7 Special Forms

7.1 The define Special Form .............................................. 34
  7.1.1 The Syntax of the define Special Form ...................... 34
  7.1.2 The Semantics of the define Special Form .................. 34

7.2 The quote Special Form .............................................. 37
  7.2.1 The Syntax of the quote Special Form ....................... 37
  7.2.2 The Semantics of the quote Special Form ................... 37
  7.2.3 Alternate Syntax for quote Special Forms .................... 39

7.3 Summary ............................................................... 39

8 Predicates

8.1 Type-Checker Predicates ............................................. 41

8.2 Comparison Predicates .............................................. 43

8.3 Summary ............................................................... 44

9 Defining Functions

9.1 Defining Functions vs. Applying Them to Inputs ................... 47

9.2 The lambda Special Form ............................................ 48

9.3 The Syntax and Semantics of Lambda Expressions ................. 50
  9.3.1 The Syntax of a Lambda Expression .......................... 50
  9.3.2 The Semantics of a Lambda Expression ....................... 51
  9.3.3 Applying a lambda Function to Input Values ............... 52

9.4 Summary ............................................................... 57

10 Some practicalities

10.1 More Fun with the Built-in printf Function ........................ 59

10.2 The Built-in load Function ......................................... 62

10.3 Comments .................................................................. 63

10.4 Summary ............................................................... 65

11 Conditional Expressions I

11.1 The if Special Form ................................................. 67

11.2 Simplifying Conditional Expressions ............................... 70

11.3 The when Special Form .............................................. 73

11.4 Summary ............................................................... 75

12 Recursion I

12.1 Defining Recursive Functions in Scheme ............................ 77

12.2 Summary ............................................................... 82

13 Conditional Expressions II

13.1 The cond Special Form .............................................. 83

13.2 Boolean Operators: AND, OR and NOT ............................ 88
  13.2.1 The Built-in not Function ..................................... 89
  13.2.2 The and Special Form ......................................... 90
  13.2.3 The or Special Form ............................................ 92

13.3 Defining Predicates using Boolean Operators ..................... 93

13.4 Simplifying Conditional and Boolean Expressions ............... 95

13.5 Summary ............................................................... 96
14 Recursion II
14.1 Recalling the Factorial Function .......................................................... 99
14.2 Tail Recursion .......................................................................................... 101
14.3 Accumulators ......................................................................................... 105
14.4 Wrapper Functions .................................................................................. 113
14.5 Summary ................................................................................................ 116

15 Local Variables, Local Environments ....................................................... 119
15.1 The let Special Form .............................................................................. 119
  15.1.1 The Syntax of the let Special Form ..................................................... 120
  15.1.2 The Semantics of the let Special Form .............................................. 121
15.2 Flipping Coins and Tossing Dice .............................................................. 124
# 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
3.1 The evaluation function in Scheme .................................................................................................. 12
3.2 Sample evaluations in Scheme ......................................................................................................... 12
3.3 A sample environment ..................................................................................................................... 13
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 \( \text{lambda} (x) \ (\times \ x) \) to the value 4 .................. 53
9.2 Applying the sphere-volume function to the value 2 ................................................................. 54
12.1 DrScheme’s evaluation of \( \text{facty-v1} 3 \) ..................................................................................... 81
13.1 Truth tables for AND, OR and NOT, expressed using truth values (above) and binary values (below) 89
14.1 DrScheme’s evaluation of \( \text{sum-to-n-acc} 3 \ 0 \) ........................................................................ 108
15.1 Evaluating a \texttt{let} special form ................................................................................................. 121
15.2 Evaluating the expression, \( \text{let} ((\ x \ (+ \ 2 \ 3))) \ (\times \ x) \), from Example 15.1.2 .... 122
15.3 Demonstrating the higher priority of the local environment in Example 15.1.4 .......... 123
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 faladfa are not legal words, and “Person tall told a me joke” 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.

![Figure 1.1: A function with no side effects (left) vs. a function with side effects (right)](image-url)
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 ??, 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; and 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 divided by six`. 
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:

\[ \text{Character Sequence} \rightarrow \text{Datum} \]

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

\[
\begin{align*}
3 & \rightarrow \text{the number three} \\
-44 & \rightarrow \text{the number negative forty-four} \\
85/6 & \rightarrow \text{the number eighty-five sixths}
\end{align*}
\]

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.

\[
\begin{align*}
0 & \rightarrow \text{the number zero} \\
000 & \rightarrow \text{the number zero} \\
000000 & \rightarrow \text{the number zero}
\end{align*}
\]

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.

\[ () \rightarrow \text{the empty list} \]

(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 display a bunch of textual 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.

In other words, there is no Scheme expression that can be put into the box below to denote the void datum.

\[ \square \rightarrow \text{the void datum} \]

How, then, can we get our hands on the void datum should we ever want to? That’s an open question for now.

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.
Again, it is important to keep in mind the difference between the typewritten character sequences (e.g., \textit{hello} and \textit{bye-bye}) and the symbols (i.e., the Scheme data) that they denote (e.g., the symbol \textit{hello}, and the symbol \textit{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 \textit{hello} does not have any parts; it is indivisible. It may help to think of it as a billiard ball with \textit{hello} written on it.

2.2 String Expressions

This section introduces the \textit{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 \textit{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, "\textit{hi}" and "\textit{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*}
"\textit{hi}" & \quad \rightarrow \quad \text{the string "\textit{hi}"} \\
"\textit{Howdy!}" & \quad \rightarrow \quad \text{the string "\textit{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 \textit{void} datum, symbols, and strings. Examples of legal expressions that denote these kinds of data are given below.

- \textbf{Numbers:} \quad 342, -81, 34/9, 21.832, etc.
- \textbf{Booleans:} \quad \#t, \#f
- \textbf{The empty list:} \quad ()
- \textbf{Symbols:} \quad \textit{x}, \textit{miles-per-gallon}, \textit{dollarsPerGallon}, *, +, /, etc.
- \textbf{Strings:} \quad "\textit{hi}"", "\textit{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 \textit{thirty-four} is represented by: \textbf{34} \textbf{$\rightarrow$} \textbf{the number \textit{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 \textit{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 \textit{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 in Fig. 3.1.

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, ?? and ??.

3.1 Evaluating Numbers, Booleans, the Empty List, the void Datum, and Strings

The evaluation function acts like the identity function when applied to numbers, booleans, the empty list, the void datum, or strings, as illustrated in Fig. 3.2. Since drawing the kinds of black boxes shown in Figs. 3.1 and 3.2 takes up so much space, from now on we’ll use a simpler, text-based notation to represent the application of the evaluation function to some datum, as illustrated below.

\[
\text{Input Datum} \implies \text{Output Datum}
\]

The double arrow (\(\implies\)) is reserved solely for representing the application of the 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 evaluation function generates the output datum when applied to a certain input datum, we may say that the output datum is the result of evaluating the input datum (or that the input datum 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 evaluation function.
Here are some more examples illustrating the trivial behavior of the evaluation function when applied to numbers, booleans, the empty list, the void datum, or strings:

- the number zero  \(\Rightarrow\) the number zero
- the boolean true  \(\Rightarrow\) the boolean true
- the empty list  \(\Rightarrow\) the empty list
- the void datum  \(\Rightarrow\) the void datum
- the string “hi there”  \(\Rightarrow\) the string “hi there”

Of course, if the evaluation function acted like the identity function for 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 important cases where the evaluation function does something a little more interesting.

### 3.2 Evaluating Symbols

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

**Environments in Scheme.** In Scheme, symbols are evaluated with respect to an environment. For example, the symbol \(x\) might evaluate to the number three in one environment, but to the boolean false in another environment. Although the word environment may sound mysterious, an environment in Scheme is really nothing more than a table of entries, where each entry pairs a symbol \(s\) with its value \(v\). For example, the sample environment illustrated in Fig. 3.3 pairs the symbol num with the value three, and the symbol xyz with the value two.
The value of a symbol \( s \) with respect to some environment is simply whatever datum appears in the entry for the symbol \( s \) in that environment. (If there is no such entry, then the value for \( s \) with respect to that environment is undefined.)

The value associated with a symbol in an environment can be a Scheme datum of any type.

For example, the symbol \( \text{num} \) evaluates to the number \( \text{three} \) with respect to the environment shown in Fig. 3.3. Similarly, the symbol \( \text{xyz} \) evaluates to the number \( \text{two} \) in that environment; and the symbol \( \text{boolie} \) evaluates to the boolean \( \text{true} \).

For another example, if an environment \( E_0 \) contains an entry that pairs the symbol \( \text{xyz} \) with the number \( \text{two} \), while another environment \( E_1 \) contains an entry that pairs the symbol \( \text{xyz} \) with the boolean \( \text{false} \), then evaluating \( \text{xyz} \) with respect to the environment \( E_0 \) will yield the number \( \text{two} \), while evaluating \( \text{xyz} \) with respect to \( E_1 \) will yield the boolean \( \text{false} \).

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 table 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 in that table.

### Evaluating symbols in the Global Environment.

If the Global Environment contains an entry that pairs the symbol \( \text{xyz} \) with the number \( \text{two} \), then the symbol \( \text{xyz} \) will evaluate to the number \( \text{two} \):

\[
\text{the symbol } \text{xyz} \implies \text{the number two}
\]

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

\[
\text{the symbol } \text{pq} \implies \text{the boolean true}
\]

\[
\text{the symbol } \text{my-empty-list} \implies \text{the empty list}
\]

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

\[
\text{the symbol } \text{foo} \implies \text{the symbol bar}
\]

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 \textit{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 relevant environment. The default environment is called the Global Environment.

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

\[
\text{the symbol } x \rightarrow \text{ the number } \textit{eighty-six}
\]

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:

\[
x \rightarrow \text{ the symbol } x \rightarrow \text{ the number } \textit{eighty-six}
\]
Chapter 4

Introduction to DrScheme

This chapter introduces the piece of software known as DrScheme.\(^1\) 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.

\[ \begin{align*}
C_{\text{in}} & \quad \Rightarrow \\
D_{\text{in}} & \quad \Rightarrow \\
D_{\text{out}} & \quad \Rightarrow \\
C_{\text{out}} &
\end{align*} \]

\(^1\)The DrScheme software is freely available from drscheme.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.
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:

```
> 3
3
> #t
#t
> ()
()   
> "Howdy!"
"Howdy!"
```

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:

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

"Howdy!" \rightarrow \text{[ the string “Howdy!” } \equiv \text{ the string “Howdy!” } \rightarrow "Howdy!"

**Example 4.1.2**

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

\[
\begin{align*}
> 0 & \rightarrow 0 \\
> 000 & \rightarrow 0 \\
> 000000 & \rightarrow 0
\end{align*}
\]

*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. (Reread 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:
18 Introduction to Computer Science via Scheme  © 2019 Luke Hunsberger  Spring 2019

Figure 4.2: Using the Run button to evaluate (the data denoted by) multiple expressions

(1) the input character sequence $C_{\text{in}}$ denotes some Scheme datum $D_{\text{in}}$;

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

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

This process is concisely summarized by:

$$C_{\text{in}} \rightarrow [D_{\text{in}} \Rightarrow D_{\text{out}}] \rightarrow C_{\text{out}}$$

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:

\[
\begin{align*}
\text{the symbol } & \quad + \quad \rightarrow \quad \text{the addition function} \\
\text{the symbol } & \quad - \quad \rightarrow \quad \text{the subtraction function} \\
\text{the symbol } & \quad * \quad \rightarrow \quad \text{the multiplication function} \\
\text{the symbol } & \quad / \quad \rightarrow \quad \text{the division function}
\end{align*}
\]

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:

- `+` → [the `+` symbol] → the addition function → #<procedure:+>
- `-` → [the `-` symbol] → the subtraction function → #<procedure:->
- `*` → [the `*` symbol] → the multiplication function → #<procedure:*>
- `/` → [the `/` symbol] → the division function → #<procedure:/>

*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.*

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>$x_1, x_2, \ldots, x_n$; zero or more 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:
Introduction to Computer Science via Scheme  © 2019 Lake Hansberger  Spring 2019 21

| Name: | – |
| Inputs: | $x_1, x_2, \ldots, x_n$: one or more numerical inputs |
| Output: | If $n = 1$ (i.e., if there is only one input), then the output is $-x_1$ otherwise, the output is the value, $x_1 - x_2 - x_3 \ldots - x_n$ |
| Side Effects: | None |

For example, applying the subtraction function to the two inputs, 10 and 3, yields the output 7; but applying it to the single input 5 yields the output $-5$.

* 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:

| Name: | quotient |
| Inputs: | numer, denom, two integers |
| Output: | The (integer) answer that results from dividing numer by denom, ignoring any remainder. |

| Name: | remainder |
| Inputs: | numer, denom, two integers |
| Output: | The (integer) remainder left over from dividing numer by denom. |

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

| Name: | integer? |
| Input: | datum, anything |
| Output: | #t if datum is an integer; otherwise, #f |

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:

```scheme
> 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.

\[
\begin{align*}
\text{the addition function} & \implies \text{the addition function} \\
\text{the multiplication function} & \implies \text{the multiplication function} \\
\text{the evaluation function} & \implies \text{the evaluation function}
\end{align*}
\]

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: str, a string
Output: the 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 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 + symbol is associated with the built-in addition function; and the * symbol is associated with the built-in multiplication function. As a result, we can effectively refer to the built-in functions by name, as illustrated below:

\[ + \rightarrow [ \text{the + symbol} \implies \text{the built-in addition function} ] \rightarrow \text{#<procedure:+>} \]

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:

\[
(3 \ 4) \quad \rightarrow \quad \text{a list containing the number three and the number four}
\]

\[
(+ \ 3 \ 4) \quad \rightarrow \quad \text{a list containing the + symbol, the number three, and the number four}
\]

\[
(\text{eval} \ (+ \ 3 \ 4)) \quad \rightarrow \quad \text{a list containing:}
\]

\[
(1) \quad \text{the symbol eval, and}
\]

\[
(2) \quad \text{a subsidiary list containing the + symbol, the number three, and the number four}
\]

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
\[ E_1 \rightarrow D_1 \]
\[ E_2 \rightarrow D_2 \]
\[ \ldots \]
\[ E_n \rightarrow D_n \]

(i.e., 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 \textit{three}, and the number \textit{four}. Thus, the character sequence, \((+ \ 3 \ 4)\), is a legal Scheme expression that denotes a list containing the + symbol, the number \textit{three}, and the number \textit{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 \textit{three}, and the number \textit{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.

\[
\begin{aligned}
> (+ \ 2 \ 3) \\
5 \\
> (* \ 3 \ 4 \ 5) \\
60 \\
> (+ \ 2 \ (* \ 3 \ 10)) \\
32 \\
> (+ \ 2 \ (* \ 3 \ (+ \ 4 \ 8 \ 6))) \\
56
\end{aligned}
\]

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:

![Addition Function Diagram]

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:

- + symbol \(\Rightarrow\) 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:

```scheme
> (+ 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
Figure 6.1: The evaluation of the list denoted by $(+ \ 2 \ 3)$

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:

- the + symbol $\Rightarrow$ the addition function
- the number two $\Rightarrow$ the number two
- the subsidiary list $\Rightarrow$ oops!

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:

- the * symbol $\Rightarrow$ the multiplication function
- the number three $\Rightarrow$ the number three
- the number ten $\Rightarrow$ the number ten

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:

- the + symbol $\Rightarrow$ the addition function
- the number two $\Rightarrow$ the number two
- the subsidiary list $\Rightarrow$ the number thirty

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, \( K_1, K_2, \ldots, K_n \):
  
  \[
  D_1 \Rightarrow K_1 \\
  D_2 \Rightarrow K_2 \\
  \ldots \\
  D_n \Rightarrow K_n
  \]

- 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., \( L \Rightarrow P \)).

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:

\[
D_1: \text{the + symbol}, \quad D_2: \text{the number one}, \quad D_3: \text{the number two}, \quad D_4: \text{the number three}
\]

These Scheme data evaluate to the following:

\[
K_1: \text{the addition function}, \quad K_2: \text{the number one}, \quad K_3: \text{the number two}, \quad K_4: \text{the number three}
\]

Since the first of these, \( K_1 \), is in fact a function, it can be applied to the inputs \( K_2, K_3, K_4 \) (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:

\[
> (+ 1 2 3) \\
6
\]

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:

\[
\begin{align*}
> & \quad (\text{quotient} \ 17 \ 3) \\
& \quad 5 \\
> & \quad (\text{remainder} \ 17 \ 3) \\
& \quad 2
\end{align*}
\]

---

**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.

\[
\begin{align*}
> & \quad (\text{printf} \ \text{"hi there"}) \\
& \quad \text{hi there} \\
> & \quad (\text{printf} \ \text{"this is a long string!"}) \\
& \quad \text{this is a long string!}
\end{align*}
\]

*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.

\[
\begin{align*}
> & \quad (\text{void}) \\
& \\
\end{align*}
\]
Figure 6.2: DrScheme’s use of different colors to distinguish side-effect printing from output values

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:

\[
\text{eval symbol} \implies \text{the evaluation function} \\
\text{+ symbol} \implies \text{the addition function}
\]

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 as, 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. Its purpose is to insert a new symbol/value pair into the Global Environment.¹

7.1.1 The Syntax of the define Special Form

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

\[(\text{define } C_1 C_2)\]

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.

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

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 \emph{evaluated}. The special rule for evaluating define special forms is illustrated below:

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

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

* 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, \boxed{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: \boxed{s \quad E}. 
Input Datum
List containing:
define symbol
The \( s \) symbol
The datum \( e \)

Evaluation Function
no output!

Global Environment
symbol value
\( s \) \( E \)

side effect

(1) New entry inserted into Global Environment
(2) \( e \rightarrow E \)
(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 \emph{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:

\[
x \longrightarrow \text{the symbol } x
\]

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

Side Effect: New Global Environment Entry: \[ \begin{array}{c|c}
\text{the symbol } x & \text{the number six} \\
\end{array} \]

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:

\[ \text{Example 7.1.1} \]

The `define` special form can also be used to insert entries into a local environment, but we shall not explore this capability.
> x
BUG! reference to undefined identifier: x
> (define x (+ 1 2 3))
> x
6
> (* x 100)
600
> x
6
> (* x 1000)
6000
> (* x x)
36
> x
6
6

Notice that the first attempt to evaluate the symbol x resulted in an error; however, after the define special form has been evaluated, each time the symbol x needs to be evaluated, the result is the value six. The subsequent expressions can be evaluated using what we have learned in previous chapters. We need the Default Rule and we need to know how to evaluate symbols. No new rules are needed. Part of the beauty of Scheme’s computational model is that once it is learned, it can be used in an unbelievably wide variety of circumstances.

Example 7.1.2: Confirming the semantics of define

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

> w
BUG! reference to undefined identifier: w
> (define w w)
> w
>
Prior to evaluating the define special form, attempting to evaluate the symbol w results in an error, because there is no entry (yet) for w in the Global Environment. However, evaluating the define special form inserts an entry for w into the Global Environment. In particular, 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

\[(quote \ C)\]

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

- \[(quote \ x)\]
- \[(quote \ (1 2 3))\]
- \[(quote \ (hi there + #t ()))\]
- \[(quote \ (1 (2 (3))))\]

7.2.2 The Semantics of the quote Special Form

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

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

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:

* A list containing the quote symbol and \(D\) evaluates to \(\ldots D\).

Notice that, according to this rule, neither the quote symbol nor the datum \(D\) are evaluated. 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:

---

\(^2\)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.
> (quote x)
  x
> (quote (1 2 3))
  (1 2 3)
> (quote (+ 2 3))
  (+ 2 3)

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:

\[
\text{(quote x)} \mapsto \left\{ \{\text{list with symbols quote and x}\} \implies \text{the symbol x}\right\} \mapsto x
\]

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:

\[
\text{(quote } C \text{)} \mapsto \left\{ \{\text{list containing quote symbol and } D\} \implies D \right\} \mapsto C'
\]

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:

> (define x (+ 1 2 3))
> x
  6
> (quote x)
  x

**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.

> (define my-list (quote (1 2 3)))
> my-list
  (1 2 3)
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, $(\text{quote } C)$ and $’C$, denote the same two-element list—namely, a list containing the quote symbol and the datum $D$:

$$(\text{quote } C) \rightarrow \text{list containing quote symbol and } D$$

$’C \rightarrow \text{list containing quote symbol and } D$$

The two character expressions are quite different, but both represent the same list! (Syntax vs. Semantics!)

Example 7.2.4

The expressions, $’\text{num}$ and $(\text{quote } \text{num})$, each represent a list containing the quote symbol and the num symbol, as illustrated below:

```
> (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.

```
> (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 generates no output value, but has 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.

```scheme
> 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.
> void?
  #<procedure: void?>
> string?
  #<procedure: string?>

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?`.¹

¹This 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><code>number?</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td><code>d</code>, any Scheme datum</td>
</tr>
<tr>
<td>Output:</td>
<td><code>#t</code> if <code>d</code> 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? +)
#t
> (procedure? '+)
#f
> (list? '(+ 1 2))
#t
> (list? ())
#t
```
> (list? +)
#f
> (void? (void))
#t
> (void? void)
#f
> (string? "abc")
#t
> (string? '("a" "b" "c")
#f
> (string? #t)
#f

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.2 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.3

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

---

2In other contexts, these predicates are commonly called relational operators.
3These predicates can also be applied to more than two inputs; however, we shall postpone discussion of such things until Chapter ??.
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.

| Name: | `eq?` |
| 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.

```
> (eq? 3 3)
#t
> (eq? 3 'x)
#f
> (eq? 'x 'x)
#t
> (eq? 'x #t)
#f
> (eq? 'x ()
#f
> (eq? () ()
#t
```

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:

```
> (< 3 4)
#t
> (< (+ 2 3) (- 10 9))
#f
```

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:

```scheme
> (number? 3)
#t
> (number? 'a)
#f
> (list? 'a)
#f
> (list? '(a b c))
#t
```
> (number? 3)
#t
> (number? '(a b c))
#f
> (list? '(a b c))
#t

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?, null?, procedure?, void?, string?.
- **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 the function 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:

\[
(lambda (x) (* x x))
\]

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, \((\text{lambda} (x) (** x x))\), is typed into the Interactions Window, DrScheme will report that its evaluation yields a function, as illustrated below:

> \((\text{lambda} (x) (** x x))\)

#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:

> \(\text{define} \ \text{square} \ \text{(lambda} (x) (** x x))\)

> square

The \text{define} special form is used to create an entry in the Global Environment that associates the \text{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, \text{square}—is not evaluated; however, the given expression—in this case, \((\text{lambda} (x) (** x x))\)—is evaluated. Thus, the value associated with the \text{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 \text{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:

> \((\text{lambda} (x) (** x x)) \ 4)\)

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 1 3)
3.14159
> (cone-volume 10 1)
104.71966666666665
```

In the cone-volume function, 3.14159 is used as an approximation of \(\pi\), and the expression, \((\ast \ 1/3\ 3.14159\ \ r\ \ r\ \ h))\), 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:

\[
(lambda \ (C_1\ C_2\ \ldots\ C_n)\ \ B)
\]

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))
• (lambda (w h) (* w h))
• (lambda (r h) (* 1/3 3.14159 r r h))
• (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:
• (lambda (x y x) (* x y))
• (lambda (x 10) (* x 10))
• (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 Ci denotes a Scheme symbol, si;
- the symbols, s1, s2, ..., sn, 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, s1, s2, ..., sn; 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 (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 (lambda (x) (* x x)) evaluates to a function, when we really mean that the list denoted by the expression (lambda (x) (* x x)) evaluates to a function.
The Evaluation of a lambda Special Form

The most important thing to know about the evaluation of a lambda special form is that the result is invariably a function; however, the evaluation of a 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 lambda functions. Thus, a lambda function is a function that resulted from having evaluated a lambda special form.

Although evaluating a 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 lambda Function to Input Values

Up to this point, the only environment that we have considered has been the Global Environment. However, when a 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.

For the purposes of this chapter, it is assumed that the lambda function was created by evaluating its 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, \((\text{lambda } (x) \, (\times \, x))\). 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, \((\times \, x)\), 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 \((\times \, x)\) therefore yields the result 16, because \(x\) evaluates to 4 in the local environment, and \(\times\) 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 lambda function to the input value 4.

This process is illustrated in Fig. 9.1.

Notice that expressions in the body of a function can refer to data that are stored in one of two places:

1. the environment within which the function was created—in this case, the Global Environment; or
2. the local environment that contains entries associated with the input parameters.

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, \(\text{pi}\), 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.
Consider the evaluation of the expression, \( \text{(sphere-volume 2)} \). 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, \( (* \ 4/3 \ pi \ r \ r \ r) \), which constitutes the body of the function, is evaluated with respect to that local environment. In the process, the \( * \) symbol evaluates to the built-in multiplication function, \( 4/3 \) evaluates to itself, the symbol \( \text{pi} \) evaluates to 3.14159, and the symbol \( r \) evaluates to 2. Applying the multiplication function to the values 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 \( \text{sphere-volume function} \) to the input value 2.

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

* When evaluating a symbol such as \( r \) or \( \text{pi} \) 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 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 r. In contrast, changing the value of the global variable, pi, has disastrous effects! (That is one of many reasons why the use of global variables should be very carefully restricted!)

> (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:

> (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 \((\text{lambda } (x) \ (* \ x \ x))\) 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

```
> (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.

```
> (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., 1) 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

\( \text{(lambda (x) \(\ast\ x\ x\))} \)

is a function whose sole input parameter is \(x\), and whose body is \(\ast\ x\ x\).

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.

\[
\text{(define square (lambda (x) \(\ast\ x\ x\)))}
\]

The application of this function to an input is handled by the evaluation of an expression such as \(\text{(square 10)}\), 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 \(\ast\ x\ x\) in the body of the squaring function, \(x\) evaluates to 10, courtesy of the local environment, whereas \(\ast\) 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 \(\ast\ x\ x\), 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.

```scheme
> (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.

- The `printf` function reacts to the character sequence, `\%`, by moving to a new line in the Interactions Window (i.e., it interprets `\%` as a newline character). The `printf` function also interprets `\n` as a newline character.

- Whenever the `printf` function encounters either of the character sequences, `\s` or `\a`, in the input string, it treats them as place-holders for pieces of data to be displayed, as discussed in Example 10.1.1, below. The only difference between `\s` and `\a` is in how they cause string data to be displayed: `\s` causes the contents of a string to be displayed within double quotes; `\a` causes the contents of a string to be displayed without any double quotes.
Because the character sequences \%, \n, \s and \a, are not interpreted literally, but involve the \texttt{printf} function \textit{escaping} from a literal interpretation, they are frequently called \textit{escape sequences}. (And the characters \% and \n that introduce escape sequences are sometimes called escape characters.) Although the \texttt{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 \texttt{printf} function to generate nicely formatted text in the Interactions Window. For this reason, the input string is frequently called a \textit{format string}—which explains the \texttt{f} in \texttt{printf}.

In summary, the \texttt{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 \s or \a is replaced by a character sequence representing the \textit{value} of the corresponding input expression.

Notice that if the format string contains \textit{n} instances of \s, then there must be \textit{n} input expressions following the format string, as follows:

\begin{verbatim}
(printf format-string expr₁ ... exprₙ)
\end{verbatim}
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:

```
> (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:

```
> (printf "Show us void: ˜s˜%" (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!

Including multiple expressions within 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.

```
> (define verbose-func
  (lambda (a b)
    (printf "Hi. This is verbose-func!˜%"))
    (printf "The value of the first input is: ˜s˜%" a)
    (printf "The value of the second input is: ˜s˜%" 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:
```
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 "\"s ==> \" 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:
Suppose the file "test.txt" contains the following expressions:

```scheme
(printf "Loading test.txt!!")

(define tester
  (lambda (datum)
    (printf "s ==> " datum)
    (eval datum)))

(define x 34)
```

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 programs, 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.

