{E}_1 \), where \( x \) has the value 5.

### 15.4 Deriving the \textit{let} Special Form from the \textit{lambda} Special Form

If you’re thinking that the evaluation of a let special form seems awfully close to the evaluation of a function call, you’re right. In fact, each \text{let} special form expression is simply a convenient abbreviation for an expression in which a lambda function is applied to some input values. Before going into all the details, we give some examples illustrating the equivalence of expressions involving \text{let} and \text{lambda}.

**Example 15.4.1**

The following Interactions Window session shows the evaluation of a \text{let} expression, followed by the evaluation of an equivalent expression involving the application of a lambda function to some inputs.

\[
\begin{align*}
> &\quad \text{(let \((x (+ 2 3))\))} \\
&\quad \quad \text{(y (* 3 4)))} \\
&\quad \quad \text{(printf "x: \text{"A"\%" x)\)}} \\
&\quad\quad\quad \text{(* x y))} \\
&\quad x: 5, \ y: 12 \\
17
\end{align*}
\]

\[
\begin{align*}
> &\quad \text{((lambda \((x y)\)\)}} \\
&\quad \quad \text{(printf "x: \text{"A"\%" x)\)}} \\
&\quad\quad\quad \text{(* x y))} \\
&\quad \quad \text{(+ 2 3)} \\
&\quad \quad \text{(* 3 4))} \\
&\quad x: 5, \ y: 12
\end{align*}
\]
The semantics for the evaluation of the first expression is identical to the semantics for the evaluation of the second expression!

In particular, for the \texttt{let} expression, a local environment is set up in which the symbol \texttt{x} is associated with the value \texttt{5} and the symbol \texttt{y} is associated with the value \texttt{12}. After that, the two expressions in the body of the \texttt{let} are evaluated with respect to that local environment yielding some side-effect printing and an output value of \texttt{17}.

The evaluation of the second expression is governed by the Default Rule for evaluating non-empty lists. The first entry in the list is a \texttt{lambda} expression. It evaluates to a function. The other entries, \texttt{(+ 2 3)} and \texttt{(* 3 4)}, evaluate to the numbers \texttt{5} and \texttt{12}, respectively. When that function is applied to those inputs, a local environment is set up in which \texttt{x} and \texttt{y} are associated with the values \texttt{5} and \texttt{12}, respectively. Then the body of the \texttt{lambda} is evaluated, yielding side-effect printing and the output value \texttt{17}.

\begin{example}

The following Interactions Window session first creates a global variable, \texttt{z}. It then evaluates a \texttt{let} expression and an equivalent expression involving the application of a \texttt{lambda} function.

\begin{verbatim}
> (define z 1000)
> (let ((x 3)
       (y 4))
    (* x y z))
12000
> ((lambda (x y)
           (* x y z))
    3 4)
12000
\end{verbatim}

Once again, the evaluation of the two expressions is the same. In particular, each involves a local environment containing entries for \texttt{x} and \texttt{y}, with the respective values \texttt{3} and \texttt{4}. In addition, each involves the evaluation of the expression \texttt{(* x y z)} with respect to that local environment. Notice that in each case, the values for \texttt{x} and \texttt{y} are drawn from the local environment, whereas the value for \texttt{z} is drawn from the Global Environment. In each case, the value of the entire expression is \texttt{12000}.

\end{example}

In general, a \texttt{let} expression of the form,

\begin{verbatim}
(let ((var\_1 val\_1)
     (var\_2 val\_2)
     ...
     (var\_n val\_n))
expr\_1
expr\_2
...
expr\_k)
\end{verbatim}

is equivalent to the following expression involving the application of a \texttt{lambda} function:

\begin{verbatim}
((lambda (var\_1...var\_n)
        expr\_1
        expr\_2
        ...
        expr\_k)
\end{verbatim}
... 
\[ \begin{align*} 
& \text{expr}_k) \\
& \text{val}_1 \ldots \text{val}_n) 
\end{align*} \]

You should convince yourself that the local environments that are created in response to evaluating these two expressions are equivalent.

* The reason we have let expressions is that they have a friendlier syntax for the cases where you want to create a local environment and then evaluate some expressions with respect to that local environment.

15.5 The let* Special Form

The syntax of the let* special form is nearly identical to that of the let special form. (The only difference is the presence of the * in let*.) However, the semantics is substantially different. In particular, the local environment is populated incrementally, as each var/val pair is processed. This difference allows a certain kind of incremental computation that turns out to be quite useful. When a let special form is evaluated, each \( \text{val}_i \) is evaluated with respect to the parent environment and, thus, none of the \( \text{val}_i \) expressions can depend on any of the variables in the nascent local environment. In contrast, when a let* special form is evaluated, each \( \text{val}_i \) is evaluated with respect to the portion of the local environment that has been created so far. As a result, the expression \( \text{val}_i \) may depend on the values of the local variables \( \text{var}_1, \ldots, \text{var}_{i-1} \) that precede it in the let* expression.

15.5.1 The Syntax of the let* Special Form

Each let* expression has the following form:

\[
(\text{let* } ((\text{var}_1 \text{ val}_1) \\
& (\text{var}_2 \text{ val}_2) \\
& \ldots \\
& (\text{var}_n \text{ val}_n)) \\
\text{expr}_1 \\
\text{expr}_2 \\
\ldots \\
\text{expr}_k)
\]

You’ll notice that the only difference is the asterisk in the name of the special form: let* instead of let.

15.5.2 The Semantics of the let* Special Form

A let* special form is evaluated as follows:

- An empty local environment is created.
- Each var/val pair is processed, in turn. In particular, an entry is created in the local environment that associates the value of \( \text{val}_i \) with the symbol \( \text{var}_i \).

\[ \Rightarrow \text{Crucially, the } i^{\text{th}} \text{ entry in the local environment is created before the } (i + 1)^{\text{st}} \text{ value is computed. Thus, the expression, } \text{val}_{i+1}, \text{ can refer to any of the preceding symbols, } \text{var}_1, \ldots, \text{var}_i. \]

- Then the expressions in the body of the let* are evaluated, in turn.
- The value of the last expression in the body of the let* serves as the value of the entire let* expression.
**Example 15.5.1**

The following Interactions Window session demonstrates the kind of incremental computation that is characteristic of a `let*` special form, but that is not possible with a (single) `let` special form:

```scheme
> (let* ((x 4)
         (y (+ x 2))
         (z (* x y))
         (w (+ x y z)))
  (printf "x: ~A, y: ~A, z: ~A, w: ~A~%" x y z w)
(x: 4, y: 6, z: 24, w: 34
68

Notice that the expression, `(+ x 2)`, that is used to compute the value for `y` refers to the local variable `x`. Similarly, the expression, `(* x y)`, that is used to compute the value for `z` refers to both `x` and `y`. Finally, the expression, `( + x y z)`, that is used to compute the value for `w` refers to `x`, `y` and `z`. Trying to do this with a `let` expression causes DrScheme to complain.

```scheme
> (let ((x 4)
         (y (+ x 2))
         (z (* x y))
         (w (+ x y z)))
  (printf "x: ~A, y: ~A, z: ~A, w: ~A~%" x y z w)
... reference to undefined identifier: x
```

The reason is due to the difference in the way `let` and `let*` expressions are evaluated (i.e., their semantics). In a `let` expression, all of the value expressions are evaluated first, before any entries are created in the local environment. Thus, none of the value expressions in a `let` can refer to any of the local variables being defined. In contrast, in a `let*` expression, the evaluation of the value expressions is interleaved with the creation of the entries in the local environment. Thus, each value expression can refer to symbols that precede it in the `let*` expression.

### 15.5.3 Deriving a Single `let*` Expression from Nested `let` Expressions

In general, a `let*` expression of the form,

```scheme
(let* ((var$_1$ val$_1$)
       (var$_2$ val$_2$)
       ...
       (var$_n$ val$_n$))
  expr$_1$
  expr$_2$
  ...
  expr$_k$)
```

is equivalent to `n` nested `let` expressions:

```scheme
(let ((var$_1$ val$_1$))
  (let ((var$_2$ val$_2$))
    ...
    (let ((var$_n$ val$_n$))
      expr$_1$
```
\[
\begin{align*}
& \text{expr}_2 \\
& \ldots \\
& \text{expr}_k \\
& \text{)))}
\end{align*}
\]

So, a \texttt{let*} expression with \( n \) variable/value pairs effectively creates a sequence of \( n \) new local environments, where each new environment is nested inside its predecessor.

The following example demonstrates the equivalence.

### Example 15.5.2

The following Interactions Window session evaluates a \texttt{let*} expression and the equivalent nested \texttt{let} expression:

```
> (let* ((x 4)
    (y (+ x 2))
    (z (* x y))
    (w (+ x y z)))
  (printf "x: \^A, y: \^A, z: \^A, w: \^A\^%" x y z w)
  (+ x y z w))

x: 4, y: 6, z: 24, w: 34
```

```
> (let ((x 4))
  (let ((y (+ x 2)))
    (let ((z (* x y)))
      (let ((w (+ x y z)))
        (printf "x: \^A, y: \^A, z: \^A, w: \^A\^%" x y z w)
        (+ x y z w))))

x: 4, y: 6, z: 24, w: 34
```

Notice that the outermost \texttt{let} expression (i.e., the one that specifies the local variable \( x \)) has a body that consists of a single \texttt{let} expression (i.e., the one that specifies the local variable \( y \)). Because the \texttt{let} expression for \( y \) is evaluated with respect to the local environment containing an entry for \( x \), it is okay for the value expression, \((+ x 2)\), to refer to \( x \). Similar remarks apply to the remaining variables.

In general, \texttt{let*} provides a simpler syntax than the equivalent set of nested \texttt{let} expressions. Thus, if you ever need to do incremental computations where the value of each local variable depends of the values of the preceding local variables, then you should consider using \texttt{let*}.

### 15.6 The \texttt{letrec} Special Form

The \texttt{letrec} special form is provided to enable the specification of local recursive functions, something that cannot be done by \texttt{let} or \texttt{let*}. The specification of a local recursive function within a \texttt{letrec} special form is quite similar to the specification of a global recursive function within a \texttt{define} special form; however, the syntax of a \texttt{letrec} expression is much closer to that of \texttt{let} and \texttt{let*}. A common use of \texttt{letrec} is to embed an accumulator-based, tail-recursive helper function \textit{within} the body of its wrapper function. In this way, the existence of the helper function (and access to it) can be hidden from the general programming public. As usual, in such scenarios, the wrapper function takes care of supplying appropriate inputs to the helper function, freeing the user to think about other things.
15.6.1 The Syntax of the \texttt{letrec} Special Form

The syntax of the \texttt{letrec} special form is identical to that of the \texttt{let} and \texttt{let*} special forms, except that the keyword is \texttt{letrec} instead of \texttt{let} or \texttt{let*}.

15.6.2 The Semantics of the \texttt{letrec} Special Form

In sharp contrast to how the \texttt{let} and \texttt{let*} special forms are evaluated, the evaluation of a \texttt{letrec} special form begins by creating the entire local environment, complete with entries for all of the local variables, before evaluating any of the value expressions. Because none of the value expressions have yet been evaluated, each local variable is initially given the dummy value, \texttt{#<undefined>}. However, since all of the local variables have corresponding entries in the local environment before any of the value expressions are evaluated, each value expression can refer to any or all of the local variables, whether they have values or not!

\begin{example}
\textbf{Example 15.6.1}

The following interactions demonstrate that the \texttt{letrec} special form sets up its local environment before evaluating any of the value expressions. Because the \texttt{let} and \texttt{let*} special forms do not do this, the corresponding instances generate errors.

\begin{verbatim}
> (let ((x y)
       (y x))
   (printf "x:~,A, y:~,A~%" x y))
ERROR: reference to undefined identifier: y
> (let* ((x y)
       (y x))
   (printf "x:~,A, y:~,A~%" x y))
ERROR: reference to undefined identifier: y
> (letrec ((x y)
       (y x))
   (printf "x:~,A, y:~,A~%" x y))
x:#,y:#
\end{verbatim}

The preceding example is illustrative, but it ignores the primary purpose of the \texttt{letrec} special form: to create local recursive functions, similar to how the \texttt{define} special form can be used to create global recursive functions. For example, a \texttt{letrec} can be used to create a local variable \texttt{funky} whose value is a function whose body includes a recursive function call of the function named \texttt{funky}.

\begin{example}
\textbf{Example 15.6.2: Using \texttt{letrec} to create a local recursive function}

The following interactions demonstrate that \texttt{letrec} can be used to define a local recursive function, whereas \texttt{let} and \texttt{let*} cannot.

\begin{verbatim}
> (let ((factyOne (lambda (n)
                   (if (<= n 1)
                     1
                     (* n (factyOne (- n 1))))))
   (printf "No error up to this point, but ..."))
(factyOne 4))
No error up to this point, but ...
ERROR: reference to undefined identifier: factyOne
> (let* ((factyTwo (lambda (n)
\end{verbatim}

\end{example}
(if (<= n 1) 1 (* n (factyTwo (- n 1)))))

No error up to this point, but ...
ERROR: reference to undefined identifier: factyTwo
> (letrec ((factyThree (lambda (n)  
(if (<= n 1) 1  
;; No problems here! :)
(* n (factyThree (- n 1)))))
(factyThree 4)) 24

In the first example, the let expression creates a local environment $E_1$ that is nested inside the Global Environment. According to the semantics for a let, the value for its variable factyOne is evaluated with respect to the parent environment—in this case, the Global Environment. The result is a lambda function created with respect to the Global Environment. As the side-effect printing indicates, the evaluation of that lambda expression does not cause an error—because the expressions in the body are not evaluated when the function is created. However, attempting to apply the function to some numerical input requires evaluating the expressions in the function body—with respect to an automatically-created local environment $E_f$ that is nested inside the Global Environment (i.e., the environment within which the function was created). Because there is no entry for factyOne in the Global Environment, this leads to an error.

Similar remarks apply to the let* expression because a let* that includes only one variable/value pair is equivalent to a let. However, for the letrec expression, there are no problems. It creates a local environment $E_1$ that contains an entry for the variable factyThree, with a placeholder value of undefined, and then evaluates the value expression (i.e., the lambda expression) with respect to the environment $E_1$. Thus, the lambda function is created with respect to the environment $E_1$. Subsequently applying this function to a numerical input causes the body of the function to be evaluated with respect to the environment $E_1$, because that is the environment within which the function was created. Since $E_1$ contains an entry for factyThree, all is well.

Although this example is also illustrative, it seems kind of silly to create a function like factyThree to use it only once. The following example highlights a more common, useful way of using letrec.

### Example 15.6.3: Using letrec to create a local recursive function within a wrapper function

The following interactions demonstrate the use of the letrec special form to create a local recursive (helper) function within the body of a wrapper function. In this case, the wrapper function is facty, and the local recursive (helper) function is the accumulator-based, tail-recursive facty-acc function. Aside from defining facty-acc, the only thing that facty does is to call facty-acc with appropriate inputs.

> (define facty  
(lambda (n)  
;; Body of FACTY starts here  
(letrec ((facty-acc (lambda (m acc)  
(if (<= m 1) acc  
(facty-acc (- m 1) (* m acc))))))  
;; Body of FACTY-ACC starts here  
(if (<= m 1)  
acc  
(facty-acc (- m 1) (* m acc)))))))
;; Body of LETREC starts here
(facty-acc n 1)))
> (facty 4)
24
> (facty 5)
120

This kind of application of letrec is commonly used to hide the existence of a recursive helper function from users who may not understand what inputs to give it, or may not want to be bothered with thinking about what inputs to give it. The helper function only exists for use by the parent function; it is not visible to the general programming public. The parent function (facty) takes care of supplying the helper function (facty-acc) with appropriate inputs.

* Take care when defining local recursive helper functions. For example, note the difference between the input $n$ to facty and the input $m$ to facty-acc. On successive recursive function calls, $m$ takes on different values, while $n$ never changes.

In-Class Problem 15.6.1

Carefully draw a diagram that shows all of the relevant environments, and the variable/value pairs in those environments, for the evaluation of (facty 4) from the preceding example.

Special Forms Introduced in this Chapter

<table>
<thead>
<tr>
<th>Special Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>let</td>
<td>Create local environment</td>
</tr>
<tr>
<td>let*</td>
<td>Create local environment, supports incremental computations</td>
</tr>
<tr>
<td>letrec</td>
<td>Create local environment, supports recursive function definitions</td>
</tr>
</tbody>
</table>

Built-in Functions Introduced in this Chapter

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random</td>
<td>Pseudo-random number generator (an impure function)</td>
</tr>
</tbody>
</table>
Chapter 16

Lists and List-Based Recursion

Previous chapters have highlighted the many important roles that non-empty lists play in Scheme’s computational model. For example, the Default Rule for evaluating non-empty lists can be used to apply functions to inputs, the `define` special form can be used to assign values to variables, the `quote` special form can be used to shield a datum from evaluation, and so on. In contrast, this chapter focuses on lists as containers of data. When viewing lists as containers of data, we typically don’t want them to be evaluated. In addition, to do any meaningful computations involving lists (e.g., to sort a list of numbers or recursively walk through a list of data), we need to be able to access the individual elements. Finally, we will often want to be able to construct lists incrementally, for example, by attaching a new element to the front of a list.

Scheme provides the following built-in functions to facilitate the use of lists as containers of data:

- `first` to access the first element of a list
- `rest` to access the rest of a list
- `cons` to construct a new list by attaching a new element to the front of an existing list

These few functions, together with the `null?` type-checker predicate from Chapter 8, will enable us to design functions that can recursively process the elements in a list.

We shall see that list-based recursion is quite similar to numerical recursion. Whereas numerical recursion is driven by the size of a numerical input, list-based recursion is driven by some feature of a list—usually whether that list is empty or not. In list-based recursion, there is a base case—usually signaled by the empty list (analogous to \( n = 0 \)); and there is a recursive case—usually signaled by a non-empty list (analogous to \( n > 0 \)). And, just as a numerical-recursive function can typically process numerical inputs of any size, a list-based recursive function can typically process lists containing any number of elements.

16.1 The Built-in Functions: `first`, `rest` and `cons`

This section describes the built-in functions, `first`, `rest` and `cons`, that Scheme provides to enable us to access parts of lists, and to attach new elements to pre-existing lists.

The `first` and `rest` accessor functions. The `first` and `rest` functions are called accessor functions because they enable us to access certain parts of a non-empty list. The contracts for these built-in functions are given below.

