Introduction to Computer Science via Scheme

Luke Hunsberger

© 2019 Luke Hunsberger
All rights reserved

Spring 2019

Contents

List of Figures

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

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

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

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

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

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

- `(define x 5)`
- `(+ 2 3)`
- `(printf "Hi there...")`

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

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

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

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

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

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

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

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

Functions

Scheme is an example of a *functional* programming language. The main thing that you, as a Scheme programmer, will do is design *functions* for solving problems. For our purposes, a function is something that takes *zero or more* inputs, and generates a *single* output, as illustrated on the lefthand side of Fig. 1.1. For example, you might define a Scheme function whose input is a scoresheet for some game, and whose output is the sum of the scores on that scoresheet.

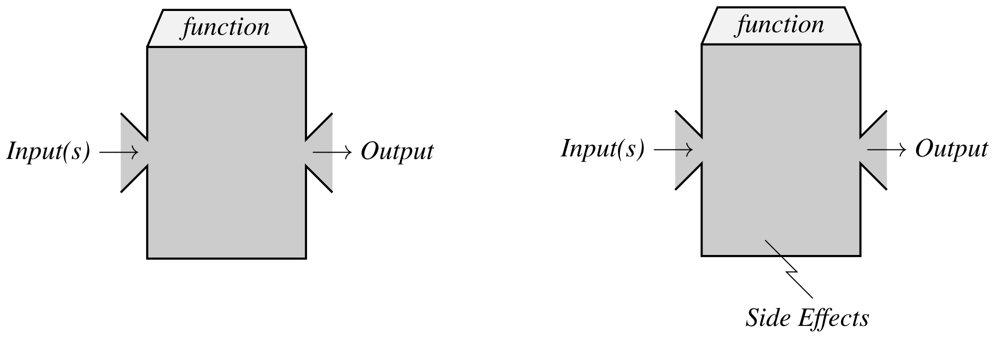


Figure 1.1: A function with no side effects (left) vs. a function with side effects (right)

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*,

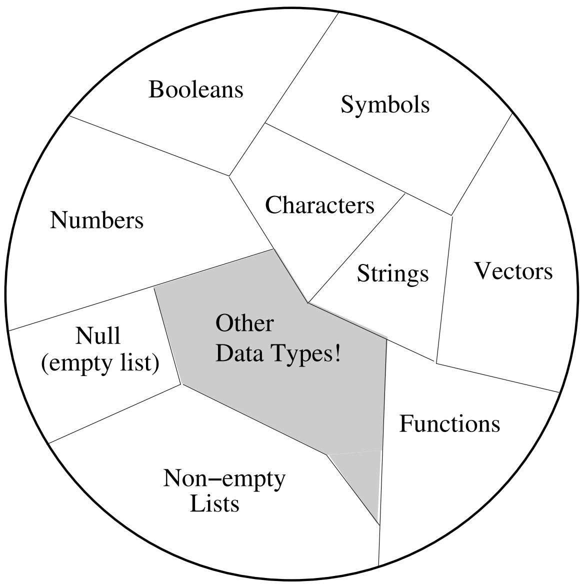


Figure 2.1: The universe of Scheme data, partitioned according to data type

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:

$$\text{Character Sequence} \longrightarrow \text{Datum}$$

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

`3`	$\longrightarrow$	the number *three*
`-44`	$\longrightarrow$	the number *negative forty-four*
`85/6`	$\longrightarrow$	the number *eighty-five sixths*

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

`0`	$\longrightarrow$	the number *zero*
`000`	$\longrightarrow$	the number *zero*
`000000`	$\longrightarrow$	the number *zero*

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

2.1.2 Booleans

According to the syntax rules of Scheme, the character sequences, `#t` and `#f`, are legal Scheme expressions. According to the semantics of Scheme, these expressions respectively denote the truth values *true* and *false*, as illustrated below:

`#t` $\longrightarrow$ the *true* truth value

`#f` $\longrightarrow$ the *false* truth value

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.*

`()` $\longrightarrow$ 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.

$\boxed{} \longrightarrow$ 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.

`hello` $\longrightarrow$ the symbol *hello*

$\star$ $\longrightarrow$ the asterisk symbol

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

2.2 String Expressions

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

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

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

`"hi"` $\longrightarrow$ the string *"hi"*

`"Howdy!"` $\longrightarrow$ the string *"Howdy!"*

2.3 Summary

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

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

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

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

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

Chapter 3

Evaluating Scheme Data

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

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

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

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

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

$$\text{Input Datum} \quad \Longrightarrow \quad \text{Output Datum}$$

The double arrow ($\Longrightarrow$) 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.

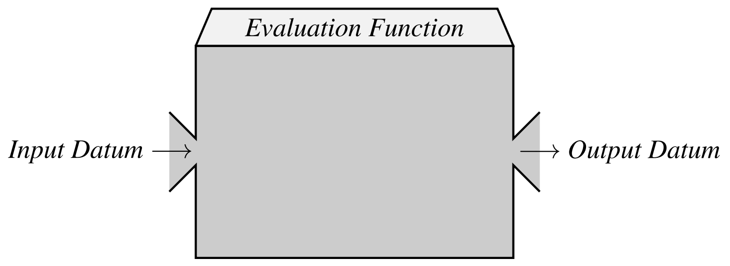


Figure 3.1: The evaluation function in Scheme

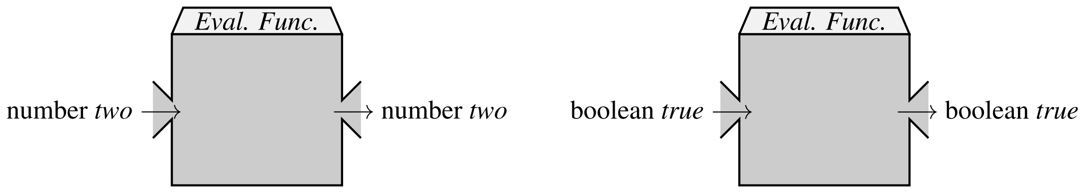


Figure 3.2: Sample evaluations in Scheme

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*	$\Longrightarrow$	the number *zero*
the boolean *true*	$\Longrightarrow$	the boolean *true*
the *empty list*	$\Longrightarrow$	the *empty list*
the *void* datum	$\Longrightarrow$	the *void* datum
the string *"hi there"*	$\Longrightarrow$	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.*

Symbol	Value
num	number *three*
xyz	number *two*
boolie	boolean *true*
stringy	string *"hello"*

Figure 3.3: A sample environment

⋆ 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 *num* evaluates to the number *three* with respect to the environment shown in Fig. 3.3. Similarly, the symbol *xyz* evaluates to the number *two* in that environment; and the symbol *boolie* evaluates to the boolean *true*.

For another example, if an environment $\mathcal{E}_0$ contains an entry that pairs the symbol *xyz* with the number *two,* while another environment $\mathcal{E}_1$ contains an entry that pairs the symbol *xyz* with the boolean *false,* then evaluating *xyz* with respect to the environment $\mathcal{E}_0$ will yield the number *two,* while evaluating *xyz* with respect to $\mathcal{E}_1$ will yield the boolean *false.*

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

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

It may help to think of an environment as a room that has a 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 *xyz* with the number *two,* then the symbol *xyz* will evaluate to the number *two*:

$$\text{the symbol } \textit{xyz} \quad \Longrightarrow \quad \text{the number } \textit{two}$$

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 *true* is the value for the symbol *pq*. Similarly, the empty list might be the value associated with the symbol *my-empty-list*, as illustrated below.

$$\text{the symbol } \textit{pq} \quad \Longrightarrow \quad \text{the boolean } \textit{true}$$

$$\text{the symbol } \textit{my-empty-list} \quad \Longrightarrow \quad \text{the } \textit{empty list}$$

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

$$\text{the symbol } \textit{foo} \quad \Longrightarrow \quad \text{the symbol } \textit{bar}$$

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

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

3.3 Summary

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

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

$$\text{the symbol } x \quad \Longrightarrow \quad \text{the number } \textit{eighty-six}$$

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:

$$\texttt{x} \quad \longrightarrow \quad \text{the symbol } x \quad \Longrightarrow \quad \text{the number } \textit{eighty-six}$$

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.

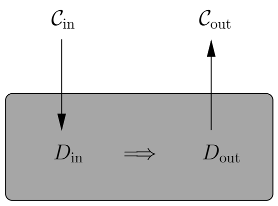
More formally, when we type a sequence of characters, $\mathcal{C}_{\text{in}}$, into the *Interactions Window*, and then hit the *Return* (or *Enter*) key, DrScheme does the following:

(1) It figures out which Scheme datum, D_{in}, is denoted by the character sequence $\mathcal{C}_{\text{in}}$;

(2) It feeds that Scheme datum as input to the evaluation function, which generates an output datum, D_{out} (i.e., D_{in} *evaluates* to D_{out}).

(3) Finally, it displays some typewritten text, $\mathcal{C}_{\text{out}}$, in the *Interactions Window* that describes the output datum, D_{out}.

This process is illustrated below.



$\mathcal{C}_{\text{in}}$
$\mathcal{C}_{\text{out}}$
$D_{\text{in}} \Longrightarrow D_{\text{out}}$

[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.

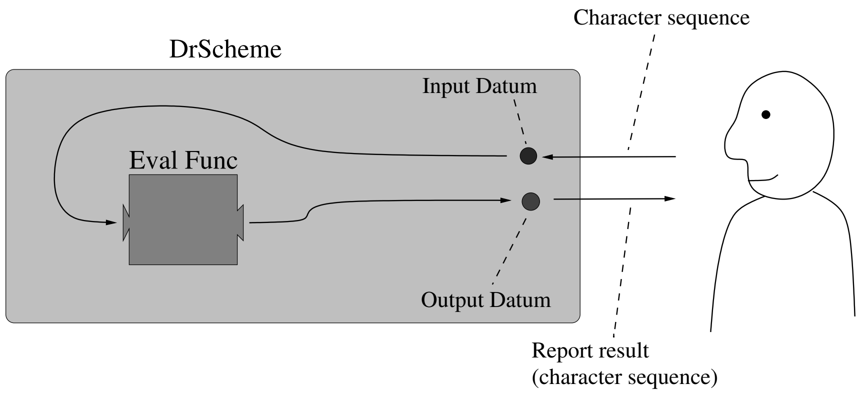


Figure 4.1: A programmer's interaction with DrScheme

Keep in mind that we only see the character sequences, $\mathcal{C}_{\text{in}}$ and $\mathcal{C}_{\text{out}}$; we do not see the Scheme data, D_{in} and D_{out}. (What does a Scheme datum look like anyway?) We can more succinctly describe this process as follows:

$$\mathcal{C}_{\text{in}} \quad \longrightarrow \quad [\quad D_{\text{in}} \quad \Longrightarrow \quad D_{\text{out}} \quad] \quad \longrightarrow \quad \mathcal{C}_{\text{out}}$$

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

⋆ When entering expressions into the Interactions Window, the datum D_{in} is evaluated with respect to the Global Environment.

