Introduction to Computer Science via Scheme

Luke Hunsberger

© 2017 Luke Hunsberger
All rights reserved

Spring 2017
# Table of Contents

7.3 Summary ............................................................................................................. 43

8 Predicates .............................................................................................................. 45
8.1 Type-Checker Predicates .................................................................................... 45
8.2 Comparison Predicates ....................................................................................... 47
8.3 Summary .............................................................................................................. 48

9 Defining Functions ................................................................................................. 51
9.1 Defining Functions vs. Applying Them to Inputs .................................................. 51
9.2 The `lambda` Special Form ................................................................................ 52
9.3 The Syntax and Semantics of Lambda Expressions ............................................. 54
9.3.1 The Syntax of a Lambda Expression .............................................................. 54
9.3.2 The Semantics of a Lambda Expression ....................................................... 55
9.3.3 Applying a `lambda` Function to Input Values .............................................. 56
9.4 Summary .............................................................................................................. 60

10 Strings, `printf`, `load`, `Run`, and Comments ...................................................... 63
10.1 Strings ............................................................................................................... 63
10.2 The `printf` Function ....................................................................................... 64
10.3 The `load` Function and the `Run` Button ......................................................... 68
10.4 Comments ......................................................................................................... 69
10.5 Summary .............................................................................................................. 70

11 Conditional Expressions ....................................................................................... 73
11.1 The `if` Special Form ....................................................................................... 73
11.2 The Boolean Operators: `and, or, and not` ......................................................... 77
11.2.1 The `not` Function ...................................................................................... 77
11.2.2 The `and` Special Form ............................................................................ 78
11.2.3 The `or` Special Form .............................................................................. 80
11.3 The `cond` Special Form .................................................................................. 81
11.4 Defining Predicates using Boolean Operators instead of Conditional Expressions .................................................................................................................. 85
11.5 Simplifying Conditional and Boolean Expressions ........................................... 87
11.6 Summary .............................................................................................................. 89

12 Recursion ............................................................................................................... 95
12.1 Defining Recursive Functions in Scheme .......................................................... 95
12.2 Tail Recursion ................................................................................................... 104
12.3 Accumulators .................................................................................................... 113
12.4 Wrapper Functions ............................................................................................ 120
12.5 Summary .............................................................................................................. 121

13 Defining Local Variables with the `let, let* and letrec` Special Forms ............... 129
13.1 The `let` Special Form ...................................................................................... 129
13.1.1 The Syntax of the `let` Special Form .......................................................... 129
13.1.2 The Semantics of the `let` Special Form .................................................... 130
13.1.3 Deriving the `let` Special Form from the `lambda` Special Form .................. 134
13.2 The `let*` Special Form ................................................................................... 141
13.2.1 The Syntax of the `let*` Special Form ....................................................... 141
13.2.2 The Semantics of the `let*` Special Form ................................................ 141
13.2.3 Deriving a Single `let*` Expression from Nested `let` Expressions .......... 142
13.3 The `letrec` Special Form ................................................................................ 144
13.3.1 The Syntax of the `letrec` Special Form ................................................... 145
13.3.2 The Semantics of the `letrec` Special Form ............................................. 145
13.4 Summary .............................................................................................................. 148
Appendices 149

A Guide to Your CS Account 151
A.1 The File System ........................................ 151
A.2 Using Terminal to Explore and Augment the File System ........................................ 152
A.3 Submitting Programming Assignments ........................................ 153
   A.3.1 Printing Text Files ........................................ 155
   A.3.2 Submitting your Files Electronically ........................................ 156

B Labs 157
B.1 Lab 1: Your first CMPU-101 lab session! ........................................ 158
List of Figures

1.1 A function with no side effects (left) vs. a function with side effects (right) ............................................. 8
2.1 The universe of Scheme data, partitioned according to data type ............................................................... 14
4.1 A programmer’s interaction with DrScheme ..................................................................................................... 22
6.1 The evaluation of the list denoted by (+ 2 3) .................................................................................................... 32
7.1 The side effect of define: inserting a new entry into the Global Environment .............................................. 39
9.1 Applying the function corresponding to \((\text{lambda} (x) (* x x))\) to the value 4 ........................................... 56
9.2 Applying the \text{sphere-volume} function to the value 2 ............................................................................ 58
12.1 DrScheme’s evaluation of \text{facty 3} ........................................................................................................... 101
12.2 DrScheme’s evaluation of \text{sum-to-n-acc 3 0} ......................................................................................... 116
A.1 A portion of a typical file system .................................................................................................................. 152
A.2 The organization of files and directories for your work in CMPU-101 ....................................................... 152
B.1 Using the cd command to navigate through a directory tree in the hunsberg account .......................... 160
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 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.