**Example 10.3.1**

```scheme
(define tester
  (lambda (datum)
```

;; Print (the value of) DATUM -- without a newline character
(printf "˜s ==> " 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 "˜s ==> " 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 "s ==> " 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 "s ==> " datum)
    ;; Evaluate DATUM
    (eval datum)))

10.4 Summary

This chapter introduced 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 `\s` 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 `\s` is interpreted as a placeholder for a piece of data. For each occurrence of `\s` in the formatting string, there must be a corresponding additional input to printf. Thus, if the formatting string includes `n` occurrences of `\s`, then there must be `n` additional inputs to printf after the formatting string, as illustrated below:

```
> (printf "One: \s, Two: \s, Three: \s\%" 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

Solving problems often involves making decisions or choosing from a set of alternatives. For example, to determine the appropriate letter grade for a given exam score, one might reason as follows: “If the grade is at least 90, output A; otherwise, if the grade is at least 80, output B, and so on.” The simplest kind of programming decision is a binary decision (i.e., choosing one of two alternatives). In English, a binary decision can be represented by a sentence of the form, “If some condition holds, then do one thing; otherwise, do something else.” For example: “If it is raining, take an umbrella; otherwise, wear sunglasses.” In this case, the condition is whether or not it is raining; the then clause is “take an umbrella”; and the else clause is “wear sunglasses”.

In Scheme, programmers use the if special form to make binary decisions. The if special form is an example of a conditional expression. Like many conditional sentences in English, an if special form has a condition, a then clause, and an else clause. For example, (if (> x y) x y) is an instance of an if special form, where the condition is (> x y), the then clause is x, and the else clause is y. Like any special form, an if special form is evaluated in its own special way. Importantly, the evaluation of an if special form depends upon whether the condition evaluates to true.

Although a single if special form can only make a binary decision, multiple if special forms can be nested 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. Chapter 13 introduces the cond special form, which facilitates making n-ary decisions.

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:

\[
\text{(if \ condExpr \ thenExpr \ elseExpr)}
\]

where:

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

Example 11.1.1

The following expressions are legal examples of the if special form:

\[
\text{(if (> 2 4) (* 8 2) (* 6 5))} \\
\text{(if (> 4 2) 'then 'else)} \\
\text{(if #f "then" "else")}
\]

The semantics of the if special form stipulates that it is evaluated as follows.

- First, the condition, \textit{condExpr}, is evaluated. Then, depending on its value, one of the following will happen.
  - **Case 1:** \textit{condExpr} \textbf{evaluates to} \texttt{#t}. In this case, \textit{thenExpr} is evaluated — and the value of the if special form is whatever \textit{thenExpr} evaluates to.
  - **Case 2:** \textit{condExpr} \textbf{evaluates to} \texttt{#f}. In this case, \textit{elseExpr} is evaluated — and the value of the if special form is whatever \textit{elseExpr} evaluates to.

Notice that the condition, \textit{condExpr}, is always evaluated; however, after that, \textit{one and only one} of the remaining expressions, \textit{thenExpr} or \textit{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 \texttt{#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.

\[
\text{> (if (> 2 4) (* 8 2) (* 6 5))} \\
30 \\
\text{> (if (> 4 2) 'then 'else)} \\
\text{then} \\
\text{> (if #f "then" "else")} \\
"else"
\]

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

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

\textit{In the third expression, the condition, \texttt{#f}, evaluates to \texttt{#f}. Thus, the \texttt{else expression}, \texttt{"else"}, is evaluated. Its value, \texttt{"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.

;;; 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 true (i.e., #f is the only Scheme datum that counts as 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 false, the conditions in these examples all count as 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 expressions involving \texttt{eq?}

Suppose that \texttt{boolie} is a symbol whose value is a boolean (i.e., either boolean true or boolean false). Then, according to the above definition, the expression, \texttt{(eq? boolie \#t)}, is equivalent to the much simpler expression, \texttt{boolie}, as demonstrated below.

\begin{verbatim}
> (define boolie \#t)
> boolie
\#t
> (eq? boolie \#t)
\#t
> (define bully \#f)
> bully
\#f
> (eq? bully \#t)
\#f
\end{verbatim}

A similar simplification also works for any expression that evaluates to a boolean value:

\begin{verbatim}
> (eq? (> 5 2) \#t)
\#t
> (> 5 2)
\#t
> (eq? (= 3 2) \#t)
\#f
> (= 3 2)
\#f
\end{verbatim}

Similarly, if \texttt{boolie} is any expression that evaluates to a boolean, then an \texttt{if} expression whose condition is \texttt{(eq? boolie \#t)} can also be simplified.

\begin{verbatim}
> (define xyz \#t)
> (if (eq? xyz \#t) 'yes 'no)
yes
> (if xyz 'yes 'no)
yes
> (define abc \#f)
> (if (eq? abc \#t) 'yes 'no)
no
> (if abc 'yes 'no)
no
> (if (= 3 2) 'yes 'no)
no
> (if (> 5 2) 'yes 'no)
yes
\end{verbatim}

* In summary, if \texttt{boolie} is any expression that evaluates to a boolean, then the following simplifications yield equivalent expressions:

\[
\begin{align*}
\texttt{(eq? boolie \#t)} & \quad \sim \quad \texttt{boolie} \\
\texttt{(if (eq? boolie \#t) thenExpr elseExpr)} & \quad \sim \quad \texttt{(if boolie thenExpr elseExpr)}
\end{align*}
\]
Furthermore, in view of the non-strict version of the `if` special form, the latter simplification works for any expression `boolie`, whether it evaluates to a boolean or not.

**Example 11.2.2: Another way of simplifying `if` expressions**

According Defn. 11.1, the expression, \((\text{if } (> \ x \ y) \ #t \ #f)\), is equivalent to the simpler expression, \((> \ x \ y)\). The following interactions demonstrate the equivalence in two different environments: one where \(x > y\), and one where \(x < y\).

```scheme
> (define x 32)  ; Setting up an environment where \(x > y\)
> (define y 4)
> (if (> x y) #t #f)
#t
> (> x y)
#t

> (define x 32)  ; Setting up an environment where \(x < y\)
> (define y 1000)
> (if (> x y) #t #f)
#f
> (> x y)
#f
```

More generally, if `boolCond` is any boolean condition, then the following simplification yields an equivalent expression:

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

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 11.2.3: Simplifying `if` expressions: non-strict truth values**

```scheme
> (if ’happy #t #f)  ; ’happy ⇒ happy, which counts as true
#t
> ’happy
happy
> (if #f #t #f)  ; The condition #f ⇒ #f, which counts as false
#f
> #f
#f
```

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, \((\text{if } ’happy \ #t \ #f)\) 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)) #t
> (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: NUM = \"s and VERBOSE? = \"s\"
        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: NUM = 3 and VERBOSE? = #t
small
> (how-big-v2 3 #f)
small
> (how-big-v2 15 #t)
Inside HOW-BIG-V2: 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:

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

For example, \( f(4) = 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \); and \( f(5) = 5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120 \).

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** \((n = 1)\): \( 1! = 1 \) \((i.e., f(1) = 1)\)

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

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, \( \text{facty-v1} \), whose definition is based on the above insights. (The function is called, \( \text{facy-v1} \), because it is the first version of the factorial function we will look at.)

\[
;;; \text{FACTY-V1}
;;; -------------------------------------------------------
;;; INPUT: N, a positive integer
;;; OUTPUT: The factorial of N \((i.e., N \cdot (N-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1)\)

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

Notice that the define special form effectively gives the name, \( \text{facty-v1} \), 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.

First, 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:

```
(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, \((\times n (\text{facty-v1} (- n 1)))\), is evaluated—according to the Default Rule. The \(*\) symbol evaluates to the multiplication function, \( n \) evaluates to 3, and \((\text{facty-v1} (- n 1))\) 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 in the local environment). 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, \((\times (\text{facty-v1} (- n 1)))\), 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, \((\times n (\text{facty-v1} (- n 1)))\), 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, \((\times n (\text{facty-v1} (- n 1)))\), in the second box cannot continue until the output value for the third box has been generated. In other words, the evaluation 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 that the value of \((\text{facty-v1} (- n 1))\) is in hand, the evaluation of the expression, \((\times n (\text{facty-v1} (- n 1)))\), 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, \((\times n (\text{facty-v1} (- n 1)))\), 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.
Figure 12.1: DrScheme’s evaluation of \((\text{facty-v1 } 3)\)
• 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.

• 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.

```scheme
> (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:

```scheme
;;; 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 nesting 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))”. In Scheme, complicated conditions can be composed from simpler ones using the boolean operators and, or and not. Like the evaluation of the if special form, the evaluation of the cond, and and or special forms is lazy, in the sense that only the computations needed to ascertain the final value are actually performed.

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.

<table>
<thead>
<tr>
<th>Example 13.1.1: Nested if expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consider the following letter-grade function, implemented using a sequence of nested if special forms to distinguish four cases.</td>
</tr>
</tbody>
</table>

```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.
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
     ...)))
Notice that, as a matter of syntax, each case of the \texttt{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 \texttt{cond} (except the last one) is typically signalled by two left parentheses. For example, the first case is represented by the list \((>= \text{num} 90) \ 'A\). The first element of that list, \((>= \text{num} 90)\), 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 \texttt{letter-grade} is demonstrated below:

\begin{verbatim}
> (letter-grade-v2 93)
A
> (letter-grade-v2 82)
B
> (letter-grade-v2 74)
C
> (letter-grade-v2 61)
D
\end{verbatim}

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

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

\begin{itemize}
  \item In most programming circumstances, the last condition in a \texttt{cond} expression should be \texttt{#t}. This ensures that at least one of the conditions in the \texttt{cond} will evaluate to \texttt{#t}. As an alternative, the last condition in a \texttt{cond} can be the \texttt{else} keyword symbol, which serves the same purpose as \texttt{#t}.
  \item If all of the conditions in a \texttt{cond} evaluate to \texttt{#f}, then the entire \texttt{cond} expression will evaluate to \texttt{void}, something that is typically not desirable. Indeed, a \texttt{cond} that expression that evaluates to \texttt{void} typically signals that the programmer forgot about a possible case.
\end{itemize}

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

\begin{verbatim}
(cond
 (cond1
  expr1)
 (cond2
  expr2)
 ...
)
\end{verbatim}
Introduction to Computer Science via Scheme

(\text{cond}_n \backslash \text{expr}_n)

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 \text{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} special form, 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.

\textbf{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:

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

\begin{verbatim}
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.)
\end{verbatim}

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 `cond` expression, each condition, `cond`, 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 `lambda` expression, it is the value of the last subsidiary expression in the selected case that serves as the value of the entire `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!!!\n")
        'A)
      ;; Case 2: Got a B
      (>= num 80)
        (printf "Well, you know, a B is pretty good!!\n")
        (printf "Nothing to be ashamed of at all!!\n")
        'B)
      ;; Case 3: Got a C
      (>= num 70)
        (printf "The student handbook says a C is average!\n")
        (printf "Thus, your grade, \~s, is average!\n" num)
        'C)
      ;; Case 4: Something else
      (else
        (printf "Hmmm... Hard to find much positive to say here.\n")
        (printf "Maybe there's been a mistake...\n")
        (printf "But until we find it, your grade stands...\n")
        '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.

```
(when condExpr
  expr1
  expr2
  ...
  exprn)
```

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

```
(cond
  (condExpr
    expr1
    expr2
    ...
    exprn))
```

For each expression, when or cond, if the condition evaluates to something that counts as true, then each of the expressions, expr1, is evaluated in turn, and the value of the last expression, exprn, 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: AND, OR and NOT

This section introduces the boolean operators, AND, OR and 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—and and or—while not is provided as a built-in function. Here, the generic term 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 true or false—boolean operators are frequently defined using truth tables. A truth table simply lists the possible inputs together with the corresponding output. Since NOT takes only one input, there are two rows in its truth table. However, AND and 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 AND, OR and NOT. These truth tables indicate that:

- the AND operator takes two boolean inputs, and outputs true if both of its inputs are true;
- the OR operator takes two boolean inputs, and outputs true if at least one of its inputs is true; and
- the NOT operator takes a single boolean input, and outputs the opposite boolean value.
In boolean logic, 0 and 1 are frequently used to represent false and true, respectively. (Not so in Scheme!) The bottom row of the figure presents the truth tables for the boolean operators using the binary values 0 and 1, instead of true and false, respectively.

### 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.

```
Example 13.2.1

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

In the first four examples, the non-boolean input counts as 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 elements in the 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)
#f

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:

```scheme
;; 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:

```scheme
;; CIVIL-TO-MIL-HOURS
;; ------------------------------
;; INPUTS: CIVIL-HOURS, an integer from 1 to 12, inclusive
;; TIME-OF-DAY, a symbol, one of AM, PM, NOON or MIDNIGHT
;; OUTPUT: An integer from 0 to 23, inclusive, representing the
;; corresponding number of hours in military notation.
```

Here are a few examples of the desired behavior:

```scheme
> (civil-to-mil-hours 3 'am) 3
> (civil-to-mil-hours 3 'pm) 15
> (civil-to-mil-hours 12 'midnight) 0
```

Hints: Use `eq?` to test whether two symbols are equal (e.g., `am` and the value of the input `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, `and`, `or`, and `not`, instead of the conditional expressions, `if` or `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 `cafe-open?`, that satisfies the following contract:

```scheme
;; CAFE-OPEN?
;; ------------------------------
;; INPUTS: DAY, a symbol, one of SUN, MON, TUE, ..., FRI, SAT
;; AM-OR-PM, a symbol, either AM or PM
;; HOUR, an integer from 1 to 12, inclusive
;; MINUTES, an integer from 0 to 59, inclusive
;; OUTPUT: #t, if the inputs specify a time at which the
;; CAFE CMPU-101 is open; #f, otherwise.
```
* 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 \texttt{cafe-open?} can be written using the boolean operators, \texttt{and}, \texttt{or}, and \texttt{not}, instead of the conditional expressions, \texttt{cond} or \texttt{if}. When approaching the definition of a predicate in this way, the following advice can be very helpful:

\begin{itemize}
  \item The body of the predicate should specify the conditions under which the predicate will output \#t (or something that counts as true).
\end{itemize}

In the preceding example, each of the cases 1 through 4 of the \texttt{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, \texttt{cafe-open?-alt}:

\begin{verbatim}
(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))))
\end{verbatim}

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

\begin{verbatim}
> (if 'sad #f #t)
#f
> (not 'sad)
#f
\end{verbatim}

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

\begin{verbatim}
(if condExpr #f #t)  ~⇒  (not condExpr)
\end{verbatim}
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:

$$\text{and} (e_1 \text{ and } e_2 \text{ and } e_3) \sim \text{and} (e_1, e_2, e_3)$$
$$\text{or} (e_1 \text{ or } e_2 \text{ or } e_3) \sim \text{or} (e_1, e_2, e_3)$$

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:

$$\text{not} (\text{and} e_1 \text{ and } e_2) \sim (\text{or} (\text{not} e_1) \text{ or } \text{not} e_2)$$
$$\text{not} (\text{or} e_1 \text{ or } e_2) \sim (\text{and} (\text{not} e_1) \text{ and } \text{not} e_2)$$

### 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 ith 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:

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

**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 cond special form instead of the if special form.

Example 14.1.1

```
;; 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
      (else (* 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 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
```
Example 14.1.2

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

\begin{verbatim}
;; FACTY
;; -----------------------------------------------
;; INPUT: N, a positive integer
;; OUTPUT: The factorial of N (i.e., N\times(N-1)\ldots\times3\times2\times1)
;; 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 = \texttt{\textbar}s\texttt{\textbar})\n")
        (* n (facty (- n 1))))))
\end{verbatim}

Notice that the \texttt{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:

\begin{verbatim}
> (facty 3)
Recursive Case (n = 3)
Recursive Case (n = 2)
Base Case (n = 1)
6
\end{verbatim}

Notice how the side-effect printing helps to demonstrate that the evaluation of \texttt{(facty 3)} mirrors the evaluation of \texttt{(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{itemize}
  \item \textbf{Base Case (} \(n = 1\): \texttt{\textbar} \(g(1) = 1\)
  \item \textbf{Recursive Case (} \(n > 1\): \texttt{\textbar} \(g(n) = g(n-1) + n^2\)
\end{itemize}

Notice that \(g(1) = 1\), \(g(2) = 1^2 + 2^2 = 5\), \(g(3) = 1^2 + 2^2 + 3^2 = 14\), \texttt{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:

\[
\text{;; SUM-SQUARES} \\
\text{;; -----------------------------------------} \\
\text{;; INPUT: \(N\), a positive integer} \\
\text{;; OUTPUT: The sum 1*1 + 2*2 + \ldots + N*N} \\
\]

\[
\text{(define sum-squares} \\
\text{(lambda (n))} \\
\text{ (cond} \\
\text{ (\text{;; Base Case: \(n = 1\)})} \\
\text{ (\text{1}))} \\
\text{ (\text{;; Recursive Case: \(n > 1\)})} \\
\text{ (#t} \\
\text{ (\text{ (+ (sum-squares (- n 1)) (* n n)))))}) \\
\text{)} \\
\]

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

\[
> (\text{sum-squares 1)} \\
1 \\
> (\text{sum-squares 2)} \\
5 \\
> (\text{sum-squares 3)} \\
14 \\
> (\text{sum-squares 4)} \\
30 \\
\]

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, \(\text{(facty 5)}\), effectively requires evaluating \(\text{(facty 4)}, \text{(facty 3)}, \text{(facty 2)}\) and \(\text{(facty 1)}\). (It may help to recall Fig. 12.1.) Similarly, evaluating \(\text{(facty 100)}\) would involve a sequence of nearly one hundred recursive function calls. For functions such as \text{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 \(\text{(facty n)}\) 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, $B$ is its body, and expr is a recursive function-call expression somewhere within $B$. We say that expr is a tail-recursive function-call expression within $B$ if, whenever evaluating $B$ requires evaluating expr, it is necessarily the case that the last step in evaluating $B$ is the evaluation of expr and, thus, the value of $B$ is identical to the value of 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 rec-func is a recursive function whose body $B$ 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 rec-func can be tail recursive is if, as shown below, the recursive function-call expression, (rec-func ...), is the last (i.e., tail) expression within the recursive case.

```
(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:

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

```scheme
;; 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 \(3! = 6, 2! = 2\) and \(1! = 1\), 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 on successive calls of a recursive function.

As each recursive function call is made, the value of the accumulator input 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, \texttt{sum-to-n}:

```scheme
;; SUM-TO-N
;; ------------------------------------------------
;; INPUT: N, number (non-negative integer)
;; OUTPUT: The value of the sum 0 + 1 + 2 + ... + 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=\(~s\) ...\~" n)
     (+ n (sum-to-n (- n 1))))))

As in prior examples, the \texttt{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=\(\sim s\) ...\~" n)
(+ n (sum-to-n (- n 1)))
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.

```
;; 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=\textasciitilde s)\textasciitilde %" acc)
      ;; Return the accumulator!
      acc)
      ;; Recursive Case: n > 0
      (#t
       (printf "Recursive Case (n=\textasciitilde s, acc=\textasciitilde s)\textasciitilde %" 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:

```
(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=\"s\")\" acc)
     ;; Return the accumulator!
     acc)
    ;; Recursive Case: n > 1
    (#t
     (printf "Recursive Case (n=\"s\", acc=\"s\")\" n acc)
     ;; Recursive function call (tail-recursive)
     (facy-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 expres-
sion 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 `(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 \((\text{facty-acc} \ 4 \ 1)\), as illustrated below:

\[
\begin{align*}
> & (\text{facty-acc} \ 4 \ 1) \\
& \text{Recursive Case (n=4, acc=1)} \\
& \text{Recursive Case (n=3, acc=4)} \\
& \text{Recursive Case (n=2, acc=12)} \\
& \text{Base Case (n=1, acc=24)} \\
& \ 24
\end{align*}
\]

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 \((\text{facty-acc} \ n \ \text{acc})\) 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 \(n! \times \text{acc}\).

*In contrast, the non-tail-recursive function, facty, seen earlier, included the recursive function-call expression, \((\text{facty} \ (- n \ 1))\), within the larger expression, \((\times n \ (\text{facty} \ (- n \ 1)))\).

---

**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=\d, acc=\d\n" n acc)
        ;; Return the accumulator!
        acc)
      ;; Recursive Case: n > 0
      (#t
```

...
(printf "Recursive Case: n=`s, acc=`s``%" 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, \(\text{sum-squares-acc 3 0}\), in the Interactions Window:

\[
\begin{align*}
> (\text{sum-squares-acc 3 0}) & \quad \leftarrow 3^2 + 2^2 + 1^2 + 0^2 = 14 \\
\text{Recursive Case: n=3, acc=0} \\
\text{Recursive Case: n=2, acc=9} \\
\text{Recursive Case: n=1, acc=13} \\
\text{Base Case: n=0, acc=14} \\
14 
\end{align*}
\]

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, \(\text{sum-squares-acc n acc}\), 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 π**

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, the
;; output is the following estimate of PI:
;; 4 * (1 - 1/3 + 1/5 - 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!)
      ((> from n) ;; Multiply the accumulator by 4:
       (* 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)
3.161198612970506
> (approx-pi-acc 1 1 10001 0) ;; = 4*(1 - 1/3 + ... + 1/10001)
3.1417926135957908
> (approx-pi-acc 1 1 1000001 0) ;; = 4*(1 - 1/3 + ... + 1/1000001)
3.1415946535859922

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

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

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, 2.66666666666, . . .

;; 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 + 1/(1!) + 1/(2!) + 1/(3!) + ... + 1/(N!)
(define approx-e-acc
  (lambda (n indy curr-denom acc)
    ;; Print out the values of the input parameters first...
    (printf "n=\n, indy=\n, curr-denom=\n, acc=\n\n" n indy curr-denom acc)
    (cond
      ;; Base Case: INDY > N (we’re done!)
      ((> indy n)
       ;; Return the accumulator!
       acc)
      ;; Recursive Case: INDY <= N
      (else
       ;; 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)) ;; accumulate current term
        )))))

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

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: \( 0!, 1!, 2!, 3!, \ldots \). 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 \texttt{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:

```scheme
> (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 \textit{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!

```scheme
;;; FACTY-WR
;;; ------------------------------------------
;;; INPUT: N, a non-negative integer
;;; OUTPUT: The factorial of N (i.e., N!)

(define facty-wr
  (lambda (n)
    ;; Call the accumulator-based helper function with ACC=1
    (facty-acc n 1)))
```

The following \texttt{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.
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}
;; APPROX-PI-WR -- wrapper function for APPROX-PI-ACC
;; -----------------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: The value of the sum:
;; 1 - 1/3 + 1/5 - 1/7 + ... (+/-) 1/N

(define approx-pi-wr
 (lambda (n)
   (approx-pi-acc 1 1 n 0)))
\end{verbatim}

Here are some examples of its use:

> (approx-pi-wr 5)
3.466666666666667
> (approx-pi-wr 10001)
3.1417926135957908

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}
;; APPROX-E-WR -- wrapper function for APPROX-E-ACC
;; -----------------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: The value of the sum:
;; 1/1! + 1/2! + 1/3! + ... + 1/N!

(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:
> (approx-e-wr 4) 2.708333333333333
> (approx-e-wr 5) 2.716666666666663
> (approx-e-wr 6) 2.718055555555554
> (approx-e-wr 100) 2.7182818284590455

Notice that the user of 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.)

```scheme
;; 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, `\((* n (\text{facty } (- n 1)))\)`. 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.

(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
       (fancy-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 if a lambda function is created by evaluating a lambda expression within the Global Environment, then any time that function is applied to appropriate inputs (i.e., any time that function is called):

1. a function-call box is automatically created that contains a local environment $E'$ that houses variable/value entries for the relevant input parameters;

2. that local environment is nested inside the Global Environment; and

3. each expression in the body of the function is evaluated with respect to that local environment $E'$.

In the process, whenever any symbol $s$ needs to be evaluated, DrScheme looks for its value, first, in the local environment $E'$. If a matching variable/value entry is found, 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 $E$ (which may or may not be the Global Environment), the new local environment $E'$ that it creates is nested inside the parent environment $E$ (i.e., $E' \subset E$). 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

\(^1\)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.
used to store the result 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.

15.1.1 The Syntax of the let Special Form

The syntax of the let special form is as follows:

\[
(\text{let } ((\text{var}_1 \ \text{val}_1) \\
\quad (\text{var}_2 \ \text{val}_2) \\
\quad \ldots \\
\quad (\text{var}_n \ \text{val}_n)) \\
\quad \text{expr}_1 \\
\quad \text{expr}_2 \\
\quad \ldots \\
\quad \text{expr}_k)
\]

where:

- \(\text{var}_1, \ldots, \text{var}_n\) are character sequences that denote \(n\) distinct symbols that will serve as the local variables, where \(n \geq 0\);
- \(\text{val}_1, \ldots, \text{val}_n\) are \(n\) expressions of any kind, called the \textit{value expressions}; and
- \(\text{expr}_1, \ldots, \text{expr}_k\) are \(k\) Scheme expressions of any kind, where \(k \geq 1\).

The expressions, \(\text{expr}_1, \ldots, \text{expr}_k\), constitute the body of the let expression.

* Notice that a let can include \textit{zero or more variable/value pairs}; however, the body of a let must include \textit{at least one expression}.

Example 15.1.1: Some legal let expressions

The following expressions are all legal let expressions:

\[
(\text{let } () \ \#t)
\]

\[
(\text{let } ((x \ (+ \ 2 \ 3))) \\
\quad (* \ x \ x))
\]

\[
(\text{let } ((x \ (+ \ 2 \ 3)) \\
\quad (y \ 3) \\
\quad (z \ (* \ 2 \ 2))) \\
\quad (\text{printf } "x: \, s, y: \, s, z: \, s{s}%" \ x \ y \ z) \\
\quad (+ \ x \ y \ z))
\]

The first let expression includes no variable/value pairs, as indicated by the empty list. Its body consists of the single expression, \#t. The second let expression includes a single variable/value pair: \((x \ (+ \ 2 \ 3))\). Its body consists of the single expression, \((* \ x)\). The third let expression includes three variable/value pairs: \((x \ (+ \ 2 \ 3)), (y \ 3)\) and \((z \ (* \ 2 \ 2))\). 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 specifies how it is evaluated. When a let expression of the form

\[
\text{let } ((\text{var}_1 \ \text{val}_1) \\
(\text{var}_2 \ \text{val}_2) \\
\ldots \\
(\text{var}_n \ \text{val}_n)) \\
\text{expr}_1 \\
\text{expr}_2 \\
\ldots \\
\text{expr}_k
\]  

is evaluated with respect to some environment \( \mathcal{E} \), the following steps are taken, as illustrated in Fig. 15.1:

1. The value expressions, \( \text{val}_1, \ldots, \text{val}_n \), are evaluated with respect to the environment \( \mathcal{E} \), generating the values \( V_1, V_2, \ldots, V_n \).

2. A local environment \( \mathcal{E}' \) is then created containing \( n \) entries—one for each of the variable/value pairs. In particular, each symbol \( \text{var}_i \) is associated with the corresponding value \( V_i \). This new environment is nested inside \( \mathcal{E} \). In other words: \( \mathcal{E}' \subset \mathcal{E} \).

3. The expressions, \( \text{expr}_1, \ldots, \text{expr}_k \), in the body of the let special form are evaluated in order with respect to the newly created local environment, \( \mathcal{E}' \). In the process of evaluating these expressions, if any symbol \( \text{var}_i \) ever needs to be evaluated, its value is drawn from the local environment \( \mathcal{E}' \). All other symbols are evaluated with respect to the parent environment \( \mathcal{E} \).

4. The value of the entire let expression is \( E_k \) (i.e., the result of evaluating the last expression in the body).
Introduction to Computer Science via Scheme  © 2019 Luke Hansberger  Spring 2019

Example 15.1.2: Evaluating sample let expressions

The following Interactions Window session demonstrates the evaluation of the sample let expressions seen earlier.