4.1 Entering Expressions into the Interactions Window

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

Example 4.1.1: DrScheme's Interactions Window

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

```
> 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:

$$\begin{array}{rclcccl}
\texttt{3} & \longrightarrow & [& \text{the number } \textit{three} & \Longrightarrow & \text{the number } \textit{three} &] \longrightarrow \texttt{3} \\
\texttt{\#t} & \longrightarrow & [& \text{the boolean } \textit{true} & \Longrightarrow & \text{the boolean } \textit{true} &] \longrightarrow \texttt{\#t} \\
\texttt{()} & \longrightarrow & [& \text{the } \textit{empty list} & \Longrightarrow & \text{the } \textit{empty list} &] \longrightarrow \texttt{()} \\
\texttt{"Howdy!"} & \longrightarrow & [& \text{the string } \textit{“Howdy!”} & \Longrightarrow & \text{the string } \textit{“Howdy!”} &] \longrightarrow \texttt{"Howdy!"}
\end{array}$$

Example 4.1.2

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

```
> 0
0
> 000
0
> 000000
0
```

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

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

4.2 DrScheme's *Run* Button

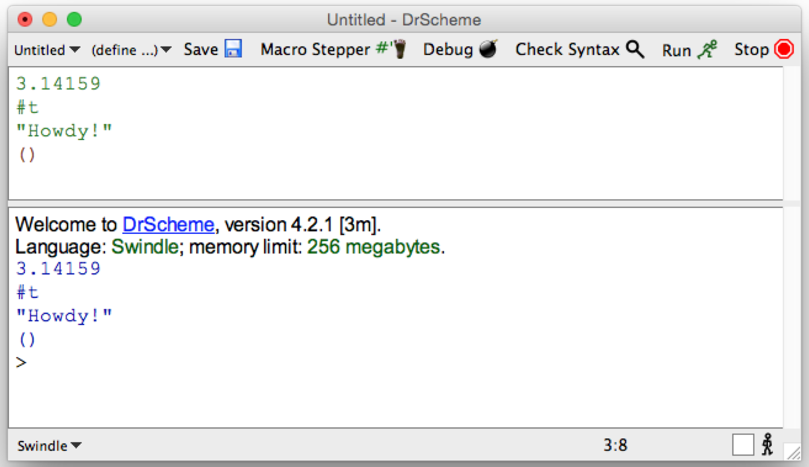
Although manually typing individual expressions into the Interactions Window and viewing DrScheme's responses can be quite useful, it is often desirable to ask DrScheme to evaluate a large number of Scheme expressions. (Re-read the above note about "evaluating expressions".) To avoid endless typing and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

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

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

4.3 Summary

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



```
3.14159
#t
"Howdy!"
()
```

```
Welcome to DrScheme, version 4.2.1 [3m].
Language: Swindle; memory limit: 256 megabytes.
3.14159
#t
"Howdy!"
()
>
```

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

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

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

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

This process is concisely summarized by:

$$\mathcal{C}_{\text{in}} \quad \longrightarrow \quad [\quad D_{\text{in}} \quad \Longrightarrow \quad D_{\text{out}} \quad] \quad \longrightarrow \quad \mathcal{C}_{\text{out}}$$

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:

the symbol `+`	$\Longrightarrow$	the *addition* function
the symbol `-`	$\Longrightarrow$	the *subtraction* function
the symbol `*`	$\Longrightarrow$	the *multiplication* function
the symbol `/`	$\Longrightarrow$	the *division* function

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

[1]Indeed, 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:[a]

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

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

$$+ \longrightarrow [\ \textit{the + symbol} \Longrightarrow \textit{the addition function}\ ] \longrightarrow \texttt{\#<procedure:+>}$$

$$- \longrightarrow [\ \textit{the - symbol} \Longrightarrow \textit{the subtraction function}\ ] \longrightarrow \texttt{\#<procedure:->}$$

$$\star \longrightarrow [\ \textit{the * symbol} \Longrightarrow \textit{the multiplication function}\ ] \longrightarrow \texttt{\#<procedure:}\star\texttt{>}$$

$$/ \longrightarrow [\ \textit{the / symbol} \Longrightarrow \textit{the division function}\ ] \longrightarrow \texttt{\#<procedure:/>}$$

[a]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:

Name:	+
Inputs:	$x_1, x_2, \ldots, x_n$; *zero or more numerical inputs*
Output:	*The sum,* $x_1 + x_2 + \ldots + x_n$
Side Effects:	*None*

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:*

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:

```
> 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.

$$\begin{aligned} \textit{the addition function} &\Longrightarrow \textit{the addition function} \\ \textit{the multiplication function} &\Longrightarrow \textit{the multiplication function} \\ \textit{the evaluation function} &\Longrightarrow \textit{the evaluation function} \end{aligned}$$

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

5.5 The Built-in Functions `printf` and `void`

Recall from Section 2.1.4 that Scheme includes a data type called *void* whose only datum is also called *void.* The purpose of the *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 *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 *void* output value. (Since *void* represents "no value", DrScheme does not feel compelled to display anything for *void*.)

⋆ If a function's output is *void*, then we may say that the function does not generate any output value.

The built-in functions, `printf` and `void`, introduced below, are examples of functions whose output value is invariably the *void* datum.

5.5.1 The `printf` Function

Scheme provides a built-in `printf` function that can be used to display information in the Interactions Window.

⋆ The display of textual information by the `printf` function is an example of a harmless *side effect.*

⋆ The output value generated by the `printf` function is always the *void* datum.

Although the `printf` function has additional capabilities that won't be explored until Chapter 10, the contract for the simplest use of the `printf` function, given below, will suffice for now.

Name:	`printf`
Input:	*str,* a string
Output:	the *void* datum (i.e., "no value")
Side Effect:	displays the contents of the string *str* in the Interactions Window (without the double-quotes)

5.5.2 The `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 *void* datum. However, should you ever need to get your hands on the *void* datum, there is a built-in function, called `void`, that does nothing but generate the *void* datum as its output. Here is its contract:

Name:	`void`
Inputs:	Any number of inputs
Output:	the *void* datum
Side Effects:	none

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:

$$+ \longrightarrow [\text{ the + symbol } \Longrightarrow \text{ the built-in } \textit{addition} \text{ function }] \longrightarrow \texttt{\#<procedure:+>}$$

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.*

$\star$ 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.

$\star$ 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)` $\longrightarrow$ a list containing the number *three* and the number *four*

`(+ 3 4)` $\longrightarrow$ a list containing the + symbol, the number *three,* and the number *four*

`(eval (+ 3 4))` $\longrightarrow$ a list containing:

(1) the symbol *eval*, and

(2) a subsidiary list containing the + symbol, the number *three,* and the number *four*

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

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

is a legal character sequence. (That's the syntax!) Furthermore, that character sequence denotes a list containing the n items denoted by $E_1, E_2, \ldots, E_n$. (That's the semantics!) Thus, if

$$E_1 \longrightarrow D_1$$
$$E_2 \longrightarrow D_2$$
$$\ldots$$
$$E_n \longrightarrow D_n$$

(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 $\mathcal{D}$ containing the n items $D_1,\ D_2,\ \ldots,\ D_n$.

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

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

6.2 Evaluating Non-Empty Lists: the Default Rule

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

Example 6.2.1

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

```
> (+ 2 3)
5
> (* 3 4 5)
60
> (+ 2 (* 3 10))
32
> (+ 2 (* 3 (+ 4 8 6)))
56
```

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

Example 6.2.2

Consider the expression `(+ 2 3)`*, which denotes a list containing three items: the + symbol, the number* two*, and the number* three*. The Default Rule for evaluating such lists has two steps. The first step is to* evaluate each item in the list. *Now, the + symbol evaluates to the built-in addition function because the Global Environment is guaranteed to contain an entry associating the + symbol with the addition function.*

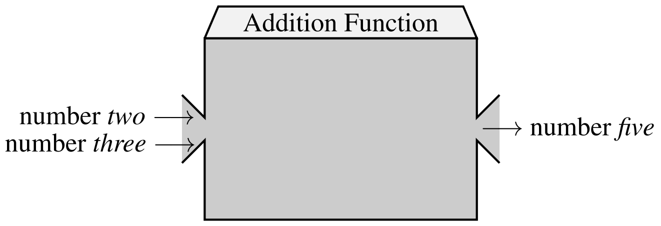
The remaining items in the list are numbers; thus, they trivially evaluate to themselves. The results of the first step are summarized below:

the + symbol ⟹ *the addition function*

the number two ⟹ *the number* two

the number three ⟹ *the number* three

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



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

`(+ 2 3)` ⟶ *[list containing + symbol, number* two, *number* three ⟹ *number* five *]* ⟶ `5`

where the evaluation is explained by:

First Step of the Default Rule:

+ symbol	⟹	*addition function*
number two	⟹	*number* two
number three	⟹	*number* three

Second Step of the Default Rule:

addition function applied to two *and* three *yields output of* five

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

Example 6.2.3

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

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

If we follow the rules we already know, we will see that nothing new is needed to explain this interaction. First, the character sequence `(+ 2 (* 3 10))` *is a legal Scheme expression that denotes a list. The denoted list contains three items: the + symbol, the number* two, *and a subsidiary list. The subsidiary list*

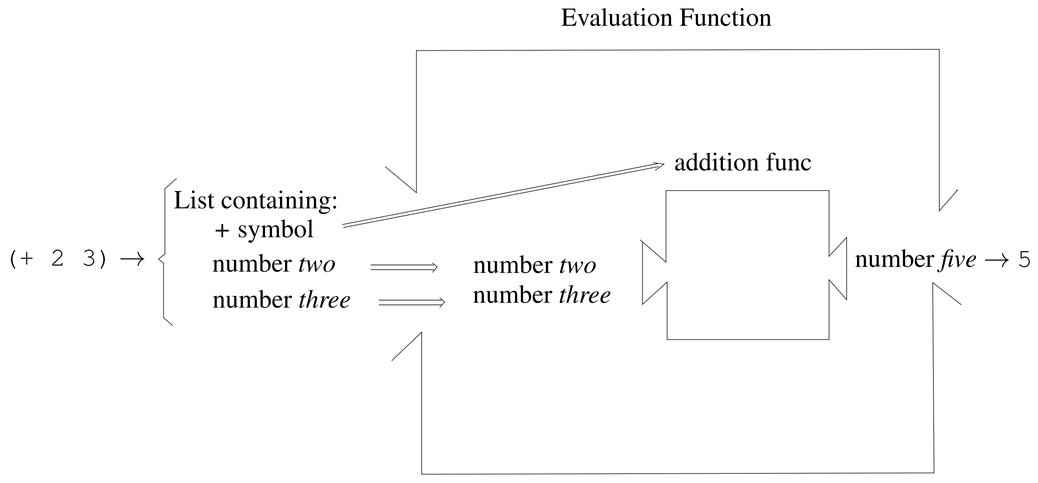


Figure 6.1: The evaluation of the list denoted by `(+ 2 3)`

*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	$\Longrightarrow$	*the addition function*
the number two	$\Longrightarrow$	*the number* two
the subsidiary list	$\Longrightarrow$	**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*	$\Longrightarrow$	*the multiplication function*
the number three	$\Longrightarrow$	*the number* three
the number ten	$\Longrightarrow$	*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	$\Longrightarrow$	*the addition function*
the number two	$\Longrightarrow$	*the number* two
the subsidiary list	$\Longrightarrow$	*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 $\mathcal{L}$ that contains n data items, $D_1, D_2, \ldots D_n$. The evaluation of the list $\mathcal{L}$ 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, $K_1, K_2, \ldots, K_n$:

  $D_1 \Longrightarrow K_1$

  $D_2 \Longrightarrow K_2$

  $\ldots$

  $D_n \Longrightarrow K_n$

- 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 $\mathcal{L}$ is defined to be that datum P (i.e., $\mathcal{L} \Longrightarrow P$).