```
;;  FIRST -- built-in function
;;  -----------------------------------------------
;;  INPUT:   LISTY, a non-empty list
;;  OUTPUT:  The FIRST element of LISTY
```
REST -- built-in function

INPUT: LISTY, a non-empty list
OUTPUT: The REST of LISTY (i.e., the portion of LISTY that contains all but its first element)

* Note that the rest of a non-empty list is necessarily a list.

---

**Example 16.1.1**

The following Interactions Window session demonstrates the use of the first and rest accessor functions to access the parts of a non-empty list.

```scheme
> (first '(a b c d e))
a
> (rest '(a b c d e))
b c d e
> (first '(64))
64
> (rest '(64))
() ← the rest is a list, even if it is empty
```

---

**Example 16.1.2: Accessing other elements of a non-empty list**

We can combine the first and rest functions to access any individual element of a list, as follows:

```scheme
> (first (rest '(a b c d e))) ← access second element
b
> (first (rest (rest '(a b c d e)))) ← access third element
c
> (first (rest (rest (rest '(a b c d e))))) ← access fourth element
d
```

Rather than re-typing these sorts of cumbersome expressions to access various elements of a list, we can define functions to simplify the process, as illustrated below:

```scheme
;; SEKUND/THURD/FORTH
;; --------------------------------
;; INPUT: LISTY, a list containing at least two elements
;; OUTPUT: The second/third/fourth element of LISTY

(define sekund
  (lambda (listy)
    (first (rest listy))))

(define thurd
  (lambda (listy)
    (first (rest (rest listy))))))

(define forth
  (lambda (listy)
    (first (rest (rest (rest listy)))))))
```
The following interactions demonstrate the use of these functions:

```scheme
> (sekund '(a b c d e))
b
> (thurd '(yes #t 383 () why))
383
> (forth '(my bonnie lies over the ocean))
over
```

Although we could continue in this fashion, defining additional accessor functions called fiffth, sicksth, and so on, we shall soon discover that there is a much easier way to access any desired element of a list: using recursion! In the meantime, you should know that Scheme provides a slew of built-in functions for accessing individual elements of a list in the manner seen above. They are called second, third, fourth, etc. As you may have guessed, the existence of these built-in functions is the reason that I gave names such as sekund, thurd and forth to the functions defined above.

In-Class Problem 16.1.1: Checking for a one-element list

Define a function, called one-elt-list?, that satisfies the following contract:

```scheme
;; ONE-ELT-LIST?
;; -----------------------------------------------
;; INPUTS: LISTY, any list
;; OUTPUT: #t if LISTY contains *exactly* one element;
;; #f otherwise.
```

Here are some examples of the desired behavior:

```scheme
> (one-elt-list? ())
#f
> (one-elt-list? '(xyz))
#t
> (one-elt-list? '(a b c d))
#f
```

**Hint:** Use some of these: null?, first, rest.

Using cons to construct a new list. The built-in cons function constructs a new list by attaching a new element onto the front of an existing list. Here is its contract:

```
;; CONS -- built-in function
;; -----------------------------------------------
;; INPUTS: FST, any Scheme datum
;; RST, a list (either empty or non-empty)
;; OUTPUT: A new list whose FIRST element is FST, and
;; the REST of whose elements are RST.
```

* When using the cons function to construct a new list, the second input must be a list!
Example 16.1.3

The following Interactions Window session demonstrates the use of the cons function.

> (cons 8 '(a b c))
(8 a b c)
> (cons 'john '(paul george ringo))
(john paul george ringo)
> (cons 64 ()) ← the second input must be a list, even if it is empty
(64)
> (define my-list '(a b c))
> (define new-list (cons 'x my-list))
> new-list
(x a b c)
> my-list
(a b c)

The last example shows that the cons function is non-destructive. The new list (x a b c) formed by attaching x to the front of my-list does not change my-list.

In-Class Problem 16.1.2: Using cons to create short lists

Define functions, called list-one and list-two, that satisfy the following contracts:

;;; LIST-ONE
;;; -----------------------------------------------
;;; INPUT:  DATUM, anything
;;; OUTPUT: A list that contains DATUM as its only element

;;; LIST-TWO
;;; -----------------------------------------------
;;; INPUTS:  ONE, TWO, anything
;;; OUTPUT: A list whose first element is ONE, and whose
;;; second element is TWO

Here are examples of the desired behavior:

> (list-one 'a)
(a)
> (define listy '(a b c))
> (define symby 'xyz)
> (list-one listy)
((a b c))
> 'listy ← quote produces different results!
listy
> (list-one symby)
(xyz)
> 'symby ← quote produces different results!
symby
> (list-two 'a 'b)
(a b)
> (list-two listy symby)
((a b c) xyz)
There is a built-in function, called list, that takes any number of inputs. It returns as its output a list containing those inputs, as illustrated below:

\[
\begin{align*}
> & \ (\text{listy symby}) \\
(\text{listy symby})
\end{align*}
\]

Hint: Use the built-in cons function.

16.2 List-based Recursion

Chapter 12 introduced recursive functions for which the recursion was driven by the size of a number. For example, in the factorial function (cf. Example 12.1.1), \(f(4)\) was computed by multiplying 4 by \(f(3)\), where \(f(3)\) was computed by multiplying 3 by \(f(2)\), where \(f(2)\) was computed by multiplying 2 by \(f(1)\), and where \(f(1) = 1\) terminated the recursion. The relevant sequence of computations is shown below:

\[
\begin{align*}
f(4) &= 4 \cdot f(3) \quad \text{Recursive call: } f(4) = 4 \cdot f(3) \\
&= 4 \cdot (3 \cdot f(2)) \quad \text{Recursive call: } f(3) = 3 \cdot f(2) \\
&= 4 \cdot (3 \cdot (2 \cdot f(1))) \quad \text{Recursive call: } f(2) = 2 \cdot f(1) \\
&= 4 \cdot (3 \cdot (2 \cdot 1)) \quad \text{Base case: } f(1) = 1 \\
&= 4 \cdot 3 \cdot 2 \quad 2 \cdot 1 = 2 \\
&= 4 \cdot 6 \quad 3 \cdot 2 = 6 \\
&= 24 \quad 4 \cdot 6 = 24
\end{align*}
\]

More generally, for any \(n > 1\), the factorial of \(n\) can be computed by making a sequence of \(n - 1\) recursive function calls, terminating in the base case, where \(f(1) = 1\). Of course, numerical recursion can take many forms. For example, the input \(n\) might start out at 0 and increase by 3 on each recursive function call until some stopping value (e.g., 90) is reached. Or the value of \(n\) might be multiplied by some value at each recursive function call. But the common feature is that deciding between the base case and the recursive case is based on the size of some number.

This section introduces list-based recursion. In list-based recursion the recursion is driven not by the size of a number, but by some feature of a list. In many cases, the relevant feature is simply whether a certain list is empty or not: if the list is empty, we’re in the base case; otherwise, we’re in the recursive case. For example, if a typical recursive function is applied to a list containing, say, five elements, then, because that list is non-empty, a recursive function call will be made on the rest of that list (i.e., a list containing four elements). And because that list is non-empty, another recursive function call will be made, this time on the rest of that list (i.e., a list containing three elements). The sequence of recursive function calls will eventually lead to the function being applied to the empty list, at which point the base case will terminate the recursion. This common kind of list-based recursion is explored in the following example.

Example 16.2.1

Suppose we are given the following contract for a function called mult-all:

\[
; ; \text{MULT-ALL}
\]
;;  -----------------------------------------------
;;  INPUT:   LISTY, a list of numbers
;;  OUTPUT:  The product of all the elements of LISTY

Here are some examples of the desired behavior:

> (mult-all '(2 3 4 10))
240
> (mult-all '(10 2 4))
80

This function can be defined recursively since:

\[
\text{(the product of all of the elements of a non-empty list)}
\]

\[
= \begin{cases} 
\text{(the first element of the list)} \\
\times \\
\text{(the product of the rest of the elements of the list)}
\end{cases}
\]

For example:

\[
\text{(the product of all of the elements of (2 3 4 10))}
\]

\[
= \begin{cases} 
2 \\
\times \\
\text{(the product of all of the elements of (3 4 10))}
\end{cases}
\]

Stated in terms of the mult-all function, where listy is a variable whose value is (2 3 4 10):

\[
\text{(mult-all listy)} \Rightarrow (\ast \text{ (first listy) (mult-all (rest listy))})
\]

Note that if this relationship is going to hold for all non-empty lists, then (mult-all ()) must evaluate to 1 (i.e., the multiplicative identity), as illustrated below:

\[
\text{(mult-all '(4))} \Rightarrow (\ast 4 \text{ (mult-all ()))} \Rightarrow (\ast 4 1) \Rightarrow 4
\]

In view of all of the above, we might imagine the evaluation of (mult-all '(2 3 4 10)) proceeding as follows, where, for example, the recursive function call on the rest of the list (2 3 4 10) is represented by (mult-all '(3 4 10)):

\[
\begin{align*}
\text{(mult-all '}(2 3 4 10)) & \quad \text{\textit{Recursive Case}} \\
\Rightarrow (\ast 2 \text{ (mult-all '}(3 4 10))) & \quad \text{\textit{Recursive Case}} \\
\Rightarrow (\ast 2 \text{ (ast 3 (mult-all '}(4 10))) & \quad \text{\textit{Recursive Case}} \\
\Rightarrow (\ast 2 \text{ (ast 3 (ast 4 (mult-all '}(10)))) & \quad \text{\textit{Recursive Case}} \\
\Rightarrow (\ast 2 \text{ (ast 3 (ast 4 (ast 10 (mult-all ())))))) & \quad \text{\textit{Base Case}} \\
\Rightarrow (\ast 2 \text{ (ast 3 (ast 4 (ast 10 1))))} & \\
\Rightarrow (\ast 2 \text{ (ast 3 (ast 4 10)))} & \\
\Rightarrow (\ast 2 \text{ (ast 3 40)}) & \\
\Rightarrow (\ast 2 \text{ 120}) & \\
\Rightarrow 240 &
\end{align*}
\]
As long as the list in question is non-empty, the recursive case evaluates an expression of the form
\((\ast \ (\text{first} \ \text{some-list}) \ \text{mult-all} \ (\text{rest} \ \text{some-list}))\). However, when the list in question is empty, the base case is reached, terminating the recursion. These sorts of considerations lead to the following solution:

\[
\text{(define mult-all}
\begin{cases}
\text{lambda (listy)} \\
\text{(cond} \\
\text{;; Base Case: LISTY is empty} \\
\text{((null? listy)} \\
\text{;; The product of all the elements of the empty list is} \\
\text{;; taken to be 1, the multiplicative identity.} \\
\text{1)} \\
\text{;; Recursive Case: LISTY is non-empty (and so we can use} \\
\text{;; the FIRST and REST accessor functions on LISTY)} \\
\text{(else} \\
\text{;; The product of all of the elements of LISTY is obtained} \\
\text{;; by multiplying the FIRST element of LISTY by the} \\
\text{;; product of all of the REST of the elements of LISTY.} \\
\text{;; The latter job is handled by the recursive func. call.} \\
\text{(* (first listy)} \\
\text{\text{mult-all} (\text{rest listy})\)))\text{)}
\end{cases}
\text{))}
\]

Example 16.2.2: Summing the numbers in a list

The following defines a \text{sum-all} function that sums the numbers in the input list. Its structure is similar to that of the \text{mult-all} function.

\[
\text{(define sum-all}
\begin{cases}
\text{lambda (listy)} \\
\text{(cond} \\
\text{;; Base Case: LISTY is empty} \\
\text{((null? listy)} \\
\text{;; The sum of all the elements of the empty list is} \\
\text{0)} \\
\text{;; Recursive Case: LISTY is non-empty} \\
\text{(else} \\
\text{;; The recursive function call computes the sum of all} \\
\text{;; the numbers in the rest of LISTY; we just add on the} \\
\text{;; first element.} \\
\text{(+ (first listy) \text{sum-all} (\text{rest listy})))})
\end{cases}
\text{))}
\]

\[
\begin{align*}
> \text{(sum-all '}(1 \ 2 \ 3 \ 4)) & \quad 10 \\
> \text{(sum-all '}(1 \ 10 \ 100 \ 1000)) & \quad 1111 \\
> \text{(sum-all '}(2 \ 5 \ 3 \ 8 \ 1)) & \quad 16
\end{align*}
\]
In-Class Problem 16.2.1

Define a function, called \texttt{add-squares}, that satisfies the following contract:

\begin{verbatim}
;; ADD-SQUARES
;; -----------------------------------------
;; INPUT: LISTY, a list of numbers
;; OUTPUT: The sum of the squares of the numbers in LISTY
\end{verbatim}

Here are some examples of the desired behavior:

\begin{verbatim}
> (add-squares ' (2 3 10)) ← 2^2 + 3^2 + 10^2 = 4 + 9 + 100 = 113
113
> (add-squares ' (1 0 5 2)) ← 1^2 + 0^2 + 5^2 + 2^2 = 1 + 0 + 25 + 4 = 30
30
\end{verbatim}

In-Class Problem 16.2.2: Computing the length of a list

Define a function, called \texttt{lengthy}, that computes the number of elements of the input list. Here is its contract:

\begin{verbatim}
;; LENGTHY
;; -------------------------------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: The number of elements of LISTY (i.e., its length)
\end{verbatim}

Here are some examples of the desired behavior:

\begin{verbatim}
> (lengthy '(a b c d e))
5
> (lengthy '(\#t () 22 xyz))
4
\end{verbatim}

Hints: Use list-based recursion. What’s the relationship between the length of listy and the length of (rest listy)? And how many elements are in the empty list?

Incidentally, now that you know how to define a function to compute the length of a list, it’s time to tell you that there is a built-in function, called \texttt{length}, that does just that!

In-Class Problem 16.2.3: Accessing the \textit{N}th element of a list

Define a function, called \texttt{fetch-nth-element}, that satisfies the following contract:

\begin{verbatim}
;; FETCH-NTH-ELEMENT
;; -----------------------------------------------
;; INPUTS: LISTY, a list
;; N, a non-negative integer treated as an "index"
\end{verbatim}
;; OUTPUT: Returns the Nth element of LISTY
;; (or #f if LISTY doesn’t have a Nth element)
;; NOTE: The elements of LISTY are indexed starting at 0.

Thus, for example, a is considered to be the zeroeth element of the list (a b c d e), while c is considered to be the element with index 2. Thus, the elements in a list containing five elements will have indices ranging from 0 to 4, inclusive. Here are some examples of the behavior of the fetch-nth-element function:

> (fetch-nth-element '(a b c d e) 0)
a
> (fetch-nth-element '(a b c d e) 2)
c
> (fetch-nth-element '(a b c d e) 8)
#f

Incidentally, now that you know how to implement the fetch-nth-element function, I can tell you that there is a built-in function, called list-ref, that does the same thing. Like fetch-nth-element, the list-ref function treats the first element of a list as having index 0.

Example 16.2.3

Suppose we want to define a function called is-elt-of? that satisfies the following contract:

;; IS-ELT-OF?
;; ----------------------------------------------------------------
;; INPUTS: ITEM, anything
;; LISTRY, a list of stuff
;; OUTPUT: #t (or something that counts as true) if ITEM
;; appears as an element of LISTORY -- as judged by EQ?
;; #f otherwise.

Here are examples of the desired behavior:

> (is-elt-of? 3 '(3 4 5))
#t
> (is-elt-of? 3 '(1 2 3 4 5))
#t
> (is-elt-of? 'x '(a b a b a))
#f

Consider the first example, where ITEM is 3, and LISTORY is (3 4 5). In this case, it is clear that ITEM appears in LISTORY because it appears as the first element. (Notice that this is a kind of base case since, once we find an occurrence of ITEM in LISTORY, there is no need to continue looking any further.) On the other hand, in the second example, where ITEM is 3, and LISTORY is (1 2 3 4 5), it is true that ITEM appears in LISTORY because, as a sequence of recursive functions call might discover, ITEM appears somewhere in the rest of LISTORY. Finally, in the third example, where ITEM is x, and LISTORY is (a b a b a), we could imagine a sequence of recursive function calls that never discover an occurrence of x, eventually leading to the base case: (is-elt-of? ’x ()), which must evaluate to #f, since nothing can appear as an element of the empty list.

In view of these considerations, we are led to the following solution:
(define is-elt-of?
  (lambda (item listy)
    (cond
     ;; Base Case 1: LISTY is EMPTY
     ((null? listy)
      ;; No occurrence of ITEM in the empty list
      #f)
     ;; Base Case 2: ITEM appears as first element of LISTY
     ((eq? item (first listy))
      ;; We found ITEM in LISTY!
      #t)
     ;; Recursive Case: Haven’t found ITEM in LISTY yet
     (else
      ;; Keep looking
      (is-elt-of? item (rest listy))))))

Notice that we must check whether LISTY is empty before trying to use first or rest, since those accessor functions can only be used on non-empty lists.

---

**Example 16.2.4: The built-in member function**

Now that you know how to define the is-elt-of? function, I can tell you that there is a built-in function, called member, that does the same thing! The only difference is that the value returned by member, in cases where it finds ITEM in LISTY, is the portion of LISTY that starts from the first occurrence of ITEM, as illustrated below:

```
> (member 3 '(1 2 3 4 5))
(3 4 5)
> (member 'x '(a b c d e f x y z))
(x y z)
```

Recall that anything other than #f counts as true. So, expressions such as the following are handled appropriately:

```
> (if (member 3 '(1 2 3 4 5)) 'say_yes 'say_no)
say_yes
```

In this case, the condition evaluated to the list (3 4 5), which counts as true, so the if special form evaluated the expression 'say_yes, generating the output value say_yes. For this reason, it does no harm for member to return something that counts as true. Furthermore, in some cases, you might be glad to have access to the list returned by member as its output.

---

**Example 16.2.5: An alternative implementation of is-elt-of?**

Recall from Section 13.3 that, when defining a predicate (i.e., a function that returns a boolean value), one can often write the body of the function using the boolean operators, and, or and not, instead of the conditional expressions, if or cond. Recall further that:

* When defining a predicate using only the boolean operators, the body of the predicate should specify the conditions under which the predicate should output the value #t (or something that counts as true).
Regarding \( (\text{is-elt-of?} \ \text{item listy}) \), we know that it will evaluate to \( \#f \) if listy is empty; therefore, it can only evaluate to \( \#t \) if listy is non-empty. However, that is not enough. In addition, we need to find \text{item} somewhere in listy. What are the possibilities? Well, \text{item} can appear either as the first element of listy, or somewhere in the rest of listy. These considerations lead to the following alternative definition of the \text{is-elt-of?} function. To distinguish the two versions, we call this one \text{is-elt-of-alt}.

\[
\begin{align*}
\text{(define is-elt-of-alt?} \ & \ \text{(lambda (item listy)} \\
& ;; \text{The following expression specifies the conditions under} \\
& ;; \text{which this function should output \#t (or something that} \\
& ;; \text{counts as true):} \\
& ;; \hspace{1em} (1) \text{LISTY must NOT be empty}; \\
& ;; \hspace{1em} \text{AND} \\
& ;; \hspace{1em} (2) \text{ITEM must appear as the FIRST element of LISTY} \\
& ;; \hspace{1em} \text{OR} \\
& ;; \hspace{1em} \text{ITEM must appear somewhere in the REST of LISTY} \\
& \hspace{1em} (\text{and} (\text{not} (\text{null? listy})) \\
& \hspace{1em} (\text{or} (\text{eq? item} (\text{first listy})) \\
& \hspace{1em} (\text{is-elt-of-alt? item} (\text{rest listy}))))
\end{align*}
\]

Try using this function in the Interactions Window to confirm that it works as advertised.

---

**In-Class Problem 16.2.4: Is a list of numbers in increasing order?**

Define a function, called \text{incr?}, that satisfies the following contract:

\[
\begin{align*}
;; \text{INCR?} \\
;; \text{---------------------------------} \\
;; \text{INPUT: LISTY, a non-empty list of numbers} \\
;; \text{OUTPUT: \#t if the numbers in LISTY are in strictly} \\
;; \hspace{1em} ^{\text{increasing}^\ast \text{ order}; \#f otherwise}
\end{align*}
\]

Here are some examples illustrating its behavior:

\[
\begin{align*}
> \ (\text{incr? } '(1 3 8 9 15)) \\
\#t \\
> \ (\text{incr? } '(1 3 4 4 6 9)) \quad \text{← Not strictly increasing} \\
\#f \\
> \ (\text{incr? } '(2 5 8 5 2)) \\
\#f \\
\end{align*}
\]

\* What’s the best way of checking whether the input list contains exactly one element?

Write one version of \text{incr?} that uses \text{if or cond}, and another that uses some combination of \text{and}, \text{or} and \text{not}. 

Example 16.2.6: Printing a histogram

The goal for this example is to define a function, called print-histy, that satisfies the following contract:

```
;; PRINT-HISTY
;; -------------------------------------------------------------
;; INPUT: LISTY, a list of non-negative integers
;; OUTPUT: None
;; SIDE EFFECT: Displays a histogram in the Interactions Window
;; based on the numbers in LISTY. In particular, for each
;; number in LISTY, prints one row of that many asterisks.
```

Here are some examples of the desired behavior:

```
> (print-histy '(3 2 8 4 6))
***
**
*******
****
******
> (print-histy '(1 2 3 4))
*
**
***
****
```

Consider the first example: (print-histy '(3 2 8 4 6)). The beauty of recursive programming is that we can write a function that explicitly does only a small part of the job, while leaving most of the work to the recursive function call. For example, to print out the desired histogram, we can just print out the first row of 3 asterisks, and then let the recursive function call take care of printing the rest of the histogram, based on the rest of the list (i.e., (2 8 4 6)). Of course, in the base case, when the list is empty, we’re all done!

```
(define print-histy
  (lambda (listy)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
        ;; Use the built-in VOID function to do ... nothing!
        (void))
      ;; Recursive Case: LISTY is non-empty
      (else
        ;; Use a helper function to print one row of the histogram
        (print-n-stars (first listy))
        ;; Then print out the rest of the histogram
        (print-histy (rest listy))))))
```

Notice that since there’s nothing to do in the base case, we just use the built-in void function to do ... nothing! (Recall from Section 2.1.4, the void function actually outputs the special void value which DrScheme interprets as “no output”.) Here’s the helper function, which is a slight re-write of the print-n-dashes function from Example 14.2.1:
(define print-n-stars
  (lambda (n)
    (cond
      ((<= n 0) (newline))
      (else
       (printf "*
")
       (print-n-stars (- n 1))))))

Finally, note that because the print-histy function does nothing in the base case, returning void as its output, the print-histy function can be re-written as follows, using the when special form.

(define print-histy
  (lambda (listy)
    ;; Base Case: LISTY is empty, do nothing.
    ;; Recursive Case: LISTY is non-empty
    (when (not (null? listy))
      ;; Use a helper function to print one row of the histogram
      (print-n-stars (first listy))
      ;; Then print out the rest of the histogram
      (print-histy (rest listy))))))

Because this simplification effectively hides the base case, a comment has been inserted to remind the reader that the base case is implicitly handled by when returning void.

16.3 Recursively Generating Lists as Output Values

So far, we have seen examples of recursive functions where the recursion is driven by a list, and the output has been a number, a boolean, or void—along with some side-effect printing. This section addresses list-based recursion where the output value is a list that has been incrementally generated by the recursive function calls. The incremental generation of lists is accomplished using the built-in cons function, introduced in Section 16.1.

Example 16.3.1: Doubling all the elements of a list

Suppose we want to define a function, called double-all, that satisfies the following contract:

;; DOUBLE-ALL
;; ----------------------------------------------
;; INPUT: LISTY, a list of numbers
;; OUTPUT: A list of numbers, each of whose elements
;; is twice the corresponding element in LISTY.

Here are some examples of the desired behavior:

> (double-all '(3 2 10 13))
(6 4 20 26)
> (double-all '(5 3 8))
(10 6 16)

Let’s apply some recursive thinking to the first example: (double-all '(3 2 10 13)). We can generate the desired output list (6 4 20 26) as follows.

(1) Consider the following pieces of the desired output list, (6 4 20 26):
146 Introduction to Computer Science via Scheme

• Its first element: 6
• The rest of its elements: (4 20 26)

(2) Fetch the corresponding pieces of the input list, (3 2 10 13):
• Its first element: 3
• The rest of the list: (2 10 13)

(3) Do the following to the corresponding pieces of the input list:

• Double the first element: (* 2 3) ⇒ 6
• Use a recursive function call to double the rest of the elements:

  (double-all ’(2 10 13)) ⇒ (4 20 26)

(4) Use the above pieces to construct the desired output list using cons:

• (cons 6 ’(4 20 26)) ⇒ (6 4 20 26)

We can more concisely describe the process outlined above, as follows. If listy is a non-empty list, the element-wise doubling of listy can be obtained by the following expression:

  (double-all listy) ⇒ (cons (* 2 (first listy))
  (double-all (rest listy)))

Before jumping to the completed function definition, we need to determine what should happen in the base case, where the input list is empty. There are two things to consider:

• The list obtained by doubling each element of the empty list is ... the empty list:

  (double-all ()) ⇒ ()

• When the input list is a one-element list, the recursive rule described above looks like this:

  (double-all ’(4)) ⇒ (cons (* 2 4) (double-all ()))
  ⇒ (cons 8 ())
  ⇒ (8)

Therefore, whether we consider the base case in isolation—what should double-all do to the empty list based on the contract?—or we consider the base case as the terminating case of a sequence of recursive function calls, we conclude that (double-all ()) should evaluate to ()

Here’s the finished product:

(define double-all
  (lambda (listy)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
        ;; The double-all of () is ...
        ())
      ;; Recursive Case: LISTY is non-empty
      (else
        ;; Double the first element and attach it to the
        ;; double-all of the rest of the list
        (cons (* 2 (first listy))
          (double-all (rest listy)))))
    ))
Example 16.3.2: Applying a given function to each element of a list

Recall the \texttt{facty} function seen in Example 12.1.1. It takes a single number as its input, and returns the factorial of that number as its output:

\begin{verbatim}
> (facty 3)
6
> (facty 5)
120
\end{verbatim}

For this exercise, we want to define a function called \texttt{mappy} that takes two inputs: (1) a function \texttt{func} that, like \texttt{facty}, can be applied to a single input, and (2) a list \texttt{listy}, each of whose elements is a suitable input for \texttt{func}. The expression \((\texttt{mappy \ func \ listy})\) should generate as its output the list whose elements are obtained by applying \texttt{func}, in turn, to each of the elements of \texttt{listy}. Here are some examples:

\begin{verbatim}
> (mappy facty '(3 4 5 6))
(6 24 120 720)
> (mappy even? '(1 2 3 4 5 6))
(#f #t #f #t #f #t)
> (mappy abs '(1 -1 2 -2 3 -3))
(1 1 2 2 3 3)
\end{verbatim}

The last expression uses the built-in \texttt{abs} function, which computes the absolute value of its input.

As in Example 16.3.1, we analyze this problem by thinking recursively, using a concrete example:

\begin{verbatim}
(mappy facty '(3 4 5 6)) ⇒ (6 24 120 720)
\end{verbatim}

1. The parts of the desired output list:
   - Its first element: 6
   - The rest of its elements: (24 120 720)

2. The corresponding parts of the input list:
   - Its first element: 3
   - The rest of its elements: (4 5 6)

3. Do the following to the pieces of the input list:
   - Apply \texttt{facty} to the first element: \((\texttt{facty 3}) ⇒ 6\)
   - Let a recursive function call apply \texttt{facty} to the rest of the elements:
     \((\texttt{mappy facty '(4 5 6)}) ⇒ (24 120 720)\)

4. Use the \texttt{cons} function to combine the above pieces:
   - \((\texttt{cons 6 '(24 120 720)}) ⇒ (6 24 120 720)\)

The above analysis suggests that for a non-empty list \texttt{listy}, the following expression will evaluate to the desired result:

\((\texttt{mappy func listy}) ⇒ \texttt{(cons (func (first listy))}) \texttt{(mappy func (rest listy)))}\)

In addition, you should convince yourself that, as in Example 16.3.1, the base case, \((\texttt{mappy func ()})\), should evaluate to \((\texttt{)}\). Here is the completed solution.
;; MAPPY
;; ----------------------------------------------------------
;; INPUTS: FUNC, a function that takes a single input
;; LISTY, a list of suitable inputs for FUNC
;; OUTPUT: A list whose elements are obtained by applying
;; FUNC to each of the elements of LISTY, in turn.

(define mappy
  (lambda (func listy)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
        ;; Applying FUNC to each element of the empty list
        ;; yields ... the empty list
        (())
      ;; Recursive Case: LISTY is non-empty
      (else
        ;; Apply FUNC to the FIRST element of LISTY, and then
        ;; use CONS to attach the result to the front of the
        ;; list obtained from the recursive function call on
        ;; the REST of LISTY.
        (cons (func (first listy))
          (mappy func (rest listy))))))))

Incidentally, now that you know how to implement the mappy function, I can tell you that there is a built-in function, called map, that does the same thing. The following example illustrates how the map function can be used to facilitate testing.

**Example 16.3.3: Using map to facilitate testing**

Suppose that you have defined a function, called square, that squares its input. Instead of writing several tester expressions to test the performance of square on several inputs, you can write just one tester expression, using map to apply square to several inputs:

```
> (tester '(map square '(1 2 3 4 10 25)))
(map (square '(1 2 3 4 10 25))) ==> (1 4 9 16 100 625)
```

**In-Class Problem 16.3.1: Removing items from a list**

Define a function, called remover, that satisfies the following contract:

;; REMOVER
;; ----------------------------------------------------------
;; INPUTS: ITEM, anything
;; LISTY, a list
;; OUTPUT: A list that contains all of the elements of
;; LISTY, except any occurrences of ITEM.

Here are some examples:

```
> (remover 3 '(1 2 3 4 5 4 3 2 1))
```
Incidentally, now that you know how to implement the `remover` function, I can tell you that there is a built-in function, called `remove`, that does the same thing, except that it only removes the first occurrence of item from listy.

The following example implements a function that takes two input lists, but uses only one to drive the recursion.

### In-Class Problem 16.3.2: Concatenating two lists

Define a function, called `conc`, that satisfies the following contract:

```scheme
;; CONC
;; -----------------------------------------------------------
;; INPUTS: LISTY, LISTZ, two lists
;; OUTPUT: A list containing all of the elements of LISTY
;; followed by all of the elements of LISTZ.
```

Here are some examples of the desired behavior:

```scheme
> (conc '(1 2 3 4) '(a b c))
(1 2 3 4 a b c)
> (conc '(a b c) '(1 2 3 4))
(a b c 1 2 3 4)
```

Hints: Let listy drive the recursion. What is the output when listy is empty?

Now that you know how to implement the `conc` function, I can tell you that there is a built-in function called `append` that does the same thing!

The preceding examples showed how a recursive function can be used to incrementally generate a new list as its output. In each case, some input list was driving the recursion. However, as the following examples show, functions whose recursion is driven by the size of a number can also be used to incrementally generate output lists.

### Example 16.3.4

The goal of this example is to define a function, called `list-down-to-zero`, that satisfies the following contract:

```scheme
;; LIST-DOWN-TO-ZERO
;; -----------------------------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: A list of the form (N N-1 N-2 ... 2 1 0)
```

Here are some examples of the desired behavior:

```scheme
> (list-down-to-zero 5)
(5 4 3 2 1 0)
> (list-down-to-zero 8)
(8 7 6 5 4 3 2 1 0)
```
Thinking recursively about the first example, we note that the list from 5 down to 0 can be constructed by attaching the number 5 to the front of the list from 4 down to 0. More generally, for any non-negative number \( n \):

\[
\text{(list-down-to-zero } n) \Rightarrow \text{(cons } n \text{ (list-down-to-zero } (- n 1))\text{)}
\]

where, for the base case, we stipulate that: \((\text{list-down-to-zero } m) \Rightarrow (), \text{ for any } m < 0\). (Alternatively, we could use \((\text{list-down-to-0 } 0) \Rightarrow (0)\) as our base case.) Here is the completed solution:

\[
\text{(define list-down-to-zero}
\begin{array}{c}
\text{(lambda (n)}
\text{(cond}
\text{;; Base Case: } N < 0
\text{((< n 0)
\text{())}
\text{;; Recursive Case: } N >= 0
\text{(else)
\text{(cons n (list-down-to-zero } (- n 1)))))})
\end{array}
\]

In-Class Problem 16.3.3

Define a function, called \text{list-up-to-n}, that satisfies the following contract:

;; LIST-UP-TO-N
;; ---------------------------------------------
;; INPUTS: \text{FROM}, a non-negative integer (starting point)
;; \text{N}, a non-negative integer (stopping point)
;; OUTPUT: A list of the form \text{(FROM FROM+1 FROM+2 ... N)}

Here are some examples of the desired behavior:

> (list-up-to-n 4 12)
(4 5 6 7 8 9 10 11 12)
> (list-up-to-n 3 7)
(3 4 5 6 7)

\text{Hint: Fill in the blanks:} The list of integers from 4 to 12 can be constructed by attaching \_
 to the front of the list of integers from \_
 to \_
. More generally: The list of integers from \text{FROM} to \text{N} can be constructed by attaching \_
 to the front of the list of integers from \_
 to \_
.

In-Class Problem 16.3.4

Define a function, called \text{random-flips}, that satisfies the following contract:

;; RANDOM-FLIPS
;; ---------------------------------------------
;; INPUTS: \text{N}, a non-negative integer
;; OUTPUT: A list containing \text{N} random flips of a coin,
;; where each flip is either H or T

Here are some examples of the desired behavior:
> (random-flips 8)
(H H T H T T H)
> (random-flips 5)
(T H H T H)

Hint: Use the flip-coin function from Example 15.2.2 as a helper. Fill in the blanks: A list of \( n \) random coin flips can be generated by attaching \( \text{_________________________} \) to the front of a list of \( \text{__________} \) random coin flips.

### 16.4 Tail Recursion, Accumulators, and Wrapper Functions Revisited

Sections 14.2 through 14.4 introduced the concepts of tail recursion, accumulators, and wrapper functions, respectively. As will be seen in this section, these concepts apply equally well to list-based recursion and the incremental generation of lists as output values.

Recall from Defn. 14.2 that a recursive function-call expression is tail recursive if, whenever its evaluation is needed as part of evaluating the parent function’s body, its evaluation is the last step in that process. And a recursive function is tail-recursive if each of its recursive function-call expressions is tail recursive.

Checking the functions implemented in Examples 16.2.1 through 16.3.4 reveals that mult-all, double-all, mappy and list-down-to-zero are not tail recursive, while is-elt-of?, is-elt-of?-alt and print-histy are tail recursive. The following examples define tail-recursive versions of mult-all, list-down-to-zero and double-all, respectively called mult-all-acc, list-down-to-zero-acc and double-all-acc. As the names indicate, each of these tail-recursive functions will take an additional input that serves to accumulate the desired answer. For mult-all-acc, the extra input will incrementally accumulate the product of the numbers in the input list, much as the accumulator in facty-acc (cf. Example 14.3.3) accumulated the factorial of its input. For list-down-to-zero-acc and double-all-acc, the extra input will incrementally accumulate a list: in particular, each tail-recursive function call will include a call to the cons function to attach a new element to the front of some list. As in Section 14.4, for each accumulator-based, tail-recursive function we shall define an accompanying wrapper function that takes care of providing appropriate initial values for any additional inputs.

**Example 16.4.1: Tail-recursive function: mult-all-acc**

Recall that the mult-all function computes the product of all of the numbers in a given list. The mult-all-acc function will work similarly, except that it will take an extra input, called acc, that will accumulate the desired product. In particular, as we walk through the given list of numbers, as each number is encountered, it will be multiplied into the accumulator. As with facty-acc from Example 14.3.3, the initial value of acc will be 1 (i.e., the multiplicative identity).

It can often help to consider a concrete example. Therefore, suppose that we want to use mult-all-acc to compute the product of the numbers in the list \((3 7 2 4)\). We start with acc equal to 1. Imagine the computation proceeding as follows, where the first input to mult-all-acc is the list of numbers, and the second input is the accumulator:

\[
\begin{align*}
(mult-all-acc \ '(3 7 2 4) 1) &\Rightarrow (mult-all-acc \ '(7 2 4) 3) \quad \text{← rec. case: “accumulate” a factor of 3} \\
&\Rightarrow (mult-all-acc \ '(2 4) 21) \quad \text{← rec. case: “accumulate” a factor of 7} \\
&\Rightarrow (mult-all-acc \ '(4) 42) \quad \text{← rec. case: “accumulate” a factor of 2} \\
&\Rightarrow (mult-all-acc \ () 168) \quad \text{← rec. case: “accumulate” a factor of 4} \\
&\Rightarrow 168 \quad \text{← base case: accumulator has the answer!}
\end{align*}
\]

Notice that the inputs for each recursive function call are:
• the rest of the current list, and
• the product of the first element of the current list and the current accumulator.

Thus, by the time the base case (i.e., the empty list) is reached, the accumulator has the desired product: $3 \cdot 7 \cdot 2 \cdot 4 = 168$. Here is the completed solution:

```
;; MULT-ALL-ACC
;; ---------------------------------------------
;; INPUTS: LISTY, a list of numbers
;; ACC, a number (accumulator of desired product)
;; OUTPUT: When called with ACC=1, the output is the product
;; of all of the numbers in LISTY. More generally, the output
;; is the product of ACC and all of the numbers of LISTY

(define mult-all-acc
  (lambda (listy acc)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy) acc)
      ;; Recursive Case: LISTY is non-empty
      (else
       ;; Tail-recursive function call on adjusted inputs:
       ;; Note: ACC "accumulates" (first listy)
       (mult-all-acc (rest listy) (* (first listy) acc))))))
```

As is often the case, describing the output for accumulator-based functions can be challenging in the general case (e.g., above, when ACC is something other than 1). Here is the accompanying wrapper function:

```
;; MULT-ALL-WR
;; ----------------------------------------
;; INPUT: LISTY, a list of numbers
;; OUTPUT: The product of the numbers in LISTY
(define mult-all-wr
  (lambda (listy) (mult-all-acc listy 1)))
```

Notice that the contract for mult-all-wr is the same as that for mult-all—except for the name of the function. That is, the two functions are equivalent.

**Example 16.4.2: Tail-recursive function: list-down-to-zero-acc**

Recall that the list-down-to-zero function takes a non-negative integer $n$ as its only input, and generates as its output a list of the form $(n \ n-1 \ n-2 \ \ldots \ 2 \ 1 \ 0)$. The list-down-to-zero-acc function will work similarly, except that it will incrementally accumulate the desired list in an extra input, acc. As in the double-all and mappy functions (cf. Examples 16.3.1 and 16.3.2, respectively) the
list-accumulator will start out as the empty list.

Consider the example where the numerical input \( n \) is 3, and we want to generate the list \((3 2 1 0)\). As in `list-down-to-zero`, the value of \( n \) will decrease by one on each recursive function call, but the accumulator will be adjusted by using the `cons` function to attach \( n \) to the front of the accumulator, as illustrated in the following sequence of evaluations:

\[
\begin{align*}
& (\text{list-down-to-zero-acc } 3 () ) \\
& \quad \Rightarrow (\text{list-down-to-zero-acc } 2 ' (3)) \quad \leftarrow \text{ attach 3 to front of acc} \\
& \quad \Rightarrow (\text{list-down-to-zero-acc } 1 ' (2 3)) \quad \leftarrow \text{ attach 2 to front of acc} \\
& \quad \Rightarrow (\text{list-down-to-zero-acc } 0 ' (1 2 3)) \quad \leftarrow \text{ attach 1 to front of acc} \\
& \quad \Rightarrow (\text{list-down-to-zero-acc } -1 ' (0 1 2 3)) \quad \leftarrow \text{ attach 0 to front of acc} \\
& \quad \Rightarrow (0 1 2 3) \quad \leftarrow \text{ acc has the answer??!}
\end{align*}
\]

Whoops! While this would be fine for generating a list from 0 to \( n \), that is not what we were aiming for! This example illustrates a common issue that arises when using list accumulators:

* When using an accumulator to incrementally generate a list, the order of the elements in the accumulator ends up being the reverse of the order in which they were attached!

There are two ways to fix this problem: (1) define a function to reverse the elements of a list; or (2) arrange to process the desired elements in the opposite order. Below, we take the second approach. Later on, we'll define a function for reversing the elements of a list.

For the `list-down-to-zero-acc` function, we can arrange to visit the numbers in the order from 0 up to \( n \) by including yet another input, called `curr` (for current number), whose value shall start out at 0 and increment by one on each recursive function call. Since 0 will be the first number to be attached to the accumulator, it will end up being the last number in the generated list, as desired. So the inputs to `list-down-to-zero-acc` will be \( n, \text{acc} \) and `curr`. In this version, the value of \( n \) will be the same for each recursive function call. That is, \( n \) serves as an upper bound on the value of `curr`. When that upper bound is reached, the recursion will terminate, as illustrated below:

\[
\begin{align*}
& (\text{list-down-to-zero-acc } 3 () 0) \\
& \quad \Rightarrow (\text{list-down-to-zero-acc } 3 ' (0) 1) \quad \leftarrow \text{ attach 0 to front of acc} \\
& \quad \Rightarrow (\text{list-down-to-zero-acc } 3 ' (1 0) 2) \quad \leftarrow \text{ attach 1 to front of acc} \\
& \quad \Rightarrow (\text{list-down-to-zero-acc } 3 ' (2 1 0) 3) \quad \leftarrow \text{ attach 2 to front of acc} \\
& \quad \Rightarrow (\text{list-down-to-zero-acc } 3 ' (3 2 1 0) 4) \quad \leftarrow \text{ attach 3 to front of acc} \\
& \quad \Rightarrow (3 2 1 0) \quad \leftarrow \text{ acc has the answer!}
\end{align*}
\]

Notice that in this version of `list-down-to-zero-acc`, the base case is signaled by `curr` being greater than \( n \)—in this example, when 4 > 3. Here is the completed solution:

```scheme
;; LIST-DOWN-TO-ZERO-ACC
;; -----------------------------------------------
;; INPUTS:  \( n \), a non-negative integer
;; ACC, a list accumulator
;; CURR, a non-negative integer
;; OUTPUT: When called with ACC=() and CURR=0, the output
;; is the list \((N N-1 N-2 ... 2 1 0)\). More generally,
;; the output is the "concatenation" of the lists
;; \((N N-1 N-2 ... \text{CURR})\) and ACC.
```

```scheme
(list-down-to-zero-acc 3 () 0)
⇒ (list-down-to-zero-acc 3 ' (0) 1) ← attach 0 to front of acc
⇒ (list-down-to-zero-acc 3 ' (1 0) 2) ← attach 1 to front of acc
⇒ (list-down-to-zero-acc 3 ' (2 1 0) 3) ← attach 2 to front of acc
⇒ (list-down-to-zero-acc 3 ' (3 2 1 0) 4) ← attach 3 to front of acc
⇒ (3 2 1 0) ← acc has the answer!
```
(define list-down-to-zero-acc
  (lambda (n acc curr)
    (cond
      ;; Base Case: CURR > N
      ((> curr n)
        ;; The accumulator has the desired list
        acc)
      ;; Recursive Case: CURR <= N
      (else
        ;; Tail-recursive function call with adjusted inputs:
        (list-down-to-zero-acc n (cons curr acc) (+ curr 1))))))

(You should convince yourself that the “more generally” part of the contract is correct.) Here is the associated wrapper function:

;; LIST-DOWN-TO-ZERO-WR
;; ---------------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: The list (N N-1 N-2 ... 2 1 0)
(define list-down-to-zero-wr
  (lambda (n)
    ;; Call the tail-recursive helper with ACC=() and CURR=0:
    (list-down-to-zero-acc n () 0)))

Before introducing the double-all-acc function, which also uses a list accumulator and, so, suffers from the same problem seen earlier regarding the order of accumulated elements, we first introduce the transfer-all and reversey functions. The latter function can be used to reverse the elements in a list.

Example 16.4.3: The transfer-all and reversey functions

The goal for this example is to define a function, called transfer-all, that satisfies the following contract:

;; TRANSFER-ALL
;; ---------------------------------------------
;; INPUTS: LISTY, LISTZ, two lists
;; OUTPUT: The list obtained by "popping" each element in turn off of the front of LISTY and "pushing" it onto the front of LISTZ.

Here are some examples of the desired behavior:

> (transfer-all '(a b c) '(1 2))
  (c b a 1 2)
> (transfer-all '(1 2) '(a b c))
  (2 1 a b c)

Notice that the elements from the first list appear in the reverse order in the output list. Here is a sample sequence of evaluations corresponding to the first example above:
(transfer-all '(a b c) '(1 2))
⇒ (transfer-all '(b c) '(a 1 2)) ← attach a to front of second list
⇒ (transfer-all '(c) '(b a 1 2)) ← attach b to front of second list
⇒ (transfer-all () '(c b a 1 2)) ← attach c to front of second list
⇒ (c b a 1 2) ← base case!

As the above example illustrates, the first list (i.e., listy) is driving the recursion, and the second list (i.e., listz) is acting like an accumulator. When listy is empty, the accumulator listz contains the desired answer. Here is the completed function definition:

(define transfer-all
  (lambda (listy listz)
    (cond
     ;; Base Case: LISTY is empty
     ((null? listy)
      ;; return the "accumulator"
      listz)
     ;; Recursive Case: LISTY is non-empty
     (else
      ;; Tail-recursive function call with adjusted inputs
      (transfer-all (rest listy)
                    (cons (first listy) listz))))))

Next, we define a “wrapper” for transfer-all which we shall call reversey, for reasons that will soon become apparent.

;;; REVERSEY -- wrapper for TRANSFER-ALL
;;;-----------------------------------------------------
;;; INPUT: LISTY, a list
;;; OUTPUT: A list that contains the same elements as
;;;         LISTY, but in the opposite order.

(define reversey
  (lambda (listy)
    ;; Call TRANSFER-ALL with LISTZ=():
    (transfer-all listy ()))))

Here are some examples that illustrate that reversey does indeed generate the reversal of its input:

> (reversey '(a b c d))
(d c b a)
> (reversey '(1 2 3 4 5 6))
(6 5 4 3 2 1)

Incidentally, now that you know how to implement the reversey function, I can tell you that there is a built-in function called reverse that does the same thing!
Example 16.4.4: Not all ways of reversing a list are equal!

This example considers an alternative approach to reversing a list, one based on repeated concatenation. Although this approach leads to a function that correctly reverses a list, it turns out to be very inefficient. First, since it is not tail recursive, it can use an awful lot of the computer’s memory when reversing long lists. Second, by repeatedly concatenating long lists, it takes a lot longer to reverse a list than the reversey function seen earlier. To illustrate the inefficiency of this approach, both functions, konk and bad-reverse, defined below, print out some information each time they are called. The konk function concatenates two lists; bad-reverse uses konk as a helper function.

;; KONK
;; --------------------------------------------------------------
;; INPUTS: LISTY, LISTZ, two lists
;; OUTPUT: A list containing all of the elements of LISTY, followed by all of the elements of LISTZ.
(define konk
  (lambda (listy listz)
    (printf "KONK: LISTY: \"A\", LISTZ: \"A\"\" listy listz)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
        ;; The concatenation of () and LISTZ is:
        (konk (null? listy) listz))
      ;; Recursive Case: LISTY is non-empty
      (else
        ;; Attach (FIRST LISTY) onto the concatenation
        ;; of (REST LISTY) and LISTZ
        (konk (rest listy) listz)
        (cons (first listy)
          (konk (rest listy) listz))))))

;; BAD-REVERSE
;; -------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: A list containing the same elements as LISTY, but in the opposite order.
(define bad-reverse
  (lambda (listy)
    (printf "BAD-REVERSE: LISTY: \"A\" listy)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
        ()
      ;; Recursive Case: LISTY is non-empty
      (else
        ;; Recursive function call reverses the REST of LISTY.
        ;; So, we need to attach (first listy) at the end.
        ;; Unfortunately this involves walking through the
        ;; potentially long list returned by the recursive
        ;; function call.
        (konk (bad-reverse (rest listy))
          (cons (first listy) ())))))
To get an idea of how inefficient bad-reverse is, try evaluating the following expression in the Interactions Window: `(bad-reverse '(a b c d e)).`

**Example 16.4.5: The `double-all-acc` function**

The goal of this example is to define a tail-recursive function that doubles all of the elements of a given list of numbers. Because we shall use a list accumulator, the doubled numbers in the accumulated list will come out in the wrong order. But we shall just use the built-in `reverse` function to reverse the order of the accumulated list before returning it as the output. Here is the completed function definition:

```scheme
;; DOUBLE-ALL-ACC
;; -----------------------------
;; INPUTS: LISTY, a list of numbers
;; ACC, a list accumulator
;; OUTPUT: When called with ACC=(), the output is
;; a list just like LISTY, except that each
;; element has been doubled.
(define double-all-acc
  (lambda (listy acc)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy) 
       ;; REVERSE the accumulator!
       (reverse acc))
      ;; Recursive Case: LISTY is non-empty
      (else
       ;; Tail-recursive function call with adjusted inputs
       (double-all-acc (rest listy)
                       ; ; "Accumulate" the first element doubled
                       (cons (* 2 (first listy))
                             acc))))))
```

As this example illustrates, the previously identified issue with list accumulators (i.e., that the accumulated elements come out in the opposite order) is easily resolved using the `reverse` function at the very last instant!

### 16.5 Sorting Algorithms

This section introduces two algorithms for sorting a list of numbers: the `insertion-sort` algorithm, and the `merge-sort` algorithm. After defining Scheme functions that implement these algorithms, they are compared by running them on long lists of randomly generated numbers. In what follows, we shall assume that the goal is to sort lists of numbers into non-decreasing order, as illustrated below:

**Before sorting:**  
(3 2 1 4 3 2 3 6 1 0 5)

**After sorting:**  
(0 1 1 2 2 3 3 3 4 5 6)

Notice that for any consecutive elements, $x$ and $y$, in the sorted list, the following holds: $x \leq y$. 
16.5.1 The Insertion-Sort Algorithm

The insertion-sort algorithm uses a helper function, called insert, that inserts a number into an already-sorted list, such that the resulting list is still sorted. Here is its contract, followed by some examples of the desired behavior:

```scheme
;; INSERT
;; ---------------------------------------------------------------
;; INPUTS: NUM, a number
;; SORTED, a list of numbers that are already sorted
;; into non-decreasing order
;; OUTPUT: The list obtained by inserting NUM into SORTED while
;; preserving the non-decreasing ordering

> (insert 3 '(5 8 9 10 11)) ← 3 goes at the front of the sorted list
(3 5 8 9 10 11)
> (insert 3 '(0 1 1 2)) ← 3 goes at the end of the sorted list
(0 1 1 2 3)
> (insert 3 '(1 2 4 5 6)) ← 3 goes somewhere in the middle
(1 2 3 4 5 6)
> (insert 3 '(1 2 2 3 4 4 4 9 12)) ← Same as above, except that there’s another 3
(1 2 2 3 3 4 4 4 9 12)
```

Intuitively, the insert function walks through the already-sorted list until it finds the proper place for the given number. (What distinguishes the “proper place” for the given number?) We’ll have more to say about how the insert function might do this—in fact, we’ll define the insert function from scratch—but, for now, we’ll just take the insert function as given.

As indicated earlier, the insertion-sort algorithm takes a (usually unsorted) list of numbers as its only input. Its goal is to generate as its output a list containing the same elements, but sorted into non-decreasing order. Here is its contract:

```scheme
;; INSERTION-SORT
;; ---------------------------------------------------------
;; INPUTS: LISTY, a list of numbers
;; OUTPUT: A list containing the same elements as LISTY,
;; but sorted into non-decreasing order

(it can be implemented using list-based recursion, as follows. First, as a base case, consider that the empty list is already sorted. Next, for the recursive case (i.e., when its input is a non-empty list), the insertion-sort algorithm applies the following recursive rule:

```scheme
(insertion-sort listy) ⇒ (insert (first listy)
(insertion-sort (rest listy)))
```

According to its contract, the recursive call on the rest of listy should generate a sorted list containing all of the elements of (rest listy). Therefore, to generate the desired output (i.e., a sorted list that contains all of the elements of listy), it only remains to find out where (first listy) should be inserted into that sorted rest of listy. And that is precisely what the call to the insert helper function does. Here is the completed definition of the insertion-sort function:

---

1 A one-element list is also already sorted, but we stick with the empty list as the base case to simplify the code slightly.

2 In general, when defining recursive functions, we assume that the recursive function call will generate the right answer. After all, it will be evaluated using the same function that we are currently defining! This sort of assumption—which, at first, may seem crazy—is justified by mathematical induction.
(define insertion-sort
  (lambda (listy)
    (cond
     ;; Base Case: LISTY is empty
     ;; The empty list is already sorted
     ((null? listy) ())
     ;; Recursive Case: LISTY is non-empty
     (else (insert (first listy) (insertion-sort (rest listy)))))))

---

**Example 16.5.1: Applying insertion-sort to a sample list**

Suppose that listy is the list (3 2 5 1 6). Then the recursive function call on the rest of listy would be, in effect,

```
(insertion-sort '(2 5 1 6))
```

Assuming that the recursive function call does the right thing, it should generate as its output the sorted list (1 2 5 6). Therefore, in this case, the above-mentioned recursive rule would, in effect, lead to the following sequence:

```
(insertion-sort '(3 2 5 1 6))
⇒ (insert 3 (insertion-sort '(2 5 1 6)))
⇒ (insert 3 '(1 2 5 6))
⇒ '(1 2 3 5 6)
```

And if we were to consider the details of each recursive function call, we would, in effect, end up with the following sequence of evaluations, using the abbreviations, i for insert, and isort for insertion-sort:

```
(isort '(3 2 5 1 6))
⇒ (i 3 (isort '(2 5 1 6)))
⇒ (i 3 (i 2 (isort '(5 1 6))))
⇒ (i 3 (i 2 (i 5 (isort '(1 6)))))
⇒ (i 3 (i 2 (i 5 (i 1 (isort '(6))))))
⇒ (i 3 (i 2 (i 5 (i 1 6 (isort ()))))))
⇒ (i 3 (i 2 (i 5 (i 1 6 ())))))
⇒ (i 3 (i 2 (i 5 '(1 6))))
⇒ (i 3 (i 2 '(1 5 6)))
⇒ (i 3 '(1 2 5 6))
⇒ '(1 2 3 5 6)
```

---

**In-Class Problem 16.5.1: The insert helper function**

Define the insert function to satisfy the contract given earlier.