```
> (let () #t)
#t
> (let ((x (+ 2 3)))
  (* x x))
25
> (let ((x (+ 2 3))
  (y 3)
  (z (* 2 2)))
  (printf "x: ~s, y: ~s, z: ~s\n\n(+ x y z))
(x: 5, y: 3, z: 4
(+ x y z))
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, as illustrated in Fig. 15.2, 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 that associate the value 5 with x, the value 3 with y, and 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 Variables vs. Global Variables

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)
```
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. 15.3.
The first expression creates a global variable \( x \) whose value is 3. The second expression is a let expression that contains two variable/value pairs: one for \( x \), and one for \( f \). According to the semantics of let expressions, the value expressions are evaluated first—with respect to the parent environment, which, in this case, is the Global Environment. In particular, as shown below, 10 evaluates to itself, and the lambda expression evaluates to a function. Next, a local environment \( E_2 \) is created that contains two entries: one in which \( x \) has the value 10, and one in which \( f \) has the recently created lambda function as its value. Finally, the body of the let expression (i.e., \((f (+ x x))\)) is evaluated with respect to the local environment \( E_2 \). In this environment, \( f \) evaluates to the recently created lambda function, and \((+ x x)\) evaluates to 20. Therefore, the Default Rule applies that lambda function to the input value 20. Since the lambda function was created in the Global Environment, the function-call box and its automatically created local environment \( E' \) are nested within the Global Environment—not within \( E_2 \). In the environment \( E' \), \( x \) evaluates to 3, and \( n \) evaluates to 20; therefore, the expression \((* x n)\) evaluates to 60. Therefore, the entire let expression evaluates to 60.

### 15.2 Flipping Coins and Tossing Dice

The following example introduces a destructive built-in function called \texttt{random} that has many uses, one of which is to demonstrate the need for the \texttt{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. It also can be used to illustrate the benefits of local variables.

**Example 15.2.1: The built-in random function**

Scheme includes a built-in function called \texttt{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 \texttt{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. Yes, 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 \texttt{let} special form (cf. Example 15.2.3, below).
The `random` function satisfies the following contract.

;;; RANDOM
;;; -----------------------------------------------
;;; INPUT: N, a positive integer
;;; OUTPUT: A pseudo-random number drawn from the set
;;;         \{0, 1, 2, \ldots, 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:

> (random 2) ← output will be 0 or 1
 0
> (random 2)
 1
> (random 2)
 0
> (random 6) ← output will be in \{0, 1, 2, 3, 4, 5\}
 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\}.

*There’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
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 destructive function such as \texttt{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 \texttt{toss-die}, which makes use of an impure function such as \texttt{random}, can itself become impure. In other words, the impurity of \texttt{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 \texttt{let} to store a randomly generated value**

The \texttt{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:

> (printf "My toss: " ~s~%" (toss-die))
3
> (printf "The square of my toss: " ~s~%" (* (toss-die) (toss-die)))
10

* Why? Because each time DrScheme evaluates \texttt{(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: \"s\"% toss)
  (printf "The square of my toss: \"s\" (* toss toss))
  (* toss 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 $E_1 \subset E_0$, where $E_0$ represents the Global Environment. 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 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.

<table>
<thead>
<tr>
<th>Example 15.3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (define x 100)</td>
</tr>
<tr>
<td>&gt; (let ((x 5))</td>
</tr>
<tr>
<td>(let ((x (* x x)))</td>
</tr>
<tr>
<td>(printf &quot;Innermost x: &quot;s&quot;% x))</td>
</tr>
<tr>
<td>(printf &quot;Middle x: &quot;s&quot;% x))</td>
</tr>
</tbody>
</table>
The let special form creates a local variable \( x \), in the environment \( E_1 \), whose value is 5. In the body of that let, the next 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 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 printf symbol in \( E_2 \) or \( E_1 \), its value is obtained from the Global Environment \( E_0 \). The second 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 remainder, then any expression being evaluated with respect to that local environment cannot access the built-in remainder function, because the local variable named remainder would have priority over the globally defined remainder function.

In general, a 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., \( E' \subset E \)). To evaluate a symbol \( s \) with respect to the new environment \( E' \), involves the following 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 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, \( E_n \subset E_{n-1} \subset \ldots \subset E_2 \subset E_1 \subset E_0 \), 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 \( E_n \).

Similar considerations apply to the local environment that is automatically created when a lambda function is applied to inputs. The main difference is:

* When a function \( f \) is applied to inputs, the local environment within which the body of that function is evaluated is nested inside the environment within which that function was created—regardless of the environment within which the function call expression is being evaluated!

In other words, if the function \( f \) was created by evaluating a lambda special form within an environment \( E \), then, when that function is applied to inputs, the local environment \( E' \) within which the body of \( f \) is evaluated is necessarily nested within the parent environment \( E' \subset E \).
Example 15.3.2

> (let ((x 5))
  (let ((f (lambda (n)
            (printf "Inside function body: x = \%s\" x)
            (* x n)))
      (let ((x 10))
        (f x))))
Inside function body: x = 5
50

As illustrated above, evaluating the first let creates a local environment $E$ in which $x$ has the value 5. Next, the second let expression is evaluated with respect to the environment $E$. According to the semantics for let expressions, the value for $f$ is obtained by evaluating the appropriate expression with respect to the parent environment $E$. In this case, that means evaluating the given lambda expression with respect to $E$. This function, which was created within the environment $E$, becomes the value for the local variable $f$ in the new local environment $E_2$, nested inside $E$. Next, the third let expression creates a local environment $E_3$ in which a new local variable $x$ has the value 10. Finally, the expression $(f x)$ is evaluated with respect to the environment $E_3$ using the Default Rule. In the environment $E_3$, the symbol $f$ evaluates to the previously created lambda function, and $x$ evaluates to 10; then that lambda function is applied to the input 10. The key point is the following: because the lambda function was created in the environment $E$, the local environment $E'$ within which the body of that function will be evaluated must be nested within $E$ (i.e., $E' \subseteq E$)—regardless of the fact that the function call expression was evaluated within the environment $E_3$. Therefore, the symbol $x$ that appears in the body of that function evaluates to 5, courtesy of the parent environment $E$, $n$ evaluates to 10, and $(\ast \ x \ n)$ evaluates to 50.

15.4 Deriving the let Special Form from the 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 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 let and lambda.
Example 15.4.1

The following Interactions Window session shows the evaluation of a let expression, followed by the evaluation of an equivalent expression involving the application of a lambda function to some inputs.

\[
\begin{align*}
\text{> (let ((x (+ 2 3))}
& \quad (y (* 3 4)))
& \quad (printf "x: \text{\textasciitilde}s, y: \text{\textasciitilde}s\%" x y)
& \quad (+ x y))
\text{x: 5, y: 12}
& \text{17}
\text{> ((lambda (x y)
& \quad (printf "x: \text{\textasciitilde}s, y: \text{\textasciitilde}s\%" x y)
& \quad (+ x y))
& \quad (+ 2 3)
& \quad (* 3 4))
\text{x: 5, y: 12}
& \text{17}
\end{align*}
\]

\[\implies\text{The semantics for the evaluation of the first expression is identical to the semantics for the evaluation of the second expression!}\]

In particular, for the let expression, a local environment is set up in which the symbol \(x\) is associated with the value 5 and the symbol \(y\) is associated with the value 12. After that, the two expressions in the body of the let are evaluated with respect to that local environment yielding some side-effect printing and an output value of 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 lambda expression. It evaluates to a function. The other entries, (+ 2 3) and (* 3 4), evaluate to the numbers 5 and 12, respectively. When that function is applied to those inputs, a local environment is set up in which \(x\) and \(y\) are associated with the values 5 and 12, respectively. Then the body of the lambda is evaluated, yielding side-effect printing and the output value 17.

Example 15.4.2

The following Interactions Window session first creates a global variable, \(z\). It then evaluates a let expression and an equivalent expression involving the application of a lambda function.

\[
\begin{align*}
\text{> (define z 1000)}
\text{> (let ((x 3)}
& \quad (y 4))
& \quad (* x y z))
\text{12000}
\text{> ((lambda (x y)
& \quad (* x y z))}
& \quad 3
& \quad 4)
\text{12000}
\end{align*}
\]

Once again, the evaluation of the two expressions is the same. In particular, each involves a local environment containing entries for \(x\) and \(y\), with the respective values 3 and 4. In addition, each involves the evaluation of the expression (* \(x\) \(y\) \(z\)) with respect to that local environment. Notice that in each case, the values for \(x\) and \(y\) are drawn from the local environment, whereas the value for \(z\) is drawn from the Global Environment. In each case, the value of the entire expression is 12000.
In general, a let expression of the 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)}
\]

is equivalent to the following expression involving the application of a lambda function:

\[
\text{((lambda \((\text{var}_1 ... \text{var}_n)
\text{ expr}_1
\text{ expr}_2
\ldots
\text{ expr}_k))
\text{ val}_1 ... \text{val}_n)}
\]

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 \(\text{var}_1, ..., \text{var}_{i-1}\) 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 \texttt{let*} Special Form

A \texttt{let*} special form is evaluated as follows:

- An empty local environment is created.

- Each \texttt{var/val} pair is processed, in turn. In particular, an entry is created in the local environment that associates the value of \texttt{val}_i with the symbol \texttt{var}_i.

\implies Crucially, the $i^{th}$ entry in the local environment is created \textit{before} the $(i+1)^{st}$ value is computed. Thus, the expression, $\texttt{val}_{i+1}$, can refer to \textit{any} of the \textit{preceding} symbols, \texttt{var}_1, \ldots, \texttt{var}_i.

- Then the expressions in the body of the \texttt{let*} are evaluated, in turn.

- The value of the last expression in the body of the \texttt{let*} serves as the value of the entire \texttt{let*} expression.

\textbf{Example 15.5.1}

The following Interactions Window session demonstrates the kind of incremental computation that is characteristic of a \texttt{let*} special form, but that is not possible with a (single) \texttt{let} special form:

\begin{verbatim}
> (let* ((x 4)
         (y (+ x 2))
         (z (* x y))
         (w (+ x y z)))
   (printf "x: \texttt{\$s}, y: \texttt{\$s}, z: \texttt{\$s}, w: \texttt{\$s}\%" x y z w)
(+ x y z w))
x: 4, y: 6, z: 24, w: 34
68
\end{verbatim}

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, $(+ y z)$, that is used to compute the value for $w$ refers to $x$, $y$ and $z$. Trying to do this with a \texttt{let} expression causes DrScheme to complain.

\begin{verbatim}
> (let ((x 4)
         (y (+ x 2))
         (z (* x y))
         (w (+ x y z)))
   (printf "x: \texttt{\$s}, y: \texttt{\$s}, z: \texttt{\$s}, w: \texttt{\$s}\%" x y z w)
(+ x y z w))
... reference to undefined identifier: x
\end{verbatim}

The reason is due to the difference in the way \texttt{let} and \texttt{let*} expressions are evaluated (i.e., their semantics). In a \texttt{let} expression, all of the value expressions are evaluated \textit{first}, before any entries are created in the local environment. Thus, none of the value expressions in a \texttt{let} can refer to any of the local variables being defined. In contrast, in a \texttt{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 \texttt{let*} expression.

15.5.3 Deriving a Single \texttt{let*} Expression from Nested \texttt{let} Expressions

In general, a \texttt{let*} expression of the form,
\( (\text{let* } ((\text{var}_1 \text{ val}_1)) \) \\
(\text{var}_2 \text{ val}_2) \) \\
... \\
(\text{var}_n \text{ val}_n)) \) \\
expr_1 \\
expr_2 \\
... \\
expr_k \) \\
is equivalent to \( n \) nested \text{let} expressions:

\( (\text{let } ((\text{var}_1 \text{ val}_1)) \) \\
(\text{let } ((\text{var}_2 \text{ val}_2)) \) \\
... \\
(\text{let } ((\text{var}_n \text{ val}_n)) \) \\
expr_1 \\
expr_2 \\
... \\
expr_k ...)) \)

So, a \text{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 \text{let*} expression and the equivalent nested \text{let} expression:

\[
\begin{align*}
> & \text{(let* } ((x 4) \\
& \quad (y (+ x 2)) \\
& \quad (z (* x y)) \\
& \quad (w (+ x y z)))) \\
& \quad (\text{printf } "x: } ^{\text{s}}s, y: } ^{\text{s}}s, z: } ^{\text{s}}s, w: } ^{\text{s}}s\%" \ x \ y \ z \ w) \\
& \quad (+ x y z w)) \\
\end{align*}
\]

\[
x: 4, y: 6, z: 24, w: 34 \\
68
\]

\[
> \text{(let } ((x 4)) \\
& \quad (\text{let } ((y (+ x 2))) \\
& \quad \text{(let } ((z (* x y))) \\
& \quad \text{(let } ((w (+ x y z)))) \\
& \quad \text{(printf } "x: } ^{\text{s}}s, y: } ^{\text{s}}s, z: } ^{\text{s}}s, w: } ^{\text{s}}s\%" \ x \ y \ z \ w) \\
& \quad (+ x y z w)))))) \\
\end{align*}
\]

\[
x: 4, y: 6, z: 24, w: 34 \\
68
\]

Notice that the outermost \text{let} expression (i.e., the one that specifies the local variable \( x \)) has a body that consists of a single \text{let} expression (i.e., the one that specifies the local variable \( y \)). Because the \text{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, \text{let*} provides a simpler syntax than the equivalent set of nested \text{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 \text{let*}. 

15.6 The letrec Special Form

The letrec special form is provided to enable the specification of local recursive functions, something that cannot be done by let or let*. The specification of a local recursive function within a letrec special form is quite similar to the specification of a global recursive function within a define special form; however, the syntax of a letrec expression is much closer to that of let and let*. A common use of letrec is to embed an accumulator-based, tail-recursive helper function 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 letrec Special Form

The syntax of the letrec special form is identical to that of the let and let* special forms, except that the keyword is letrec instead of let or let*.

15.6.2 The Semantics of the letrec Special Form

In sharp contrast to how the let and let* special forms are evaluated, the evaluation of a 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, #<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!

```
Example 15.6.1

The following interactions demonstrate that the letrec special form sets up its local environment before evaluating any of the value expressions. Because the let and let* special forms do not do this, the corresponding instances generate errors.

> (let ((x y) (y x))
   (printf "x:~s, y:~s~%" x y))
ERROR: reference to undefined identifier: y
> (let* ((x y) (y x))
   (printf "x:~s, y:~s~%" x y))
ERROR: reference to undefined identifier: y
> (letrec ((x y) (y x))
   (printf "x:~s, y:~s~%" x y))
   x:#<undefined>, y:#<undefined>
```

The preceding example is illustrative, but it ignores the primary purpose of the letrec special form: to create local recursive functions, similar to how the define special form can be used to create global recursive functions. For example, a letrec can be used to create a local variable funky whose value is a function whose body includes a recursive function call of the function named funky.
Example 15.6.2: Using letrec to create a local recursive function

The following interactions demonstrate that letrec can be used to define a local recursive function, whereas let and let* cannot.

> (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)
        (if (<= n 1)
            1
            (* n (factyTwo (- n 1)))))))
    (printf "No error up to this point, but ..."))
    (factyTwo 4))
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 are more common, useful way of using letrec.
Example 15.6.3: Using \texttt{letrec} to create a local recursive function within a wrapper function

The following interactions demonstrate the use of the \texttt{letrec} special form to create a local recursive (helper) function within the body of a wrapper function. In this case, the wrapper function is \texttt{facty}, and the local recursive (helper) function is the accumulator-based, tail-recursive \texttt{facty-acc} function. Aside from defining \texttt{facty-acc}, the only thing that \texttt{facty} does is to call \texttt{facty-acc} with appropriate inputs.

\begin{verbatim}
> (define facty
  (lambda (n)
    ;; Body of FACTY starts here
    (letrec ((facty-acc (lambda (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
\end{verbatim}

This kind of application of \texttt{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 (\texttt{facty}) takes care of supplying the helper function (\texttt{facty-acc}) with appropriate inputs.

* Take care when defining local recursive helper functions. For example, note the difference between the input \texttt{n} to \texttt{facty} and the input \texttt{m} to \texttt{facty-acc}. On successive recursive function calls, \texttt{m} takes on different values, while \texttt{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 \texttt{(facty 4)} from the preceding example.

Special Forms Introduced in this Chapter

\begin{verbatim}
let Create local environment
let* Create local environment, supports incremental computations
letrec Create local environment, supports recursive function definitions
\end{verbatim}

Built-in Functions Introduced in this Chapter

\begin{verbatim}
random Pseudo-random number generator (an impure function)
\end{verbatim}
Guide to Your CS Account

All of the programming work you do in this course will be done using your CS computer account which you can access from any of the classroom or lab computers in the CS Department. The name of your account is typically the same as the first part of your Vassar email, although there can be exceptions. For example, my CS account name is hunsberg, which harkens back to the days when account names were limited to eight characters! Every student in this course has their own CS account. In addition, the CMPU-101 course itself also has an account, called cs101. All of the computer files and directories (a.k.a. folders) for all of the CS account holders are organized into a single tree-like structure called a file system. All of the computer programs you write for this course will be computer files that are stored within your portion of the CS file system. Thus, it will be important to understand how to navigate through the file system, create new files and directories, start up the DrScheme software, and print out and electronically submit your program files. All of this will be enabled by simply opening up a Terminal window and entering the appropriate commands at the prompt. (Since the computers are running the Linux operating system, we may refer to these commands as Linux commands.) The rest of this chapter describes the file system, how to explore the file system using the commands issued from a Terminal window, and how to format, submit and print out your assignment files.

A.1 The File System

The file system is organized into a tree-like hierarchy of computer files and directories. A directory (or folder) is a collection of computer files that typically have something in common. For example, a directory called lab1 might contain all of the program files associated with your first programming lab. A directory may also contain subsidiary directories (a.k.a. sub-directories or sub-folders), thereby enabling directories to be organized into a tree-like hierarchy.

At the root of the file system is a special directory, called the root directory, that is the topmost ancestor of every other file and directory in the entire file system. For convenience, the root directory is frequently denoted by a single forward slash: /. As indicated in Fig. A.1, the root directory typically contains lots of directories with strange names (e.g., bin, dev, etc and mnt). These directories are used by the Linux operating system to handle things that will not concern us. However, one of the directories in the root directory is relevant for us: the home directory. As its name suggests, the home directory contains the “home” directories of every CS account. For example, the home directory contains two directories, called hunsberg and cs101, which are the respective home directories of my CS account and that of the CMPU-101 course.

Full pathnames. Each file or directory can be referred to by an absolute address, called its full pathname. The full pathname for a file or directory, X, represents the unique path from the root directory to X in the file system’s hierarchy. For example, the full pathname for my home directory is /home/hunsberg, since the root directory contains the home directory, and the home directory contains the hunsberg directory. Similarly, the full pathname for the cs101 home directory is /home/cs101.
The Desktop directory. As illustrated in Fig. A.1, the home directory for each CS account contains a subdirectory called Desktop. Although my Desktop directory has the same name as your Desktop directory, they are in fact distinct directories. The operating system has no trouble distinguishing them because their full pathnames are unique. For example, the full pathname for my Desktop directory is `/home/hunsberg/Desktop`, while the full pathname for the Desktop directory belonging to the cs101 account is `/home/cs101/Desktop`.

* Most of the files and directories located within your Desktop directory will have a corresponding icon that is automatically displayed on your computer screen’s Desktop.

All of the files you create for your work in this course should be organized within your Desktop directory, as illustrated in Fig. A.2. Notice that this organization allows room for growth should you decide to take subsequent Computer Science courses (e.g., CMPU-102, CMPU-145, and so on).

A.2 Using Terminal to Explore and Augment the File System

The Linux operating system provides numerous commands that enable you to navigate through the file system. These commands are processed by a program called Terminal. When you start the Terminal program, it opens up a Terminal window. When a command is typed into the Terminal window, and the Enter key is tapped, the Terminal program will attempt to execute the command.
When using Linux commands in a Terminal window to navigate the file system, the Terminal program keeps track of your current location within the directory tree. That current location is called your working directory. The name of the working directory is often automatically displayed as part of the prompt in the Terminal window. Below are listed some of the most useful Linux commands for navigating the file system and creating new directories. The use of these commands is covered by Lab 1.

- **pwd** – Print the Working Directory (i.e., display where you are in the tree of directories). When you first open the Terminal window, the working directory is typically set to be the home directory of your account. Thus, if I open up a terminal window in my account and immediately enter the `pwd` command, it will cause the following to be displayed: `/home/hunsberg`.

- **ls** – List the contents (i.e., files and sub-directories) of the working directory.

- **cd** – Change Directory. If used by itself, this command returns you to your account’s home directory (i.e., it sets the working directory to be your home directory). If you give it an input (e.g., a full pathname), then the `cd` command will set the working directory to be whatever directory you specify.

- **mkdir** – Make (i.e., create) a new Directory. This command takes one input: either a full pathname for the new directory or just a simple name for it. For example, the following command would create a new directory named `tmp` within my `Desktop` directory:
  ```bash
  mkdir /home/hunsberg/Desktop/tmp
  ```
  Alternatively, if I was already in the `Desktop` directory (i.e., if the working directory was set to be my `Desktop` directory), then the following simpler command would have the same effect:
  ```bash
  mkdir tmp
  ```

As already mentioned, Lab 1 will demonstrate the use of these and other Linux commands in more detail.

### A.3 Submitting Programming Assignments

This section describes the process of submitting programming assignments. Typically, this will involve two steps: (1) printing out your definitions and interactions files; and (2) electronically submitting the directory that contains these two files.

- When doing any lab or assignment, be sure to save your definitions file periodically so that you don’t lose it should something go wrong! Give it a name such as `yourName-asmt3-defns.txt`.

#### Before Printing or Electronically Submitting your Files

Before printing or electronically submitting your files, you should carefully review the following guidelines.

- Your definitions and interactions must be saved as plain-text files! (If you are unsure about this, review the relevant portions of Lab 1.)

- Your definitions window should be nicely formatted. See the code-from-class postings on the course website for examples of nicely formatted code. Or look at the posted solutions to any lab or assignment. In particular:
  ```bash
  ;; ===========================================
  ;; CMPU-101, Fall 2019
  ;; Asmt. or Lab Info
  ;; Your Name
  ;; ===========================================
  ```

  where Asmt. or Lab Info is replaced by the relevant assignment or lab number (e.g., Asmt. 3 or Lab 5), and Your Name is replaced by your name!
Make sure that the first Scheme expression in your definitions file is: (load "asmt-helper.txt").

Make sure that the second Scheme expression in your definitions file involves an application of the header function to appropriate inputs, for example, something having the form:

(header "Your Name" "Asmt. 3").

When you hit the Run button, you should see a nicely displayed header at the top of your interactions.

Make sure that each problem is introduced by an invocation of the problem function, surrounded by commented lines of dashes, as illustrated below:

;;; -----------------------------------
;;; (problem "Description")
;;; -----------------------------------

Make sure that each function you define is preceded by a “contract” (i.e., a block of comments that specifies the name of the function, the names and descriptions of the input parameters, a brief description of the output, and, if your function has side effects, a brief description of those too. Make sure that your contract clearly distinguishes the output value of the function from any side effects it might have. The contract should have the following form:

;;; FUNCTION-NAME
;;; -----------------------------------
;;; INPUTS: names and descriptions of inputs
;;; OUTPUT: description of output value (or "none")
;;; SIDE EFFECTS: description of side effects (if any)

In your function definition, the names for your function and its inputs should match the names that appear in the contract!

Make sure that your code is properly indented. This is easiest to do by selecting the Scheme menu item and choosing Reindent All.

Make sure that your code does not include long lines of text that wrap around to the next line! Instead, break up long lines by using the Enter key, and taking advantage of DrScheme's automatic indentation!

When needed, your code should be augmented with concise comments explaining (briefly) what your code does. (See code-from-class postings for examples.) For example, if your function uses the cond special form (cf. Chapter 11), then each case of your cond should be preceded by a brief comment describing that case.

Make sure that you have thoroughly tested your functions to demonstrate that they work as desired. This is typically done by providing a bunch of tester expressions that test a variety of cases beyond those that are given in the lab or assignment instructions.

Make sure that there are blank lines between the problem expression and the contract, between the contract and the function definition, between the function definition and the tester expressions, and between the tester expressions and the following problem (if any). Again, see code-from-class postings for examples.

When you are confident that your definitions file adheres to the above guidelines, then do the following:

Save your definitions window one last time.
Hit the Run button one last time.
Save your interactions as plain text! (Use the Save Other and Save Interactions as Text... menu items in DrScheme.)
Double-check that your interactions begin with a nice block of text generated by the header function. The top of your interactions should have the following form:
Asmt. or Lab Info
Your Name

where “Asmt. or Lab Info” is replaced by the relevant information, and “Your Name” is replaced by your name. If this information does not appear at the top of your interactions, check that your definitions file includes a call to the header function as described earlier.

* The contents of your interactions should be laid out nicely using the problem and tester functions, as described earlier. If not, go back to your Definitions Window and make the needed changes.

* Double-check that each tester expression is properly displaying both the input(s) and output—and that each is generating the right answer! If you spot any errors, go back to your function definition and make needed changes. If you make any changes to your Definitions Window, you will need to save your definitions, hit the Run button again, and then save your interactions (as plain text) again.

Congratulations! You should now be ready to print out and electronically submit your work!

### A.3.1 Printing Text Files

**Warning!** The information in this section applies only to printing out files containing plain text! The commands given below should not be used to print out pdf, doc, jpg, or any other non-plain-text files.

For most programming assignments, you will need to print out only two files: your definitions file and your interactions file. (It is not necessary to print out anything for labs.) Both of these files should be plain-text files. If either appears with a bunch of gibberish then you should review the instructions for saving your definitions or interactions as plain-text files. You do not need to turn in printouts of the asmt-helper.txt file, since you are not expected to make changes to that file. In addition, you should not print out any file whose name ends with a `~` character (e.g., myfile.txt~); those files are automatically generated backup files that can be safely ignored.

<table>
<thead>
<tr>
<th>Example A.3.1</th>
</tr>
</thead>
</table>

Suppose that `~hunsberg/Desktop/my101/labs/lab2/hun-lab2.txt` is the full pathname for a plain-text file called `hun-lab2.txt`. The following command can be used within a Terminal window to print out that file to the printer called Asprey, which is located in Room SP 307:

```
enscript -P Asprey `~hunsberg/Desktop/my101/labs/lab2/hun-lab2.txt
```

Since typing out full pathnames can be quite tedious, there’s an even easier way. First, cd into the desired directory—in this case, my lab2 directory; and then issue the following, simpler command:

```
enscript -P Asprey hun-lab2.txt
```

(See Section A.2 if you need a refresher on cd-ing into a desired directory.)

In general, if you are currently in a directory $D$ that contains a plain-text file named `myfile.txt`, then you can print out that file using the following command:

```
enscript -P Asprey myfile.txt
```

If you have any trouble printing, ask a coach for help.

* After printing your definitions and interactions, make sure to staple them—with the definitions on top!

* The Asprey printer should only be used to print out Computer Science labs or assignments.
A.3.2 Submitting your Files Electronically

Assignment files must be electronically submitted using the submit101 command from a Terminal window. This command has the following syntax:

```
submit101  AsmtSubmissionName  YourAsmtDir
```

where AsmtSubmissionName is the name for this assignment for submission purposes (which is typically given to you as part of the assignment instructions) and YourAsmtDir is the name of your assignment directory. (That’s right: you must submit the entire directory; the submit101 command cannot be used to submit individual files.)

---

**Example A.3.2**

Suppose that the AsmtSubmissionName is h-asmt3 and your assignment directory is called asmt3. (We may also say that h-asmt3 is the name of the dropbox into which you are going to submit your assignment.) Suppose further that your asmt3 directory is contained within a directory called asmts. Then you would electronically submit your asmt3 directory by first cd-ing into your asmts directory, and then executing the following command:

```
submit101  h-asmt3  asmt3
```

Note that it is very important that you be in the parent directory of the directory that you want to submit! (The asmts directory is called the parent of the asmt3 directory because asmts contains asmt3.) If you are in the asmt3 directory, then you should execute the following command to cd into the parent asmts directory:

```
cd ..
```

The two periods denote the parent directory of the working directory.

---

If you have any trouble using the submit101 command, ask me or a coach during lab or office/coaching hours.
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.

> (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:

> (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:

;; SEKUND/THURD/FOURTH
;; --------------------------------
;; 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:

> (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 fifth, 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:

;;; 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:

> (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:

```scheme
;; 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)
```
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:

```scheme
> (list 'a (+ 2 3) #f)
(a 5 #f)
```

Notice the difference between result obtained from the above example and that obtained by evaluating the following quote special form.