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 $\mathcal{L}$ be the list containing the following data:

D_1*: the + symbol,* D_2*: the number* one*,* D_3*: the number* two*,* D_4*: the number* three

These Scheme data evaluate to the following:

K_1*: the* addition *function,* K_2*: the number* one*,* K_3*: the number* two*,* K_4*: the number* three

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

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

Example 6.2.6

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

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

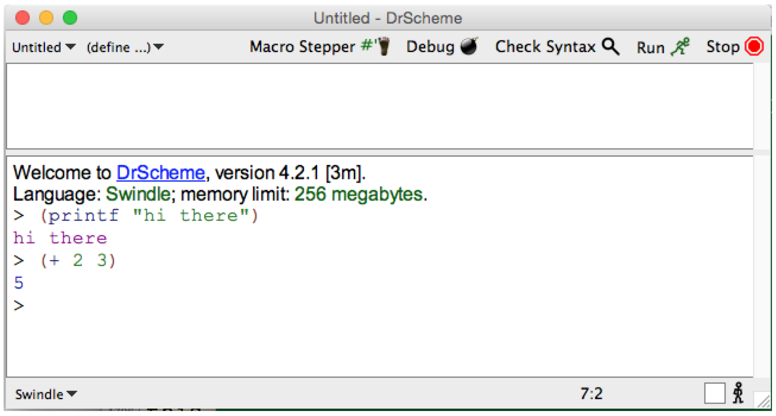
Note that the textual information displayed by DrScheme in each case is side-effect printing, *not a Scheme output value. More interesting uses of the* `printf` *function will be described in Chapter 10.*

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

Example 6.2.7

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

```
> (void)
>
```



```
Welcome to DrScheme, version 4.2.1 [3m].
Language: Swindle; memory limit: 256 megabytes.
> (printf "hi there")
hi there
> (+ 2 3)
5
>
```

Figure 6.2: DrScheme's use of different colors to distinguish side-effect printing from output values

```
> (void 3 #t () "xyz")
>
```

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:*

$$\begin{array}{lcl} \text{eval } \textit{symbol} & \Longrightarrow & \textit{the evaluation function} \\ + \textit{ symbol} & \Longrightarrow & \textit{the addition function} \end{array}$$

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:

```
> (+ 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:

```
(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

$$\texttt{(define } \mathcal{C}_1 \ \mathcal{C}_2\texttt{)}$$

where $\mathcal{C}_1$ is an expression denoting some Scheme symbol s, and $\mathcal{C}_2$ can be any expression denoting any Scheme datum, e, as illustrated below.

$$\mathcal{C}_1 \longrightarrow s \qquad \text{and} \qquad \mathcal{C}_2 \longrightarrow e$$

Therefore:

$$\texttt{(define } \mathcal{C}_1 \ \mathcal{C}_2\texttt{)} \longrightarrow \text{List containing the } \texttt{define} \text{ symbol, the } s \text{ symbol, and the datum } e$$

For example, `(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.

```
(define addn-func +)

(define zero 0)

(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 *evaluated.* The special rule for evaluating `define` special forms is illustrated below:

$$\texttt{(define } \mathcal{C}_1 \ \mathcal{C}_2\texttt{)} \longrightarrow [\ \text{List containing } \texttt{define}, s \text{ and } e \quad \Longrightarrow \quad ▨\ ] \longrightarrow ▨$$

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 *void* datum as its output. Recall from Section 2.1.4 that the *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 *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, $\boxed{s \mid \textit{void}}$, into the Global Environment, where *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: $e \Longrightarrow E$.

(3) Insert E as the value for s in the Global Environment: $\boxed{s \mid E}$.

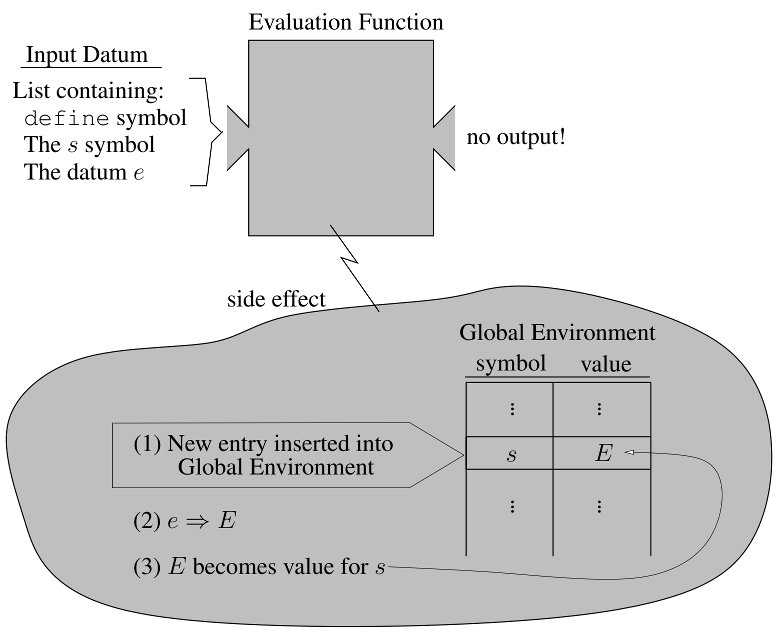


Figure 7.1: The side effect of `define`: inserting a new entry into the Global Environment

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

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

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

Of course, something *has* happened!

Example 7.1.1

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

`x` $\longrightarrow$ the symbol *x*

`(+ 1 2 3)` $\longrightarrow$ [a list containing the + symbol and the numbers one, two and three] $\Longrightarrow$ the number *six*

Side Effect: New Global Environment Entry:

the symbol x	the number six

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

[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
```

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*:* $\texttt{w} \Longrightarrow$ 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` $\mathcal{C}$`)`

where $\mathcal{C}$ 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` $\mathcal{C}$`)`, denotes a list containing two items: the *quote* symbol and whatever $\mathcal{C}$ 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 $\mathcal{C}$ denotes some datum, D, then `(quote` $\mathcal{C}$`)` denotes a list containing the *quote* symbol and D. Using the arrow notation, we can say:

If:	$\mathcal{C}$	$\longrightarrow D$
Then:	`(quote` $\mathcal{C}$`)`	$\longrightarrow$ 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:

$\star$ A list containing the *quote* symbol and D evaluates to ... D.

Notice that, according to this rule, neither the *quote* symbol nor the datum D are evaluated.[2] 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:*

`(quote x)` $\longrightarrow$ *[* {list *with symbols* quote *and* x} $\Longrightarrow$ *the symbol* x *]* $\longrightarrow$ `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 $\mathcal{C}$ is an expression denoting some datum D, then entering the expression, `(quote` $\mathcal{C}$`)`, into DrScheme will cause the following to happen:

`(quote` $\mathcal{C}$`)` $\longrightarrow$ [{*list* containing *quote* symbol and D} $\Longrightarrow$ D] $\longrightarrow$ $\mathcal{C}'$

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, $\mathcal{C}'$, to describe D to us; however, $\mathcal{C}'$ 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 $\mathcal{C}$ is any Scheme expression denoting some datum D, then the expressions, `(quote` $\mathcal{C}$`)` and `'`$\mathcal{C}$, denote the *same* two-element list—namely, a list containing the *quote* symbol and the datum D:

$$\begin{array}{lll} \texttt{(quote } \mathcal{C}\texttt{)} & \longrightarrow & \text{list containing } \textit{quote} \text{ symbol and } D \\ \texttt{'}\mathcal{C} & \longrightarrow & \text{list containing } \textit{quote} \text{ symbol and } D \end{array}$$

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

Example 7.2.4

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

```
> (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:

```
> (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.[1] And so on.

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

Example 8.1.1

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

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

[1]The *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?`.[a]

[a]This text uses the terms, *function* and *procedure,* interchangeably; however, the term *function* seems better suited given that Scheme is typically referred to as a *functional* programming language.

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

Example 8.1.2

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

Name:	`number?`
Input:	*d, any Scheme datum*
Output:	`#t` *if* `d` *is a number; otherwise,* `#f`

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.

```
> (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]

Example 8.2.1

```
> (> 3 4)
#f
> (> 4 3)
#t
> (>= 4 3)
#t
> (= 3 4)
#f
> (= 3 3)
#t
```

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

[3]These predicates can also be applied to more than two inputs; however, we shall postpone discussion of such things until Chapter **??**.

DrScheme also provides a comparison predicate called `eq?` that is more general that 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:

```
> (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, 6π.

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

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

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

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

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

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

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

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

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

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

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

Example 9.2.2

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

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

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

Example 9.2.3

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

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

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

9.3 The Syntax and Semantics of Lambda Expressions

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

9.3.1 The Syntax of a Lambda Expression

A lambda expression has the following syntax:

`(lambda (`$\mathcal{C}_1$ $\mathcal{C}_2$ `...` $\mathcal{C}_n$`)` $\mathcal{B}$`)`

where:

- each $\mathcal{C}_i$ 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
- $\mathcal{B}$ is a character sequence denoting a Scheme datum, $\mathcal{D}$, of any kind.

Thus, $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ specify n distinct *input parameters* for the lambda expression, and $\mathcal{B}$ 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 $\mathcal{C}_i$ denotes a Scheme symbol, s_i;
- the symbols, $s_1, s_2, \ldots, s_n$, are distinct; and
- $\mathcal{B}$ denotes some Scheme datum $\mathcal{D}$,

then a lambda expression of the form

$$\texttt{(lambda (}\mathcal{C}_1\ \mathcal{C}_2\ \ldots\ \mathcal{C}_n\texttt{)}\ \mathcal{B}\texttt{)}$$

denotes a Scheme *list* whose elements are as follows:

- the `lambda` symbol;
- a list containing n distinct symbols, $s_1, s_2, \ldots, s_n$; and
- the Scheme datum, $\mathcal{D}$

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, `(lambda (x) (* x x))`*. As noted above, it evaluates to a Scheme function. When this* lambda function *is applied to some input value, say* `4`*, the following things happen:*

- *A* local environment *is created that contains a single entry in which the symbol* `x` *has the value* `4`*.*
- *The expression,* `(* x x)`*, which constitutes the body of the function, is evaluated* with respect to the newly created local environment. *This means that: (1) any occurrence of the symbol* `x` *is evaluated using the entry for* `x` *in the local environment, ignoring any entry for* `x` *that might exist in the Global Environment; and (2) all other symbols are evaluated with respect to the Global Environment. The evaluation of* `(* x x)` *therefore yields the result* `16`*, because* `x` *evaluates to* `4` *in the local environment, and* `*` *evaluates to the built-in multiplication function in the Global Environment.*
- *That value,* `16`*, is taken to be the output value that results from applying the* `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, `pi`*, to hold the value* 3.14159*. It then defines a function, named* `sphere-volume`*. Finally, it applies this function to some sample input values.*

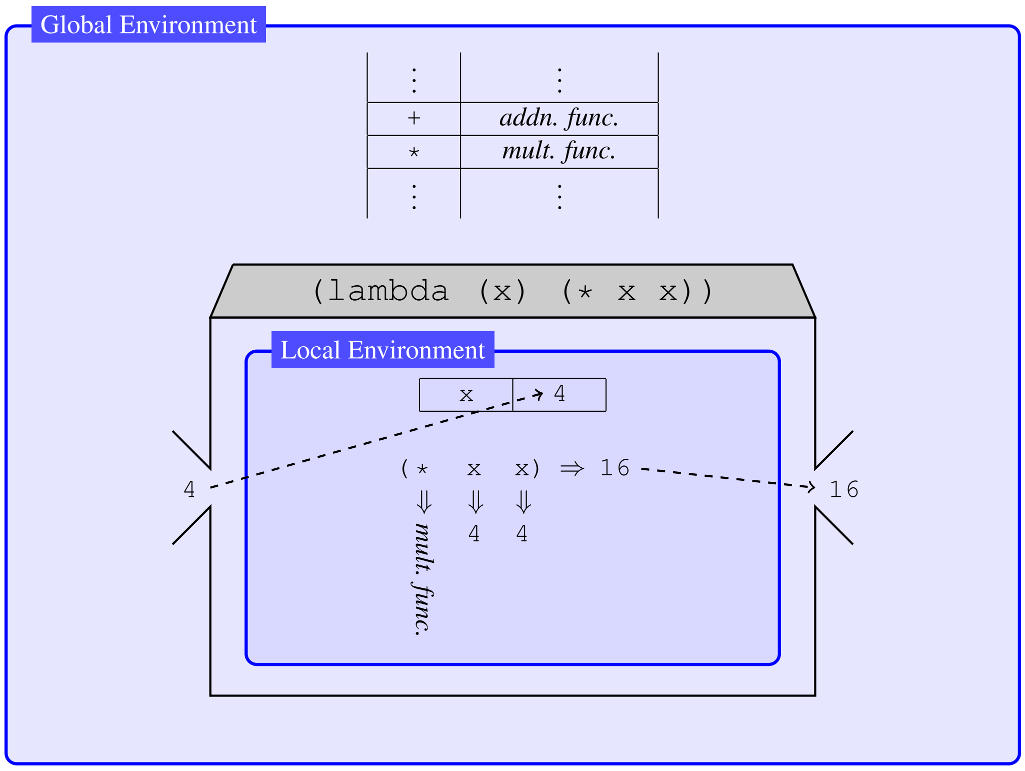


Figure 9.1: Applying the function corresponding to `(lambda (x) (* x x))` to the value `4`

```
> (define pi 3.14159)
> (define sphere-volume (lambda (r) (* 4/3 pi r r r)))
> (sphere-volume 1)
4.188786666666666
> (sphere-volume 2)
33.51029333333333
```