**Hints:** Use recursion to walk through sorted until you find the proper place for num. How will you recognize the proper place for num? Consider (first listy) and num. Finally, what should you do if sorted is empty?
In-Class Problem 16.5.2: Generating long lists of random numbers

Define a function, called `list-of-n-random-numbers`, that satisfies the following contract:

```scheme
;; LIST-OF-N-RANDOM-NUMBERS
;; ----------------------------------------------
;; INPUT: N, a positive integer
;; OUTPUT: A list containing N numbers, each randomly generated
;;         from the set {0, 1, 2, ..., 99999}
```

Here are some examples of the desired behavior:

```scheme
> (list-of-n-random-numbers 10)
(18980 44224 94176 57470 23568 47609 70753 77870 98756 11729)
> (list-of-n-random-numbers 5)
(68856 3578 85898 27820 87029)
```

**Hint:** In the recursive case, use the built-in `random` function with an appropriate input.

This function can be used to randomly generate lists of numbers for `insertion-sort` to sort, as illustrated below:

```scheme
> (let* ((list-o-randies (list-of-n-random-numbers 5))
         (sorted (insertion-sort list-o-randies)))
   (printf "BEFORE: \"A\" %" list-o-randies)
   (printf "AFTER: \"A\" %" sorted)
   sorted)
BEFORE: (68502 79284 50452 31764 48239)
AFTER: (31764 48239 50452 68502 79284)
(31764 48239 50452 68502 79284)
> (let* ((list-o-randies (list-of-n-random-numbers 5))
         (sorted (insertion-sort list-o-randies)))
   (printf "BEFORE: \"A\" %" list-o-randies)
   (printf "AFTER: \"A\" %" sorted)
   sorted)
BEFORE: (51897 96352 87874 82047 17760)
AFTER: (17760 51897 82047 87874 96352)
(17760 51897 82047 87874 96352)
```

Of course, it will be more interesting to see how long it takes `insertion-sort` to sort really long lists of numbers (e.g., lists having thousands of elements). In such cases, you wouldn’t want to print out the before and after lists!

To avoid excessive memory usage, it is better to implement accumulator-based tail-recursive versions of the `insert` and `insertion-sort` functions.

In-Class Problem 16.5.3: Accumulator-based tail-recursive version of the `insert` function

For this problem, the goal is to define an accumulator-based tail-recursive version of the `insert` function, called `insert-acc`. Recall that the `insert` function aims to insert a given number `num` into its proper place in an already-sorted list, `sorted`. The main idea behind the accumulator-based tail-recursive approach is to walk through `sorted`, accumulating all of its elements that are smaller than `num`, as
illustrated below:

\[
\begin{align*}
\text{(insert-acc num sorted acc)} \\
\text{(insert-acc 5 '(1 2 4 6 12 15) () )} \\
\text{(insert-acc 5 '(2 4 6 12 15) '(1))} \\
\text{(insert-acc 5 '(4 6 12 15) '(2 1))} \\
\text{(insert-acc 5 '(6 12 15) '(4 2 1))}
\end{align*}
\]

Notice that when all of the numbers smaller than num have been accumulated, the proper place for num has been found (i.e., the base case has been reached). The only thing that remains is to assemble the pieces into the final sorted list. In the above example, the desired list is (1 2 4 5 6 12 15), which can be built as follows:

1. Use cons to attach num to the front of sorted, yielding (5 6 12 15).
2. Use transfer-all (from Example 16.4.3) to transfer all of the elements of acc onto the result of Step 1, yielding (1 2 4 5 6 12 15).

Using the approach outlined above, define the insert-acc to satisfy the following contract:

;; INSERT-ACC
;; -----------------------------------------------------------
;; INPUT: NUM, a number
;; SORTED, a list of numbers that are already sorted
;; into non-decreasing order
;; ACC, a list of numbers in non-increasing order,
;; where each number in ACC is less than NUM
;; OUTPUT: When called with ACC = (), the output is a list
;; containing NUM and all the numbers in SORTED,
;; all sorted into non-decreasing order.

Here are some examples of its use:

\[
\begin{align*}
& \text{> (insert-acc 5 '(1 2 4 6 12 15) () )} \\
& (1 2 4 5 6 12 15) \\
& \text{> (insert-acc 3 '(1 1 2 2 3 3 4 4 5 5) () )} \\
& (1 1 2 2 3 3 3 4 4 5 5)
\end{align*}
\]

Finally, define a wrapper function, called insert-wr, that satisfies the following contract, and exhibits the behavior shown below:

;; INSERT-WR -- wrapper function for INSERT-ACC
;; -----------------------------------------------------------
;; INPUT: NUM, a number
;; SORTED, a list of numbers that are already sorted
;; into non-decreasing order
;; OUTPUT: A list containing NUM and all the numbers in SORTED,
;; all sorted into non-decreasing order.

\[
\begin{align*}
& \text{> (insert-wr 5 '(1 2 4 6 12 15))} \\
& (1 2 4 5 6 12 15) \\
& \text{> (insert-wr 3 '(1 1 2 2 3 3 4 4 5 5))} \\
& (1 1 2 2 3 3 3 4 4 5 5)
\end{align*}
\]
In-Class Problem 16.5.4: Tail-recursive version of insertion-sort

For this problem, we seek a tail-recursive version of the insertion-sort algorithm. For convenience, we call it isort-acc. The following sequence of recursive function calls illustrates the approach, which uses an extra accumulator argument to accumulate the sorted list. At each step the first element of the unsorted list is inserted into its proper place in the sorted list:

$$\Rightarrow (\text{isort-acc } '(4\ 9\ 2\ 6)\ ())$$ ← recursive case  
$$\Rightarrow (\text{isort-acc } '(9\ 2\ 6)\ (\text{insert-wr}\ 4\ ())$$ ← recursive case  
$$\Rightarrow (\text{isort-acc } '(9\ 2\ 6)\ '(4))$$ ← recursive case  
$$\Rightarrow (\text{isort-acc } '(2\ 6)\ (\text{insert-wr}\ 9\ '(4))$$ ← recursive case  
$$\Rightarrow (\text{isort-acc } '(2\ 6)\ '(4\ 9))$$ ← recursive case  
$$\Rightarrow (\text{isort-acc } '(6)\ (\text{insert-wr}\ 2\ '(4\ 9))$$ ← recursive case  
$$\Rightarrow (\text{isort-acc } '(6)\ '(2\ 4\ 9))$$ ← recursive case  
$$\Rightarrow (\text{isort-acc } ()\ (\text{insert-wr}\ 6\ '(2\ 4\ 9))$$ ← base case  
$$\Rightarrow (\text{isort-acc } ()\ '(2\ 4\ 6\ 9))$$  
$$\Rightarrow (2\ 4\ 6\ 9)$$

Once your isort-acc function is working properly, define a wrapper function called isort-wr that calls isort-acc with an appropriate value for the accumulator.

16.5.2 The Merge-Sort Algorithm

The merge-sort algorithm, like the insertion-sort algorithm, takes a (typically unsorted) list of numbers as its input, and generates a sorted version of that list as its output. Here is its contract:

```
;; MERGE-SORT
;; --------------------------------------------------------
;; INPUTS: LISTY, a list of numbers
;; OUTPUT: A list containing the same elements as LISTY, but sorted into non-decreasing order
```

However, the merge-sort algorithm takes a very different approach to sorting lists, as follows. First, its base case handles the case where listy is a one-element list which, of course, must already be sorted. Second, when listy is non-empty, it uses recursion, as follows:

1) Split listy into two lists, lefty and righty, of roughly the same size;

2) Use the merge-sort function to sort lefty, yielding a sorted list, sorted-lefty; and use merge-sort to sort righty, yielding a sorted list, sorted-righty; and then

3) Merge the two sorted lists, sorted-lefty and sorted-righty, into a single sorted list, which will be the desired output.

As indicated above, the merge-sort function uses two helper functions: split and merge. These helpers will be defined shortly. For now, we will assume that they are available, and that they satisfy the following contracts:

```
;; SPLIT
;; --------------------------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: A list of the form (LEFTY RIGHTY) where LEFTY and RIGHTY are two subsidiary lists such that the elements of LISTY have been allocated as evenly as possible to LEFTY and RIGHTY, but with no regard to their order.
```
;; MERGE
;; -----------------------------------------------
;; INPUT: SORTED-ONE, SORTED-TWO, two lists of numbers
;; that are already sorted into non-decreasing order.
;; OUTPUT: A single list that contains all of the elements
;; of SORTED-ONE and SORTED-TWO, sorted into
;; non-decreasing order.

Example 16.5.2: The split and merge helper functions

Here are some examples of the behavior of the split and merge helper functions:

> (split '(5 3 1 2 8 4 9 4))  ← Input has an even number of elements
   ((4 4 2 3) (9 8 1 5))
> (split '(5 3 1 2 7))       ← Input has an odd number of elements
   ((7 1 5) (2 3))
> (merge '(1 3 5 7) '(2 4 6 8))
   (1 2 3 4 5 6 7 8)
> (merge '(1 1 2 3 3 3 5 9) '(2 3 3 4 8 8 9))
   (1 1 2 2 3 3 3 3 3 4 5 8 8 9 9)

In the case of the split function, notice that the order of the elements in the input list and the two subsidiary lists in the output do not matter at all. The reason is that split will typically be applied to unsorted lists—so the order of the elements doesn’t matter. Also, if the input list has an even number of elements, then the two lists in the output will have the same number of elements; otherwise, one of the output lists will have the odd element. For the merge function, the two input lists must already be sorted, but they may have duplicate elements, and the two input lists need not have the same number of elements.

Example 16.5.3: Applying merge-sort to a sample list

Here, we consider the application of the merge-sort function to the input list (8 2 5 9 3 4 6 1). As described previously, there are three steps to the recursive case:

1. Split listy into two lists, lefty and righty, of roughly the same size. Here:
   
   lefty = (6 3 5 8)
   righty = (1 4 9 2)

2. Use the merge-sort function to sort lefty, yielding a sorted list, sorted-lefty; and use merge-sort to sort righty, yielding a sorted list, sorted-righty. Here:
   
   sorted-lefty = (3 5 6 8)
   sorted-righty = (1 2 4 9)

3. Merge the two sorted lists, sorted-lefty and sorted-righty, into a single sorted list, which will be the desired output. Here:
   
   (merge '(3 5 6 8) '(1 2 4 9)) ⇒ (1 2 3 4 5 6 8 9).

Here is the completed definition of the merge-sort function:

```
(define merge-sort
 (lambda (listy)
   (cond
```


Base Case: LISTY has exactly one element
((null? (rest listy))
 ;; A one-element list is already sorted
 listy)

Recursive Case: LISTY has at least two elements
(else
 (let* ((list-o-lists (split listy))
        ;; Access the two subsidiary lists in LIST-O-LISTS
        (lefty (first list-o-lists))
        (righty (second list-o-lists))
        ;; Recursively sort LEFTY and RIGHTY
        (sorted-lefty (merge-sort lefty))
        (sorted-righty (merge-sort righty)))
 ;; Body of the LET*: MERGE the two sorted lists
 (merge sorted-lefty sorted-righty))))

Notice that most of the work is done in the variable-declaration part of the let* special form. The body of the let* just applies the merge function to the two sorted lists.

Now it is time to define the split and merge helper functions needed by merge-sort.

### In-Class Problem 16.5.5: The `split` helper function

Define the split helper function to satisfy the contract seen earlier. Here are some hints:

1. Define an accumulator-based helper function, called `split-acc`, that includes two extra inputs, lefty and righty. These will serve as accumulators for the two subsidiary lists.

2. In the base case, use the `list-two` function defined in In-Class Problem 16.1.2 to create the desired list of lists. (Alternatively, use the built-in `list` function; or use a couple of calls to the `cons` function.)

3. Define `split` as a wrapper function that simply calls `split-acc` with appropriate initial values for its accumulator inputs.

### In-Class Problem 16.5.6: The `merge` helper function

Define the merge helper function to satisfy the contract seen earlier. Here are some hints:

1. When either list is empty, the answer is easy.

2. When both lists are non-empty, compare their first elements to see which one comes first.

Define two versions of the merge function: one that is not tail recursive (and perhaps easier to define), and one that is just a wrapper for a tail-recursive helper function called `merge-acc`. The contract for `merge-acc` is given below.

```scheme
;; MERGE-ACC
;; ---------------------------------------------
;; INPUTS: SORTED-LEFTY, SORTED-RIGHTY, two lists of
;; numbers, each sorted into non-decreasing order
;; ACC, a list-accumulator
;; OUTPUT: When called with ACC=(), the output is a
;; single list containing all of the elements
;; of SORTED-LEFTY and SORTED-RIGHTY, sorted
```
16.5.3 Comparing the Performance of Insertion Sort and Merge Sort

This section shows how we can write Scheme functions to automate a rigorous comparison of the insertion-sort and merge-sort algorithms. Some considerations include:

- We want to test these algorithms on really long lists of randomly generated numbers.
- For each randomly generated list, we want to test both algorithms on the same list.
- We’d like to know how long it takes each algorithm to sort the lists.

We already have the list-of-n-random-numbers function, from In-Class Problem 16.5.2. And since the two sorting algorithms are non-destructive, we can simply store the randomly generated list of numbers in a local variable, and then apply each sorting algorithm to the same list. As for timing their performance, Scheme provides a special form, called time, described below.

The time special form. The purpose of the time special form is to report how long it takes to evaluate a given expression. The syntax and semantics of the time special form are simple.

(Syntax) Any expression of the form (time expr) is a legal instance of the time special form.

(Semantics – Output Value) Any expression of the form (time expr) evaluates to whatever expr evaluates to.

(Semantics – Side Effect) The evaluation of an expression of the form (time expr) causes three pieces of timing information to be displayed in the Interactions Window:

- cpu time how many milliseconds DrScheme spent evaluating expr. (CPU is an acronym for the computer’s central processing unit.)
- real time how many milliseconds elapsed while expr was evaluated.
- gc time how many milliseconds were spent in a memory-management process called garbage collection. (Garbage collection is an extremely interesting and important concept in the management of a computer’s memory, but a discussion of it is beyond the scope of this book.)

The cpu time is typically a bit less than the real time because a computer’s CPU typically does more than one thing during any given time interval; thus, the time the CPU devotes to DrScheme’s evaluation of expr will typically be less than the elapsed time. For our purposes, the cpu time is the most relevant, because it most accurately reflects how much time DrScheme spent evaluating the given expression.

Example 16.5.4: Using the time special form

Here are some examples of the time special form in action:

```
> (time (list-of-n-random-numbers 10000))
cpu time: 4 real time: 5 gc time: 0
(19207 53390 65067 65764 68321 75622 81451 38038 86109 ...)
> (time (insertion-sort (list-of-n-random-numbers 10000)))
cpu time: 7643 real time: 7849 gc time: 62
```
> (let ((listy (list-of-n-random-numbers 10000))
    (time (insertion-sort listy)))
cpu time: 7519 real time: 7674 gc time: 61
(2 9 14 16 26 31 32 37 38 40 84 85 113 114 115 119 171 ...)