```scheme
> '(a (+ 2 3) #f)
(a (+ 2 3) #f)
```

## 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:

\[
\begin{align*}
f(4) &= 4 \cdot f(3) & \text{Recursive call: } f(4) = 4 \cdot f(3) \\
     &= 4 \cdot (3 \cdot f(2)) & \text{Recursive call: } f(3) = 3 \cdot f(2) \\
     &= 4 \cdot (3 \cdot (2 \cdot f(1))) & \text{Recursive call: } f(2) = 2 \cdot f(1) \\
     &= 4 \cdot (3 \cdot (2 \cdot 1)) & \text{Base case: } f(1) = 1 \\
     &= 4 \cdot (3 \cdot 2) & 2 \cdot 1 = 2 \\
     &= 4 \cdot 6 & 3 \cdot 2 = 6 \\
     &= 24 & 4 \cdot 6 = 24
\end{align*}
\]

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 \texttt{mult-all}:

\begin{verbatim}
;;  MULT-ALL
;;  ------------------------------
;;  INPUT:  LISTY, a list of numbers
;;  OUTPUT: The product of all: the elements of LISTY
\end{verbatim}

Here are some examples of the desired behavior:

\begin{verbatim}
> (mult-all '(2 3 4 10))
240
> (mult-all '(10 2 4))
80
\end{verbatim}

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:

\[
\text{(the product of all of the elements of (2 3 4 10))} = 2 \times \text{(the product of all of the elements of (3 4 10))}
\]

Stated in terms of the \texttt{mult-all} function, where \texttt{listy} is a variable whose value is \texttt{(2 3 4 10)}:

\[
\text{(mult-all listy)} \Rightarrow (* \text{(first listy)} \text{(mult-all (rest listy))})
\]

Note that if this relationship is going to hold for all non-empty lists, then \texttt{(mult-all ()}) must evaluate to 1 (i.e., the multiplicative identity), as illustrated below:

\[
\text{(mult-all '(4))} \Rightarrow (* 4 \text{(mult-all ())}) \Rightarrow (* 4 1) \Rightarrow 4
\]

In view of all of the above, we might imagine the evaluation of \texttt{(mult-all '(2 3 4 10))} proceeding as follows, where, for example, the recursive function call on the rest of the list \texttt{(2 3 4 10)} is represented by \texttt{(mult-all '(3 4 10))}:

\[
\begin{align*}
\text{(mult-all '(2 3 4 10))} & \quad \text{Recursive Case} \\
\Rightarrow (* 2 \text{(mult-all '(3 4 10))}) & \quad \text{Recursive Case} \\
\Rightarrow (* 2 (* 3 \text{(mult-all '(4 10))})) & \quad \text{Recursive Case} \\
\Rightarrow (* 2 (* 3 (* 4 \text{(mult-all ()))})) & \quad \text{Recursive Case} \\
\Rightarrow (* 2 (* 3 (* 4 (* 10 \text{(mult-all ())})))) & \quad \text{Base Case} \\
\Rightarrow (* 2 (* 3 (* 4 (* 10 1)))) & \\
\Rightarrow (* 2 (* 3 (* 4 10))) & \\
\Rightarrow (* 2 (* 3 40)) & \\
\Rightarrow (* 2 120) & \\
\Rightarrow 240 &
\end{align*}
\]
As long as the list in question is non-empty, the recursive case evaluates an expression of the form
\((* \text{ (first some-list)} \text{ (mult-all (rest some-list))})\). 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:

\[
\text{(define mult-all}
\text{ (lambda (listy))}
\text{ (cond}
\text{ ;; Base Case: LISTY is empty}
\text{ (null? listy)}
\text{ ;; The product of all the elements of the empty list is}
\text{ ;; taken to be 1, the multiplicative identity.}
\text{ 1})
\text{ ;; Recursive Case: LISTY is non-empty (and so we can use}
\text{ ;; the FIRST and REST accessor functions on LISTY)}
\text{ (else}
\text{ ;; The product of all of the elements of LISTY is obtained}
\text{ ;; by multiplying the FIRST element of LISTY by the}
\text{ ;; product of all of the REST of the elements of LISTY.}
\text{ ;; The latter job is handled by the recursive func. call.}
\text{ (* (first listy)}
\text{ (mult-all (rest listy))))})
\]

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.

\[
\text{(define sum-all}
\text{ (lambda (listy))}
\text{ (cond}
\text{ ;; Base Case: LISTY is empty}
\text{ (null? listy)}
\text{ ;; The sum of all the elements of the empty list is}
\text{ 0})
\text{ ;; Recursive Case: LISTY is non-empty}
\text{ (else}
\text{ ;; The recursive function call computes the sum of all}
\text{ ;; the numbers in the rest of LISTY; we just add on the}
\text{ ;; first element.}
\text{ (+ (first listy) (sum-all (rest listy))))})
\]

\[
> \text{(sum-all }\text{ '(1 2 3 4)})
10
> \text{(sum-all }\text{ '(1 10 100 1000)})
1111
\]
In-Class Problem 16.2.1

Define a function, called `add-squares`, that satisfies the following contract:

```scheme
;; ADD-SQUARES
;; -----------------------------------------
;; INPUT: LISTY, a list of numbers
;; OUTPUT: The sum of the squares of the numbers in LISTY
```

Here are some examples of the desired behavior:

```scheme
> (add-squares '(2 3 10)) 113
> (add-squares '(1 0 5 2)) 30
```

In-Class Problem 16.2.2: Computing the length of a list

Define a function, called `lengthy`, that computes the number of elements of the input list. Here is its contract:

```scheme
;; LENGTHY
;; -------------------------------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: The number of elements of LISTY (i.e., its length)
```

Here are some examples of the desired behavior:

```scheme
> (lengthy '(a b c d e)) 5
> (lengthy '(#t () 22 xyz)) 4
```

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 `length`, that does just that!

In-Class Problem 16.2.3: Accessing the \(N\)th element of a list

Define a function, called `fetch-nth-element`, that satisfies the following contract:

```scheme
;; FETCH-NTH-ELEMENT
;; -----------------------------------------
```
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:

```scheme
;;; IS-ELT-OF?
;;; --------------------------------------------
;;; INPUTS: ITEM, anything
;;; LISTY, a list of stuff
;;; OUTPUT: #t (or something that counts as true) if ITEM
;;; appears as an element of LISTY -- 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 LISTY is (3 4 5). In this case, it is clear that ITEM appears in LISTY 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 LISTY, there is no need to continue looking any further.) On the other hand, in the second example, where ITEM is 3, and LISTY is (1 2 3 4 5), it is true that ITEM appears in LISTY because, as a sequence of recursive functions call might discover, ITEM appears somewhere in the rest of LISTY. Finally, in the third example, where ITEM is x, and LISTY 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 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 (is-elt-of? 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 item somewhere in listy. What are the possibilities? Well, 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 is-elt-of? function. To distinguish the two versions, we call this one is-elt-of-alt?.

```
(define is-elt-of-alt?
  (lambda (item listy)
    ;; The following expression specifies the conditions under
    ;; which this function should output #t (or something that
    ;; counts as true):
    ;; (1) LISTY must NOT be empty;
    ;; AND
    ;; (2) ITEM must appear as the FIRST element of LISTY
    ;; OR
    ;; ITEM must appear somewhere in the REST of LISTY
    (and (not (null? listy))
         (or (eq? item (first listy))
             (is-elt-of-alt? item (rest listy))))
  ))
```

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 incr?, that satisfies the following contract:

```
;; INCR?
;; -----------------------------------------------
;; INPUT: LISTY, a non-empty list of numbers
;; OUTPUT: #t if the numbers in LISTY are in strictly
;; *increasing* order; #f otherwise
```

Here are some examples illustrating its behavior:

```
> (incr? '(1 3 8 9 15))
#t
> (incr? '(1 3 4 4 6 9))  ← Not strictly increasing
#f
> (incr? '(2 5 8 5 2))
#f
```

* What’s the best way of checking whether the input list contains exactly one element?
Write one version of `incr?` that uses `if` or `cond`, and another that uses some combination of `and`, `or`, and `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)
        (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):
• 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:

\[
\text{(double-all listy)} \Rightarrow \text{(cons (* 2 \text{first listy}) (double-all \text{rest listy}))}
\]

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
  ;; The double-all of () is ...
  (null? listy) ()
  ;; Recursive Case: LISTY is non-empty
  ...))
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: (facty 3) ⇒ 6
- Let a recursive function call apply \texttt{facty} to the rest of the elements:
  (mappy facty '(4 5 6)) ⇒ (24 120 720)

(4) Use the \texttt{cons} function to combine the above pieces:

- (cons 6 '(24 120 720)) ⇒ (6 24 120 720)
The above analysis suggests that for a non-empty list \( \text{listy} \), the following expression will evaluate to the desired result:

\[
(m\text{appy \ func \ listy}) \Rightarrow (\text{cons \ (func \ (first \ listy))}) \\
(m\text{appy \ func \ (rest \ listy)))
\]

In addition, you should convince yourself that, as in Example 16.3.1, the base case, \((m\text{appy \ func \ ()})\), should evaluate to \((\))\. 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 \text{mappy} function, I can tell you that there is a built-in function, called \text{map}, that does the same thing. The following example illustrates how the \text{map} function can be used to facilitate testing.

Example 16.3.3: Using \text{map} to facilitate testing

\text{Suppose that you have defined a function, called \text{square}, that squares its input. Instead of writing several tester expressions to test the performance of \text{square} on several inputs, you can write just one tester expression, using \text{map} to apply \text{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

\text{Define a function, called \text{remover}, that satisfies the following contract:}
\begin{document}

\section*{In-Class Problem 16.3.2: Concatenating two lists}

\textit{Define a function, called \texttt{conc}, that satisfies the following contract:}

\begin{verbatim}
;; CONC
;; -------------------------------
;; INPUTS: LISTY, LISTZ, two lists
;; OUTPUT: A list containing all of the elements of LISTY
;; followed by all of the elements of LISTZ.
\end{verbatim}

\textit{Here are some examples of the desired behavior:}

\begin{verbatim}
> (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)
\end{verbatim}

\textit{Hints: Let listy drive the recursion. What is the output when listy is empty?}

\section*{Example 16.3.4}

\textit{The goal of this example is to define a function, called \texttt{list-down-to-zero}, that satisfies the following contract:}

\begin{verbatim}
;; LIST-DOWN-TO-ZERO
\end{verbatim}

\end{document}
;; -----------------------------------------------
;; 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:

> (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:

(list-down-to-zero n) = (cons n (list-down-to-zero (- n 1))

where, for the base case, we stipulate that: (list-down-to-zero m) = (), for any m < 0. (Alternatively, we could use (list-down-to-0 0) = (0) as our base case.) Here is the completed solution:

(define list-down-to-zero
(lambda (n)
(cond
    ;; Base Case: N < 0
    ((< n 0) ()
    ;; Recursive Case: N >= 0
    (else
      (cons n (list-down-to-zero (- n 1))))))))

In-Class Problem 16.3.3

Define a function, called list-up-to-n, that satisfies the following contract:

;; LIST-UP-TO-N
;; -----------------------------------------------
;; INPUTS: FROM, a non-negative integer (starting point)
;; N, a non-negative integer (stopping point)
;; OUTPUT: A list of the form (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)

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 from to 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 random-flips, that satisfies the following contract:

;;; RANDOM-FLIPS
;;; -------------------------------
;;; INPUTS: N, a non-negative integer
;;; OUTPUT: A list containing 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 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 ___________ to the front of a list of _______ 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:

\[(\text{mult-all-acc } '(3 7 2 4) 1)\]
\[(\text{mult-all-acc } '(7 2 4) 3) \leftarrow \text{ rec. case: "accumulate" a factor of } 3\]
\[(\text{mult-all-acc } '(2 4) 21) \leftarrow \text{ rec. case: "accumulate" a factor of } 7\]
\[(\text{mult-all-acc } '(4) 42) \leftarrow \text{ rec. case: "accumulate" a factor of } 2\]
\[(\text{mult-all-acc } () 168) \leftarrow \text{ base case: accumulator has the answer!}\]

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:

\[
\begin{align*}
\text{;; MULT-ALL-ACC} \\
\text{;; ---------------------------------------------------------------} \\
\text{;; INPUTS: LISTY, a list of numbers} \\
\text{;; ACC, a number (accumulator of desired product)} \\
\text{;; OUTPUT: When called with ACC=1, the output is the product} \\
\text{;; of all of the numbers in LISTY. More generally, the output} \\
\text{;; is the product of ACC and all of the numbers of LISTY}\end{align*}
\]

\[
\begin{align*}
\text{(define mult-all-acc} \\
\text{ (lambda (listy acc)} \\
\text{ (cond} \\
\text{ (null? listy) \\
\text{ (acc)} \\
\text{ (else} \\
\text{ (mult-all-acc (rest listy) \\
\text{ (* (first listy) acc))))))})\end{align*}
\]

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:

\[
\begin{align*}
\text{;; MULT-ALL-WR} \\
\text{;; ----------------------------------------} \\
\text{;; INPUT: LISTY, a list of numbers} \\
\text{;; OUTPUT: The product of the numbers in LISTY}\end{align*}
\]

\[
\begin{align*}
\text{(define mult-all-wr} \\
\text{ (lambda (listy) \\
\text{ (mult-all-acc listy 1))})}
\end{align*}
\]
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:

\[
\begin{align*}
\text{(list-down-to-zero-acc 3 ()}) \\
\Rightarrow \text{(list-down-to-zero-acc 2 '(3))} & \quad \text{attach 3 to front of acc} \\
\Rightarrow \text{(list-down-to-zero-acc 1 '(2 3))} & \quad \text{attach 2 to front of acc} \\
\Rightarrow \text{(list-down-to-zero-acc 0 '(1 2 3))} & \quad \text{attach 1 to front of acc} \\
\Rightarrow \text{(list-down-to-zero-acc -1 '(0 1 2 3))} & \quad \text{attach 0 to front of acc} \\
\Rightarrow \text{(0 1 2 3)} & \quad \text{acc has the answer?!!}
\end{align*}
\]

**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`, `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:

\[
\begin{align*}
\text{(list-down-to-zero-acc 3 () 0)} \\
\Rightarrow \text{(list-down-to-zero-acc 3 '(0) 1)} & \quad \text{attach 0 to front of acc} \\
\Rightarrow \text{(list-down-to-zero-acc 3 '(1 0) 2)} & \quad \text{attach 1 to front of acc} \\
\Rightarrow \text{(list-down-to-zero-acc 3 '(2 1 0) 3)} & \quad \text{attach 2 to front of acc} \\
\Rightarrow \text{(list-down-to-zero-acc 3 '(3 2 1 0) 4)} & \quad \text{attach 3 to front of acc} \\
\Rightarrow \text{(3 2 1 0)} & \quad \text{acc has the answer!}
\end{align*}
\]

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:
;; 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 ... CURR) and ACC.

(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
⇒ (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 `reversely` 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)
        listz)
      ;; Recursive Case: LISTY is non-empty
      (else
        (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 (rest listy) listz)))))
```
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)).`

```
(konk (bad-reverse (rest listy))
     (cons (first listy) ())))))
```

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:

```
;; 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 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 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
```

```scheme
> (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.\(^1\) 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)`.\(^2\) 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
      ((null? listy)
       ( ))
      ;; The empty list is already sorted
      ()
      ;; 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)) Recursive case
⇒ (i 3 (isort '(2 5 1 6))) Recursive case
⇒ (i 3 (i 2 (isort '(5 1 6)))) Recursive case
⇒ (i 3 (i 2 (i 5 (isort '(1 6)))))) Recursive case
⇒ (i 3 (i 2 (i 5 (i 1 (isort '(6))))))) Recursive case
⇒ (i 3 (i 2 (i 5 (i 1 (i 6 (isort ())))))) Base case!
⇒ (i 3 (i 2 (i 5 (i 1 (i 6 ())))))) Insert 6 into ()
⇒ (i 3 (i 2 (i 5 (i 1 '6)))))) Insert 1 into (6)
⇒ (i 3 (i 2 (i 5 '(1 6)))))) Insert 5 into (1 6)
⇒ (i 3 (i 2 '(1 5 6))) Insert 2 into (1 5 6)
⇒ (i 3 '(1 2 5 6)) Insert 3 into (1 2 5 6)
⇒ '(1 2 3 5 6) Done!

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:

;; 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:

> (list-of-n-random-numbers 10)
(18980 44224 94176 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:

> (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)
BEFORE: (68502 79284 50452 31764 48239)
AFTER: (31764 48239 50452 68502 79284)
(31764 48239 50452 68502 79284)

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
The approach is to walk through sorted, accumulating all of its elements that are smaller than num, as illustrated below:

\[
\begin{align*}
& (\text{insert-acc} \ 5 \ '1\ 2\ 4\ 6\ 12\ 15) \ () \\
& (\text{insert-acc} \ 5 \ '2\ 4\ 6\ 12\ 15) \ '(1) \\
& (\text{insert-acc} \ 5 \ '4\ 6\ 12\ 15) \ '(2\ 1) \\
& (\text{insert-acc} \ 5 \ '6\ 12\ 15) \ '(4\ 2\ 1)
\end{align*}
\]

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:

\[
\begin{align*}
& ;; \ \text{INSERT-ACC} \\
& ;; ----------------------------------------------------------- \\
& ;; \ \text{INPUT: NUM, a number} \\
& ;; \hspace{1cm} \text{SORTED, a list of numbers that are already sorted} \\
& ;; \hspace{1cm} \text{into non-decreasing order} \\
& ;; \hspace{1cm} \text{ACC, a list of numbers in non-increasing order,} \\
& ;; \hspace{1cm} \text{where each number in ACC is less than NUM} \\
& ;; \ \text{OUTPUT: When called with ACC = (), the output is a list} \\
& ;; \hspace{1cm} \text{containing NUM and all the numbers in SORTED,} \\
& ;; \hspace{1cm} \text{all sorted into non-decreasing order.}
\end{align*}
\]

Here are some examples of its use:

\[
\begin{align*}
& > (\text{insert-acc} \ 5 \ '(1\ 2\ 4\ 6\ 12\ 15) \ ()) \\
& \hspace{1cm} (1\ 2\ 4\ 5\ 6\ 12\ 15) \\
& > (\text{insert-acc} \ 3 \ '(1\ 1\ 2\ 2\ 3\ 3\ 4\ 4\ 5\ 5) \ ()) \\
& \hspace{1cm} (1\ 1\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 5\ 5)
\end{align*}
\]

Finally, define a wrapper function, called insert-wr, that satisfies the following contract, and exhibits the behavior shown below:

\[
\begin{align*}
& ;; \ \text{INSERT-WR -- wrapper function for INSERT-ACC} \\
& ;; ----------------------------------------------------------- \\
& ;; \ \text{INPUT: NUM, a number} \\
& ;; \hspace{1cm} \text{SORTED, a list of numbers that are already sorted} \\
& ;; \hspace{1cm} \text{into non-decreasing order} \\
& ;; \hspace{1cm} \text{OUTPUT: A list containing NUM and all the numbers in SORTED,} \\
& ;; \hspace{1cm} \text{all sorted into non-decreasing order.}
\end{align*}
\]

\[
\begin{align*}
& > (\text{insert-wr} \ 5 \ '(1\ 2\ 4\ 6\ 12\ 15)) \\
& \hspace{1cm} (1\ 2\ 4\ 5\ 6\ 12\ 15) \\
& > (\text{insert-wr} \ 3 \ '(1\ 1\ 2\ 2\ 3\ 3\ 4\ 4\ 5\ 5)) \\
& \hspace{1cm} (1\ 1\ 2\ 2\ 3\ 3\ 3\ 4\ 4\ 5\ 5)
\end{align*}
\]
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:

\[
\begin{align*}
\Rightarrow & \ (\text{isort-acc} \ 4 \ 9 \ 2 \ 6 \ ()) \quad \text{recursive case} \\
\Rightarrow & \ (\text{isort-acc} \ 9 \ 2 \ 6 \ (\text{insert-wr} \ 4 \ ())) \quad \text{recursive case} \\
\Rightarrow & \ (\text{isort-acc} \ 2 \ 6 \ (\text{insert-wr} \ 9 \ (4))) \quad \text{recursive case} \\
\Rightarrow & \ (\text{isort-acc} \ 2 \ 6 \ (4 \ 9)) \quad \text{recursive case} \\
\Rightarrow & \ (\text{isort-acc} \ 6 \ (\text{insert-wr} \ 2 \ (4 \ 9))) \quad \text{recursive case} \\
\Rightarrow & \ (\text{isort-acc} \ 6 \ (2 \ 4 \ 9)) \quad \text{recursive case} \\
\Rightarrow & \ (\text{isort-acc} \ () \ (\text{insert-wr} \ 6 \ (2 \ 4 \ 9))) \quad \text{recursive case} \\
\Rightarrow & \ (\text{isort-acc} \ () \ (2 \ 4 \ 6 \ 9)) \quad \text{recursive case} \\
\Rightarrow & \ (2 \ 4 \ 6 \ 9) \quad \text{base case}
\end{align*}
\]

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:

\[
\begin{align*}
;; \ \text{MERGE-SORT} \\
;; \ \text{---------------------------------------------------------} \\
;; \ \text{INPUTS: LISTY, a list of numbers} \\
;; \ \text{OUTPUT: A list containing the same elements as LISTY,} \\
;; \ \text{but sorted into non-decreasing order}
\end{align*}
\]

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:

\[
\begin{align*}
;; \ \text{SPLIT} \\
;; \ \text{---------------------------------------------------------} \\
;; \ \text{INPUT: LISTY, any list} \\
;; \ \text{OUTPUT: A list of the form (LEFTY RIGHTY) where LEFTY} \\
;; \ \text{and RIGHTY are two subsidiary lists such that the} \\
;; \ \text{elements of LISTY have been allocated as evenly as} \\
;; \ \text{possible to LEFTY and RIGHTY, but with no regard to} \\
;; \ \text{their order.}
\end{align*}
\]
Here are some examples of the behavior of the split and merge helper functions:

\[
\begin{align*}
&> (\text{split } '(5 3 1 2 8 4 9 4)) \quad \text{← Input has an even number of elements} \\
&\quad ((4 4 2 3) (9 8 1 5)) \\
&> (\text{split } '(5 3 1 2 7)) \quad \text{← Input has an odd number of elements} \\
&\quad ((7 1 5) (2 3)) \\
&> (\text{merge } '(1 3 5 7) ' (2 4 6 8)) \\
&\quad (1 2 3 4 5 6 7 8) \\
&> (\text{merge } '(1 1 2 3 3 3 5 9) ' (2 3 3 4 8 8 9)) \\
&\quad (1 1 2 2 3 3 3 3 3 4 5 8 8 9 9)
\end{align*}
\]

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:

\[
\begin{align*}
\text{lefty} &= (6 3 5 8) \\
\text{righty} &= (1 4 9 2)
\end{align*}
\]

(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:

\[
\begin{align*}
\text{sorted-lefty} &= (3 5 6 8) \\
\text{sorted-righty} &= (1 2 4 9)
\end{align*}
\]

(3) Merge the two sorted lists, sorted-lefty and sorted-righty, into a single sorted list, which will be the desired output. Here:

\[
(\text{merge } '(3 5 6 8) ' (1 2 4 9)) \Rightarrow (1 2 3 4 5 6 8 9).
\]

Here is the completed definition of the merge-sort function:

\[
\begin{align*}
\text{(define merge-sort} \\
\quad (\lambda (\text{listy})
\end{align*}
\]

\[
\begin{align*}
\text{MERGE} \\
\text{---------------------------} \\
\text{INPUT: SORTED-ONE, SORTED-TWO, two lists of numbers} \\
\text{that are already sorted into non-decreasing order.} \\
\text{OUTPUT: A single list that contains all of the elements} \\
\text{of SORTED-ONE and SORTED-TWO, sorted into} \\
\text{non-decreasing order.}
\end{align*}
\]
(cond
   ;; Base Case: LISTRY has exactly one element
   (null? (rest listy))
   ;; A one-element list is already sorted
   listy)
   ;; Recursive Case: LISTORY has at least two elements
   (else
    (let* (;; LIST-O-LISTS has the form (LEFTY RIGHTY)
           (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:

```
;; 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
```

;; 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
Introduction to Computer Science via Scheme

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 ...)
```
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.

*To increase readability, the output lists have been cut off.*

---

**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.

```scheme
;; 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 ...\% ")
      (time (insertion-sort listy))
      (printf "%Running merge-sort ...\% ")
      (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 \texttt{merge-sort} and \texttt{insertion-sort}

Use the \texttt{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 recommend? Try running the faster of the two algorithms on really long lists (e.g., with 100,000 elements, or even a million elements).

Example 16.5.6: The built-in \texttt{sort} function

Scheme provides a built-in function, called \texttt{sort}, whose contract is given below, followed by some examples of its use.

\begin{verbatim}
;;; SORT -- built-in function
;;; -----------------------------------------------
;;; INPUTS: LISTY, a list of stuff
;;; COMPARER, a predicate that can be applied to
;;; any pair of elements in LISTY
;;; 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) \Rightarrow \#t, then AAA
;;; comes before BBB in the output list.

> (sort '(5 2 1 3 3 2 5) <) \leftarrow \text{sort into non-decreasing order}
(1 2 2 3 3 5 5)
> (sort '(5 2 1 3 3 2 5) >) \leftarrow \text{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))))
(-6 5 -4 3 -2 1)
\end{verbatim}

In the last case, the \texttt{COMPARER} predicate is specified by a \texttt{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 \texttt{first} and \texttt{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 \texttt{first} and \texttt{rest} parts. The rest of this section explores that structure, revealing the central role of a data structure called a \texttt{cons cell}—also known as a \texttt{pair}.

#### 16.6.1 Data Structures

In Computer Science, the term, \textit{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 \textit{indexed} so that any particular slot can be accessed by its corresponding (numerical) index. For example, the slots in \textit{vectors}—to be discussed in Chapter 18—are indexed in this way. In other data structures, the slots for holding data are \textit{named} so that any particular slot can be accessed by its name. Named slots are often called \textit{fields}. For example, a \textit{bank-account} data structure might have fields called \texttt{password} and \texttt{balance}. The rest of this section restricts attention to a very simple field-based data structure that, for historical reasons, is called a \texttt{cons cell}. Each cons cell has only two fields. For this reason, cons cells are also called \texttt{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 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:

- cons: For constructing a new cons cell
- cons?: Type-checker predicate for cons cells

---

**Example 16.6.1: The cons function revisited**

Here is a more accurate contract for the 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 cons function is indeed a cons cell, as confirmed by the built-in 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.

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.

```scheme
> (cons 3 (cons 4 (cons 6 ())))
(3 4 6)
> (cons 1 (cons 2 (cons 3 (cons 4 ()))))
(1 2 3 4)
> (cons 'x (cons 'y (cons 'z ())))
(x y z)
```

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:

```scheme
> (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:

```scheme
;;; 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:

> (well-formed-list? ()
#t
> (well-formed-list? '(a b c d))
#t
> (well-formed-list? (cons 1 (cons 2 3)))
#f
> (well-formed-list? ’xyz)
#f

Since this function is a predicate, you should be able to define it using and, or and not, without using if or 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.