Consider the evaluation of the expression, `(sphere-volume 2)`*. It involves the following steps:*

- *First, a local environment is set up containing a single entry in which the symbol* `r` *has the value* `2`*.*
- *Next, the expression,* `(* 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* `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* `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* `pi` *came from the Global Environment.*

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

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

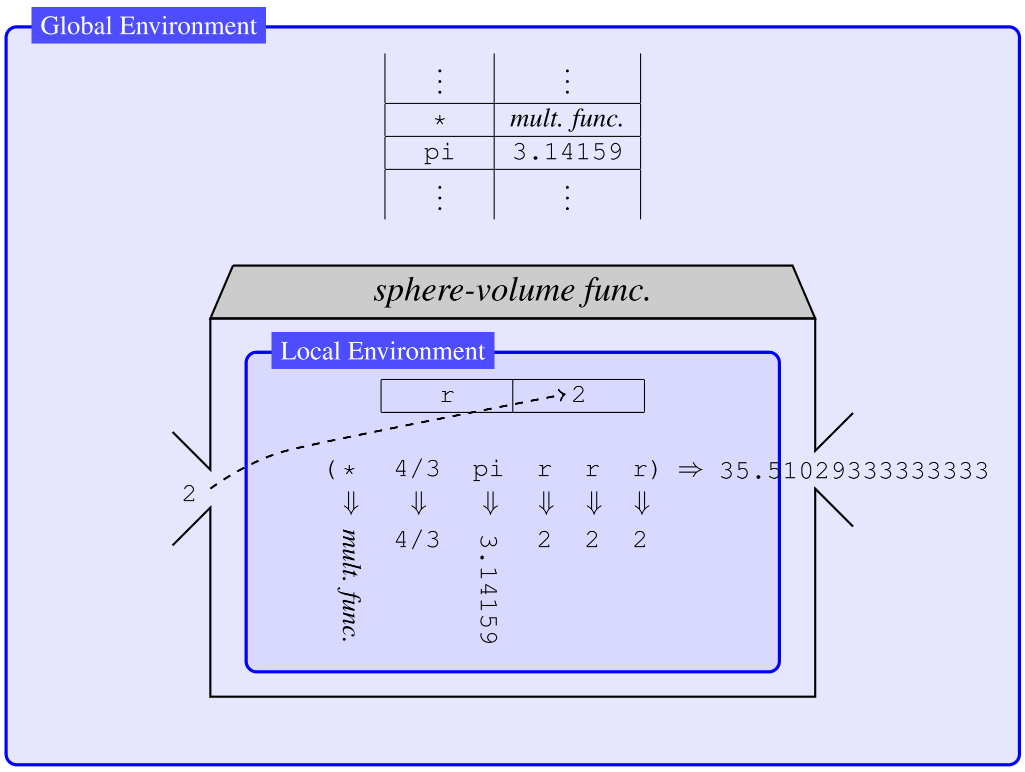


Figure 9.2: Applying the `sphere-volume` function to the value `2`

The following Interactions Window session (continuing from the one given above) illustrates that the existence of a global variable named `r` *has no effect on the local variable that also happens to be named* `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`*, evaluations to* `10`*; and*
- *The squaring function is applied to* `10`*, yielding the output value* `100`*.*

Example 9.3.6

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

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

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

Differences Between Mathematical Notation and Lambda Notation

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

In Scheme, we can use a lambda special form to define a function without giving it a name. For example, we might evaluate `(lambda (x) (* x x))` to create a squaring function. However, we cannot replace the parameter `x` in that lambda expression by arbitrary expressions. For example, `(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:

$f(x) = x^2$

$f(3) = 9$

$f(2+3) = 25$

$f(f(10)) = 10000$

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.,* `()`*) serves as the output value. This function is kind of silly since the results of evaluating the first four expressions in its body are thrown away.*

⋆ *The only way that intermediate expressions in the body of a function could have any impact is if they caused* side effects.

Up to this point, the only function that we have seen that has side effects is the built-in `printf` function. It displays the contents of a string in the Interactions Window. This is a harmless side effect that can be very useful.

9.4 Summary

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

(1) the `lambda` symbol;

(2) a list of input parameters; and

(3) one or more expressions constituting the *body* of the function.

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

In Scheme, the following are distinct:

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

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

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

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

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

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

Special Forms Introduced in this Chapter

`lambda`	Used to specify functions of our own design.

Chapter 10

Some practicalities

This chapter introduces the following practicalities:

- Further capabilities of the built-in `printf` function. This function, which takes a *string* as one of its inputs, can be used to display nicely formatted information in DrScheme's Interactions Window. Its functionality is similar to that of the format/print functions found in many programming languages.
- The built-in `load` function. This function causes the Scheme expressions in a specified file to be evaluated as though they had been manually typed into the Interactions Window. As such, these expressions are evaluated with respect to the Global Environment. In this way, a library of useful Scheme definitions can be incorporated into your own program quite easily. The name of the file is specified by a *string*.
- Comments. A comment is a piece of syntax that DrScheme completely ignores. Comments are used by programmers to help clarify—for people—what the program/code is supposed to do.

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

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

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

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

Escape sequences. In the above example, the `printf` function effectively copied the contents of the input string into the Interactions Window *verbatim.* However, the `printf` function sometimes deviates from this simple behavior. In particular, as the `printf` function walks through the input string, it reacts to a few special character sequences in special ways.

- 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 `printf` function *escaping* from a literal interpretation, they are frequently called *escape sequences*. (And the characters `~` and `\` that introduce escape sequences are sometimes called escape characters.) Although the `printf` function can deal with a variety of other escape sequences, these are the only ones that we'll need for this course. Their use enables the `printf` function to generate nicely formatted text in the Interactions Window. For this reason, the input string is frequently called a *format string*—which explains the `f` in `printf`.

In summary, the `printf` function causes the contents of the format string (i.e., its first input) to be displayed verbatim in the Interactions Window, except that:

- the quotation marks are omitted;
- each instance of `~%` or `\n` is interpreted as a newline character and, thus, causes subsequent text to be displayed on the next line in the Interactions Window; and
- each instance of `~s` or `~a` is replaced by a character sequence representing the *value* of the corresponding input expression.

Notice that if the format string contains n instances of `~s`, then there must be n input expressions following the format string, as follows:

(`printf` *format-string* *expr*$_1$ … *expr*$_n$)

Example 10.1.1: Formatted printing with `printf`

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

```
> (printf "Hi there!\nBye there!")
Hi there!
Bye there!
> (printf "Oh, I get it!~%This sentence begins on a new line!")
Oh, I get it!
This sentence begins on a new line!
> (printf "First thing: ~s, second thing: ~s~%" (+ 2 3) (* 6 7))
First thing: 5, second thing: 42
> (printf "Line One!~%  Line Two!!~%    Line Three!!!~%")
Line One!
  Line Two!!
    Line Three!!!
> (printf "First ===> ~s, Second ===> ~s, Third ===> ~s~%"
          (+ 4 2) (- 9 6.3) (* 4 100))
First ===> 6, Second ===> 2.7, Third ===> 400
> (printf "A symbol: ~s, a string: ~s, a boolean: ~s~%"
          'I-am-a-symbol
          "I am a String!"
          (> 4 2))
A symbol: I-am-a-symbol, a string: "I am a String!", a boolean: #t
> (printf "String shown by tilde A: ~a~%" "I am a string!")
String shown by tilde A: I am a string!
> (printf "String shown by tilde S: ~s~%" "I am a string!")
String shown by tilde S: "I am a string!"
```

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:
```

```
12
>
```

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

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

Example 10.1.4: Defining a useful `tester` function

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

```
(define tester
  (lambda (datum)
    (printf "~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:

Name:	`load`
Input:	*filename*, a string
Output:	The result of evaluating the *last* expression in the file named *filename*.
Side Effect:	Whatever side effects result from evaluating *all* of the expressions in the file named *filename*.

Example 10.2.1

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

```
(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

```
(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:

`(if` *condExpr thenExpr elseExpr* `)`

where:

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

Example 11.1.1

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

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

(if (> 4 2) 'then 'else)

(if #f "then" "else")
```

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

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

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

Example 11.1.2

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

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

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

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

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

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

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

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

```
;;  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 `eq?`

Suppose that `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,* `(eq? boolie #t)`*, is equivalent to the much simpler expression,* `boolie`*, as demonstrated below.*

```
> (define boolie #t)
> boolie
#t
> (eq? boolie #t)
#t
> (define bully #f)
> bully
#f
> (eq? bully #t)
#f
```

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

```
> (eq? (> 5 2) #t)
#t
> (> 5 2)
#t
> (eq? (= 3 2) #t)
#f
> (= 3 2)
#f
```

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

```
> (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
```

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

`(eq?` *boolie* `#t)` $\leadsto$ *boolie*

`(if (eq?` *boolie* `#t)` *thenExpr elseExpr*`)` $\leadsto$ `(if` *boolie thenExpr elseExpr*`)`

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, `(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 $x < y$.*

```
> (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:

`(if` *boolCond* `#t #f)` $\leadsto$ *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

```
> (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,* `(if 'happy #t #f)` *and* `'happy`*, are equivalent in the sense that they both evaluate to things that count as true. The third and fourth lines demonstrate the case where the condition of an* `if` *expression evaluates to* `#f`*.*

In-Class Problem 11.2.1

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

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

Hint: Use an unsimplified *conditional expression!*

11.3 The `when` Special Form

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

```
(when condExpr thenExpr)
```

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

Example 11.3.1

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

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

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

Example 11.3.2

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

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

(define how-big-v2
  (lambda (num verbose?)
    ;; Do some side-effect printing?
    (when verbose?
      (printf "Inside HOW-BIG-V2: 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, $expr_1$, …, $expr_n$, 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, $expr_1$, …, $expr_n$, 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, `(symbol? 'x)` is a function-call expression that calls the built-in *symbol?* function. In contrast, the expressions, `(define myVar 3)` and `(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, `facty-v1`*, whose definition is based on the above insights. (The function is called,* `facty-v1`*, because it is the first version of the factorial function we will look at.)*

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

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

Notice that the `define` *special form effectively gives the name,* `facty-v1`*, to the function defined by the* `lambda` *special form. Notice, too, that the body of this function includes a conditional expression that distinguishes the base case (i.e., when* `n = 1`*) from the recursive case (i.e., when* `n > 1`*). Finally, notice* *that the body includes a function-call expression that calls* `facty-v1`*. (We'll have more to say about this!)*

Okay, so what happens when the above expression is evaluated? Well, the expression is a `define` *special form. So, the symbol,* `facty-v1`*, is* not *evaluated. Only the third element of the* `define` *special form (i.e., the* `lambda` *expression) is evaluated. Like any* `lambda` *expression, the one above evaluates to a function. However:*

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

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

(1) an entry for `facty-v1` *in the Global Environment is created with a temporary value:* void*;*

(2) the `lambda` *expression is evaluated, which yields a function; and*

(3) that function is entered into the Global Environment as the value associated with `facty-v1`*.*

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

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.*[a]

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

Since the value of `n` *is* `3` *in the local environment, the condition* `(= n 1)` *evaluates to* `#f`*. Thus, the* then *expression,* `1`*, is skipped, and the* else *expression,* `(* n (facty-v1 (- n 1)))`*, is evaluated—according to the Default Rule. The* `*` *symbol evaluates to the* multiplication *function,* `n` *evaluates to* `3`*, and* `(facty-v1 (- n 1))` *evaluates to ... Gosh, we need a new paragraph!*

The subsidiary expression, `(facty-v1 (- n 1))`*, is evaluated according to the Default Rule. First, the* `facty-v1` *symbol evaluates to the* `facty-v1` *function; and* `(- n 1)` *evaluates to* `2` *(since* `n` *has the value* `3` *in the local environment). Next, the* `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,* `(* (facty-v1 (- n 1)))`*, in the top function-call box cannot continue until the subsidiary expression,* `(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 `facty-v1` *function to the value* `2`*, depicted in the second function-call box in the figure, is similar to the application of the* `facty-v1` *function to* `3` *in the top box, except that the local environment in the second box associates the input parameter,* `n`*, with the value* `2`*.*

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

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

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

The application of the `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* `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,* `(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* `(facty-v1 (- n 1))` *is in hand, the evaluation of the expression,* `(* n (facty-v1 (- n 1)))`*, in the middle box can continue. To wit, the* mulitplication *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,* `(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* `(facty-v1 (- n 1))` *is in hand, the evaluation of the expression,* `(* n (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!