The first example shows that it doesn’t take DrScheme long to generate a list of 10,000 random numbers. The second example shows how long it takes to generate and sort a list of numbers, using the insertion-sort function. The last example is the most important: it shows how long the sorting process takes; it ignores the time needed to generate the original list of random numbers.

---

Example 16.5.5: Comparing the performance of the sorting algorithms

The following function can be used to compare the performance of the insertion-sort and merge-sort algorithms.

;; COMPARE-SORTING-ALGS
;; ---------------------------------------
;; INPUT: N, a positive integer
;; OUTPUT: None
;; SIDE EFFECT: Reports how long it took for the
;; insertion-sort and merge-sort algorithms to sort
;; the same randomly generated list of N numbers.
(define compare-sorting-algs
  (lambda (n)
    (let ;; Generate a list of n randomly generated numbers
        (listy (list-of-n-random-numbers n))
      (printf "Running insertion-sort ...\n")
      (time (insertion-sort listy))
      (printf "\nRunning merge-sort ...\n")
      (time (merge-sort listy))
    ;; Return VOID (so we don’t see a long list of numbers)
    (void))))

Here is an example:

> (compare-sorting-algs 1000)
Running insertion-sort ...
cpu time: 87 real time: 93 gc time: 0

Running merge-sort ...
cpu time: 6 real time: 6 gc time: 0

In-Class Problem 16.5.7: A thorough comparison of merge-sort and insertion-sort

Use the compare-sorting-algs function to compare the performance of the two sorting algorithms on lists of the following lengths: 1000, 2000, 4000, 8000, 16000, etc. Which algorithm would you recom-
Example 16.5.6: The built-in sort function

Scheme provides a built-in function, called sort, whose contract is given below, followed by some examples of its use.

;; SORT -- built-in function
;; ---------------------------------------------
;; INPUTS: LISTY, a list of stuff
;; OUTPUT: A list containing the same elements as LISTY,
;; but sorted such that for any elements AAA and BBB
;; in LISTY, if (COMPARER AAA BBB) ==> #t, then AAA
;; comes before BBB in the output list.

> (sort '(5 2 1 3 3 2 5) <)  ; sort into non-decreasing order
(1 2 2 3 3 5 5)
> (sort '(5 2 1 3 3 2 5) >)  ; sort into non-increasing order
(5 5 3 3 2 2 1)
> (sort '(1 3 5 -2 -4 -6)  ; (lambda (x y) (> (* x x) (* y y)))
  (lambda (x y) (> (* x x) (* y y))))
(-6 5 -4 3 -2 1)

In the last case, the COMPARER predicate is specified by a lambda special form. The sorting function uses this predicate to sort the numbers such that their squares are non-increasing.

16.6 The Underlying Structure of Non-Empty Lists

Up to this point, we have seen that non-empty lists can often be effectively processed recursively using only the first and rest accessor functions. The reason for this is that the underlying structure of non-empty lists in Scheme is, in fact, based on decomposing them into their first and rest parts. The rest of this section explores that structure, revealing the central role of a data structure called a cons cell—also known as a pair.

16.6.1 Data Structures

In Computer Science, the term, data structure, refers to any organized (or structured) collection of data. Typically, each data structure has one or more slots for holding data. In some data structures, the slots for holding data are indexed so that any particular slot can be accessed by its corresponding (numerical) index. For example, the slots in vectors—to be discussed in Chapter 18—are indexed in this way. In other data structures, the slots for holding data are named so that any particular slot can be accessed by its name. Named slots are often called fields. For example, a bank-account data structure might have fields called password and balance. The rest of this section restricts attention to a very simple field-based data structure that, for historical reasons, is called a cons cell. Each cons cell has only two fields. For this reason, cons cells are also called pairs. General field-based data structures will be addressed thoroughly in Chapter 19.

16.6.2 Cons Cells (a.k.a. Pairs)

A cons cell is a field-based data structure structure that has only two fields: one named first, and one named rest. (Yes, that’s right! Stay tuned for the relationship between cons cells and non-empty lists.) Scheme provides the
following built-in functions for computing with cons cells, one of which we have already seen:

\[
\begin{align*}
\text{cons} & \quad \text{For constructing a new cons cell} \\
\text{cons?} & \quad \text{Type-checker predicate for cons cells}
\end{align*}
\]

---

**Example 16.6.1: The \texttt{cons} function revisited**

Here is a more accurate contract for the \texttt{cons} function. Notice that the second input need not be a list.

```scheme
;; CONS -- built-in function
;; ------------------------------------------------------
;; INPUTS: FST, RST, any Scheme data
;; OUTPUT: A cons cell whose FIRST field contains FST,
;; and whose REST field contains RST.
```

The following Interactions Window session demonstrates that the output generated by the \texttt{cons} function is indeed a cons cell, as confirmed by the built-in \texttt{cons?} type-checker predicate:

```scheme
> (cons 1 2)
(1 . 2)
> (cons? (cons 1 2))
#t
> (cons 'x "1232")
(x . "1232")
> (cons? (cons 'x "1232"))
#t
> (cons #t 'abc)
(#t . abc)
> (cons? (cons #t 'abc))
#t
```

Notice that if the output value is a cons cell, DrScheme displays the result using the dotted-pair notation. For example, a cons cell whose first field contains 1 and whose rest field contains 2 is displayed as (1 . 2) by DrScheme.

* DrScheme uses the dotted-pair notation when the rest field of a cons cell is something other than a list.
* The dotted-pair notation is not legal Scheme syntax; so we cannot use it in our Scheme programs or in the Interactions Window.

It must be stressed that:

* Although the dotted-pair notation shown above utilizes parentheses, it does not represent a list!

However:

* When the rest field of a cons cell contains a list, then that cons cell is a non-empty list!

In such cases, the Scheme datum is both a cons cell and a non-empty list. This does not contradict the statement made long ago—in Chapter 2—that a datum can only belong to one data type because:

* The set of non-empty lists is an example of a compound data type. Each non-empty list is, in fact, a cons cell that has special contents, in particular, one whose rest field contains a list.
Figure 16.1: The non-empty list, \( (3 \ 4 \ 6) \), as a single cons cell—with very particular contents

### Example 16.6.2: Cons cells vs. non-empty lists

The following interactions demonstrate that a non-empty list is a cons cell whose rest field contains a list, whereas a cons cell whose rest field contains some other kind of data is not a list.

```scheme
> (cons? '(2 3 4))
#t
> (list? (rest '(2 3 4)))
#t
> (cons? (rest '(2 3 4)))
#t
> (cons 1 2)
(1 . 2)
> (list? (cons 1 2)) ← A dotted pair is not a list
#f
```

Furthermore, as seen previously, when the rest field of a cons cell contains a list, DrScheme displays that cons cell using the familiar list notation:

```scheme
> (cons 1 '(2 3 4))
(1 2 3 4)
> (cons 'x '(y z))
(x y z)
> (cons 1 ())
(1)
```

Fig. 16.1 shows one way of depicting the non-empty list, \( (3 \ 4 \ 6) \)—namely, as a single cons cell having very particular contents. In this case, the list is indeed represented as a single cons cell—the biggest one in the picture. The first field in this cons cell contains the datum 3; the rest field of this cons cell contains another cons cell—one that represents the rest of the list (i.e., \( (4 \ 6) \)). The first field of that cons cell contains the datum 4; the rest field contains ... yet another cons cell! The first field of the innermost cons cell contains the datum 6; the rest field contains the empty list, which signals that we have reached the end of the list \( (3 \ 4 \ 6) \). Notice that the list represented by these three nested cons cells has three elements: 3, 4 and 6. Notice further that the first field of each cons cell contains one of the elements of the list.

* In general, if a list contains \( n \) elements, it can be represented by a nested structure of \( n \) cons cells.

### Example 16.6.3: The structure of non-empty lists

The following interactions demonstrate that a list containing \( n \) elements can be represented by a nested structure of \( n \) cons cells.
Although Fig. 16.1 provides an accurate depiction of the nested structure of cons cells that can be used to represent a non-empty list, this kind of picture would get awfully difficult to draw for lists containing more than, say, five or ten elements. For this reason, we prefer to depict non-empty lists as chains of cons cells, using arrows, as illustrated in Fig. 16.2. It is important to realize that the non-empty list depicted by this figure is the same list as that depicted in Fig. 16.1 (i.e., we have two kinds of picture-syntax for one semantic list!). Instead of showing the rest of the list as a cons cell nested inside the rest field, this depiction uses an arrow from the rest field of one cons cell to the next cons cell in the chain. Similarly, the rest field of the second cons cell points to the third cons cell in the chain. Finally, the rest field of the last cons cell, which contains the empty list, is often depicted as a box with an X in it, signalling the end of the chain.

So... is a non-empty list a single cons cell? Or is it a chain of cons cells? The answer is: it depends on how you look at it! For example, according to the cons? type-checker predicate, a non-empty list is most definitely a single cons cell:

```
> (cons? '(2 3 4))
#t
```

On the other hand, if the rest field of a given cons cell $C_1$ contains a nested cons cell $C_2$, then the thing that actually gets written into the rest field of $C_1$ in the computer’s memory is undoubtedly the address of $C_2$ (i.e., the location in the computer’s memory where $C_2$ can be found). In other words, the rest field of $C_1$ contains a pointer to $C_2$—which can be represented by an arrow, as in Fig. 16.2! In short, you can look at it both ways. For our purposes, thinking of non-empty lists as chains of cons cells will be most convenient.

### In-Class Problem 16.6.1: Defining our own type-checker predicate for lists

Define a predicate that satisfies the following contract:

```
;; WELL-FORMED-LIST?
;; -----------------------------------------------
;; INPUT:   DATUM, anything
;; OUTPUT:  #t if DATUM is an empty or non-empty list.
;;          If non-empty, DATUM should be a chain of cons
;;          cells, each of whose *rest* slot is filled by
;;          a well-formed list.
```
Here are some examples of its use:

\[
\begin{align*}
&> \text{well-formed-list? } () \text{\#t} \\
&> \text{well-formed-list? } '(a \ b \ c \ d) \text{\#t} \\
&> \text{well-formed-list? } (\text{cons} \ 1 \ (\text{cons} \ 2 \ 3)) \text{\#f} \\
&> \text{well-formed-list? } 'xyz \text{\#f}
\end{align*}
\]

Since this function is a predicate, you should be able to define it using \texttt{and}, \texttt{or}, \texttt{and-not}, \texttt{without using if \hspace{1mm} or \hspace{1mm} cond.}

* Now that we have explored the underlying structure of non-empty lists in terms of cons cells, you should review all of the examples from earlier in this chapter to make sure that you understand the underlying structures of the lists involved.

Example 16.6.4: The double-all function revisited

Recall the definition of the double-all function seen in Example 16.3.1 which takes a list of numbers as its input, and generates a list of the same length whose elements are obtained by doubling the corresponding elements from the input list.

\[
\begin{align*}
\text{(define double-all)} \\
&\quad (\lambda \text{listy}) \\
&\quad \text{(cond} \\
&\quad ;; \text{Base Case: LISTY is empty} \\
&\quad \text{((null? listy))} \\
&\quad ;; \text{The double-all of } () \text{ is ...} \\
&\quad () \\
&\quad ;; \text{Recursive Case: LISTY is non-empty} \\
&\quad \text{(else}} \\
&\quad ;; \text{Double the first element and attach it to the} \\
&\quad \text{double-all of the rest of the list} \\
&\quad \text{(cons } (* \ 2 \ (\text{first listy})\text{)} \\
&\quad \text{(double-all (rest listy))))})
\end{align*}
\]

Here’s an example of its behavior:

\[
\begin{align*}
&> \text{double-all } '(3 \ 1 \ 4 \ 7) \\
&\quad (6 \ 2 \ 8 \ 14)
\end{align*}
\]

In general, the double-all function returns a list containing the same number of elements as its input. Equivalently, we may say that the double-all function is length preserving. This can be formally proved using the technique of mathematical induction; however, we shall content ourselves with a less formal analysis.

First, note that for any datum \(d\) and any list \(\ell\), the list \((\text{cons} \ d \ \ell)\) has one more element than \(\ell\). Thus, for example, the list \((3 \ 1 \ 4 \ 7)\), which is equivalent to \((\text{cons} \ 3 \ '(1 \ 4 \ 7))\), has one more element than \((1 \ 4 \ 7)\). But now consider \((\text{double-all } '(3 \ 1 \ 4 \ 7))\). By the recursive case, \((\text{double-all } '(3 \ 1 \ 4 \ 7))\) effectively evaluates to \((\text{cons} \ 6 \ (\text{double-all } '(1 \ 4 \ 7)))\), which has one more element than \((\text{double-all } '(1 \ 4 \ 7))\). Therefore, if we want to show that \((\text{double-all } '(3 \ 1 \ 4 \ 7))\) and \((3 \ 1 \ 4 \ 7)\) have the same number of elements, we need only...
show that (cons 6 (double-all '(1 4 7))) and (cons 3 '(1 4 7)) have the same number of elements, which is equivalent to showing that (double-all '(1 4 7)) and (1 4 7) have the same number of elements. But then, by a similar line of reasoning, this will hold if and only if (double-all '(4 7)) and (4 7) have the same number of elements. And that will hold if and only if (double-all '7) and 7 have the same number of elements. And that will hold if and only if (double-all ()) and () have the same number of elements. And that holds—since (double-all ()) evaluates to ()!

The technique described in the preceding example can be used to show that the built-in map function is also length preserving. For example, (1 2 3 4) and the list generated by evaluating (map facty '(1 2 3 4)) must have the same length.

In-Class Problem 16.6.2: Picturing the length preserving nature of double-all and map

Draw the chain of cons cells corresponding to the list (3 1 4 7). Draw a circle around the portion of that chain that corresponds to the rest of the list. Then draw the chain of cons cells corresponding to the list (6 2 8 14) generated by evaluating (double-all '(3 1 4 7)). Draw a circle around the portion of the chain corresponding to the rest of that list. Notice that the first cons cell in (3 1 4 7) is matched by the first cons cell in (6 2 8 14); and that the rest of the cons cells in (3 1 4 7) are matched by the rest of the output list (6 2 8 14) generated by the recursive function call. In other words, each call to double-all effectively consumes one cons cell from the input list and produces one cons cell in the output list. For that reason, the input and output lists must have the same number of cons cells and, hence, the same number of elements.

16.7 Hierarchical/Deep/Nested Lists

The syntax of Scheme expressions allows lists that contain other lists as elements. Indeed, lists may contain lists that contain other lists that contain other lists, and so on, to any desired depth.

* A list that has at least one element that is itself a list is called a hierarchical (or deep or nested) list.

* A list that does not contain any lists as elements is sometimes called a flat list.