(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))))))

Here’s an example of its behavior:

> (double-all ’(3 1 4 7))
(6 2 8 14)

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 \((\text{cons} \ 6 \ (\text{double-all} \ '((1 \ 4 \ 7)))\) and \((\text{cons} \ 3 \ (1 \ 4 \ 7))\) have the same number of elements, which is equivalent to showing that \((\text{double-all} \ '(1 \ 4 \ 7))\) 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 \((\text{double-all} \ '(4 \ 7))\) and \((4 \ 7)\) have the same number of elements. And that will hold if and only if \((\text{double-all} \ ()\) and \()\) have the same number of elements. And that holds—since \((\text{double-all} \ ())\) evaluates to \()\)!

The technique described in the preceding example can be used to show that the built-in \text{map} function is also length preserving. For example, \((1 \ 2 \ 3 \ 4)\) and the list generated by evaluating \((\text{map} \ \text{facty} \ '(1 \ 2 \ 3 \ 4))\) must have the same length.

**In-Class Problem 16.6.2: Picturing the length preserving nature of \text{double-all} and \text{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 \((\text{double-all} \ '(3 \ 1 \ 4 \ 7))\). 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 \text{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 \textbf{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., \text{sum-all*} instead of \text{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.

```scheme
;; SUM-ALL*
;; ---------------------------------------------
;; INPUT: HLISTY, a (possibly hierarchical) list of numbers
;; OUTPUT: The sum of all of the numbers appearing anywhere
;; within HLISTY
> (sum-all* '(1 (2 (3 (4) 5) 6)))
21
> (sum-all* '((((((10 100)))) 1))))
111
```

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., (first hlisty)) is itself a list.

```scheme
(define sum-all*
  (lambda (hlisty)
    (cond
      ;; Base Case: HLISTY is empty
      (null? hlisty) 0
      ;; Recursive Case 1: First element of HLISTY is a list
      ((list? (first hlisty))
        (+ (sum-all* (first hlisty))
          (sum-all* (rest hlisty))))
      ;; Recursive Case 2: First element of HLISTY is not a list
      (else
        (+ (first hlisty)
          (sum-all* (rest hlisty)))))))
```

Notice that when (first hlisty) is itself a list, it follows that both (first hlisty) and (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 (first hlisty) is the list (1 2 3) and (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.

⇒ 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 (first hlisty) is a list can be terribly inefficient because, in cases where (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.
 QUICK-LIST?

README

INPUT: DATUM, anything
OUTPUT: #t if DATUM is either () or a cons cell;
       #f otherwise.

(define quick-list?
  (lambda (datum)
    (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:

 LENGTHY

README

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) 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)))))))
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.
;; 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:**

```scheme
> (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:

```
;; 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:

```
> (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:

```
;; 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:

```
> (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.

```
> (+) ← 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

Multiplying 1 2 3 4 5 yields 120

Multiplying 10 10 10 yields 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.

```scheme
(lambda args expr expr2 ... exprk)
```

where `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 `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.

```scheme
;; 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 `my-multy` in hand, the desired function, `my-multy-multy`, can be easily defined using the new syntax for the lambda special form, as follows.

```scheme
;; MY-MULTY-MULTY
;; --------------------------------------------------------
;; INPUTS: Any number of numbers
;; OUTPUT: The product of those numbers
```

```scheme
;; MY-MULTY-MULTY
;; --------------------------------------------------------
;; INPUTS: Any number of numbers
;; OUTPUT: The product of those numbers
```
\(\text{(define my-multy-multy}\\\(\text{(lambda args}\\\quad;; \text{Since ARGs is a LIST of numbers...}\\\quad\text{(my-multy args)\})}\)\)

The following interactions demonstrate the difference between \text{my-multy} and \text{my-multy-multy}.

\begin{verbatim}
> (my-multy '(1 2 3 4))
24
> (my-multy-multy 1 2 3 4)
24
> (my-multy '(10 10 10))
1000
> (my-multy-multy 10 10 10)
1000
\end{verbatim}

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 \text{my-multy-multy} by using the built-in \text{apply} function.

\textbf{Example 16.8.2: The built-in apply function}

\begin{verbatim}
The built-in apply function satisfies the following contract:

;; 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.

\end{verbatim}

\begin{verbatim}
> (+ 100 10 1)
111
> (apply + ' (100 10 1))
111
> (* 1 2 3 4 5)
120
> (apply * ' (1 2 3 4 5))
\end{verbatim}

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: ARG is empty
      ((null? args) 1)
      ;; Recursive Case: ARG is non-empty
      (else
       (* (first args)
          (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 $n^{th}$ 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
Part II

Destructive Programming in Scheme
Chapter 17

Iteration

Like recursion, iteration is a technique that enables a programmer to make the computer do things repetitively. However, unlike recursion, iteration typically involves destructive programming. This chapter presents several special forms that facilitate incorporating iteration in Scheme programs. The `set!` special form causes the value of a variable to be destructively modified. The `while` special form iteratively evaluates the expressions in its body as long as some condition holds. (Destructive programming is required if a condition that initially evaluates to `true` is eventually going to evaluate to `false`.) The `dotimes` and `dolist` special forms, respectively, automate kinds of iteration that are analogous to numerical and list-based recursion.

17.1 The `set!` Special Form

The purpose of the `set!` special form is to destructively modify (i.e., change) the value of a variable (i.e., the value associated with a symbol in some environment). After using `set!` to change the value of a variable, its previous value will be lost forever—unless it was saved elsewhere prior to setting the new value.

<table>
<thead>
<tr>
<th>Example 17.1.1: The <code>set!</code> special form</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (define x 100) ← Create a global variable x with value 100</td>
</tr>
<tr>
<td>&gt; x 100</td>
</tr>
<tr>
<td>&gt; (set! x (* 5 5)) ← Change the value associated with x</td>
</tr>
<tr>
<td>&gt; x 25</td>
</tr>
<tr>
<td>&gt; (set! y 50) ← No can do! There is no entry for a variable named y.</td>
</tr>
<tr>
<td>Error!</td>
</tr>
</tbody>
</table>

As the last example demonstrates, DrScheme will report an error if you try to use `set!` for a symbol that has no entry in the relevant environment.

The syntax of the `set!` special form. The `set!` special form has the following syntax:

```
(set! var newVal)
```

where `var` is any symbol expression, and `newVal` can be any Scheme expression. Each of the following are legal instances of the `set!` special form:

```
(set! x 3)
(set! myVar (* 8 10))
(set! yourVar (member 3 '(1 2 3 4 5)))
```
Global Environment

<table>
<thead>
<tr>
<th>:</th>
<th>:</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xxx</td>
</tr>
<tr>
<td>y</td>
<td>yyy</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Local Environment

| x | 0 |

(set! x 'newX)
(set! y 'newY)
(list x y)

Local Environment

| x | newX |

(set! x 'newX)
(set! y 'newY)
(list x y) ⇒ (newX newY)

Figure 17.1: The environments for Example 17.1.2 before (left) and after (right) evaluating the set! expressions

The semantics of the set! special form. Like most Scheme expressions, the evaluation of a set! special form depends on the environment in which it is being evaluated. The simplest case is that of a set! special form being evaluated with respect to the Global Environment. In that case, the evaluation proceeds as follows.

1. \(\text{newVal}\) is evaluated in the Global Environment, generating some Scheme datum \(D\).

2. \(D\) is inserted as the new value for \(\text{var}\) in the Global Environment.

3. The set! special form itself evaluates to \text{void} (i.e., no value).

Using set! within a local environment. Recall that when a symbol is evaluated within a local environment, the local environment takes precedence over any parent environments. For example, if a symbol \(x\) is being evaluated within a local environment, then drScheme looks first in that local environment for an entry for \(x\). If it finds one, then it uses the associated value; otherwise, it checks the parent environment(s).

The same sort of precedence of local environments applies when using the set! special form within a local environment to change the value for a symbol. For example, suppose that \((\text{set! var newVal})\) is being evaluated with respect to a local environment \(E_1\) that is nested directly inside the Global Environment (i.e., \(E_1 \subseteq E_0\)). In this case, \(\text{newVal}\) will be evaluated with respect to the local environment \(E_1\), generating some Scheme datum \(D\). Next, the appropriate variable named \(\text{var}\) must be located. If there is an entry for the symbol \(\text{var}\) in the local environment \(E_1\), then \(D\) will be inserted as the value for \(\text{var}\) in that entry. Otherwise, \(D\) will be inserted as the value for \(\text{var}\) in the Global Environment. (If neither environment contains an entry for \(\text{var}\), then attempting to use set! to change its value would cause an error.)

Example 17.1.2: Using set! within a local environment

The following interactions begin by creating global variables named \(x\) and \(y\), and then using a let to create a local variable named \(x\).

\[
\begin{align*}
> & \ (\text{define } x \ '\text{xxx}) \quad ;; \text{Global variable, } x \\
> & \ (\text{define } y \ '\text{yyy}) \quad ;; \text{Global variable, } y \\
> & \ (\text{let } \ ((x \ 0)) \quad ;; \text{Local variable, } x \\
& \quad \quad \text{(set! } x \ '\text{newX}) \\
& \quad \quad ;; \text{Change value of LOCAL variable, } x \\
& \quad \quad \text{(set! } y \ '\text{newY}) \\
& \quad \quad ;; \text{Change value of GLOBAL variable, } y \\
& \quad \quad \text{(list } x \ y)) \\
& \quad \quad \text{(newX newX newY)}
\end{align*}
\]
In the body of the `let`, the two `set!` expressions are evaluated with respect to the local environment, as illustrated in Fig. 17.1. Because that local environment has an entry for \( x \), the expression `(set! x 'newX)` changes the value of \( x \) in the local environment. In contrast, there is no entry for \( y \) in the local environment; therefore, the expression `(set! y 'newY)` changes the value of \( y \) in the Global Environment. Note that the list generated as the output value for the `let` expression contains the new values for the local variable \( x \) and the global variable \( y \). Finally, evaluating \( x \) in the Global Environment shows that the global variable \( x \) has not been affected, whereas the global variable \( y \) has a new value.

**(Optional) Using `set!` within deeply nested environments.** Suppose that `(set! var newVal)` is being evaluated with respect to an environment \( E_n \) that is nested inside other environments, as follows: \( E_n \subset E_{n-1} \subset \ldots \subset E_2 \subset E_1 \subset E_0 \). The following steps are carried out:

1. The expression `newVal` is evaluated with respect to the environment \( E_n \), generating some datum \( D \).
2. The environments, \( E_n, E_{n-1}, \ldots, E_0 \), are scanned, in order, to find the first one that contains an entry for \( var \). Call that environment \( E_j \).
3. \( D \) is inserted as the new value for \( var \) in the environment \( E_j \).
4. The `set!` special form itself evaluates to `void`.

**Example 17.1.3: (Optional) Using `set!` within deeply nested environments**

Consider the following multiply nested `let` expressions. As illustrated in Fig. 17.2, each of the first three `let` expressions creates a local environment with a variable named \( x \). The fourth `let` creates a local environment with a variable named \( w \). The `set!` special form is evaluated with respect to the innermost local environment, \( E_4 \), but ends up changing the value of \( x \) in the environment, \( E_3 \), because that is the ancestor environment that is closest to \( E_4 \) and contains a variable named \( x \).

```scheme
> (let ((x 10)) ;; Environment \( E_1 \)
  (let ((x 20)) ;; Environment \( E_2 \)
    (let ((x 30)) ;; Environment \( E_3 \)
      (let ((w 40)) ;; Environment \( E_4 \)
        (printf "Evaluating x in env. E4 before: \"A\~%\" x)"
        (set! x 99)
        (printf "Evaluating x in env. E4 after: \"A\~%\" x))
        (printf "Evaluating x in env. E3 after: \"A\~%\" x))
        (printf "Evaluating x in env. E2 after: \"A\~%\" x))
        (printf "Evaluating x in env. E1 after: \"A\~%\" x))
      Evaluating x in env. E4 before: 30
      Evaluating x in env. E4 after: 99
      Evaluating x in env. E3 after: 99
      Evaluating x in env. E2 after: 20
      Evaluating x in env. E1 after: 10
```

Notice that there is no way to use `set!` in environments \( E_4 \) or \( E_3 \) to change the value of the variable named \( x \) in either \( E_2 \) or \( E_1 \), because those variables are effectively blocked by the presence of the variable named \( x \) in \( E_3 \).
Although it is important to understand which variable is affected when a \texttt{set!} special form is evaluated in a particular environment, a programmer typically avoids difficult cases by not having multiple variables with the same name in different environments.

\textbf{What now?} Okay, so we can use \texttt{set!} to change the value of a variable. How can we use that capability to our advantage? The next section introduces the \texttt{while} special form which, together with \texttt{set!}, enables a kind of computation called \textit{iteration} that can be a convenient alternative to recursion.

\section{The \texttt{while} Special Form}

The purpose of the \texttt{while} special form is to enable a kind of \textit{looping} behavior called \textit{iteration}. A typical example might be glossed as: “As long as (while) some condition \textit{Condy} holds, do some action \textit{Acty}.” The semantics for this example could be summarized as follows:

1. Evaluate the condition \textit{Condy}.
2. If \textit{Condy} evaluates to \textit{true} (or something that counts as true) then:
   3. Do the action \textit{Acty} and go back to Step 1;
   4. Otherwise, stop.

For example, \textit{Condy} might be the condition “the value of \textit{x} is positive”, and \textit{Acty} might be the compound action “first print out the value of \textit{x}; then decrease \textit{x}’s value by one.” As this example suggests, the \texttt{while} special form only makes sense in the context of destructive programming because, without destructive programming, the condition \textit{Condy} would always evaluate to the same thing. In particular, in the context of non-destructive programming, if \textit{Condy} evaluated to \textit{true} in Step 1, then it would forever evaluate to \textit{true}, leading to a situation where the action \textit{Acty} would be repeated forever. However, as in the example, if the value of \textit{x} decreases by one on each \textit{iteration}, then the condition might eventually evaluate to \textit{false}.

* \textit{Acty} is the action that is done \textit{iteratively}. Each doing of Step 3 is called an \textit{iteration}. 

*
The syntax of the while special form. The while special form has the following syntax:

\[
(\text{while } \text{condExpr} \\
\quad \text{expr}_1 \\
\quad \text{expr}_2 \\
\quad \ldots \\
\quad \text{expr}_k) 
\]

where \text{condExpr} is the condition (a.k.a., Condy); and the expressions, \text{expr}_1, \ldots, \text{expr}_k, together constitute the body (a.k.a., the compound action Acty). Notice that the syntax of the while special form is identical to that of the when special form—except for its name, of course. The essential difference is in the semantics.

The semantics of the while special form. A while special form is evaluated as follows:

1. Evaluate \text{condExpr}.
2. If \text{condExpr} evaluates to true (or something that counts as true),
3. Then evaluate each of the expressions, \text{expr}_1, \ldots, \text{expr}_k, in order, then go back to Step 1.
4. Otherwise, return \text{void} as the output of the while expression.

Note that the only way out of a while loop is via Step 4. Hence, a while special form always evaluates to void—unless the condition stays true forever, leading to an infinite loop.

---

**Example 17.2.1**

The following interactions demonstrate that:

1. the expressions in the body of a while may be evaluated zero times, and
2. a while expression evaluates to void.

```scheme
> (while #f (printf "hi!"))  ← Body evaluated zero times
#t
> (void? (while #f (printf "hi!"))))  ← The while expr. evaluates to void
```

---

**Example 17.2.2: A typical while loop**

This example illustrates how let, while and set! can be combined to create a useful while loop.

```scheme
> (let ((counter 0)) 
  (while (< counter 4) 
    ;; BODY of the WHILE: 
    (printf "counter = \"A\"%" counter) 
    ;; Increment the value of COUNTER 
    (set! counter (+ counter 1))) 
  ;; AFTER the WHILE: 
  counter)
counter = 0
counter = 1
counter = 2
counter = 3
4
```
In this example, the `let` special form creates a local environment in which the variable `counter` has an initial value of 0. As long as the value of `counter` is less than 4, the expressions in the body of the `while` are evaluated, leading to side-effect printing. In addition, on each iteration, the value of `counter` is incremented by one. As a result, the condition `(<= counter 4)` will eventually evaluate to `#f`, stopping the loop. Notice that after the `while` loop completes, `counter` has the value 4 (i.e., the value that caused the condition `(<= counter 4)` to become false).

* If you forget to modify the value of a counter variable in a while loop, then the condition that controls the while loop may forever evaluate to `#t`, resulting in an infinite loop. Whoops!

Example 17.2.3: Summing numbers iteratively

The following combination of `let`, `while` and `set!` computes the sum of the numbers from 1 to 100. Notice the use of an accumulator variable whose value is destructively modified on each iteration.

```scheme
> (let ((counter 0)
         (acc 0))
   (while (<= counter 100)
       ;; accumulate the current value of counter
       (set! acc (+ acc counter))
       ;; increment the value of counter
       (set! counter (+ counter 1)))
   ;; After the WHILE loop: the accumulator has the answer acc
5050
```

In-Class Problem 17.2.1

Define a function, called `sum-iter`, that takes a positive integer n as its only input and returns as its output the sum of the numbers from 1 to n. Define another function, called `facty-iter`, that takes a positive integer n as its only input and returns as its output the factorial of n (i.e., the product of the numbers from 1 to n). For each function, use `let`, `while` and `set!`, as demonstrated in the previous example, to implement the desired iteration. Here are some examples of the desired behavior.

```scheme
> (sum-iter 4)
10
> (facty-iter 4)
24
```

In-Class Problem 17.2.2: An iterative version of `transfer-all`

Recall the accumulator-based, tail-recursive `transfer-all` function from Example 16.4.3. Define an iterative version of this function, called `transfer-all-iter`. Your function should not be recursive; instead, it should use `let`, `while` and `set!`, to iteratively transfer all of the elements of its first input onto its second input.
In-Class Problem 17.2.3: Iterative version of the insert function

Recall the accumulator-based, tail-recursive insert-acc function from In-Class Problem 16.5.3. It began by recursively accumulating all of the numbers from the sorted list that were smaller than item, at which point it had effectively located the appropriate insertion point. Define an iterative function, called insert-iter, that uses let to create a local variable called acc that starts off empty. Next, it should use a while loop to iteratively accumulate all of the numbers from the sorted list that are smaller than item, at which point it will have located the appropriate insertion point for item. After the while loop, it can then use transfer-all (cf. Example 16.4.3) or transfer-all-iter (cf. In-Class Problem 17.2.2) to transfer all of the accumulated numbers onto the remains of the sorted list. Note that your insert-iter function should satisfy the same contract as the insert-wr function, not the insert-acc function. Instead of taking an extra acc input, it creates a local variable called acc.

In-Class Problem 17.2.4: Iterative version of insertion-sort

Using the same approach as in In-Class Problem 17.2.3, define an iterative version of the insertion-sort algorithm. Call your function isort-iter. You may wish to review the isort-acc and isort-wr functions from In-Class Problem 16.5.4.

Example 17.2.4: Using random within a while loop

This example illustrates that the value of a variable can be set to a random value on each iteration, leading to a while loop having an unpredictable number of iterations.

```scheme
> (let ((val (random 4)))
  (while (< val 3)
    (printf "val: ˜A˜%" val)
    (set! val (random 4)))
  ;; after the WHILE:
  val)
val: 2
val: 0
val: 2
val: 1
3
> (let ((val (random 4)))
  (while (< val 3)
    (printf "val: ˜A˜%" val)
    (set! val (random 4)))
  ;; after the WHILE:
  val)
val: 1
val: 0
val: 1
val: 2
val: 1
val: 0
3
```
Example 17.2.5: (Optional) Implementing our own version of `while`

The following `my-while` function provides essentially the same behavior as the `while` special form. However, it is a little clunkier to use because, unlike special forms, function-call expressions are evaluated by the Default Rule. Therefore, to ensure proper behavior, the condition and the body must be encapsulated within lambda functions, called `cond-func` and `body-func`, each of which takes zero inputs. The `my-while` function causes the desired condition to be evaluated by applying `cond-func` to zero inputs; and it causes the desired body expressions to be evaluated by applying `body-func` to zero inputs. Notice that, unlike almost all of the recursive functions seen in Part I of this book, the recursive call to `my-while` is applied to the same inputs! In so doing, the `my-while` function is implicitly relying on `cond-func` or `body-func` to be destructive, to avoid going into an endless loop.

```
;; MY-WHILE
;;  __________________________________________________________
;; INPUTS:  COND-FUNC, a function that takes zero inputs
;;          BODY-FUNC, a function that takes zero inputs
;; OUTPUT:  VOID
;; SIDE EFFECT: As long as (COND-FUNC) evaluates to
;;              (something that counts as) true, MY-WHILE evaluates
;;              (BODY-FUNC).
(define my-while
  (lambda (cond-func body-func)
    (when (cond-func)    ; Apply cond-func to zero inputs
      (body-func)       ; Apply body-func to zero inputs
    ;; Recursively call MY-WHILE with the same inputs!
    (my-while cond-func body-func)))))
```

The following interaction demonstrates the use of the `my-while` function.

```
> (let ((ctr 3))
  (printf "MY-WHILE example:˜%")
  (my-while (lambda ()              ; Apply cond-func to zero inputs
             (> ctr 0))
    (lambda ()                     ; Apply body-func to zero inputs
      (printf " ctr: ˜A˜%" ctr)
      (set! ctr (1- ctr)))
  )
MY-WHILE example:
  ctr: 3
  ctr: 2
  ctr: 1
```

Note that because `my-while` is a function, the expression, `(my-while...)`, is evaluated by the Default Rule. As a result, both lambda expressions are evaluated before calling the `my-while` function. Therefore, the corresponding lambda functions are created in the context of the local environment that includes an entry for the symbol `ctr`. Thus, whenever these lambda functions are eventually called, their bodies are evaluated in an environment that is nested within the environment that has an entry for `ctr`. Thus, any occurrences of the symbol `ctr` in the bodies of those functions will refer to the local variable `ctr`, as desired.

Here is the equivalent example done using the `while` special form.

```
> (let ((ctr 3))
  (printf "WHILE example:˜%")
```
17.3 Converting Tail Recursion to Iteration

Recall from Section 14.2 that for any tail-recursive function, DrScheme can employ a memory-saving trick whereby a single function-call box is repeatedly recycled, instead of creating a new function-call box for each recursive function call. In effect, what DrScheme does is to convert a tail-recursive function call into iteration. This section describes the process.

We begin by recalling the tail-recursive print-n-dashes function, seen previously in Example 14.2.1, and then showing how it can be implemented iteratively using while and set!.

Example 17.3.1: Implementing print-n-dashes iteratively

Here is the print-n-dashes function. Note that it is tail recursive.

;; 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 "-
")
       ;; Let the recursive func call print the rest of the dashes
       (print-n-dashes (- n 1))))))

Here is an equivalent function, called print-n-dashes-iter, that is implemented using iteration.

(define print-n-dashes-iter
  (lambda (n)
    ;; Iterative Case: N > 0
    (while (> n 0)
      (printf "-
")
      (set! n (- n 1)))
    ;; Base Case: N <= 0
    (newline)))
The following interactions demonstrate that the two functions provide the same functionality.

> (print-n-dashes 5)
-----
> (print-n-dashes 8)
--------
> (print-n-dashes-iter 5)
-----
> (print-n-dashes-iter 8)
--------

For the recursive function, the recursive case involves the printing of a single dash followed by a tail-recursive function call with the input \((- n 1)\). For the iterative function, each iteration involves the printing of a single dash followed by destructively decrementing the value of \(n\) by one. For both functions, the base case is reached when the value of \(n\) is zero, resulting in a newline being generated. For the recursive function, the base case is one of the cases that is explicitly handled by the \texttt{cond} expression; for the iterative function, the base case is what happens after the \texttt{while} loop is completed.

Unlike in the above example, most of the times we want to create a tail-recursive function, we start by defining a tail-recursive helper function that takes extra inputs (e.g., accumulators). Afterward, we define a wrapper function that calls the tail-recursive helper function with suitable inputs. These more typical cases of tail-recursion can also be easily converted into iteration, as demonstrated by the following example.

### Example 17.3.2

Recall the \texttt{sum-to-n-acc} function, seen previously in Example 14.3.2.