[a] 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.

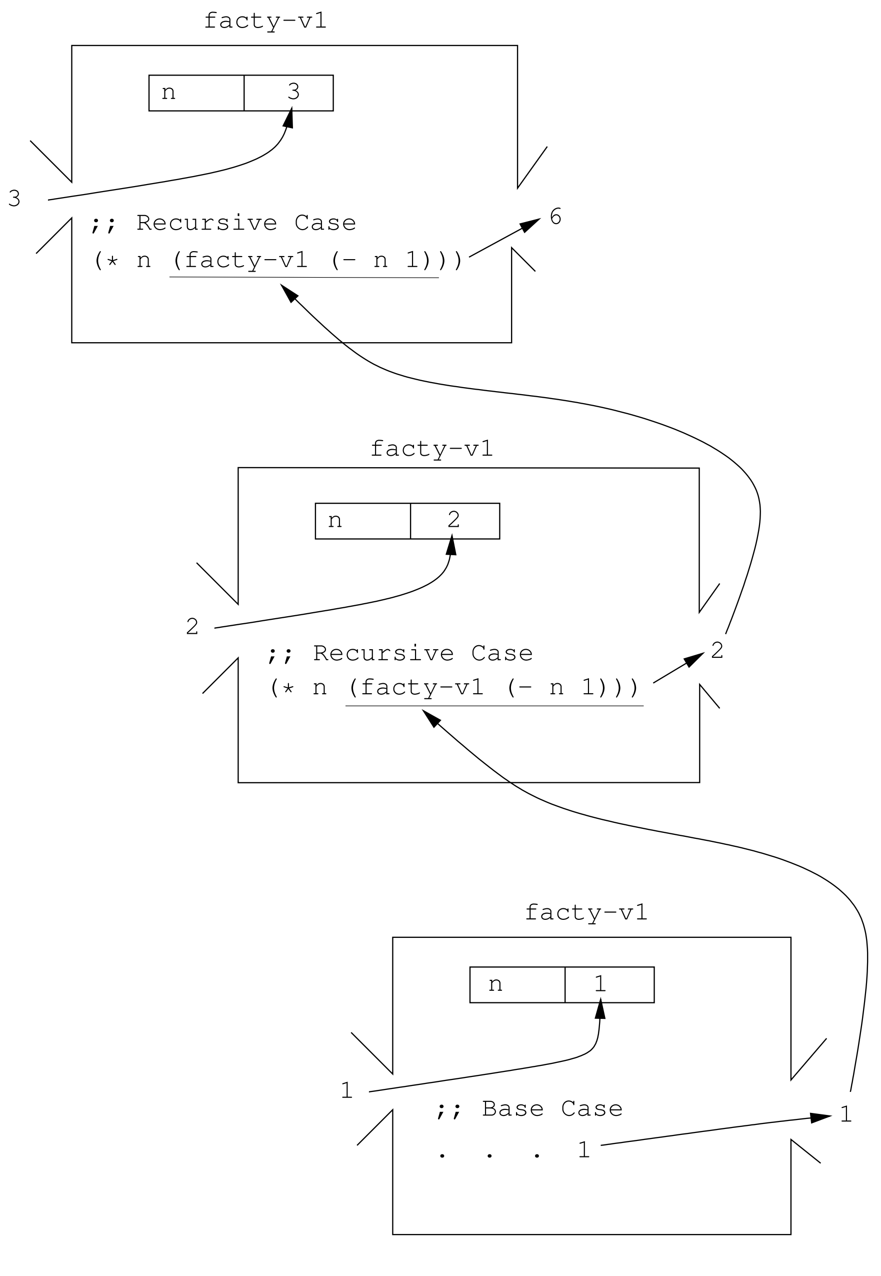


Figure 12.1: DrScheme's evaluation of `(facty-v1 3)`

- 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.*

```
> (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:*

```
;;  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.[1] 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.

Example 13.1.1: Nested if expressions

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

```
;;  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)
```

[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.

```
                       'C
                       'D)))))
```

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`s

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

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

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

```
          ((>= num 70)
           'C)
          ;; Case 4:  Got a D
          (#t
           'D))))
```

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

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

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

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

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

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

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

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

```
(cond
  (cond₁
   expr₁)
  (cond₂
   expr₂)
     ⋮
```

```
    (cond_n
     expr_n)
   )
```

where:

- each $\mathtt{cond}_i$ is a (strict or non-strict) condition;
- (in most circumstances) the last condition, $\mathtt{cond}_n$, is either `#t` or `else`; and
- each $\mathtt{expr}_i$, called an *answer expression,* can be any Scheme expression.

The value of a `cond` expression is determined as follows:

- Each condition, $\mathtt{cond}_i$, is evaluated in turn until one is found that evaluates to `#t`—or something that counts as true.
- The value of the `cond` expression is the value of the corresponding answer expression, $\mathtt{expr}_i$.
- If every condition evaluates to `#f`, then the entire `cond` expression evaluates to *void.*

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

Example 13.1.3

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

```
> (cond)                ←— No cases in this cond
> (cond
    ((> 2 3)            ←— The conditions in both cases evaluate to #f
     '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
```

In many programming circumstances, when a `cond` *expression evaluates to* 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* `cond` *use* `else` *or* `#t` *as its condition, thereby enabling it to serve as a "catch-all" (or default) case, ensuring that no cases are missed. (Exceptions will be discussed below.)*

The `cond` special form, even more generally! Recall that the body of a `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, $cond_i$ can be followed by multiple subsidiary expressions. Typically, having multiple expressions for a single condition only makes sense if the expressions before the last one have side effects. As with the body of a `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!!!~%")
      'A)
     ;; Case 2:  Got a B
     ((>= num 80)
      (printf "Well, you know, a B is pretty good!!~%")
      (printf "Nothing to be ashamed of at all!!~%")
      'B)
     ;; Case 3:  Got a C
     ((>= num 70)
      (printf "The student handbook says a C is average!~%")
      (printf "Thus, your grade, ~s, is average!~%" num)
      'C)
     ;; Case 4:  Something else
     (else
      (printf "Hmmm...  Hard to find much positive to say here.~%")
      (printf "Maybe there's been a mistake...~%")
      (printf "But until we find it, your grade stands...~%")
      'D))))
```

The behavior of this function is illustrated below:

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

In each case, the conditions were evaluated sequentially until one was found that evaluated to `#t`*. The subsidiary expressions associated with that condition were then evaluated sequentially, and the value of the last subsidiary expression was given as the value of the entire* `cond` *expression. (Although DrScheme reports the output value in a different color, it is hard to see the differences in color in a black-and-white transcript of an Interactions Window session.)*

For example, the expression, `(cond-effects 86)`*, was evaluated as follows. First, the condition,* `(>= num 90)`*, was evaluated. Since it evaluated to* `#f`*, the second condition,* `(>= num 80)`*, was evaluated. This one evaluated to* `#t`*. Thus, the associated subsidiary expressions were evaluated in turn. The value of the last subsidiary expression was* `B`*. Thus,* `B` *was returned as the output value for the entire* `cond` *expression. Notice that only the subsidiary expressions associated with the condition,* `(>= num 80)`*, were evaluated. The subsidiary expressions associated with the other conditions were ignored. The remaining conditions (i.e.,* `(>= num 70)` *and* `#t`*) were also ignored.*

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

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

```
(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, $expr_i$, is evaluated in turn, and the value of the last expression, $expr_n$, is taken to be the value of the entire `when` or `cond` expression. However, if the condition evaluates to `#f`, then the value of the entire expression is *void.*

13.2 Boolean Operators: *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.

x	y	$AND(x, y)$
false	*false*	*false*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

x	y	$OR(x, y)$
false	*false*	*false*
false	*true*	*true*
true	*false*	*true*
true	*true*	*true*

x	$NOT(x)$
false	*true*
true	*false*

x	y	$AND(x, y)$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$OR(x, y)$
0	0	0
0	1	1
1	0	1
1	1	1

x	$NOT(x)$
0	1
1	0

Figure 13.1: Truth tables for *AND*, *OR* and *NOT*, expressed using truth values (above) and binary values (below)

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:*

```
> (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:*

```
;;  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

```
> (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:

```
;;  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:*

```
;;  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:

```
> (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:*

```
;;  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 `cafe-open?` *can be written using the boolean operators,* `and`, `or` *and* `not`, *instead of the conditional expressions,* `cond` *or* `if`. *When approaching the definition of a predicate in this way, the following advice can be very helpful:*

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

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

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

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

13.4 Simplifying Conditional and Boolean Expressions

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

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

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

`(if` *condExpr* `#f #t)` ⇝ `(not` *condExpr*`)`

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

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

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

  (and e_1 (and e_2 e_3)) $\leadsto$ (and e_1 e_2 e_3)

  (or e_1 (or e_2 e_3)) $\leadsto$ (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* stipluate that the following equivalences hold, but we can't really call them simplifications:

(not (and e_1 e_2)) $\leadsto$ (or (not e_1) (not e_2))

(not (or e_1 e_2)) $\leadsto$ (and (not e_1) (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 i^{th} case is the *first* case whose condition evaluates to (non-strict) *true,* then the expressions in the body of that case are evaluated; and all subsequent cases are ignored.

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

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

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

`(if` *someExpr* `#t #f)`	$\leadsto$	*someExpr*
`(if` *someExpr* `#f #t)`	$\leadsto$	`(not` *someExpr*`)`
`(and` e_1 `(and` e_2 e_3`))`	$\leadsto$	`(and` e_1 e_2 e_3`)`
`(or` e_1 `(or` e_2 e_3`))`	$\leadsto$	`(or` e_1 e_2 e_3`)`

Special Forms Introduced in this Chapter

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

Built-in Functions Introduced in this Chapter

`not`	Toggles boolean values

Chapter 14

Recursion II

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

14.1 Recalling the Factorial Function

We begin by considering an equivalent version of the `facty-v1` function, called `facty-v2`, that uses the `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
      (#t
       (* n (facty-v2 (- n 1)))))))
```

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

```
> (facty-v2 4)
24
```

Example 14.1.2

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

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

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

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

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

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

Example 14.1.3: Summing Squares

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

Base Case ($n = 1$): $g(1) = 1$

Recursive Case ($n > 1$): $g(n) = g(n-1) + n^2$

Notice that $g(1) = 1,\ g(2) = 1^2 + 2^2 = 5,\ g(3) = 1^2 + 2^2 + 3^2 = 14,$ *and so on.*

In Scheme, we can define a function, called `sum-squares`*, that computes the sum of the squares from* 1 *to its input value* n*, as follows:*

```
;;  SUM-SQUARES
;; ----------------------------------------
;;  INPUT:  N, a positive integer
;;  OUTPUT: The sum 1*1 + 2*2 + ... + N*N

(define sum-squares
  (lambda (n)
    (cond
      ;; Base Case:  n = 1
      ((= n 1)
       1)
      ;; Recursive Case:  n > 1
      (#t
       (+ (sum-squares (- n 1)) (* n n))))))
```

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

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

14.2 Tail Recursion

Typically, the evaluation of a recursive function-call expression leads to a sequence of recursive function calls. For example, evaluating the expression, `(facty 5)`, effectively requires evaluating `(facty 4)`, `(facty 3)`, `(facty 2)` and `(facty 1)`. (It may help to recall Fig. 12.1.) Similarly, evaluating `(facty 100)` would involve a sequence of nearly one hundred recursive function calls. For functions such as `facty`, the evaluation of each recursive function call is *suspended* pending the evaluation of all of the subsidiary function calls. Keeping track of all of these suspended evaluations requires storing relevant information somewhere in the computer's memory. Thus, if the value of `n` gets large enough, DrScheme's evaluation of `(facty n)` would eventually cause problems. In particular, at some point, the operating system would refuse to grant DrScheme more memory to hold the needed information.