For example, the expression (x (2 (3) 2) #t) denotes a hierarchical list whose three elements are: the symbol x, the subsidiary list (2 (3) 2), and the boolean #t. This section demonstrates that recursively processing hierarchical lists is frequently only slightly more complicated than recursively processing flat lists. Indeed, when recursively processing the items in a deep list, it often happens that one need only insert one extra case to handle the possibility that the item currently under consideration is itself a list.

⇒ By convention, functions that recursively process hierarchical lists frequently have names ending in an asterisk (e.g., sum-all* instead of sum-all).

Example 16.7.1: Summing the items in a hierarchical list

Summing all of the items in a hierarchical list turns out to be only slightly more involved that summing the items in a flat list. (You may wish to review the sum-all function defined in Example 16.2.2.) The contract for the hierarchical version, called sum-all*, is given below, followed by some examples of its use.

;; SUM-ALL*
;; -------------------------------------------------------------
;; INPUT: HLISTY, a (possibly hierarchical) list of numbers
;; OUTPUT: The sum of all of the numbers appearing anywhere
You may recall that the sum-all function contained a cond expression with two cases: a base case and a recursive case. Below, the sum-all* function includes an extra recursive case that handles the possibility that the item currently under consideration (i.e., \(\text{first hlisty}\)) is itself a list.

\[
\text{(define sum-all* }
\text{ (lambda (hlisty)}
\text{ (cond}
\text{ ;; Base Case: HLISTY is empty}
\text{ ((null? hlisty)}
\text{ 0))}
\text{ ;; Recursive Case 1: First element of HLISTY is a list}
\text{ ((list? (first hlisty))}
\text{ (+ (sum-all* (first hlisty))}
\text{ (sum-all* (rest hlisty))))}
\text{ ;; Recursive Case 2: First element of HLISTY is not a list}
\text{ (else)}
\text{ (+ (first hlisty)}
\text{ (sum-all* (rest hlisty))))}
\text{))}
\]

Notice that when \((\text{first hlisty})\) is itself a list, it follows that both \((\text{first hlisty})\) and \((\text{rest hlisty})\) are lists. Therefore, the sum-all* function can be recursively applied to both of these lists, and the results added together to generate the desired sum. For example, if hlisty is the list \((1 2 (3)) 4 (5 1))\), then \((\text{first hlisty})\) is the list \((1 2 (3))\) and \((\text{rest hlisty})\) is the list \((4 (5 1))\). Recursively applying sum-all* to these two lists yields the results, 6 and 10, respectively. The sum of these two numbers (i.e., 16) is the sum of all of the numbers in hlisty.

\[\Rightarrow \text{Notice that, as usual, we let the recursive function calls do most of the work!}\]

Note. Using the list? predicate (e.g., in Recursive Case 1, above) to check whether \((\text{first hlisty})\) is a list can be terribly inefficient because, in cases where \((\text{first hlisty})\) happens to be a long list, the list? predicate will walk down its entire length, checking that it is a well formed chain of cons cells. Instead, if we assume that hlisty does not contain any malformed chains of cons cells, we can greatly increase the efficiency of Recursive Case 1 by using the quick-list? predicate, defined below.

\[
\text{;; QUICK-LIST?}
\text{;; -----------------------------------------------}
\text{;; INPUT: DATUM, anything}
\text{;; OUTPUT: #t if DATUM is either () or a cons cell;}
\text{;; #f otherwise.}
\text{(define quick-list?}
\text{ (lambda (datum)}
\text{ (or (null? datum) (cons? datum))))}
\]

Unlike list?, the quick-list? predicate does not walk down any chains of cons cells; instead, if datum is a cons cell, it simply assumes that it is the first cons cell in a well formed chain (i.e., that it is a non-empty list).
The rest of the examples in this section assume that all hierarchical lists are well formed (i.e., that they do not contain any malformed chains of cons cells).

**Example 16.7.2: Top-level elements vs. leaf items in hierarchical lists**

Recall In-Class Problem 16.2.2, whose goal was to define a function to compute the number of elements in a flat list. Here is one solution:

```scheme
;; LENGTHY
;; -------------------------------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: The number of elements of LISTY (i.e., its length)
(define lengthy
  (lambda (listy)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
        ;; No elements in the empty list
        0)
      ;; Recursive Case: LISTY is non-empty
      (else
        ;; (FIRST LISTY) is one element; the recursive
        ;; function call counts the REST of the elements
        (+ 1 (lengthy (rest listy)))))))
```

As demonstrated below, the `lengthy` function does not care whether the individual elements of `listy` are symbols, numbers, booleans, or … even other lists! Thus, it counts what we sometimes call the top-level elements of `listy`.

```
> (lengthy '(a b c d e))
5
> (lengthy '(x (1 1) (2 (3) 2) y))
4
> (lengthy '((((((3 3 3)))))))
1
```

For contrast, the function, `num-leaf-items*`, counts the number of so-called leaf items in a possibly hierarchical list—that is, the items that appear at any level of the hierarchy.

```scheme
;; NUM-LEAF-ITEMS*
;; ------------------------------------------------------
;; INPUT: HLISTY, a (possibly hierarchical) list
;; OUTPUT: The number of items that appear in HLISTY
;; at any level of the hierarchy.
```

Here is how `num-leaf-items*` treats the same lists encountered above:

```
> (num-leaf-items* '(a b c d e))
5
> (num-leaf-items* '(x (1 1) (2 (3) 2) y))
7
> (lengthy '((((((3 3 3)))))))
3
```
Notice that for flat lists such as `(a b c d e)`, where each item occurs as a top-level element, `num-leaf-items*` outputs the same answer as `lengthy`. However, `num-leaf-items*` treats hierarchical lists much differently. Note that it does not count subsidiary lists, but only the primitive data that appear within them. Thus, `(num-leaf-items* '(x (1 1) (2 (3) 2) y))` outputs 7, for the seven leaf items: `x, 1, 1, 2, 3, 2 and y`.

Although `num-leaf-items*` descends into the hierarchy of the input list, counting all the leaf items it finds along the way, defining this function is not difficult—as long as we let recursive function calls do most of the work! The following solution demonstrates that `num-leaf-items*` need only include one additional case, to handle the possibility that the element currently under consideration is itself a list:

```scheme
(define num-leaf-items*
  (lambda (hlisty)
    (cond
      ;; Base Case: HLISTY is empty
      ((null? hlisty) 0)
      ;; Recursive Case 1: (FIRST HLISTY) is itself a list!
      ((quick-list? (first hlisty))
        ;; Recursive calls on (FIRST HLISTY) and (REST HLISTY)
        ;; compute the numbers of items in each part of HLISTY.
        (+ (num-leaf-items* (first hlisty))
           (num-leaf-items* (rest hlisty))))
      ;; Recursive Case 2: (FIRST HLISTY) is NOT a list
      (else
        ;; Count 1 for (FIRST HLISTY); let the recursive
        ;; function call count the items in (REST HLISTY).
        (+ 1 (num-leaf-items* (rest hlisty)))))))
```

Notice that the Base Case and Recursive Case 2 are completely analogous to the Base Case and Recursive Case for `lengthy`. The only difference is the insertion of Recursive Case 1, which handles the possibility that `(first hlisty)` is itself a list. And that case is easily handled because, in that case, `(first hlisty)` and `(rest hlisty)` are both lists. Recursively applying `num-leaf-items*` to both of those lists, and then summing the results, gives the desired answer.

---

**In-Class Problem 16.7.1: A hierarchical version of the map function**

Define a function, called `map*`, that satisfies the following contract:

```scheme
;; MAP*
;; ---------------------------------------------
;; INPUTS:  FUNC, a function that expects one input
;;          HLISTY, a (possibly hierarchical) list of
;;          suitable inputs for FUNC
;; OUTPUT: A list with the same structure as HLISTY, where
;;         each item is obtained by applying FUNC to the
;;         corresponding item in HLISTY.
```

Here are some examples of its behavior:

```scheme
> (map* abs `((((-1) (2 -3) (-4 ((5)))))
 (((1) (2 3) (4 ((5))))))
> (map* (lambda (x) (* x x)) '(1 (2 (3 4 5) 6) 7))
(1 (4 (9 (16 25) 36) 49)
```
### In-Class Problem 16.7.2: Flattening a hierarchical list

Define a function, called `flatten`, that satisfies the following contract:

```scheme
;;; FLATTEN
;;; ------------------------------------------------------------
;;; INPUT: HLISTY, a (possibly hierarchical) list
;;; OUTPUT: A flat (i.e., non-hierarchical) list that contains
;;; all of the items from HLISTY "in the same order".
```

Here are some examples of its behavior:

```scheme
> (flatten* '((4 2) 3 (x (y))))
(4 2 3 x y)
> (flatten* '(1 (2 (3) 4) 5))
(1 2 3 4 5)
```

Hint: In one case, use the built-in `append` function; in another, use `cons`.

### 16.8 Functions that can be Applied to Variable Numbers of Inputs

Recall that many of the built-in functions can be applied to variable numbers of inputs. For example, the built-in addition and multiplication functions can each be applied to zero or more inputs, as illustrated below.

```scheme
> (+) ← Adding no numbers together yields zero
0
> (+ 10 20)
30
> (+ 100 10 1)
111
> (+ 1000 200 30 4)
1234
> (*) ← Multiplying no numbers together yields one
1
> (* 1 2 3 4 5)
120
> (* 10 10 10)
1000
```

Similarly, the built-in subtraction and division functions can each be applied to one or more inputs.

Given that function application in Scheme is provided through the evaluation of non-empty lists, it might not surprise you to learn that defining a function that can be applied to variable numbers of inputs can be handled by collecting the variable number of inputs into a list. As will be seen below, a slight extension to the syntax for the `lambda` special form enables this new capability.

**Extending the Syntax of the lambda Special Form.** In addition to the syntax shown in Chapter 7, the `lambda` special form also supports the following syntax.

```
(lambda args
  expr_1
  expr_2
  ...
  expr_k)
```
where \( \textit{args} \) can be any symbol expression. When such a function is applied to some number of inputs, those inputs are packaged together into a list, and that list of inputs becomes the value for the symbol \( \textit{args} \) in the local environment inside the function-call box. Thus, inside the function-call box, this function behaves as though it received a list as its only input.

**Example 16.8.1: Defining a function that can be applied to a variable number of inputs**

For this example, we aim to define a function that can take any number of numerical inputs. To make things simple, this function will simply multiply those inputs together. We begin by defining a similar function that takes a single input that contains a list of numbers.

;; MY-MULTY
;; -----------------------------------------------
;; INPUTS: A list of numbers
;; OUTPUT: The product of the numbers in that list
(define my-multy
  (lambda (listy)
    (if (null? listy)
        1
        (* (first listy)
           (my-multy (rest listy))))))

With \( \textit{my-multy} \) in hand, the desired function, \( \textit{my-multy-multy} \), can be easily defined using the new syntax for the \( \textit{lambda} \) special form, as follows.

;; MY-MULTY-MULTY
;; -----------------------------------------------
;; INPUTS: Any number of numbers
;; OUTPUT: The product of those numbers
(define my-multy-multy
  (lambda args
    ;; Since ARGS is a LIST of numbers...
    (my-multy args)))

The following interactions demonstrate the difference between \( \textit{my-multy} \) and \( \textit{my-multy-multy} \).

\[
\begin{align*}
> & \ (\text{my-multy '}(1 \ 2 \ 3 \ 4)) \\
&& 24 \\
> & \ (\text{my-multy-multy} \ 1 \ 2 \ 3 \ 4) \\
&& 24 \\
> & \ (\text{my-multy '}(10 \ 10 \ 10)) \\
&& 1000 \\
> & \ (\text{my-multy-multy} \ 10 \ 10 \ 10) \\
&& 1000
\end{align*}
\]

Using the above example as a guide, we could convert any function that takes a single input that is a list into an equivalent function that can be applied to inputs that are drawn from such a list. However, we can also take a more direct approach to defining a function like \( \textit{my-multy-multy} \) by using the built-in \( \textit{apply} \) function.
**Example 16.8.2: The built-in apply function**

*The built-in apply function satisfies the following contract:*

```scheme
;; APPLY -- built-in
;; --------------------------------------------------------------
;; INPUTS: FUNC, a function
;; LISTY, a list of suitable inputs for FUNC
;; OUTPUT: The result of applying FUNC to the inputs in LISTY
```

The following interactions demonstrate the difference between applying a function (e.g., the built-in addition function) to a variable number of inputs versus using apply to apply that same function to the elements of a given list.

```
> (+ 100 10 1)
111
> (apply + '(100 10 1))
111
> (* 1 2 3 4 5)
120
> (apply * '(1 2 3 4 5))
```

There is little mystery behind the built-in apply function...

**Example 16.8.3: Implementing our own version of apply**

```scheme
;; MY-APPLY
;; ------------------------------------------------------
;; INPUTS: FUNC, a function
;; LISTY, a list of suitable inputs for FUNC
;; OUTPUT: The result of applying FUNC to the elements of LISTY

(define my-apply
  (lambda (func listy)
    (eval (cons func listy)))))
```

```
> (my-apply + '(1 2 3 4))
10
> (my-apply * '(10 10 10))
1000
```

**Example 16.8.4: A more direct approach to defining a function that can be applied to a variable number of inputs**

```scheme
;; MY-MULTY-MULTY-V2
;; --------------------------------------------------------------
;; INPUTS: Any number of numerical inputs
;; OUTPUT: The product of those numbers
```
(define my-multy-multy-v2
  (lambda args
    (cond
      ;; Base Case: ARGS is empty
      ((null? args)
        1)
      ;; Recursive Case: ARGS is non-empty
      (else
        (* (first args)
           ;; Since (REST ARGS) is a LIST of numbers...
           (apply my-multy-multy-v2 (rest args)))))))

Note the use of apply in the last line. It is needed because my-multy-multy-v2 is supposed to be applied to any number of numerical inputs, not a single input that is a list of numbers. Here are some examples of my-multy-multy-v2 in action.

> (my-multy-multy-v2 1 2 3 4)
24
> (my-multy-multy-v2 10 10 10)
1000

Special Forms Introduced in this Chapter

* time Displays timing information

Built-in Functions Introduced in this Chapter

* abs Computes the absolute value of its input
* first, rest Accessor functions for lists
* cons Create a new list by attaching a new item to the front of a given list
* cons? Type-checker for cons cells
* second, third, fourth, etc. Additional accessor functions for lists
* list Create a list containing the specified items
* member Does an item appear in a list?
* map Apply given function to each element of a list, in turn
* length Compute the number of elements in a list
* list-ref Fetch the \textit{n}^{th} element of a list—general purpose accessor function
* append Concatenate two lists
* reverse Reverse the elements of a list
* sort Sort a list according to a given comparison function
* apply Apply a function to the elements of a given list