```
;; 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=\texttt{~A})\" acc)
        ;; Return the accumulator!
        acc)
      ;; Recursive Case: n > 0
      (#t
        (printf "Recursive Case (n=\texttt{~A}, acc=\texttt{~A})\" n acc)
        ;; Make recursive function call with updated inputs
        (sum-to-n-acc (- n 1) (+ acc n))))))
```

Here is the corresponding wrapper function:
;; SUM-TO-N-WR
;; -------------------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: The sum, 0 + 1 + 2 + ... + N

(define sum-to-n-wr
  (lambda (n)
    ;; Call the accumulator-based helper with ACC=0:
    (sum-to-n-acc n 0)))

And here is an iterative function, sum-to-n-iter, that performs the same computation:

;; SUM-TO-N-ITER
;; -------------------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: The sum, 0 + 1 + 2 + ... + N

(define sum-to-n-iter
  (lambda (n)
    ;; Create local variable ACC whose initial value is 0
    (let ((acc 0))
      ;; Iterative Case: N > 0
      (while (> n 0)
        ;; Accumulate the current value of N
        (set! acc (+ acc n))
        ;; Decrement N by one
        (set! n (- n 1)))
      ;; After the WHILE, ACC has the answer
      acc)))

When there are multiple base cases and multiple recursive cases, the conversion from recursion to iteration can still be done quite easily.

Example 17.3.3: (Optional) A more complex example of converting from recursion to iteration

The following tail-recursive function walks through a list of numbers, searching for an occurrence of num. Along the way, it prints out information about the numbers it passes by: + for numbers bigger than num, - for numbers smaller than num. As its output, it returns #f if num wasn’t found in the list; otherwise, it returns the index of the position where num was found. Although this function can easily be implemented without multiple base cases or multiple recursive cases, implementing it in this way enables us to demonstrate the general process of converting a tail-recursive function to an iterative function.

;; INDEX-OF-NUM-IN-LIST-ACC
;; -------------------------------------------------------
;; INPUTS: NUM, a number
;; LISTY, a list of numbers
;; INDY, current index
;; OUTPUT: When called with INDY = 0, the output is
;; the index of the first occurrence of NUM in LISTY;
;; or #f if NUM does not occur in LISTY.
Chapter 18

Vectors

A vector is a data structure that, like a list, contains an ordered sequence of data. However, there are many significant differences between vectors and lists.

Recall that lists can be incrementally extended, one element at a time, using the cons function. As a result, as illustrated in Fig. 18.1, the individual cons cells in which the various elements of a list are stored frequently end up being scattered haphazardly about the computer’s memory. For this reason, accessing elements of a list can be relatively slow. For example, to access the one-thousandth element of a list typically requires recursively walking through the first thousand cons cells in the chain. However, as has been demonstrated repeatedly in previous chapters, much recursive processing of lists can be done while only accessing the first and rest of a list. For these sorts of applications, lists are quite handy. Furthermore, many list-based functions can be written non-destructively, which facilitates testing and debugging, since the performance of a non-destructive function depends only on the code within its body.

In sharp contrast to lists, vectors are stored within a single, contiguous block of memory—which has both advantages and disadvantages. The principal advantage is that every item of a vector can be accessed very quickly, whether the first element or the thousandth. To demonstrate why, Fig. 18.2 illustrates the layout of the elements in a typical vector. Notice that each slot has a numerical index, starting at zero. A crucial feature of the layout is that each slot takes up the same amount of space (e.g., four bytes per slot in the figure). Since each slot has the same size, the memory address of any given slot is very easy to compute. For example, if the start of the vector is located at memory address 1000, and each slot is four bytes wide, then the address of the \(i\)th slot is \(1000 + 4 \times i\). (Thankfully, DrScheme takes care of such low-level details. A Scheme programmer need never deal directly with memory addresses.) An important feature of the “Random Access Memory” (RAM) found in modern computers is that the contents of any memory address can be fetched in the same (very small) amount of time. (The term “random” here is used to indicate that the contents of any randomly selected memory address can be fetched in the same, very small amount of time.) Thus, the time required to fetch the first element of a vector (i.e., the element with index 0) is the same as the time required to fetch the thousandth element.

One disadvantage of vectors is that the block of memory that will hold the vector’s elements must be allocated in one chunk. As a result, vectors cannot be easily extended to accommodate new elements. In particular, if a vector is created to hold up to \(N\) elements, then it will never be able to hold more than \(N\) elements. There is no analog to the cons function for vectors. Each vector has a fixed length.

Since not all of a vector’s elements may be known at the time the block of memory is allocated, destructive programming is typically used with vectors. For example, a Scheme function might decide to set the 24th element of a vector to be \#t, then later set the 36th element to be 34.2, and still later set the 24th element to be \#f (erasing the prior contents of the 24th slot). Just as accessing any desired slot of a vector is very efficient, so too is the operation of destructively modifying the contents of any desired slot. (Again, this is a feature of Random Access Memory.) Thus, the speed of accessing and modifying the contents of a vector is balanced by the challenge of using destructive programming, which can make testing and debugging your functions a little more difficult. Nonetheless, the tradeoff can be quite worthwhile; and if care is taken, the risks associated with using destructive programming can be mitigated.
18.1 Vector Expressions

Chapter 2 introduced several types of primitive data expressions, including numbers, booleans, the empty list, and symbols. For each type of expression, the syntactic rules for character sequences denoting that type of data were described. Then, Chapter 3 described how instances of those data types are evaluated. Similarly, Chapter 6 presented the syntactic rules for character sequences that denote non-empty lists, and described how the Default Rule evaluates non-empty lists. Following this trend, this section presents the \#(...) syntax—called the pound syntax—for character sequences that denote vectors, as well as the semantics of evaluation for vectors. (Vectors evaluate to themselves.) It should be noted, however, that the pound syntax has limited use because the vectors it denotes are immutable (i.e., their contents cannot be changed). The pound syntax is typically only useful for testing functions that look at the contents of vectors, and perhaps do some computations based on those contents, but do not try to change the contents in any way. After presenting the pound syntax, the next section presents built-in functions for creating mutable vectors (i.e., vectors whose contents can be changed), which turn out to be more useful in practice.

A vector is a datum. Scheme provides a special syntax, the pound syntax, for denoting vectors. Just as the expression \((a \ b \ c)\) can be used to represent (or denote) a list containing the three elements \(a\), \(b\) and \(c\), the expression \#(a b c) can be used to represent (or denote) a vector containing the three elements \(a\), \(b\) and \(c\). In general, any expression of the form \#(\(e_1\ \ e_2\ \ \ldots\ \ e_n\)) can be used to represent a vector containing the \(n\) elements denoted by the expressions \(e_1\), \(e_2\), \ldots, \(e_n\). Thus, for example, the expression \#(\(x\ \ #t\ \ ()\ 32\)) denotes a vector containing four elements: a symbol, a boolean, the empty list, and a number.

One important fact about the \#(...) syntax is that the vectors it represents are immutable (i.e., their contents cannot be changed). This limits the usefulness of the \#(...) syntax. However, it can be useful when testing functions that don’t need to modify their vector inputs (e.g., functions that print out the contents of a vector). Immutable vectors can also be useful as the values of global constants. For example, the vector \#(clubs hearts diamonds spades) could be used to denote the suits in a deck of cards, and the vector

\[
\begin{array}{|c|c|c|}
\hline
\text{Index} & \text{Element} & \text{Memory Address} \\
\hline
0 & 32 & 1000 \\
1 & #t & 1004 \\
2 & () & 1008 \\
3 & 3.14 & 1012 \\
\vdots & \vdots & \vdots \\
1 & 26 & 1000 + 1*4 \\
\vdots & \vdots & \vdots \\
\hline
\end{array}
\]

Note: This example presumes that each element of the vector occupies four bytes of memory.
#(sun mon tue wed thu fri sat) could be used to denote the days of the week.

* Unlike lists, **vectors evaluate to themselves!**

**Example 18.1.1: Demonstrating that vectors evaluate to themselves**

The following interactions use the #(...) syntax to demonstrate that vectors evaluate to themselves.

```scheme
> #(1 2 a b #t (+ 2 3))  ← The pound syntax denotes a vector
#(1 2 a b #t (+ 2 3))  ← That vector evaluates to itself!
> #(a b #(c d e) (x y z))
#(a b #(c d e) (x y z))
```

With the #(...) syntax, there is no need to quote the subsidiary expressions. For example, #(a b c) simply denotes the vector containing the three symbols a, b and c. When the vector gets evaluated, its elements are not evaluated! The vector simply evaluates to itself—without evaluating any of its elements. The Default Rule for evaluating non-empty lists does not apply to vectors!

Incidentally, DrScheme uses the #(...) syntax to report results that are vectors, whether they are immutable or mutable.

* Remember: Vectors denoted by the # syntax are **immutable!** Once created, their contents can’t be changed.

### 18.2 Constructing Vectors

Scheme provides two built-in functions, called **vector** and **make-vector**, that can be used to create new vectors. The **vector** function is similar to the built-in **list** function, mentioned briefly in Section 16.1. The **make-vector** function is typically the most practical. It enables the programmer to create a vector of any specified length.

#### 18.2.1 The Built-in **vector** Function

The **vector** function is a built-in function that works very much like the built-in **list** function, mentioned briefly in Section 16.1. Whereas **list** constructs a list containing the specified elements, **vector** constructs a vector containing the specified elements. Unlike the vectors denoted by the #(...) syntax, vectors created by the **vector** function are **mutable** (i.e., their contents can be changed—we’ll see how momentarily).

**Example 18.2.1: Using the **vector** function**

Here’s the contract for the **vector** function, followed by some examples of its use.

```scheme
;;  VECTOR -- built-in function
;;  -----------------------------------------------
;;  INPUTS:  ELT1, ELT2, ELT3, ... : any number of inputs
;;  OUTPUT:  A *mutable* vector containing the specified elements

> (vector 1 (+ 2 3) ’a)
#(1 5 a)
> (list 1 (+ 2 3) ’a)
(1 5 a)
> (define vecky (vector 1 (+ 2 3) ’a))
> vecky
```
#(1 5 a)
> (define vecky-two (vector 100 vecky 200))
> vecky-two
#(100 #(1 5 a) 200)

Note that since vector is a built-in function, expressions such as (vector 1 (+ 2 3) ’a) are evaluated by the Default Rule. Thus, all input expressions are evaluated before being passed as input to the vector function—which is why the a in the first two examples had to be quoted.

18.2.2 The make-vector Function

The make-vector function is a built-in function that can be used to create a vector of any specified length. It is the most common way of creating a vector because it can be used, for example, to easily create a vector with, say, 1000 slots. Like with the vector function, vectors created by make-vector are mutable (i.e., their contents can be changed). The make-vector function can be called with either one or two inputs (i.e., the second input is optional). The first input specifies how many slots the new vector should contain. The second input, if provided, specifies the initial value for all of the slots. If the second input is not provided, make-vector fills each slot with a default value of 0.

Example 18.2.2: Using the make-vector function

Here’s the contract for the make-vector function, followed by some examples of its use.

;; MAKE-VECTOR -- built-in function
;; ___________________________________________________________
;; INPUTS: NUM-SLOTS, a non-negative integer
;; INIT-VALUE, an *optional* input, can be anything
;; OUTPUT: A vector with NUM-SLOTS slots, each initially
;; filled with INIT-VALUE (or 0 if INIT-VALUE
;; not provided)

> (make-vector 5) ← called with one input
#(0 0 0 0 0)
> (make-vector 15)
#(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> (make-vector 5 ’a)
#(a a a a)
> (make-vector 15 ’_)
#(_ _ _ _ _ _ _ _ _ _ _ _ _ _ _)

18.3 Accessing Information Stored in a Vector

To be of any use, the information stored in a vector must be accessible to the programmer. Scheme provides the vector-ref function to access the contents of any specified slot in a vector, and the vector-length function to access the (fixed) number of elements in a vector.
18.3.1 The \texttt{vector-ref} Function

Each element of a vector is identified by its numerical index. The built-in \texttt{vector-ref} function provides an easy way to access any element of a vector by its index.

\begin{verbatim}
Example 18.3.1: Using \texttt{vector-ref}

Here is the contract for the \texttt{vector-ref} function, followed by some examples of its use.

\begin{verbatim}
;;\texttt{VECTOR-REF} -- built-in function
;;-----------------------------------------------
;;INPUTS: VECKY, a vector
;;INDY, an index
;;OUTPUT: The item of VECKY stored at slot INDY

> (define vecky #(a b c d e))
> (vector-ref vecky 0)
a
> (vector-ref vecky 2)
c
\end{verbatim}
\end{verbatim}

\end{verbatim}

* Although the \texttt{vector-ref} and \texttt{list-ref} functions may appear similar syntactically, they operate quite differently. The \texttt{vector-ref} function accesses the specified element of a vector nearly instantaneously, while the \texttt{list-ref} function walks through each element of the specified list until it finds the one with the desired index. Thus, \texttt{vector-ref} is much more efficient than \texttt{list-ref}.

18.3.2 The \texttt{vector-length} Function

As already mentioned, each vector has a fixed length. For this reason, the length of the vector can be stored with the vector itself, when it is created. Thus, the built-in \texttt{vector-length} function does not need to walk through the entire vector to figure out how long it is; instead it can simply look up the \texttt{length} information that is stored with the vector. (The \texttt{length} of a vector is an example of a \textit{field} in a data structure, which will be discussed in the next chapter.) Thus, using \texttt{vector-length} is \textit{very fast} with any vector, no matter how long. This is quite different from the built-in \texttt{length} function for lists which must walk all the way through a list in order to figure out how many elements it has.

\begin{verbatim}
Example 18.3.2: Using \texttt{vector-length}

Here is the contract for \texttt{vector-length}, followed by some examples of its use.

\begin{verbatim}
;;\texttt{VECTOR-LENGTH} -- built-in function
;;-----------------------------------------------
;;INPUT: VECKY, a vector
;;OUTPUT: The number of elements/slots in that vector

> (define vecky #(a b c d e))
> (vector-length vecky)
5
> (vector-length (make-vector 25))
25
\end{verbatim}
\end{verbatim}

\end{verbatim}
In-Class Problem 18.3.1: Fetching a random element from a vector

Define a function, called `fetch-random-element`, that satisfies the following contract:

```
;; FETCH-RANDOM-ELEMENT
;; -----------------------------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: One of the elements of VECKY, chosen at random
```

Here are some examples of the desired behavior:

```
> (fetch-random-element #(a b c d e f))
c
> (fetch-random-element #(a b c d e f))
a
> (fetch-random-element #(a b c d e f))
d
```

Hint: Use the built-in `vector-length`, `random` and `vector-ref` functions.

### 18.4 Recursively or Iteratively Processing Vectors

Since a vector is an ordered sequence of elements, we can define recursive functions to walk through vectors in much the same way that we defined recursive functions to walk through lists. One example of this will be given below. However, writing recursive functions to walk through vectors would quickly grow tiresome because it would involve repeating the same kind of pattern over and over again. To avoid this kind of repetition, we will instead use the `do` special form to enable us to iteratively walk through any vector. However, before introducing this use of `do`, we first demonstrate how to manually write a recursive function to walk through some or all of a vector.

#### Example 18.4.1: (Optional) Manually walking through a vector

The following function prints out the contents of a vector from a given starting index. On each recursive function call, the value of the index is incremented, until it goes past the end of the vector. Recall that if a vector has length `n`, then the legal indices range from 0 to `n - 1`.

```
;; PRINT-VECTOR-FROM
;; -----------------------------------------------
;; INPUTS: VECKY, a vector
;; I, a non-negative integer, no greater than
;; the length of VECKY
;; OUTPUT: None
;; SIDE EFFECT: Prints out the elements of VECKY
;; from index I onward.

(define print-vector-from
  (lambda (vecky i)
    (cond
      ;; Base Case: I is too big
      ;; Note: The last legal index of VECKY is LENGTH-1
      (>= i (vector-length vecky))
      (void))
  )
)
;; Recursive Case: I is a legal index
(else
    ;; Print one element
    (printf "Element \^A of VECKY is: \^A\^%" i (vector-ref vecky i))
    ;; Let recursion print the rest of the elements
    (print-vector-from vecky (+ i 1)))))))

> (print-vector-from #(a b c d) 0)
Element 0 of VECKY is: a
Element 1 of VECKY is: b
Element 2 of VECKY is: c
Element 3 of VECKY is: d
> (print-vector-from #(a b c d) 2)
Element 2 of VECKY is: c
Element 3 of VECKY is: d

The wrapper function, print-vector-wr, can then be defined as follows:

;;; PRINT-VECTOR-WR
;;; -------------------------------------------------------------
;;; INPUT: VECKY, a vector
;;; OUTPUT: None
;;; SIDE EFFECT: Prints out the elements of VECKY

(define print-vector-wr
    (lambda (vecky)
        (print-vector-from vecky 0)))

> (print-vector-wr #(a b c))
a
b
c

* In general, any function that needs to recursively process the elements of a vector can do so by defining a helper function that includes an extra input i whose value is initially zero and increments by one on each recursive function call until it exceeds the last legal index for the given vector.

However, the dotimes special form simplifies the task of walking through a vector.

### Example 18.4.2: Printing the contents of a vector using dotimes

The print-vector function, below, illustrates the use of the dotimes special form to walk through a vector, printing out its contents. Note the use of the vector-length function to specify the number of iterations to perform.

;;; PRINT-VECTOR
;;; -------------------------------------------------------------
;;; INPUT: VECKY, a vector
;;; OUTPUT: None
;;; SIDE EFFECT: Displays the contents of VECKY in the
(define print-vector
  (lambda (vecky)
    ;; I takes on the values: 0, 1, 2, ..., LENGTH-1
    (dotimes (i (vector-length vecky))
      ;; Print out the Ith element of VECKY
      (printf "A: A\n" i (vector-ref vecky i)))))

> (define vecky #(a b c d e))
> (print-vector vecky)
a
b
c
d
e

Example 18.4.3: Printing the contents of a vector in reverse order

The following function prints out the contents of a given vector in reverse order. Notice the use of the local variable rev-indy, whose value is the index of the next element to be printed. For example, you should convince yourself that for a vector of length four, the counter variable indy will range from 0 to 3, while the local variable rev-indy will range from 3 to 0.

;; PRINT-IN-REVERSE
;; --------------------------------------------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: None
;; SIDE EFFECT: Prints out the contents of VECKY
;; in reverse order

(define print-in-reverse
  (lambda (vecky)
    ;; LEN = number of elements in VECKY
    (let ((len (vector-length vecky)))
      (dotimes (indy len)
        ;; INDY takes on values from 0 to LEN-1
        ;; REV-INDY takes on values from LEN-1 to 0
        (let ((rev-indy (- len indy 1)))
          (printf "vecky[\nA\n] = A\n" rev-indy
                  (vector-ref vecky rev-indy))))))

> (print-in-reverse #(a b c d))
vecky[3] = d
vecky[2] = c
vecky[1] = b
vecky[0] = a
In-Class Problem 18.4.1: Using `dotimes` to print out every other element of a vector

Define a function, called `print-every-other-one-veck`, that takes a vector as its only input. It should not return any output value. Instead, it should print out every other element of the given vector. Here is the contract, followed by some examples:

```scheme
;; PRINT-EVERY-OTHER-ONE-VECK
;; ------------------------------------------
;; INPUT:  VECK, a vector
;; OUTPUT: None
;; SIDE EFFECT: Prints out every other element of VECK

> (print-every-other-one-veck #(a b c d e))
a
c
e
> (print-every-other-one-veck #(a a b b c c d d))
a
b
c
d
```

Hint: Use the `even?` function. If the current index is even, then print out the corresponding element.

In-Class Problem 18.4.2: Testing whether two vectors are “equal”

Define a function, called `vector-equal?`, that satisfies the following contract:

```scheme
;; VECTOR-EQUAL?
;; ------------------------------------------
;; INPUTS: VECK-ONE and VECK-TWO, any vectors
;; OUTPUT: #t if VECK-ONE and VECK-TWO have the same elements, in the same order; #f otherwise.
```

Here are some examples:

```scheme
> (vector-equal? #(a b c) #(a b c))
#t
> (vector-equal? (make-vector 3) (vector 0 0 0))
#t
> (vector-equal? #(a b c) #(a b c d))
#f
```

Notice that the two input vectors cannot be equal if they have different lengths. Therefore, `vector-equal?` can immediately return `#f` if the two vectors have different lengths. On the other hand, if they do have the same length, then it can call a helper function, `vector-equal?-helper`, to manually walk through the vectors in parallel, comparing their corresponding elements. Note that using `dotimes` is not an option for this problem because the helper function should be able to stop early if it ever discovers corresponding elements that are not the same. Here’s the contract for the helper function:

```scheme
;; VECTOR-EQUAL?-HELPER
;; ------------------------------------------
```
Notice that the helper function is only ever called on vectors having the same length.

After you've implemented this function, you may wish to know that the built-in equal? function can be used to test the equality of vectors, as illustrated below.

```scheme
> (equal? #(1 2 3) #(1 2 3))
#t
> (equal? #(a b c) (vector 'a 'b 'c))
#t
> (equal? #(0 0 0) (make-vector 4))
#t
```

### 18.5 Destructively Modifying a Vector

The `vector-set!` function is provided to enable a programmer to destructively modify the contents of a specified slot in a vector. It is frequently called a mutator function, because it enables a programmer to mutate the contents of a vector.

* The name of the function ends with an exclamation point to remind us of its destructive side effect.

#### Example 18.5.1: The `vector-set!` function

Here is the contract for the `vector-set!` function, followed by an example of its use.

```scheme
;; VECTOR-SET! -- Built-in Function
;; -------------------------------------------
;; INPUTS: VECKY, a vector
;; INDY, a numerical index
;; NEW-VAL, anything
;; OUTPUT: *void*
;; SIDE EFFECT: Destructively changes the contents of VECKY
;; at slot INDY to become NEW-VAL

> (define vecko (vector 0 10 20 30))
> vecko
#(0 10 20 30)
> (vector-set! vecko 2 'x)
> vecko
#(0 10 x 30)
```
In-Class Problem 18.5.1: Initializing a vector

Define a function, called `init-veck`, that satisfies the following contract:

```
;; INIT-VECK
;; -----------------------------------------------
;; INPUT: VECK, a vector
;; OUTPUT: Don’t care
;; SIDE EFFECT: Sets the value of slot 0 to 0, the value of slot 1 to 1, and so on.
```

Here is an example of its use:

```
> (define vecky (make-vector 5))
> vecky
#(0 0 0 0 0)
> (init-veck vecky)
> vecky
#(0 1 2 3 4)
```

Hint: Use `dotimes` and `vector-set!`.

In-Class Problem 18.5.2: Swapping elements of a vector

Define a destructive function, called `vector-swap!`, that destructively modifies a vector by swapping two of its elements as specified by the following contract:

```
;; VECTOR-SWAP!
;; -----------------------------------------------
;; INPUTS: VECKY, a vector
;; I, J, two numerical indices
;; OUTPUT: don’t care
;; SIDE EFFECT: Destructively swaps the contents of VECKY at slots I and J.
```

Here are some examples of its use:

```
> (define vecky (vector 'a 'b 'c 'd 'e 'f))
> vecky
#(a b c d e f)
> (vector-swap! vecky 1 4)
> vecky
#(a e c d b f)
> (vector-swap! vecky 0 4)
> vecky
#(b e c d a f)
```
18.6 Summary

This chapter introduced vectors that, like lists, are ordered sequences of data. However, vectors and lists have many important differences. For example, whereas the elements of a list reside in cons cells that may be scattered haphazardly throughout a computer’s memory, the contents of a vector reside within a single, contiguous block of memory. As a result, accessing the individual elements of a vector is typically much faster than accessing the individual elements of a list. Whereas lists are especially amenable to recursive processing using non-destructive programming, vectors are especially amenable to iterative processing using destructive programming. Although care must be taken when using destructive programming with vectors, the efficiency of vectors often makes the tradeoff worthwhile.

Scheme provides a special syntax, the pound syntax, for denoting immutable vectors. For example, the expression #(1 a x) denotes a vector containing three elements. The pound syntax is of limited use, for example, when testing functions that look at vectors but do not try to change their contents, or for denoting global constants (e.g., a vector containing the names of the days of the week).

To enable programmers to do meaningful computations with vectors, Scheme provides a small handful of built-in functions. The functions, vector and make-vector, are constructor functions that can be used to create new vectors, either by explicitly specifying all of the individual elements or by specifying only the total number of slots. The functions, vector-ref and vector-length, are accessor functions that can be used to access the information contained in a vector: vector-ref fetches a specified element from a vector, while vector-length returns the number of elements contained in a vector. The function, vector-set!, is a mutator function: it can be used to destructively modify the contents of an individual slot in a vector. (There is also a type-checker predicate for vectors, called vector?, but it is rarely needed.)

Because the individual elements of a vector are accessed by their numerical indices, the dotimes special form frequently comes in handy when needing to perform computations on the contents of vectors.

Built-In Functions Introduced in this Chapter

- vector Construct a vector containing the specified items
- make-vector Construct a vector of a specified length
- vector-ref Fetches a specified element of a given vector
- vector-length Fetches the length of a given vector
- vector-set! Sets the value of a specified slot of a given vector
- vector->list Convert vector into a list (cf. Problem 18.10)
- list->vector Convert list into a vector (cf. Problem 18.11)
Chapter 19

Data Structures

The early chapters of this book introduced several kinds of primitive data in Scheme, including numbers, booleans, the empty list, void, and symbols. Each instance of primitive data is indivisible in the sense that it does not have any parts that the programmer can access or modify. In contrast, a data structure is an organized collection of data whose parts the programmer can access or modify. Data structures come in many varieties. For example:

- A vector is an example of a data structure whose slots are accessed by their numerical indices. Such data structures may be called index-based data structures.

- A cons cell is an example of a data structure whose slots are accessed by name. The named slots in such a data structure are called fields; and the data structures are called field-based data structures. For example, the fields in a cons cell are called first and rest.¹

- A list is an example of a composite data structure that is recursively defined. In particular, a list is defined by the following two rules:
  
  (Base Case) The empty list is a list.
  
  (Recursive Case) A cons cell whose rest field contains a list is a list.

  The recursive nature of lists is what enables us to define recursive functions that process any list, no matter how complicated.

This chapter focuses on field-based data structures.

Example 19.0.1: Motivating field-based data structures, I

Suppose you wanted to write a program that needed to represent dates, such as October 22, 1958 or December 7, 1941. Since each date includes a month, a day, and a year, you could easily store each date in a vector with three slots. However, to help distinguish your dates-as-vectors from other small vectors, you might want to include an extra slot whose value would be some easily recognizable symbol, such as i-am-a-date!. Using this approach, the above-mentioned dates might be represented by the following vectors:

<table>
<thead>
<tr>
<th>0</th>
<th>i-am-a-date!</th>
<th>0</th>
<th>i-am-a-date!</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1958</td>
<td>1</td>
<td>1941</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

The following interactions demonstrate how you might use such dates-as-vectors:

¹Although a vector is primarily an index-based data structure, it also contains a field, called length, whose value is accessed by the vector-length function.
> (define my-birthday (vector 'i-am-a-date! 1958 10 22))
> my-birthday
> #(i-am-a-date! 1958 10 22)
> (vector-ref my-birthday 2)
10
> (vector-ref my-birthday 1)
1958

*Notice that to access a particular item requires knowing which index to use (e.g., 2 for month, and 1 for year).*

The following example takes a more structured approach to storing date information in vectors.

### Example 19.0.2: Motivating field-based data structures, II

*First, we define some useful constants.*

```scheme
;; The unique identifier for dates
(define *date-id* 'i-am-a-date!)

;; Names for the various indices
(define *date-year-index* 1)
(define *date-month-index* 2)
(define *date-day-index* 3)

;; *MONTHS* -- A vector of names for the months
;; -----------------------------------------------------
;; To enable using the standard numbers for months,
;; we use a vector of length 13, ignoring slot 0.
;; And, since the names of the months won’t change,
;; we can use an *immutable* vector.
(define *months*  
  #(0 Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec))
```

*Next, we define a constructor function (i.e., a function that creates an instance of a date-as-vector).*

```scheme
;; MAKE-DATE -- a constructor function
;; --------------------------------------------------------------------------
;; INPUTS: YEAR, an integer
;; MONTH, an integer between 1 and 12
;; DAY, a positive integer between 1 and 31
;; OUTPUT: A "date-as-vector" containing the given information

(define make-date
  (lambda (year month day)
    (vector *date-id* year month day)))
```

*Although it may not be used all the much, here’s a type-checker predicate for a date-as-vector.*
;; DATE? -- type-checker predicate for dates
;; ---------------------------------------------------------------
;; INPUT: DATUM, anything
;; OUTPUT: #t, if DATUM is a "date-as-vector"

(define date? (lambda (datum)
    ;; Output #t if DATUM is a vector with four slots,
    ;; and whose zeroeth slot contains the *DATE-ID*:
    (and (vector? datum)
        (= 4 (vector-length datum))
        (eq? *date-id* (vector-ref datum 0))))))

Next, we define some accessor functions (i.e., functions that enable us to access the information stored in a date-as-vector).

;; DATE-YEAR -- Accessor Function
;; ---------------------------------------------------------------
;; INPUT: DATEY, a "date-as-vector"
;; OUTPUT: The year stored in DATEY

(define date-year (lambda (datey) (vector-ref datey *date-year-index*)))

;; DATE-MONTH -- Accessor Function
;; ---------------------------------------------------------------
;; INPUT: DATEY, a "date-as-vector"
;; OUTPUT: The month stored in DATEY

(define date-month (lambda (datey) (vector-ref datey *date-month-index*)))

;; DATE-DAY -- Accessor Function
;; ---------------------------------------------------------------
;; INPUT: DATEY, a "date-as-vector"
;; OUTPUT: The day stored in DATEY

(define date-day (lambda (datey) (vector-ref datey *date-day-index*)))

And, finally, some mutator functions (i.e., functions that enable us to destructively modify the contents of a date-as-vector).

;; SET-DATE-YEAR! -- Mutator Function
;; ---------------------------------------------------------------
;; INPUTS: DATEY, a "date-as-vector"
;; NEW-VAL
;; OUTPUT: Don’t care
;; SIDE EFFECT: Destructively modifies the contents of
;; the YEAR slot to be NEW-VAL.
(define set-date-year!
  (lambda (datey new-val)
    (vector-set! datey *date-year-index* new-val)))

;; SET-DATE-MONTH! -- Mutator Function
;; ---------------------------------------------------------------------------
;; INPUTS: DATEY, a "date-as-vector"
;; NEW-VAL
;; OUTPUT: Don’t care
;; SIDE EFFECT: Destructively modifies the contents of
;; the MONTH slot to be NEW-VAL.
(define set-date-month!
  (lambda (datey new-val)
    (vector-set! datey *date-month-index* new-val)))

;; SET-DATE-DAY! -- Mutator Function
;; ---------------------------------------------------------------------------
;; INPUTS: DATEY, a "date-as-vector"
;; NEW-VAL
;; OUTPUT: Don’t care
;; SIDE EFFECT: Destructively modifies the contents of
;; the DAY slot to be NEW-VAL.
(define set-date-day!
  (lambda (datey new-val)
    (vector-set! datey *date-day-index* new-val)))

The following interactions demonstrate the use of dates-as-vectors to store date information:

> (define my-birthday (make-date 1958 10 22)) ← Creating a date
> (date? my-birthday) ← Using the type checker
#t
> (date? (make-date 1941 12 7))
#t
> (date? (make-vector 4))
#f
> (date-year my-birthday) ← Accessing the contents
1958
> (date-month my-birthday)
10
> (date-day my-birthday)
22
> (set-date-month! my-birthday 11) ← Changing the contents
> my-birthday
#{i-am-a-date! 1958 11 22}
> (set-date-year! my-birthday 1968)
> my-birthday
#{i-am-a-date! 1968 11 22}
19.1 The define-struct Special Form

The preceding examples demonstrate that vectors can be used as data structures to store and manage any kind of data, and that a variety of functions can be defined to facilitate common tasks associated with using these vectors as data structures: constructing new instances of the data structures, and accessing or mutating their contents. Although these functions are not complicated, providing their definitions is both time-consuming and unilluminating. For this reason, Scheme provides a special form, called define-struct, that makes defining new data structures quite easy. In particular, evaluating a define-struct special form results in the automatic definition of all the needed constructor, accessor, mutator, and type-checker functions.