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

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

Example 14.2.1: Printing Dashes

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

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

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

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

```
> (print-n-dashes 5)
-----
> (print-n-dashes 12)
------------
```

Consider the evaluation of the expression, `(print-n-dashes 5)`*. According to the Default Rule for evaluating non-empty lists, evaluating this list requires applying the* `print-n-dashes` *function to the input value* `5`*. Thus, a function-call box must be set up with a local environment containing an entry for the input parameter,* `n`*, whose value is* `5`*. Next, the body of the function is evaluated. Since* `n` *has the value* `5` *in this function-call box, we are in the recursive case. Thus, the two printing expressions must be evaluated in turn. Recall, too, that the value of the last expression will be the output for this function call. Evaluating the first expression,* `(printf "-")`*, causes a single dash to be displayed in the Interactions Window. Evaluating the second expression,* `(print-n-dashes (- n 1))`*, requires making a recursive function call.*

At this point, we would normally require a new *function-call box to process the recursive application of* `print-n-dashes` *to the value* `4`*. However, we make the following crucial observation:*

⋆ *When the value of the recursive function-call expression,* `(print-n-dashes (- n 1))`*, is known, it will be the output value for the original expression,* `(print-n-dashes 5)`*. Thus, we don't really need the information in the first function-call box anymore. As a result, we can simply re-use the function-call box for the second function call.*

Thus, instead of creating a new function-call box for the application of `print-n-dashes` *to the value* `4`*, DrScheme simply re-uses the function-call box it already has at hand. This will require DrScheme to* erase *the value* `5` *for the local parameter* `n` *and replace it with the value* `4`*, and then proceed to evaluate the body of the function with respect to this new local environment.*

⋆ *You may object that DrScheme is engaged in destructive programming. And you are right! However, that does not have any bearing on the non-destructiveness of the* `print-n-dashes` *function.*

> *The semantics of Scheme stipulates that each recursive function call gets a new function-call box. Thus, according to the semantics of Scheme, the* `print-n-dashes` *function is non-destructive. However, DrScheme is privately re-using a single block of memory, using destructive techniques to perform a sequence of computations that are* equivalent *to those it would have performed if it were using the* non-destructive *techniques. Because DrScheme's use of destructive computation is equivalent to the desired non-destructive computation, this is a* safe *use of destructive computing. Notice, too, that* our hands are clean! *We are writing non-destructive functions!*

To reiterate: From a theoretical viewpoint, the evaluation of tail-recursive function calls is no different from the evaluation of non-tail-recursive function calls: neither is destructive. However, the DrScheme software makes efficient use of memory when evaluating tail-recursive function calls. At a very low-level, this can be construed as destructive; however, our Scheme programs are nonetheless non-destructive!

If I were to ask you to draw a sequence of function-call boxes for all of the expressions, `(print-n-dashes 5)`, `(print-n-dashes 4)`,..., `(print-n-dashes 0)`, *you would probably get tired—especially when you realized that you would lose no information by simply re-using a single function-call box for processing the entire sequence of recursive function calls. That's all that DrScheme is doing when it processes a tail-recursive function call.*

The `print-n-dashes` function is an example of a tail-recursive function. But what exactly do we mean by the term, tail recursive?

Definition 14.1: Tail-recursive function

Suppose that f is a function, $\mathcal{B}$ is its body, and expr *is a recursive function-call expression somewhere within $\mathcal{B}$. We say that* expr *is a tail-recursive function-call expression within $\mathcal{B}$ if, whenever evaluating $\mathcal{B}$ requires evaluating* expr, *it is necessarily the case that the* last *step in evaluating $\mathcal{B}$ is the evaluation of* expr *and, thus, the value of $\mathcal{B}$ 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 $\mathcal{B}$ consists of a single* `cond` *expression. Suppose further that this* `cond` *has only two cases: a base case and a recursive case. The only way that* `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:*[a]

```
(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.*

[a]This is actually the `facty-v2` function, but the same points apply to all versions of the `facty` function seen earlier.

In-Class Problem 14.2.1

Define a function that satisfies the following contract:

```
;;  PRINT-FUNC-VALS
;; ----------------------------------------------------------------
;;  INPUTS:  FUNC, a function that expects a single numerical input
;;           FROM, a starting input
;;           TO, an ending input
;;  OUTPUT:  None
;;  SIDE EFFECT:  Prints the values of FUNC when applied to
;;                the successive inputs from FROM to TO.
```

Tail-recursive functions like `print-n-dashes` do not generate interesting output values; instead, their primary purpose is to display information in the Interactions Window as a side effect. Functions that generate interesting output values can also be tail recursive; however, they typically require one or more additional input parameters. Frequently, those additional input parameters are called *accumulators* because they are used to incrementally accumulate values of interest. Section 14.3 addresses accumulator-based tail-recursive functions.

14.3 Accumulators

In the factorial example, seen earlier, each recursive function call generated an output value that represented a solution to a simpler problem. For example, the evaluation of `(facty 4)` (i.e., 4!) resulted in the recursive function calls, `(facty 3)`, `(facty 2)` and `(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, `sum-to-n`*:*

```
;;  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)~%")
       0)
      ;; Recursive Case:  n > 0
      (#t
       (printf "Recursive Case (n=~s) ...~%" n)
       (+ n (sum-to-n (- n 1)))))))
```

As in prior examples, the `printf` *expressions serve only to display information about the recursive function calls; they do not affect the output value, as illustrated below.*

```
> (sum-to-n 3) ;; compute 0 + 1 + 2 + 3
Recursive Case (n=3) ...
Recursive Case (n=2) ...
Recursive Case (n=1) ...
Base Case (n=0)
6
```

Notice that the evaluation of `(sum-to-n 3)` *involved a sequence of function calls—namely:* `(sum-to-n 3)`, `(sum-to-n 2)`, `(sum-to-n 1)` *and* `(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 + ... + N.
;;           More generally, the output is the value of
;;             ACC + 0 + 1 + 2 + ... + N.

(define sum-to-n-acc
  (lambda (n acc)
    (cond
      ;; Base Case:  n = 0
      ((= n 0)
       (printf "Base Case (n=0, acc=~s)~%" acc)
       ;; Return the accumulator!
       acc)
      ;;Recursive Case:  n > 0
      (#t
       (printf "Recursive Case (n=~s, acc=~s)~%" 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`*:*

```
;;  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)
       (facty-acc (- n 1) (* n acc))))))
```

An expression of the form, `(facty-acc n 1)`*, will evaluate to the factorial of* `n`*. In other words, the initial value of the accumulator must be* `1` *(i.e., the multiplicative identity) for this function to achieve its desired result.*

Notice that the function, `facty-acc`*, is tail recursive, as evidenced by the fact that the recursive function-call expression,* `(facty-acc (- n 1) (* n acc))`*, by itself constitutes the last expression in the recursive case. It is not a subsidiary expression within some larger expression. Thus, the value of the recursive function-call expression is the output value for the original function call-box.*[a]

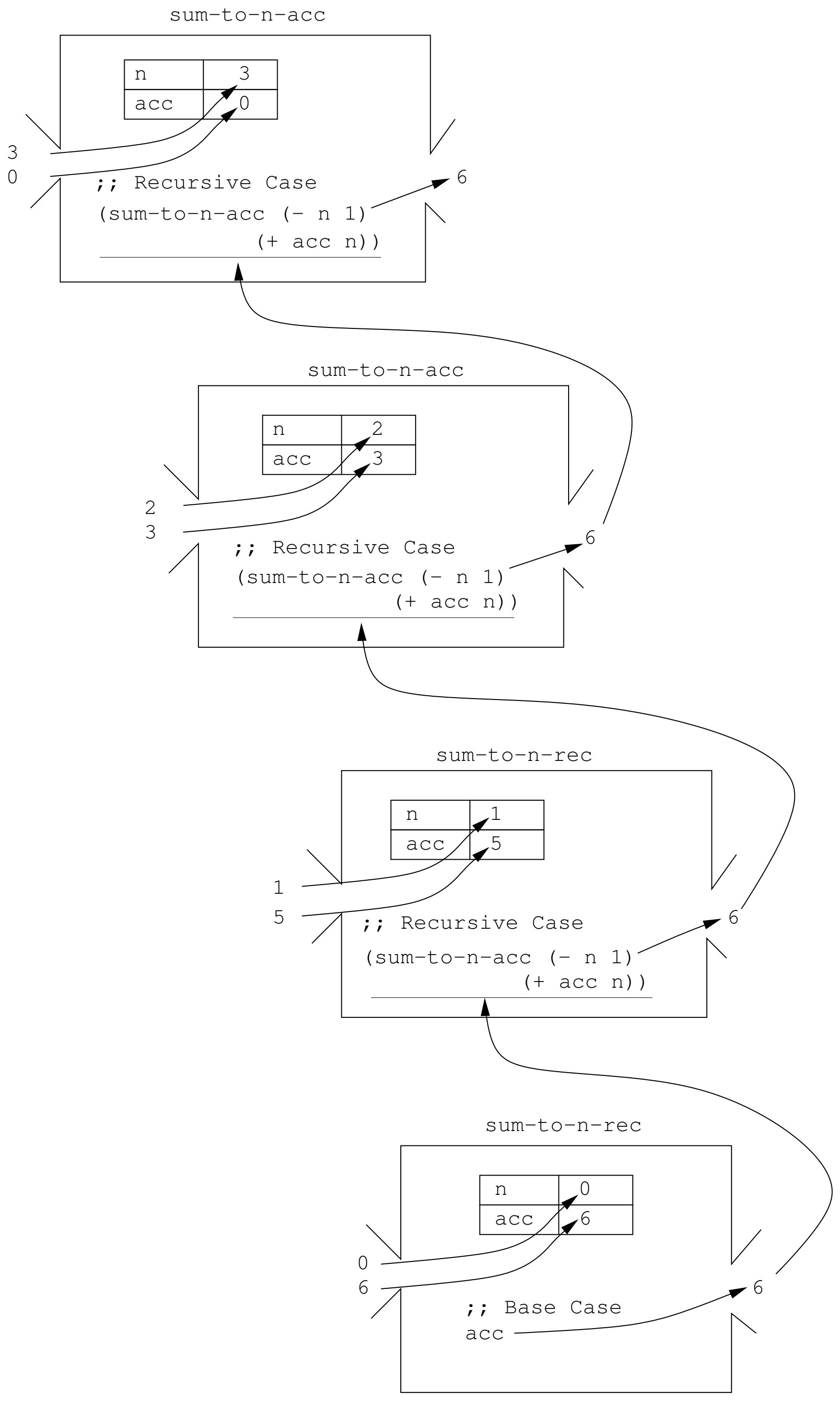


Figure 14.1: DrScheme's evaluation of `(sum-to-n-acc 3 0)`

For `facty-acc`*, the current accumulator,* `acc`*, is* multiplied *by* `n` *to generate the value of the accumulator for the recursive function call. Since* `facty-acc` *involves multiplying the current accumulator to generate the value of the next accumulator, the appropriate initial value for the accumulator is* `1`*. Thus, to use* `facty-acc` *to compute* $4!$*, we would evaluate an expression such as* `(facty-acc 4 1)`*, as illustrated below:*

```
> (facty-acc 4 1)
Recursive Case (n=4, acc=1)
Recursive Case (n=3, acc=4)
Recursive Case (n=2, acc=12)
Base Case (n=1, acc=24)
24
```

Remember that each function call-box includes its own local environment that contains two parameters, `n` *and* `acc`*. The parameters in each call-box may have the same names as the parameters in the other call-boxes; however they are quite distinct. Thus, there are four distinct parameters named* `n`*, having the values* `4`*,* `3`*,* `2` *and* `1`*. Similarly, there are four separate parameters named* `acc`*, having the values* `1`*,* `4`*,* `12` *and* `24`*. Notice that by the time the base case is reached, in the final function call, the accumulator,* `acc`*, has the desired value* `24`*.*

Incidentally, the following description of the output value for the function, `facty-acc`*, is more general, in that it allows the accumulator to have values other than* `1`*:*

$\star$ *The output value for* `(facty-acc n acc)` *is equal to the factorial of* `n` *multiplied by* `acc`*.*

Notice that if `acc` *equals* `1`*, then the output value is indeed* $n!$ *However, if* `acc` *is something other than* `1`*, then the value is* $n! * acc$*.*

[a]In contrast, the non-tail-recursive function, `facty`, seen earlier, included the recursive function-call expression, `(facty (- n 1))`, within the larger expression, `(* n (facty (- n 1)))`.

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*:*

```
;;  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=~s, acc=~s~%" 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, `(sum-squares-acc 3 0)`*, in the Interactions Window:*

```
> (sum-squares-acc 3 0)        ⟵  3² + 2² + 1² + 0² = 14
Recursive Case: n=3, acc=0
Recursive Case: n=2, acc=9
Recursive Case: n=1, acc=13
Base Case: n=0, acc=14
14
```

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,* `(sum-squares-acc n acc)`*, evaluates to* $0^2 + 1^2 + \ldots + n^2 + acc$*.*

For example, when $n = 2$ *and* $acc = 9$*, the result is* $0^2 + 1^2 + 2^2 + 9$ *(i.e.,* 14*). Similarly, when* $n = 0$ *and* $acc = 14$*, the result is* $0^2 + 14$ *(i.e.,* 14*).*

Example 14.3.5: Approximating π

Mathematicians tell us that the value of π *can be approximated using sums of the form shown below:*

$$\pi \;\approx\; 4 \cdot (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots \pm \frac{1}{n})$$

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* `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* `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* `sign` *will alternate between* 1 *and* -1 *and, thus, keeps track of the sign of the current term. The input* `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* `acc` *is used to accumulate the desired sum. Here is the contract:*

```
;;  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)
3.1611986129870506
> (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)
3.1415946535856922
```

Notice how big the input `n` *must be to get even modestly accurate approximations of* π*.*

Example 14.3.6: Approximating e

Mathematicians tell us that the number e *is well approximated by sums of the form*

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

In particular, as the value of n *gets larger, the sum gets closer and closer to the value of* e*. Below, we define a function,* `approx-e-acc`*, that involves several input parameters that can be construed as accumulators. (Sometimes accumulators accumulate really interesting stuff; sometimes they accumulate boring stuff.) For this function:*

- *the input parameter* `n`*, which indicates the last term in the sum, will stay the same across all recursive* *function calls;*

- *the input parameter* `indy` *will take on the values,* $0, 1, 2, \ldots, n$*, 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.,* $1, 1, 2, 6, 24, \ldots, n!$*); and*
- *the input parameter* `acc` *will accumulate the desired sum; it will take on the values* $1, 2, 2.5, 2.66666666666, \ldots$.

```
;;  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=~s, indy=~s, curr-denom=~s, acc=~s~%"
            n indy curr-denom acc)
    (cond
      ;; Base Case:  INDY > N (we're done!)
      ((> indy n)
       ;; Return the accumulator!
       acc)
      ;; Recursive Case:  INDY <= N
      (#t
       ;; Tail-recursive function call with adjusted inputs
       (approx-e-acc
          n                          ;; n doesn't change
          (+ 1 indy)                 ;; increment indy
          (* (+ 1 indy) curr-denom)  ;; update current denom
          (+ acc (/ 1.0 curr-denom)) ;; 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
```

```
n=4, indy=3, curr-denom=6, acc=2.5
n=4, indy=4, curr-denom=24, acc=2.6666666666666665
n=4, indy=5, curr-denom=120, acc=2.708333333333333
2.708333333333333
```

Notice that the `n` *parameter stays fixed at* `4` *across all the recursive function calls. This parameter is used only to signal the end of the sum. The parameter* `indy` *takes on the integer values from* `0` *to* `5`*. It identifies the current term. Thus, when* `indy` *is greater than* `n`*, no further accumulation of terms is necessary. The parameter* `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* `curr-denom` *is simply multiplied by* `(+ indy 1)` *to generate the next factorial. Finally, the parameter* `acc` *accumulates the desired sum. By the time the base case is reached (i.e., when* `indy > n`*), the accumulator holds the desired answer. Thus, the accumulator is simply returned as the output value for this function.*

If the `printf` *expression is commented out, then the function can be used to compute a very close approximation of* `e` *without a lot of excess printing, as demonstrated below:*

```
> (approx-e-acc 20 0 1 0)
2.7182818284590455
```

14.4 Wrapper Functions

One annoying characteristic of accumulator-based functions is that the accumulators need to be given appropriate initial values to ensure the desired results. Fortunately, this problem is easily overcome by providing *wrapper functions.* In this setting, a wrapper function is designed to properly initialize any accumulators so that the user of an accumulator-based function need not remember the appropriate values. This section gives wrapper functions for some of the accumulator-based functions seen earlier.

Example 14.4.1: A wrapper for `facty-acc`

The following defines a wrapper function, `facty-wr`*, for the accumulator-based function,* `facty-acc`*, defined earlier. Notice that the wrapper function simply calls* `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!*

```
;;  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 Interactions Window session demonstrates how the wrapper function shields the user from the accumulator. In fact, the user of `facty-wr` *may not even be aware that an accumulator is being used at all.*

```
> (facty-wr 3)
6
> (facty-wr 4)
24
> (facty-wr 5)
120
```

Example 14.4.2: A wrapper for `approx-pi-acc`

The function, `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,* `approx-pi`*, shields the user from having to know the appropriate initial values for these additional inputs:*

```
;;  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)))
```

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 `approx-e-acc`

The function, `approx-e-acc`*, from Example 14.3.6, involved several accumulators. The following wrapper function,* `approx-e-wr`*, shields the user from having to know the appropriate initial values for these accumulators:*

```
;;  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)))
```

Here's what it looks like in the Interactions Window:

```
> (approx-e-wr 4)
2.708333333333333
> (approx-e-wr 5)
2.7166666666666663
> (approx-e-wr 6)
2.7180555555555554
> (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.)*

```
;;  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!~%")))))

> (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:

```
(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, `(facty (- n 1))`, is embedded within a larger expression, `(* n (facty (- n 1)))`. The evaluation of the larger expression is suspended while waiting for `(facty (- n 1))` to be evaluated. After `(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.

$$\begin{aligned} g(5) &= 1^2 + 2^2 + 3^2 + 4^2 + 5^2 \\ &= (1^2 + 2^2 + 3^2 + 4^2) + 5^2 \\ &= g(4) + 5^2 \end{aligned}$$

In turn, this suggests that $g(n) = g(n-1) + n^2$ for each $n > 1$, which leads to the following solution in 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
      (facty-acc (- n 1) (* n acc))))))
```

Instead of taking the answer returned by the recursive function call and multiplying it by n, this solution uses an extra input, called an *accumulator,* to accumulate the desired answer. The main computations involve determining the values to be fed to the recursive function call—in this case, `(- n 1)` and `(* n acc)`. In the base case, the accumulator is returned as the output value, since it has, by that time, accumulated the desired answer.

Because tail-recursive functions often require extra inputs (e.g., accumulators), it is frequently desirable to provide *wrapper* functions that take care of the annoying job of giving appropriate values to the extra inputs. For example, a wrapper function for the `facty-acc` function would take care of calling `facty-acc` with an initial value of `1` for `acc`.

Built-in Functions Introduced in this Chapter

`even?`:	Returns `#t` if its input is an even number
`odd?`:	Returns `#t` if its input is an odd number
`sin`:	Returns the *sine* of its input
`log`:	Returns the *natural logarithm* of its input

Chapter 15

Local Variables, Local Environments

Recall that, in Scheme, every expression is evaluated with respect to some environment. Up to this point, most of the expressions we have encountered have been evaluated with respect to the Global Environment. For example, expressions entered into the Interactions Window are evaluated with respect to the Global Environment, as are expressions from the Definitions Window when the *Run* button is pressed. Now, when evaluating a symbol with respect to the Global Environment, there is only one place to look for that symbol's value: in the Global Environment.

However, we have also seen that 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 $\mathcal{E}'$ 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 $\mathcal{E}'$.

In the process, whenever any symbol s needs to be evaluated, DrScheme looks for its value, first, in the local environment $\mathcal{E}'$. 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 $\mathcal{E}$ (which may or may not be the Global Environment), the new local environment $\mathcal{E}'$ that it creates is nested inside the parent environment $\mathcal{E}$ (i.e., $\mathcal{E}' \subset \mathcal{E}$). Each of the expressions in the *body* of the `let` special form is evaluated with respect to that new local environment $\mathcal{E}'$. As a result, when any symbol s needs to be evaluated, DrScheme gives priority to the new environment $\mathcal{E}'$. 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:

```
(let ((var1 val1)
      (var2 val2)
       ...
      (varn valn))
  expr1
  expr2
   ...
  exprk)
```

where:

- $var_1, \ldots, var_n$ are character sequences that denote n distinct symbols that will serve as the local variables, where $n \geq 0$;
- $val_1, \ldots, val_n$ are n expressions of any kind, called the *value expressions;* and
- $expr_1, \ldots, expr_k$ are k Scheme expressions of any kind, where $k \geq 1$.

The expressions, $expr_1, \ldots, expr_k$, constitute the *body* of the `let` expression.

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

Example 15.1.1: Some legal `let` expressions

The following expressions are all legal `let` *expressions:*

```
(let () #t)


(let ((x (+ 2 3)))
  (* x x))


(let ((x (+ 2 3))
      (y 3)
      (z (* 2 2)))
  (printf "x: ~s, y: ~s, z: ~s~%" x y z)
  (+ 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 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)`*.*

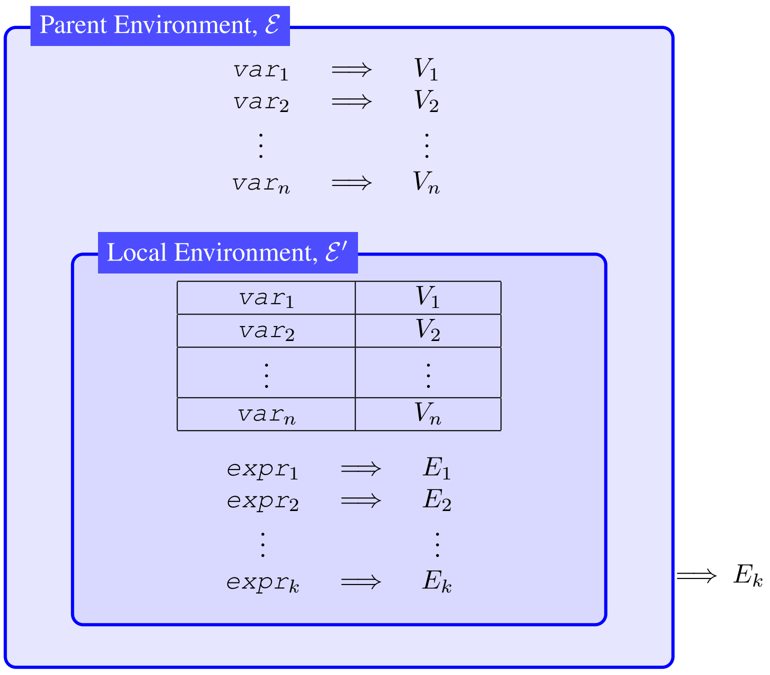

Parent Environment, $\mathcal{E}$
$\mathtt{var}_1 \Longrightarrow V_1$
$\mathtt{var}_2 \Longrightarrow V_2$
$\mathtt{var}_n \Longrightarrow V_n$
Local Environment, $\mathcal{E}'$
$\mathtt{var}_1$ V_1
$\mathtt{var}_2$ V_2
$\mathtt{var}_n$ V_n
$\mathtt{expr}_1 \Longrightarrow E_1$
$\mathtt{expr}_2 \Longrightarrow E_2$
$\mathtt{expr}_k \Longrightarrow E_k$
$\Longrightarrow E_k$

Figure 15.1: Evaluating a `let` special form

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

```
(let ((var₁ val₁)
      (var₂ val₂)
       ...
      (varₙ valₙ))
  expr₁
  expr₂
   ...
  exprₖ)
```

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

(1) The value expressions, $\mathtt{val}_1, \ldots, \mathtt{val}_n$, 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 $\mathtt{var}_i$ 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, $\mathtt{expr}_1, \ldots, \mathtt{expr}_k$, 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 $\mathtt{var}_i$ 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).