A define-struct special form has the following syntax:

\[
\text{(define-struct } \text{structName} \ (\text{fname}_1 \ \text{fname}_2 \ldots \ \text{fname}_k))
\]

where:
- \text{structName} is a symbol that will be the name of the data structure; and
- \text{fname}_1, \text{fname}_2, \ldots, \text{fname}_k are \(k\) symbols that will be the names of the fields of the new data structure.

The semantics of the define-struct special form stipulates that its evaluation generates no output, but has the side effect of defining all of the following functions:
- a constructor function, named \text{make-structName};
- a type-checker predicated, named \text{structName}?;
- \(k\) accessor functions, named \text{structName–fname}_1, \ldots, \text{structName–fname}_k; and
- \(k\) mutator functions, named \text{set-structName–fname}_1!, \ldots, \text{set-structName–fname}_k!.

Example 19.1.1: Using define-struct to define a date data structure

The following expression is all that is needed to define a date data structure in Scheme:

\[
(\text{define-struct date} \ (\text{year} \ \text{month} \ \text{day}))
\]

Its evaluation generates all of the following functions:
- **Constructor**: make-date
- **Type checker**: date?
- **Accessors**: date-year, date-month, date-day
- **Mutators**: set-date-year!, set-date-month!, set-date-day!

The following interactions demonstrate their use:

```
> (define-struct date (year month day))
> (define my-bday (make-date 1958 10 22))
> (date-year my-bday)
1958
> (date-month my-bday)
10
> (set-date-year! my-bday 1978)
> (date-year my-bday)
1978
>my-bday
#<date>
```
The last expression demonstrates that DrScheme is not terribly helpful when displaying instances of our new data structure: it just reports that it is such an instance. However, we can define our own display function, as follows.

```scheme
;; PRINT-DATE
;; -------------------------------------------------------
;; INPUT: DATEY, an instance of a DATE data structure
;; OUTPUT: None
;; SIDE EFFECT: Displays the date stored in DATEY in
;; the following format: Month Day, Year.
(define print-date
  (lambda (datey)
    (printf "~A ˜A, ˜A˜%"
      ;; Fetch name of month from *MONTHS* vector
      (vector-ref *months* (date-month datey))
      (date-day datey)
      (date-year datey))))
```

Here are some examples of its use:

```
> (print-date my-bday)
Oct 22, 1958
> (print-date (make-date 1941 12 7))
Dec 7, 1941
```

**Example 19.1.2: Implementing a deck of cards**

Below, we define a deck data structure that has two fields: cards and num-left. The cards field will be a vector that is partitioned into two sections, as illustrated in Fig. 19.1. The first section will contain...
all of the cards that have not yet been dealt; the second section will contain those that have already been dealt. Thus, the second section of the vector is effectively a “discard pile”. The num-left field serves two purposes. First, it keeps track of the number of cards that have not yet been dealt (i.e., that remain available for dealing). Second, it corresponds to the index at the top of the discard pile. For example, as shown in the figure, suppose that there are ten cards in the deck and num-left is 6. This means that there are six cards that have not yet been dealt. Those cards are located in the first part of the cards vector, at positions 0, 1, . . . , 5. The “discard pile” begins at index 6. The four cards in the discard pile are at positions 6, 7, 8 and 9. In general, the cards available for dealing have indices between 0 and (num-left – 1), while the discard pile starts at index num-left.

The deck data structure is defined below. Notice that it is preceded by a block of comments that together specify the name of the data structure and, for each field, the name of that field along with a brief explanation of what that field is for.

;;; A DECK data structure
;;; --------------------------------------------------------
;;; CARDS: A vector of cards
;;; NUM-LEFT: Number of cards still left in the deck
;;; (i.e., still available for dealing)

(define-struct deck (cards num-left))

For testing purposes, we define a print-deck function that prints out the entire contents of a deck data structure. Later on (e.g., when using the deck data structure as part of a program that implements a card game) we might define a different function that does not show the players the order of the cards in the deck!

;;; PRINT-DECK
;;; --------------------------------------------------------
;;; INPUT: DECKY, a DECK data structure
;;; OUTPUT: None
;;; SIDE EFFECT: Displays contents of DECKY

(define print-deck
  (lambda (decky)
    (printf "Deck of cards: "
        (deck-cards decky) "A", num-left: "A", num-left decky)))

Next, we define a destructive function, called deal!, that deals one randomly chosen card from the deck. To preserve the partition between undealt cards and the discard pile, the deck data structure is modified as follows. First, the randomly selected card (i.e., the one to be dealt) is swapped with the card at the bottom of the not-yet-dealt partition (i.e., the card whose index is (num-left – 1)). Next, the value of the num-left field is decremented. In this way, the newly dealt card is effectively moved to the “top” of the “discard pile”.

;;; DEAL!
;;; --------------------------------------------------------
;;; INPUT: DECKY, a DECK data structure
;;; OUTPUT: One of the cards from DECKY, selected
;;; at random
;;; SIDE EFFECT: Modifies DECKY so that the newly dealt
;;; card joins the "discard" pile (i.e., becomes dealt)
(define deal!
  (lambda (decky)
    (let* (;; The number of cards left in the deck
           (num-lefty (deck-num-left decky))
           ;; A random index into the cards still left in the deck
           (rnd-indy (random num-lefty))
           ;; The CARDS vector from DECKY
           (veck-o-cards (deck-cards decky))
           ;; The card we shall output
           (dealt-card (vector-ref veck-o-cards rnd-indy))
           ;; The index of card to be swapped with DEALT-CARD
           (indy-to-move (- num-lefty 1))
           ;; The card to be moved
           (card-to-move (vector-ref veck-o-cards indy-to-move)))
      ;; Swap DEALT-CARD and CARD-TO-MOVE
      ;; (i.e., move newly dealt card into discard pile)
      (vector-set! veck-o-cards rnd-indy card-to-move)
      (vector-set! veck-o-cards indy-to-move dealt-card)
      ;; Decrement the NUM-LEFT field
      (set-deck-num-left! decky indy-to-move)
      ;; Output:
      dealt-card))

Because the deal! function randomly selects cards from those that have not yet been dealt, there is no need to simulate any “shuffling”; instead, shuffling the deck is simply a matter of resetting the num-left field so that all cards become available for dealing.

;;; SHUFFLE!
;;; ------------------------------------------------------
;;; INPUT: DECKY, a DECK data structure
;;; OUTPUT: None
;;; SIDE EFFECT: Makes all cards available for dealing

(define shuffle!
  (lambda (decky)
    ;; Reset the NUM-LEFT field to its initial value
    (set-deck-num-left! decky (vector-length (deck-cards decky)))
    ;; Output the modified DECK
    decky))

The following interactions demonstrate the use of the deck data structure, using a test deck that contains only ten cards.

> (define decky
   (make-deck (vector 'a 'b 'c 'd 'e 'f 'g 'h 'i 'j) 10))
> (print-deck decky)
Deck of cards: #(a b c d e f g h i j), num-left: 10
> (deal! decky)
d
> (print-deck decky)
Deck of cards: #(a b c j e f g h i d), num-left: 9
> (deal! decky)
  e
> (print-deck decky)
Deck of cards: #(a b c j i f g h e d), num-left: 8
> (deal! decky)
  c
> (print-deck decky)
Deck of cards: #(a b h j i f g c e d), num-left: 7
> (deal! decky)
  g
> (print-deck decky)
Deck of cards: #(a b h j i f g c e d), num-left: 6
> (shuffle! decky)
#<deck>
> (print-deck decky)
Deck of cards: #(a b h j i f g c e d), num-left: 10
> (deal! decky)
  a
> (print-deck decky)
Deck of cards: #(d b h j i f g c e a), num-left: 9

### 19.2 Summary

This chapter introduced field-based data structures (i.e., data structures whose slots (a.k.a. fields) are accessed by name). Although field-based data structures can be implemented using vectors, doing so requires a programmer to manually define a variety of constructor, accessor and mutator functions. To avoid such mundane tasks, Scheme provides the `define-struct` special form. The evaluation of a single `define-struct` special form automatically creates the needed constructor, accessor and mutator functions. The chapter illustrated the use of field-based data structures, first to represent dates, and second to represent a deck of cards.

**Special Forms Introduced in this Chapter**

- `define-struct`  For specifying new data structures
Chapter 20

The Model-View-Controller Paradigm

The Model-View-Controller (MVC) paradigm is a paradigm for software development that seeks to decompose a complex programming problem into relatively independent components. This paradigm can be used to simplify the task of creating Apps for mobile devices, including games. To simplify our presentation, we shall focus on games, too. In the context of developing a game:

- the Model consists of the data structures that contain all of the information needed to keep track of the game;
- the Controller consists of the functions that are used to play the game (e.g., to select moves or take action); and
- the View consists of functions for displaying the current state of the game to the player(s).

The relative independence comes from the following:

- the Model typically contains no functions, or just some basic wrapper functions that facilitate the use of the data structures it defines;
- the Controller functions can access or modify the contents of the data structures defined in the Model; and
- the View functions only access the contents of the data structures in the Model; they do not modify them.

An important responsibility of the model is to keep track of the current state of the game (e.g., whose turn it is, how many rolls of the dice the current player has, which slots of the scoresheet are already filled in, and so on). Although the controller functions can access and modify the contents of the data structures in the model, they do not get involved with displaying any information to the player(s). Similarly, the view functions only show the player whatever information is relevant for the current state of the game; they do not get involved in taking action (e.g., rolling dice or choosing cards).

20.1 Implementing a Deck of Cards using the MVC Paradigm

The implementation of a deck of cards seen in the previous chapter was done according to the Model-View-Controller paradigm, as follows:

- **Model**: The deck data structure
- **View**: The print-deck function
- **Controller**: The deal! and shuffle! functions

Notice that:

- the deck data structure contains all of the information needed to represent the deck of cards;
20.2 Implementing a Simple Game using the MVC Paradigm

This section presents a lengthy example that implements a simple dice-throwing game, called dice-dice, as an illustration of the use of the Model-View-Controller paradigm.

Example 20.2.1: Implementing the dice-dice game using the MVC paradigm

The rules of the dice-dice game are as follows. It is a one-player game. The player gets five turns. On each turn, the player seeks to accumulate points by repeatedly throwing a pair of dice. The player can throw the dice as many times as he or she wants; however, if a toss ever comes up doubles (e.g., a pair of threes), the turn is immediately ended, with the player receiving a score of zero for that turn. The overall score is the sum of the points accumulated across all five turns. The goal is to achieve a high overall score.

The flow of the game can be represented in many ways. We will use the diagram in Fig. 20.1 as a guide. In the figure, the boxes with rounded corners represent the different states that the game could be in. The three states are:

- **ready-to-play**: It is time for the player to either roll the dice or score the current total.
- **waiting**: Waiting for the player to begin the next turn.
- **game-over**: The game is over!

The game starts in the ready-to-play state, which is indicated by the > symbol pointing at that rounded box.

The arrows in the figure represent transitions from one state to another. A transition is typically triggered by some action of the player, but the particular transition may also depend on the outcome of the action (e.g., whether the player happened to roll doubles). Actions are implemented as Scheme functions (e.g.,
roll! to roll the dice, save! to save the total for the current turn, continue! to start the next turn, and reset! to start a new game. For example, if the game is in the ready-to-play state, then the player can choose either to roll the dice or save the current score. If the player chooses to roll the dice and anything other than doubles turn up, then the total of that pair of dice is accumulated for the current turn, and the state remains ready-to-play. However, if doubles turn up, then the current turn is ended with a score of zero, and the state changes to waiting. When the state is ready-to-play, if the player decides to save the points accumulated so far for the current turn, then the current turn will end, and a new turn will begin, with the state remaining ready-to-play. From the waiting state, the player’s only option is to do the continue! action. If there are no more turns left, then the state changes to game-over; otherwise, it changes to ready-to-play. From the game-over state, the only option is to start a new game.

Now that the flow of the game has been described, it is time to talk about the model, the view, and the controller.

- The model consists primarily of a dice-dice data structure that contains the information needed to keep track of the game; however, it is also convenient to define some global constants and to provide a wrapper function for make-dice-dice that facilitates creating a new game.

- The view consists of a non-destructive function, print-dd, that can be used to display the current contents of the dice-dice data structure in a way that is useful for the player.

- The controller consists of destructive functions that implement the different actions available to the player (e.g., roll!, save!, continue! and reset!).

We begin with the model. Notice that the definition of the dice-dice data structure is introduced by a block of comments explaining what each of the fields in the data structure is for.

;;; DICE-DICE struct
;;; ---------------------------------------------
;;; STATE: a symbol: READY-TO-PLAY, WAITING, or GAME-OVER
;;; CURR-TURN: an integer from 0 to 4 (the "current turn")
;;; LAST-TOSS: a pair (T1 T2) showing last toss of the dice;
;;; or #f if at the start of a new turn
;;; CURR-TOTAL: the points accumulated so far for the current turn
;;; SCORES: a vector of saved totals (5 entries, one per turn)
;;; GRAND TOTAL: the sum of all totals saved so far

(define-struct dice-dice (state curr-turn last-toss curr-total scores grand-total))

;;; Global constants for the different states of the game

(define *ready-to-play* 'ready-to-play)
(define *waiting* 'waiting)
(define *game-over* 'game-over)

;;; A global constant for an empty slot

(define *mt* '_)

;;; NEW-GAME -- wrapper for MAKE-DICE-DICE
;;; ---------------------------------------------
;;; INPUT: none
;; OUTPUT: A new DICE-DICE struct whose fields have been
;;     initialized for a new game

(define new-game
  (lambda ()
    (make-dice-dice
     *ready-to-play* ;; state
     0 ;; curr-turn
     #f ;; last-toss
     0 ;; curr total
     (make-vector 5 *mt*) ;; scores
     0 ;; grand total
  ))

The view function, print-dd, is next. Notice that the information displayed by print-dd depends
on which state the game is in. (See the cond cases in print-dd.) Not only does print-dd display
information about the current state of the game, it also displays the actions that are now available to the
player.

;; PRINT-DD
;; ------------------------------------------------------------------
;; INPUT: DD, a dice-dice struct
;; OUTPUT: None
;; SIDE EFFECT: Displays the information about the current
;;     state of the game of dice-dice, as represented by DD.

(define print-dd
  (lambda (dd)
    ;; Some local variables, just for convenience
    (let ((st (dice-dice-state dd))
     (last-toss (dice-dice-last-toss dd))
     (turn (dice-dice-curr-turn dd))
     (scores (dice-dice-scores dd))
     (grand-total (dice-dice-grand-total dd))
     (curr-total (dice-dice-curr-total dd)))
      (cond
       ;; -----------------------------
       ;; Case 1: Starting new turn
       ;; -----------------------------
       ;; (and (eq? st *ready-to-play*)
       ;;     ;; Haven’t tossed any dice yet
       ;;     (not last-toss))
       (printf "Start turn \A. Scores: \A. Grand total: \A\n"
       turn scores grand-total)
       (printf "==> ROLL! or SAVE!\n")
       ;; -----------------------------
       ;; Case 2: Turn already in progress
       ;; -----------------------------
       ;; (eq? st *ready-to-play*)
       (printf "Turn \A. Last Toss: \A, Curr Total: \A\n"
       turn last-toss curr-total)
       (printf "==> ROLL! or SAVE!\n")
       ;; -----------------------------
       ;; Case 3: Whoops! Zeroed out!
       )
  )
Finally, the controller functions are listed below.

### RESET!

---

**INPUT:** DD, a DICE-DICE struct  
**OUTPUT:** DD, destructively modified  
**SIDE EFFECT:** Resets contents of DD to a fresh game

```
(define reset!  
  (lambda (dd)  
    ;; Reset contents of DD in preparation for a new game  
    (set-dice-dice-state! dd *ready-to-play*)  
    (set-dice-dice-curr-turn! dd 0)  
    (set-dice-dice-last-toss! dd #f)  
    (set-dice-dice-curr-total! dd 0)  
    (set-dice-dice-scores! dd (make-vector 5 *mt*))  
    (set-dice-dice-grand-total! dd 0)  
    ;; Return the modified struct as output  
    dd))
```

### ROLL!

---

**INPUT:** DD, a DICE-DICE struct  
**OUTPUT:** none  
**SIDE EFFECT:** If appropriate, rolls the dice and updates game

```
(define roll!  
  (lambda (dd)  
    (let ((st (dice-dice-state dd)))  
      (cond  
        ;; Good Case: It’s time to roll  
        ;; ------------------------------  
        ((eq? st *ready-to-play*)  
          ;; Roll the dice  
          (let* ((toss-one (+ 1 (random 6)))  
              (toss-two (+ 1 (random 6))))  
            ;; Update LAST-TOSS  
            (set-dice-dice-last-toss! dd (list toss-one toss-two))  
            (cond  
              ;; ------------------------------  
```
;; Case 1: Doubles!
;; -------------------------------
((eq? toss-one toss-two)
 ;; CURR-TOTAL goes to zero
 (set-dice-dice-curr-total! dd 0)
 ;; Set state to WAITING so player can see results
 (set-dice-dice-state! dd *waiting*)
;; -------------------------------
;; Case 2: Not doubles!
;; -------------------------------
(else
 ;; Update CURR-TOTAL
 (set-dice-dice-curr-total! dd (+ (dice-dice-curr-total dd)
 toss-one
toss-two))
 ;; No change to state
))));

;; ------------------------------------------------
;; Bad Cases
;; ------------------------------------------------
(else
 ;; Print out an error message
 (printf "Sorry! Can’t roll now!\n")
))

;; SAVE!
;; -------------------------------------
;; INPUT: DD, a DICE-DICE struct
;; OUTPUT: None
;; SIDE EFFECT: If appropriate, saves curr-total to score vector
;; and prepares for the next turn (or the end of the game)

(define save!
 (lambda (dd)
 ;; For convenience, define some local variables
 (let ((st (dice-dice-state dd))
 (vecky (dice-dice-scores dd))
 (indy (dice-dice-curr-turn dd)))
 (cond
 ;; Good case: Saving is allowed
 ;; -------------------------------------
 ((eq? st *ready-to-play*)
 ;; Store the score in the vector
 (vector-set! vecky indy (dice-dice-curr-total dd))
 ;; Update the grand total
 (set-dice-dice-grand-total! dd
 (+ (dice-dice-curr-total dd)
 (dice-dice-grand-total dd)))
 (cond
 ;; Case 1: That was the last turn
;;; ------------------------------------
;; The game is OVER!
;; ------------------------------------
(set-dice-dice-state! dd *game-over*)

;;; Case 2: That was not the last turn
;;; ------------------------------------
(else
 ;; Prepare for next turn
 (set-dice-dice-curr-turn! dd (+ indy 1))
 (set-dice-dice-last-toss! dd #f)
 (set-dice-dice-curr-total! dd 0)))

;;; ------------------------------------------------
;;; Bad Cases
;;; ------------------------------------------------
(else
 ;; Print out an error message
 (printf "Sorry! Can’t save now! "))))

;;; CONTINUE!
;;; -------------------------------------------------------
;;; INPUT: DD, a DICE-DICE struct
;;; OUTPUT: None
;;; SIDE EFFECT: When appropriate, prepares for the
;;; next turn (or game over)
(define continue!
 (lambda (dd)
 ;; For convenience, create some local variables
 (let ((st (dice-dice-state dd))
      (indy (dice-dice-curr-turn dd))
      (vecky (dice-dice-scores dd)))
 (cond
 ;; Good Case: Waiting to go to next turn (or game over)
 ;; -------------------------------------------------------
 ((eq? st *waiting*)
 (cond
 ;; Case 1: That was the last turn
 ;; -------------------------------------------------------
 ((= indy 4)
 (set-dice-dice-state! dd *game-over*)
 ;; ---------------------------------------
 (else
 ;; Prepare for the next turn
 (set-dice-dice-state! dd *ready-to-play*)
 (set-dice-dice-curr-turn! dd (+ indy 1))
 (set-dice-dice-last-toss! dd #f)
 (set-dice-dice-curr-total! dd 0)
Now that the model, view and controller modules have been defined, we can provide a few wrapper functions that make it easier to play the game. Each wrapper function simply calls the corresponding controller function, and then calls the `print-dd` view function to display the results.

;; No need to update grand total :(

;; ROLL-WR!, SAVE-WR!, CONTINUE-WR!, RESET-WR!
;; ---------------------------------------------
;; INPUT: DD, a DICE-DICE struct
;; OUTPUT: None
;; SIDE EFFECT: Carries out the corresponding controller
;; function, which destructively modifies DD, and then
;; calls PRINT-DD to see the results.

(define roll-wr!
  (lambda (dd)
    (roll! dd)
    (print-dd dd)))

(define save-wr!
  (lambda (dd)
    (save! dd)
    (print-dd dd)))

(define continue-wr!
  (lambda (dd)
    (continue! dd)
    (print-dd dd)))

(define reset-wr!
  (lambda (dd)
    (reset! dd)
    (print-dd dd)))

The following interactions demonstrate the full implementation of dice-dice.

> (define g (new-game))
> (print-dd g)
Start turn 0. Scores: #(_ _ _ _ _). Grand total: 0
=> ROLL! or SAVE!
> (roll-wr! g)
Turn 0. Last Toss: (5 6), Curr Total: 11
=> ROLL! or SAVE!
> (roll-wr! g)
Turn 0. Last Toss: (4 6), Curr Total: 21
=> ROLL! or SAVE!
> (save-wr! g)
Start turn 1. Scores: #(21 _ _ _ _). Grand total: 21
=> ROLL! or SAVE!
> (roll-wr! g)
Turn 1. Last Toss: (4 2), Curr Total: 6
20.3 Summary

This chapter introduced the Model-View-Controller paradigm for software development. Its aim is to simplify the development process by dividing the software into three relatively independent modules. For example, in the context of game development, the Model specifies the data structures needed to represent the state of the game; the View defines functions for displaying the state of the game to the player/user; and the Controller defines functions/actions that the player can use to change the state of the game. One advantage of the MVC paradigm is that a game defined with a text-based interface can typically be changed into a game that incorporates a graphical
user interface by introducing a new View module, without making changes to the Model or Controller modules. However, some additional work is required to ensure that onscreen actions (e.g., button clicks) can be used to call the desired controller functions.
Appendix A

Guide to Your CS Account

All of the programming work you do in this course will be done using your CS computer account which you can access from any of the classroom or lab computers in the CS Department. The name of your account is typically the same as the first part of your Vassar email, although there can be exceptions. For example, my CS account name is hunsberg, which harkens back to the days when account names were limited to eight characters! Every student in this course has their own CS account. In addition, the CMPU-101 course itself also has an account, called cs101. All of the computer files and directories (a.k.a. folders) for all of the CS account holders are organized into a single tree-like structure called a file system. All of the computer programs you write for this course will be computer files that are stored within your portion of the CS file system. Thus, it will be important to understand how to navigate through the file system, create new files and directories, start up the DrScheme software, and print out and electronically submit your program files. All of this will be enabled by simply opening up a Terminal window and entering the appropriate commands at the prompt. (Since the computers are running the Linux operating system, we may refer to these commands as Linux commands.) The rest of this chapter describes the file system, how to explore the file system using the commands issued from a Terminal window, and how to format, submit and print out your assignment files.

A.1 The File System

The file system is organized into a tree-like hierarchy of computer files and directories. A directory (or folder) is a collection of computer files that typically have something in common. For example, a directory called lab1 might contain all of the program files associated with your first programming lab. A directory may also contain subsidiary directories (a.k.a. sub-directories or sub-folders), thereby enabling directories to be organized into a tree-like hierarchy.

At the root of the file system is a special directory, called the root directory, that is the topmost ancestor of every other file and directory in the entire file system. For convenience, the root directory is frequently denoted by a single forward slash: /. As indicated in Fig. A.1, the root directory typically contains lots of directories with strange names (e.g., bin, dev, etc and mnt). These directories are used by the Linux operating system to handle things that will not concern us. However, one of the directories in the root directory is relevant for us: the home directory. As its name suggests, the home directory contains the “home” directories of every CS account. For example, the home directory contains two directories, called hunsberg and cs101, which are the respective home directories of my CS account and that of the CMPU-101 course.

Full pathnames. Each file or directory can be referred to by an absolute address, called its full pathname. The full pathname for a file or directory, X, represents the unique path from the root directory to X in the file system’s hierarchy. For example, the full pathname for my home directory is /home/hunsberg, since the root directory contains the home directory, and the home directory contains the hunsberg directory. Similarly, the full pathname for the cs101 home directory is /home/cs101.
The Desktop directory. As illustrated in Fig. A.1, the home directory for each CS account contains a subdirectory called Desktop. Although my Desktop directory has the same name as your Desktop directory, they are in fact distinct directories. The operating system has no trouble distinguishing them because their full pathnames are unique. For example, the full pathname for my Desktop directory is /home/hunsberg/Desktop, while the full pathname for the Desktop directory belonging to the cs101 account is /home/cs101/Desktop.

* Most of the files and directories located within your Desktop directory will have a corresponding icon that is automatically displayed on your computer screen’s Desktop.

All of the files you create for your work in this course should be organized within your Desktop directory, as illustrated in Fig. A.2. Notice that this organization allows room for growth should you decide to take subsequent Computer Science courses (e.g., CMPU-102, CMPU-145, and so on).

A.2 Using Terminal to Explore and Augment the File System

The Linux operating system provides numerous commands that enable you to navigate through the file system. These commands are processed by a program called Terminal. When you start the Terminal program, it opens up a Terminal window. When a command is typed into the Terminal window, and the Enter key is tapped, the Terminal program will attempt to execute the command.
When using Linux commands in a Terminal window to navigate the file system, the Terminal program keeps track of your current location within the directory tree. That current location is called your working directory. The name of the working directory is often automatically displayed as part of the prompt in the Terminal window. Below are listed some of the most useful Linux commands for navigating the file system and creating new directories. The use of these commands is covered by Lab 1.

- **pwd** – Print the Working Directory (i.e., display where you are in the tree of directories). When you first open the Terminal window, the working directory is typically set to be the home directory of your account. Thus, if I open up a terminal window in my account and immediately enter the `pwd` command, it will cause the following to be displayed: `/home/hunsberg`.

- **ls** – List the contents (i.e., files and sub-directories) of the working directory.

- **cd** – Change Directory. If used by itself, this command returns you to your account’s home directory (i.e., it sets the working directory to be your home directory). If you give it an input (e.g., a full pathname), then the `cd` command will set the working directory to be whatever directory you specify.

- **mkdir** – Make (i.e., create) a new Directory. This command takes one input: either a full pathname for the new directory or just a simple name for it. For example, the following command would create a new directory named `tmp` within my `Desktop` directory:
  ```
  mkdir /home/hunsberg/Desktop/tmp
  ```
  Alternatively, if I was already in the `Desktop` directory (i.e., if the working directory was set to be my `Desktop` directory), then the following simpler command would have the same effect:
  ```
  mkdir tmp
  ```

As already mentioned, Lab 1 will demonstrate the use of these and other Linux commands in more detail.