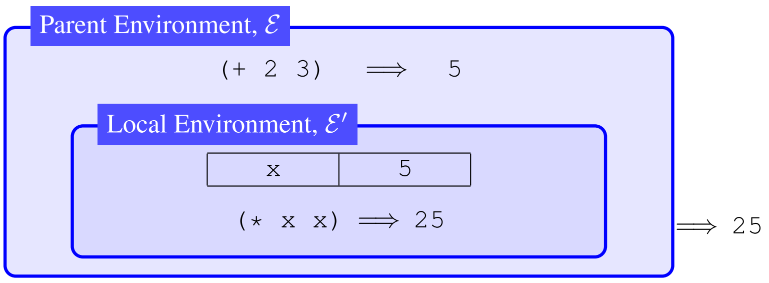

Parent Environment, $\mathcal{E}$
(+ 2 3) ⟹ 5
Local Environment, $\mathcal{E}'$
x
5
(* x x) ⟹ 25
⟹ 25

Figure 15.2: Evaluating the expression, `(let ((x (+ 2 3))) (* x x))`, from Example 15.1.2

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~%" x y z)
    (+ x y z))
x: 5, y: 3, z: 4
12
```

In the first expression, the local environment contains no entries. Thus, when the body of the `let` is evaluated, the result is the same as if it were evaluated outside the `let`. In particular, the expression, `#t`, evaluates to `#t`, which is reported as the value of the entire `let` expression. Since the purpose of a `let` expression is to set up a local environment, it is rare to see a `let` expression that contains no `var`/`val` pairs.

In the second expression, 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)
```

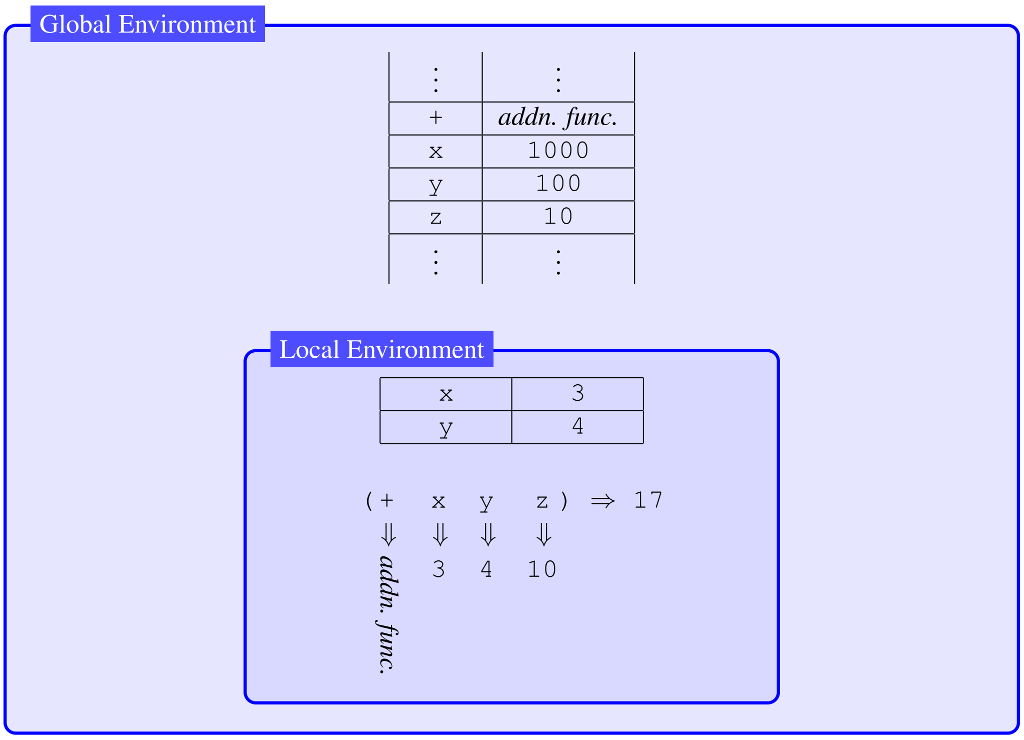


Figure 15.3: Demonstrating the higher priority of the local environment in Example 15.1.4

```
> (define y 100)
> (define z 10)
> (+ x y z)
1110
> (let ((x 3)
        (y 4))
    (+ x y z))
17
```

The first three expressions use the `define` *special form to create three global variables, named* `x`*,* `y` *and* `z`*. The last expression uses a* `let` *to create a local environment containing two local variables, named* `x` *and* `y`*. When the single expression in the body of the* `let` *is evaluated, the values for* `x` *and* `y` *are drawn from the local environment, whereas the values for* `+` *and* `z` *are drawn from the Global Environment. The entries for* `x` *and* `y` *in the Global Environment play no role in the evaluation of the expression* `(+ x y z)` *in the body of the this* `let` *expression, as illustrated in Fig. 15.3.*

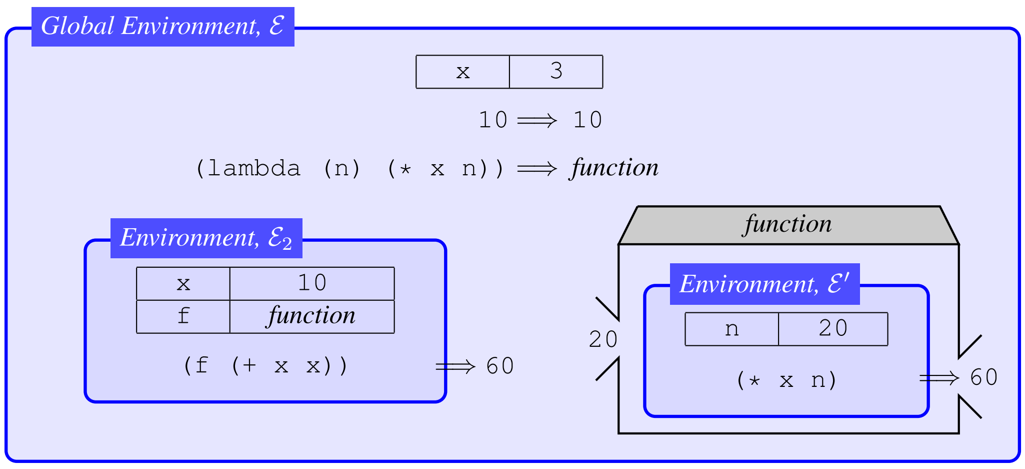
Example 15.1.4: Local Functions

Consider the following interactions:

```
> (define x 3)
> (let ((x 10)
        (f (lambda (n)
             (* x n))))
```

```
         (f (+ x x)))
    60
```

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 $\mathcal{E}_2$ 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 $\mathcal{E}_2$. 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 $\mathcal{E}'$ are nested within the Global Environment—not within $\mathcal{E}_2$. In the environment $\mathcal{E}'$,* `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`.


Global Environment, $\mathcal{E}$
x 3
10 ⟹ 10
(lambda (n) (* x n)) ⟹ function
Environment, $\mathcal{E}_2$
x 10
f function
(f (+ x x)) ⟹ 60
function
Environment, $\mathcal{E}'$
20
n 20
(* x n) ⟹ 60

15.2 Flipping Coins and Tossing Dice

The following example introduces a *destructive* built-in function called `random` that has many uses, one of which is to demonstrate the need for the `let` special form. I know... this part of the book is supposed to only deal with non-destructive functions. But, this one exception is too much fun to postpone any further. 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 `random` *that can be used to generate* pseudo-random *numbers. Unlike all of the functions that we have seen so far in this book, the* `random` *function has the unusual property that successive applications of it to the* same input *can generate different output values! This can happen because the computations it performs to generate its output depend on the values of* secret *global variables that it* destructively *modifies. Yep, it's a* destructive *function! Despite being destructive, it is introduced here for three reasons:*

(1) it is fun;

(2) it can be quite useful when programming games; and

(3) it provides a nice demonstration of the need for the `let` *special form (cf. Example 15.2.3, below).*[a]

The `random` *function satisfies the following contract.*

```
;;  RANDOM
;; ---------------------------------------------------------------
;;  INPUT:   N, a positive integer
;;  OUTPUT:  A pseudo-random number drawn from the set
;;              {0, 1, 2, ..., N-1}
;;  SIDE EFFECTS:  Destructively modifies some secret global
;;    variables that enable it to (possibly) generate a
;;    different output the next time it is called---even if
;;    it is called with the same input!
```

Here are some examples demonstrating its behavior:

```
> (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 $\{0, 1, 2, \ldots, n-1\}$.*

[a]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
;; -----------------------------------------------------------
;;  INPUTS:  None
;;  OUTPUT:  A randomly chosen number, one of: {1,2,3,4,5,6}

(define toss-die
  (lambda ()
    ;; Since (RANDOM 6) generates a number in {0,1,2,3,4,5},
    ;; we must add one to get a number in {1,2,3,4,5,6}.
    (+ 1 (random 6))))
```

Here are some examples of their use:

```
> (flip-coin)
H
> (flip-coin)
T
> (flip-coin)
H
> (toss-die)
3
> (toss-die)
1
> (toss-die)
6
```

⋆ One of the most reliable features of non-destructive programming is that no matter how many times you apply a given function f to the same inputs, you will always get the same output. In other words, non-destructive functions are truly functions, in the mathematical sense. Such functions are sometimes called *pure* functions. In contrast, a destructive function such as `random`, which has the potential to generate a different output every time it is called on the same input, is sometimes called an *impure* function.

⋆ The preceding example demonstrates that a function such as `toss-die`, which makes use of an impure function such as `random`, can itself become impure. In other words, the impurity of `random` can infect the otherwise pure function that calls it.

⋆ Because impure functions can be difficult to debug (i.e., find errors and fix them), introducing impure functions into a program should be done with great care! A good rule of thumb is: Do as much as you can with pure (non-destructive) functions; only introduce impure (destructive) functions when they are absolutely necessary—or, as in this chapter, when they are fun!

Example 15.2.3: Using `let` to store a randomly generated value

The `toss-die` *function is fine, but suppose that you toss a die and want to do several things with the result (e.g., print out the value, print out the square of the value, and so on). The following attempt does not work:*

```
> (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 `(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:*

```
> (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.)*

Index