### A.3 Submitting Programming Assignments

This section describes the process of submitting programming assignments. Typically, this will involve two steps: (1) printing out your definitions and interactions files; and (2) electronically submitting the directory that contains these two files.

- When doing any lab or assignment, be sure to save your definitions file periodically so that you don’t lose it should something go wrong! Give it a name such as `yourName-asmt3-defns.txt`.

### Before Printing or Electronically Submitting your Files

Before printing or electronically submitting your files, you should carefully review the following guidelines.

- Your definitions and interactions must be saved as plain-text files! (If you are unsure about this, review the relevant portions of Lab 1.)

- Your definitions window should be nicely formatted. See the code-from-class postings on the course website for examples of nicely formatted code. Or look at the posted solutions to any lab or assignment. In particular:

  - Make sure that your definitions file begins with a block of comments like this:
    ```
    ;; ==============================================================
    ;; CMPU-101, Fall 2019
    ;; Asmt. or Lab Info
    ;; Your Name
    ;; ==============================================================
    ```
    where Asmt. or Lab Info is replaced by the relevant assignment or lab number (e.g., Asmt. 3 or Lab 5), and Your Name is replaced by your name!
• Make sure that the first Scheme expression in your definitions file is: (load "asmt-helper.txt").

• Make sure that the second Scheme expression in your definitions file involves an application of the header function to appropriate inputs, for example, something having the form:

  (header "Your Name" "Asmt. 3").

When you hit the Run button, you should see a nicely displayed header at the top of your interactions.

• Make sure that each problem is introduced by an invocation of the problem function, surrounded by commented lines of dashes, as illustrated below:

  ;; -----------------------------------
  ;; (problem "Description")
  ;; -----------------------------------

• Make sure that each function you define is preceded by a “contract” (i.e., a block of comments that specifies the name of the function, the names and descriptions of the input parameters, a brief description of the output, and, if your function has side effects, a brief description of those too. Make sure that your contract clearly distinguishes the output value of the function from any side effects it might have. The contract should have the following form:

  ;; FUNCTION-NAME
  ;; ------------------------------------------------------
  ;; INPUTS: names and descriptions of inputs
  ;; OUTPUT: description of output value (or "none")
  ;; SIDE EFFECTS: description of side effects (if any)

• In your function definition, the names for your function and its inputs should match the names that appear in the contract!

• Make sure that your code is properly indented. This is easiest to do by selecting the Scheme menu item and choosing Reindent All.

• Make sure that your code does not include long lines of text that wrap around to the next line! Instead, break up long lines by using the Enter key, and taking advantage of DrScheme’s automatic indentation!

• When needed, your code should be augmented with concise comments explaining (briefly) what your code does. (See code-from-class postings for examples.) For example, if your function uses the cond special form (cf. Chapter 11), then each case of your cond should be preceded by a brief comment describing that case.

• Make sure that you have thoroughly tested your functions to demonstrate that they work as desired. This is typically done by providing a bunch of tester expressions that test a variety of cases beyond those that are given in the lab or assignment instructions.

• Make sure that there are blank lines between the problem expression and the contract, between the contract and the function definition, between the function definition and the tester expressions, and between the tester expressions and the following problem (if any). Again, see code-from-class postings for examples.

• When you are confident that your definitions file adheres to the above guidelines, then do the following:

  • Save your definitions window one last time.
  • Hit the Run button one last time.
  • Save your interactions as plain text! (Use the Save Other and Save Interactions as Text... menu items in DrScheme.)
  • Double-check that your interactions begin with a nice block of text generated by the header function. The top of your interactions should have the following form:
CMPU-101, Fall 2019
Asmt. or Lab Info
Your Name

where “Asmt. or Lab Info” is replaced by the relevant information, and “Your Name” is replaced by your name. If this information does not appear at the top of your interactions, check that your definitions file includes a call to the header function as described earlier.

* The contents of your interactions should be laid out nicely using the problem and tester functions, as described earlier. If not, go back to your Definitions Window and make the needed changes.

* Double-check that each tester expression is properly displaying both the input(s) and output—and that each is generating the right answer! If you spot any errors, go back to your function definition and make needed changes. If you make any changes to your Definitions Window, you will need to save your definitions, hit the Run button again, and then save your interactions (as plain text) again.

Congratulations! You should now be ready to print out and electronically submit your work!

### A.3.1 Printing Text Files

* **Warning!** The information in this section applies only to printing out files containing plain text! The commands given below should not be used to print out pdf, doc, jpg, or any other non-plain-text files.

For most programming assignments, you will need to print out only two files: your definitions file and your interactions file. (It is not necessary to print out anything for labs.) Both of these files should be plain-text files. If either appears with a bunch of gibberish then you should review the instructions for saving your definitions or interactions as plain-text files. You do not need to turn in printouts of the asmt-helper.txt file, since you are not expected to make changes to that file. In addition, you should not print out any file whose name ends with a `~` character (e.g., myfile.txt~); those files are automatically generated backup files that can be safely ignored.

Example A.3.1

Suppose that `hunsberg/Desktop/my101/labs/lab2/hun-lab2.txt` is the full pathname for a plain-text file called hun-lab2.txt. The following command can be used within a Terminal window to print out that file to the printer called Asprey, which is located in Room SP 307:

```bash
enscript -P Asprey hunsberg/Desktop/my101/labs/lab2/hun-lab2.txt
```

Since typing out full pathnames can be quite tedious, there’s an even easier way. First, cd into the desired directory—in this case, my lab2 directory; and then issue the following, simpler command:

```bash
enscript -P Asprey hun-lab2.txt
```

(See Section A.2 if you need a refresher on cd-ing into a desired directory.)

In general, if you are currently in a directory $D$ that contains a plain-text file named myfile.txt, then you can print out that file using the following command:

```bash
enscript -P Asprey myfile.txt
```

If you have any trouble printing, ask a coach for help.

* After printing your definitions and interactions, make sure to **staple** them—with the definitions on top!

* The Asprey printer should **only** be used to print out Computer Science labs or assignments.
A.3.2 Submitting your Files Electronically

Assignment files must be electronically submitted using the `submit101` command from a Terminal window. This command has the following syntax:

```
submit101  AsmtSubmissionName  YourAsmtDir
```

where `AsmtSubmissionName` is the name for this assignment for submission purposes (which is typically given to you as part of the assignment instructions) and `YourAsmtDir` is the name of your assignment directory. (That’s right: you must submit the entire directory; the `submit101` command cannot be used to submit individual files.)

<table>
<thead>
<tr>
<th>Example A.3.2</th>
</tr>
</thead>
</table>

Suppose that the `AsmtSubmissionName` is `h-asmt3` and your assignment directory is called `asmt3`. (We may also say that `h-asmt3` is the name of the dropbox into which you are going to submit your assignment.) Suppose further that your `asmt3` directory is contained within a directory called `asmts`. Then you would electronically submit your `asmt3` directory by first `cd`-ing into your `asmts` directory, and then executing the following command:

```
submit101  h-asmt3  asmt3
```

Note that it is very important that you be in the parent directory of the directory that you want to submit! (The `asmts` directory is called the parent of the `asmt3` directory because `asmts` contains `asmt3`.) If you are in the `asmt3` directory, then you should execute the following command to `cd` into the parent `asmts` directory:

```
cd ..
```

The two periods denote the parent directory of the working directory.

If you have any trouble using the `submit101` command, ask me or a coach during lab or office/coaching hours.
(define index-of-num-in-list-acc
 (lambda (num listy indy)
  (cond
   ;; Base Case 1: LISTY empty
   ((null? listy)
    ;; NUM not found
    #f)
   ;; Base Case 2: NUM found!
   ((= num (first listy))
    ;; return the current index
    indy)
   ;; Recursive Case 1: (FIRST LISTY) > NUM
   ((> (first listy) num)
    (printf "+")
    (index-of-num-in-list-acc num (rest listy) (+ indy 1)))
   ;; Recursive Case 2: (FIRST LISTY) < NUM
   (else
    (printf "-")
    (index-of-num-in-list-acc num (rest listy) (+ indy 1))))))

;; INDEX-OF-NUM-IN-LIST -- wrapper function
;; ------------------------------------------
;; INPUTS: NUM, a number
;; LISTY, a list of numbers
;; INDY, current index
;; OUTPUT: The index of the first occurrence of NUM
;; in LISTY; or #f if NUM does not occur in LISTY.
(define index-of-num-in-list
 (lambda (num listy)
  ;; Call tail-recursive helper with INDY = 0:
  (index-of-num-in-list-acc num listy 0)))

Below, the index-of-num-in-list-iter function is an iterative implementation that exhibits the same behavior. Notice the use of two extra variables, continue? and answer. The while loop keeps going as long as the value of continue? is #t. When the base case is reached, continue? is set to #f to stop the while loop; and, because the while expression evaluates to void, the answer variable is used to store the desired answer so that it can be recalled after the while loop is finished.

(define index-of-num-in-list-iter
 (lambda (num listy)
  (let ((indy 0)
    (answer #f)
    (continue? #t))
   (while continue?
    (cond
     ;; Base Case 1: LISTY empty
     ((null? listy)
      ;; Stop the WHILE loop
      (set! continue? #f))
     ;; Num not found
     (set! answer #f))
    ;; Base Case 2: NUM found!}
The following interactions demonstrate that the recursive and iterative functions have the same behavior.

```
> (index-of-num-in-list 3 '(5 4 9 0 0 2 3 8 7 6 9))
+++---6
> (index-of-num-in-list-iter 3 '(5 4 9 0 0 2 3 8 7 6 9))
+++---6
> (index-of-num-in-list 3 '(2 5 1 6 3 8 8))
-+-+-+4
> (index-of-num-in-list-iter 3 '(2 5 1 6 3 8 8))
-+-+-+4
```

### 17.4 The `dotimes` Special Form

As its name suggests, the `dotimes` special form enables something to be done a certain number of times. In particular, for a given value $n$, the `dotimes` special form will evaluate the expressions in its body $n$ times, once for each value in the set $\{0, 1, 2, \ldots, n - 1\}$. The `dotimes` special form will be particularly useful for iteratively walking through vectors, to be discussed in the next chapter.

**The syntax of the `dotimes` special form.** The `dotimes` special form has the following syntax:

```
(dotimes (var numExpr)
  expr1
  expr2
  ...
  exprk)
```

where:

- `var` is a symbol that will be the name of a counter variable in a local environment created by the `dotimes`;
- `numExpr` is any expression that evaluates to a non-negative integer, say, $n$, that will specify the number of iterations to be performed by the `dotimes`; and
• The expressions, $expr_1, expr_2, \ldots, expr_k$, are any $k$ expressions that together constitute the body of the dotimes.

The semantics of the dotimes special form. A dotimes special form is evaluated as follows. First, the expression $numExpr$ is evaluated, resulting in a non-negative integer $n$. Second, a local environment $E$ is created containing a variable $var$ whose value is initially set to zero. Next, the following steps are performed $n$ times:

• The expressions, $expr_1, expr_2, \ldots, expr_k$, are evaluated with respect to that new local environment.

• The value of $var$ in the local environment is incremented by one.

Thus, the expressions in the body of the dotimes are evaluated $n$ times, once for each value of $var$ in the range, \{0, 1, 2, \ldots, n − 1\}. Note that the expressions in the body may refer to the variable $var$. When they are evaluated, the current value of $var$ will be taken from the local environment.

Finally, the dotimes special form evaluates to void. Therefore, the usefulness of dotimes comes not from any output value, but from the side effects that occur by evaluating the expressions in its body with respect to the new local environment.

Example 17.4.1: Illustrating the dotimes special form

The following examples illustrate how the dotimes special form can be used.

> (dotimes (i 5)
  ;; The body:
  (printf "i: ^A~:" i))

i: 0
i: 1
i: 2
i: 3
i: 4

> (dotimes (i (+ 1 2))
  ;; The body:
  (printf "^A + ^A = ^A~:" i i (+ i i)))

0 + 0 = 0
1 + 1 = 2
2 + 2 = 4

In these examples, the body consists of a single expression; however, that need not be the case in general. Notice that in the first example, the numerical expression 5 ensures that the expression in the body will be evaluated five times, once for each value of i in the range \{0, 1, 2, 3, 4\}. In the second example, the body of the dotimes is evaluated three times, since (+ 1 2) evaluates to 3.

Example 17.4.2: (Optional) Implementing our own version of dotimes

The my-dotimes function, defined below, implements essentially the same behavior as the dotimes special form. However, like my-while, defined earlier, it uses a lambda function, called body-func, to encapsulate the expressions in the body of the dotimes.

;; MY-DOTIMES
;; -------------------------------
;; INPUTS: N, a non-negative integer
;; BODY-FUNC, a function that takes zero inputs
Here is a simple demonstration of its behavior:

```
> (my-dotimes 4 (lambda (i)
    (printf "--- i = \~A\~" i)))
--- i = 0
--- i = 1
--- i = 2
--- i = 3
```
ELT: C

> (dolist (elt (cons 1 (cons 2 (cons 3 ()))))
  (printf "elt: ~A" elt)
  (printf "<---~%")
elt: 1 <---
elt: 2 <---
elt: 3 <---

Note that unlike many recursive functions on lists, there is no way to break out of a `dolist` before processing all of the elements in the given list.

Example 17.5.2: (Optional) Implementing our own version of `dolist`

The following function provides the same functionality as the `dolist` special form, except that the expressions in the body are encapsulated within a `lambda` function that expects a single input, elt.

;;; MY-DOLIST
;;; ---------------------------------------------------------------
;;; INPUTS: LISTY, a list
;;; BODY-FUNC, a function that takes a single input, for example, any element of LISTY
;;; OUTPUT: void
;;; SIDE EFFECT: For each element ELT of LISTY, MY-DOLIST applies BODY-FUNC to ELT.

(define my-dolist
  (lambda (listy body-func)
    ;; As long as LISTY is non-empty...
    (while (not (null? listy))
      ;; Evaluate the "body" with ELT = (FIRST LISTY)
      (body-func (first listy))
      ;; The next iteration will deal with (REST LISTY)
      (set! listy (rest listy)))))

Here are some examples of its use.

> (my-dolist '(a b c)
  (lambda (elt)
    (printf "elt: ~A~%" elt)))
elt: a
elt: b
elt: c

> (my-dolist (cons 1 (cons 2 (cons 3 ())))
  (lambda (elt)
    (printf "elt: ~A" elt)
    (printf "<---~%")))
elt: 1 <---
elt: 2 <---
elt: 3 <---
17.6 Summary

This chapter introduced a kind of repetitive processing called iteration, that is an alternative to recursion. Iterative programming typically requires destructive programming. In particular, it frequently requires the programmer to create and modify various variables to control the iterative process. This can make tracking down errors in iterative programming more difficult. However, for many tasks, iterative solutions can be elegant and efficient.

The set! special form allows a programmer to change the value of an existing variable to any desired value. Frequently, set! is used to increment or decrement the value of a counter variable in looping constructs.

The while special form provides a basic form of iteration where the programmer must explicitly specify the condition that determines how many iterations are performed, and explicitly modify the values of variables upon which the condition typically depends. The let, while and set! special forms are frequently used together. For example, a let special form can be used to create some local variables (e.g., a counter variable and one or more accumulators), a while special form can then be placed within the body of the let to provide the desired repetitive behavior, and the set! special form can be used within the body of that while to control its iterative behavior. With such low-level control, the programmer can specify a wide variety of iterative computational behaviors that can be extremely useful; however, care must be taken to ensure that, for example, counter variables are correctly updated to avoid entering into an infinite loop.

Next, this chapter demonstrated how tail-recursive functions can be transformed into equivalent iterative functions. The equivalence of tail-recursion and iteration is what DrScheme relies on when it recycles a single function call box for all of the recursive function calls of a tail-recursive function.

Because certain iterative computations are so frequently needed, the dotimes and dolist special forms are provided. They shield the programmer from the low-level details of, for example, setting up a local environment for a counter variable, and managing the changing of its value throughout the iterative process. The dotimes special form enables a programmer to make some set of computations happen a specified number of times, once for each value of a counter variable, for example, $i \in \{0, 1, 2, \ldots, n - 1\}$. In each iteration, the expressions in the body of the dotimes may refer to the current value of $i$. Similarly, the dolist special form enables a programmer to make some set of computations happen, once for each element of a given list.

Special Forms Introduced in this Chapter

- **set!** Destructively modify the value of a variable
- **while** As long as some condition holds, evaluate expressions in body
- **dotimes** Evaluate expressions in the body $n$ times, once for each value in $\{0, 1, \ldots, n - 1\}$
- **dolist** Evaluate expressions in the body, once for each element of a given list
Appendix B

Lab 1: Your first CMPU-101 lab session!

The purpose of this lab is to demonstrate the basics of navigating your Computer Science account, creating files and directories, saving them, and so on.

- If you get stuck anywhere along the line, please ask for help!

You will access your CS account through computers that are running the Linux operating system. The following instructions introduce the basic Linux commands that you will use from within your CS account to download files, create files, organize your files, and so on, for all future labs and programming assignments.

Part One: Logging into your CS account

Sit down at one of the computers. Log into your account using the following information:

Username: The same as the first part of your Vassar email address.
Password: Look at the whiteboard!

Once the “Desktop” appears on-screen, right-click on any unused portion of the Desktop. A little menu will pop up. Select “Open in Terminal”. A Terminal window should appear on-screen. (If you aren’t able to open up a Terminal window, ask for help.)

The Terminal window acts a lot like DrScheme’s Interactions Window that you have seen in class. In the Interactions Window, you type a Scheme expression at the prompt, followed by hitting the Enter key. In response, the Scheme datum denoted by that Scheme expression is evaluated and, usually, some information is displayed. In the Terminal window, you type Linux commands at the prompt. When you hit the Enter key, the Terminal program tries to execute the command you entered. Of course, if you enter something wrong, it may complain vigorously.

One of the main jobs of commands entered into the Terminal window is to enable you to navigate the files and directories not only in your account, but also the entire file system for all of the CS accounts.

Before proceeding, be sure to read Appendix A through Section A.2.

Table B.1 lists a sequence of Linux commands, along with explanations for each. For each command shown, type the command into the Terminal window, and then hit the Enter key. You should enter the commands one at a time, in the order shown. If you get mixed up, just go back to the first command—or ask for help.

After entering the entire sequence of commands listed in Table B.1, you should end up in a newly created directory, called lab1, within your CS account. The full pathname for this lab1 directory should be displayed as `~/Desktop/my101/labs/lab1` or `/home/yourAcctName/Desktop/my101/labs/lab1`. (The character `~` is frequently used as a convenient abbreviation for your home directory.)

If you get stuck and want to return to your home directory, just type: cd ~. Alternatively, you could just type cd without any inputs because it assumes you want to go to your home directory by default.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description of what it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>pwd</em></td>
<td>Display the <em>working directory</em> (i.e., the directory you are in right now). You are probably in your account’s home directory, which may be displayed as: <code>/home/yourAcctName.</code></td>
</tr>
<tr>
<td><em>cd ~</em></td>
<td>Move (conceptually) to your “home” directory (i.e., set the working directory to be your home directory). (This is probably not necessary since you were probably already in your home directory.) Note: <em>cd</em> stands for “change directory”; and ~ is a convenient shorthand for your home directory.</td>
</tr>
<tr>
<td><em>ls</em></td>
<td>List the contents of your working directory. It should contain a sub-directory called Desktop.</td>
</tr>
<tr>
<td><em>cd Desktop</em></td>
<td>Change the working directory to be your Desktop directory.</td>
</tr>
<tr>
<td><em>ls</em></td>
<td>List the contents of the working directory—which should be the Desktop directory at this point. Note that most of the contents of the Desktop directory have a corresponding icon on the Desktop!</td>
</tr>
<tr>
<td><em>mkdir my101</em></td>
<td>Create a new directory called my101. Note that because my101 is <em>not</em> a full pathname, the new directory will created within the working directory—in this case, the Desktop directory. And since the new directory is located within the Desktop directory, an icon will automatically appear on the Desktop!</td>
</tr>
<tr>
<td><em>ls</em></td>
<td>This should show you that the Desktop directory now contains a sub-directory called my101.</td>
</tr>
<tr>
<td><em>cd my101</em></td>
<td>Travel into the my101 directory.</td>
</tr>
<tr>
<td><em>pwd</em></td>
<td>Show that you are now in the my101 sub-directory. It will probably be displayed before the prompt as ~/Desktop/my101 or /home/yourAcctName/Desktop/my101.</td>
</tr>
<tr>
<td><em>ls</em></td>
<td>Show the (non-existent) contents of your new my101 directory.</td>
</tr>
<tr>
<td><em>mkdir labs</em></td>
<td>Create a new directory called labs inside the my101 directory. (It is created within the my101 directory because that is the working directory—i.e., the directory where you are right now.)</td>
</tr>
<tr>
<td><em>ls</em></td>
<td>Show that you indeed have created labs.</td>
</tr>
<tr>
<td><em>cd labs</em></td>
<td>Move into the labs directory.</td>
</tr>
<tr>
<td><em>pwd</em></td>
<td>Show that you are indeed there.</td>
</tr>
<tr>
<td><em>ls</em></td>
<td>Show that the labs directory is currently empty.</td>
</tr>
<tr>
<td><em>mkdir lab1</em></td>
<td>Create a sub-directory called lab1.</td>
</tr>
<tr>
<td><em>ls</em></td>
<td>Show that lab1 is there.</td>
</tr>
<tr>
<td><em>cd lab1</em></td>
<td>Move into the lab1 directory.</td>
</tr>
<tr>
<td><em>pwd</em></td>
<td>Show where you are.</td>
</tr>
</tbody>
</table>

Table B.1: A sequence of Linux commands to create the directory structure for Lab 1
If you want to “back up” to the “parent” of your working directory, use the following command (with two periods): \texttt{cd ..}

Examples of using the \texttt{cd} command to navigate through a directory tree are shown in Fig. B.1. The figure presumes that the account name—and hence the name of the home directory—is \texttt{hunsberg}.

**Part Two: Firing Up DrScheme**

Use the \texttt{pwd} command to verify that you are currently in your \texttt{lab1} directory. Then, while still in your \texttt{lab1} directory, type the following command into the Terminal window to start up the DrScheme program:

```

drscheme&
```

If you forget to type the \texttt{&} character, then the Terminal window will freeze until the DrScheme program is closed/finished. If you include the \texttt{&} character, you can continue to use the Terminal window while the DrScheme program is running.

⇒ Since this is the first time that you have opened DrScheme, you will need to “choose” the “Full Swindle” language, as follows. First, in the DrScheme menu bar, click on the Language menu item, and then select Choose Language. Then, in the pop-up window, under Swindle, select Full Swindle.

**Using the Interactions Window.** The lower window pane in DrScheme is called the Interactions Window. Type a legal Scheme expression at the prompt—which is usually a \texttt{>} character—and then hit the \texttt{Enter} or \texttt{Return} key. See what DrScheme does. Then try a few more expressions. Afterward, click the Run button to clear the Interactions Window.

**Using the Definitions Window.** The upper window pane in DrScheme is called the Definitions Window. Type some legal Scheme expressions in there, say, one per line. Then click the Run button. DrScheme should clear the Interactions Window and then evaluate the (data denoted by the) expressions in the Definitions Window, just as though you had typed them in manually. Type some more legal expressions in the Definitions Window. Then click Run again. DrScheme again clears the Interactions Window and then evaluates the (data denoted by the) expressions in the Definitions Window.

Enter the following text at the top of your Definitions Window:
254 Introduction to Computer Science via Scheme  © 2019 Luke Hunsberger  Fall 2019

But replace YOUR NAME!! with your name! Click on the Run button. Notice that DrScheme ignores any line of text that begins with a semi-colon. Such lines of text are called comments. Comments are useful for people reading your programs. DrScheme doesn’t care what you write in your comments. Comments don’t even count as expressions in Scheme. They are totally ignored.

Next, hit the Enter (or Return) key to insert a blank line, and then enter the following expressions into your Definitions Window:

```scheme
(printf "================================\n")
(printf " CMPU-101, Fall 2019\n")
(printf " Lab 1\n")
(printf " YOUR NAME!!\n")
(printf "================================\n\n")
```

Again, replace YOUR NAME!! with your actual name. Click the Run button one more time to see the results. Hopefully you will like what you see in the Interactions Window. If not, make some adjustments to the expressions in your Definitions Window and click Run again. And feel free to ask for help.

Finally, enter another blank line, followed by a few Scheme expressions of your own, one expression per line. Hit the Run button to see the results of evaluating those expressions.

Saving the Contents of the Definitions Window to a File.  It is important to save your work frequently. Under the File menu of DrScheme, select the Save Definitions menu item. A pop-up window will ask you to enter a name for the new file into which the contents of your Definitions Window will be saved. Enter yourName-lab1-defns.txt, where yourName is replaced by ... your name! By default, DrScheme will save the file within whatever directory you were in when you started up DrScheme. That’s why I wanted you to move into the lab1 directory before firing up DrScheme. If you didn’t, you will have to explicitly choose the lab1 directory from the pop-up window. Ask for help if you don’t see how to do this.

Once your Definitions have been saved to a file, you can save any subsequent changes by clicking on the little blue icon that appears near the top of DrScheme’s window. (It only appears when you have made changes.) Make a few changes, then save your file and click the Run button.

When you are sure that you have successfully saved your file, close the DrScheme program either by clicking on the X in the upper-right corner of the main window, or by selecting the File/Quit menu item. If you didn’t save your Definitions, DrScheme should ask you to do so before actually closing down.

Opening a file in DrScheme.  From the Terminal window, fire up DrScheme one more time. Then, under the File menu of DrScheme, select the Open item. When prompted, select the file yourName-lab1-defns.txt that you saved earlier. The contents of the file should appear in the Definitions Window. Normally, the Definitions Window occupies the upper half of DrScheme’s main window, with the Interactions Window in the bottom half; however, until you click the Run button, the Definitions Window may be all you see.

Saving the Contents of the Interactions Window to a File.  Click on the Run button one more time. When you see nicely displayed results in the Interactions Window, you can save it to a file as follows.

- It is very important to save the Interactions as plain text! Otherwise, there could be all sorts of junk written to your file.

Click on the File menu, select Save Other, and then select Save Interactions as Text.... It will prompt you for a name for your file. You should name the file as follows: yourName-lab1-inters.txt, where yourName is replaced by ... your name.
Printing out your files. Printing is done from the Terminal window, not from DrScheme. So, you may wish to close DrScheme, or move its window out of the way. In the terminal window, type `pwd` to make sure that you are in the `lab1` directory. Then type `ls` to see the list of files that are currently in there. You can ignore the files whose names end with a “tilde” character (i.e., `˜`); they are backup files that are created automatically.

To print a file named `myfile.txt`, type the following command in the Terminal window:

```
enscript -P Asprey myfile.txt
```

Print out one of the files you saved and then go get the printout in SP307. (Asprey is the name of the printer in room SP307.) You can keep this print-out. It was just for practice.

⇒ Only use `enscript` to print out PLAIN TEXT files!!

⇒ Do not use `enscript` to print out any other kind of file!

Submitting your files electronically. Submitting a lab or assignment electronically is also done from the Terminal window. However, to submit your `lab1` files electronically, you should `cd` into the *parent* of the `lab1` directory (i.e., the directory that *contains* the `lab1` directory). So, if you are currently in the `lab1` directory, type: `cd ..` (with two periods!). Then type the following command:

```
submit101 h-lab1 lab1
```

`h-lab1` is the name of the dropbox for this lab (“h” for Hunsberger); `lab1` is the directory of files you are submitting. If you see a bunch of hopeful messages fly by, it probably worked. But you should ask me or one of the coaches to come over to check things out.

That’s it!