In a similar way, people use a programming language to communicate with a computer. And 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 (i.e., what computations the computer will perform). For most computer programming languages, the constituents of the language, whether they are called expressions, statements or entire programs, are usually 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 {
  
-}

For another example, the following character sequences are legal building blocks of a Scheme program:

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

The semantic rules for a programming language specify what the legal expressions (or statements or programs) in that language mean (i.e., what computations the computer will perform in response). For example, the semantic rules of Java stipulate that the legal statement, int x = 5;, results in the computer creating space for a variable named x whose value will be the integer five. Similarly, the semantic rules of Scheme stipulate that the legal expression, (define x 5), results in the computer creating a new variable named x whose value will be the integer five.
Although people can effectively communicate using the English 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 the computer is performing.

To enable us to enter the world of programming as quickly and painlessly as possible, it is helpful to use a programming language for which the syntax and semantic rules 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 computer language in the future.

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

**Functions**

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

In certain cases, we may also consider functions that generate side effects, as illustrated on the righthand side of Fig. 1.1. An example of a harmless, but very useful side effect is that of causing information to be displayed onscreen. For example, the above-mentioned function might not only compute the sum of the scores on a given scoresheet, but also have the side effect of displaying the contents of that scoresheet on a computer screen.

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

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

Non-Destructive Programming in Scheme
Chapter 2

Primitive Data Expressions

In our daily lives, we frequently use character sequences to denote both concrete and abstract data. For example, the character sequence dog can be used to denote a dog. Similarly, the character sequence 34 can be used to denote the number thirty-four. Of course, this book itself consists of a bunch of character sequences that denote all sorts of things. Well, actually, it is a piece of paper with ink marks on it. The ink marks represent characters which, 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 and what they mean.

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.

⋆ Each Scheme program is a sequence of characters.

⋆ The building blocks of a Scheme program are (typically much shorter) character sequences called 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 semantic rules of Scheme tell us which datum each legal expression denotes. For example, in Scheme, the legal expressions 3, #t and () respectively denote the number three, the truth value true, and the empty list. (More will be said about truth values and lists later on.)

Although Scheme expressions can be more complicated, it makes sense to start with simpler ones. Thus, we begin with primitive data expressions. Each primitive data expression denotes a Scheme datum of a particular kind. As illustrated in Fig. 2.1, the universe of Scheme data is populated by numbers, truth values (called booleans), symbols and primitive functions, among many others. Importantly, each datum has a unique data type. For example, a Scheme datum might be a number or a symbol, but cannot be both. Stated differently, the universe of Scheme data is partitioned according to data type. Each section below addresses a different type of primitive data.

⋆ 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.

2.1 Numbers

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

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

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

```
Character Sequence → Datum
```

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

```
3 → the number three
-44 → the number negative forty-four
85/6 → 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`.

```
0 → the number zero
000 → the number zero
000000 → the number zero
```

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

### 2.2 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 → the true truth value
#f → 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.

## 2.3 The Empty List (or Null)

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

() → the empty list

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

## 2.4 Symbols

Another kind of primitive data in Scheme is a symbol. Symbols are frequently used as variables in Scheme programs. To explicitly write down all of the syntax rules specifying which character sequences are legal symbol expressions is not necessary. For our purposes, it suffices to say that practically any sequence of letters, whether lower-case, upper-case or a mixture of the two, is a legal symbol expression in Scheme. For example, hello, goodBye and gasMileage are legal symbol expressions in Scheme. In addition, any character sequence consisting of letters and hyphens is a legal symbol expression in Scheme—as long as it begins with a letter! For example, brave-new-world, gas-mileage and xyz-prq-abc are legal symbol expressions in Scheme. Finally, commonly used one-character expressions, such as *, +, - and /, also constitute legal symbol expressions in Scheme.

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

hello → the symbol hello
* → the asterisk symbol

Again, it is important to keep in mind the difference between the typewritten character sequences (e.g., hello and bye-bye) and the symbols (i.e., the Scheme data) that they denote (e.g., the symbol hello and the symbol bye-bye). This distinction is hard to write down because we use symbols to denote character sequences, and we also use symbols to denote the symbols denoted by character sequences.)

## 2.5 Summary

This chapter introduced the syntax and semantics for a variety of types of primitive data: numbers, booleans, the empty list, and symbols. Examples of legal syntax for these kinds of data are given below.

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

For each legal expression (i.e., piece of syntax), the semantics specifies the datum denoted that expression. This book uses a single arrow (→) to represent denotation. For example, the fact that the character sequence 34 denotes the number thirty-four is represented by: 34 → the number thirty-four.
Chapter 3

Evaluation

We’ve 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 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 briefly.

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

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

* In most cases, the application of the evaluation function to a Scheme datum does not directly generate any side effects. However, there are some important exceptions which shall be highlighted as they are encountered—in Chapters 7, ?? and ??.

3.1 Applying the *Evaluation Function* to Numbers, Booleans, or the Empty List

The evaluation function acts like the *identity function* when applied to numbers, booleans or the *empty list*, as illustrated below.
Since drawing all of these black boxes takes up so much space, from now on we’ll use a simpler, text-based notation to represent the application of the evaluation function to some datum, as illustrated below.

\[
\text{Input Datum} \quad \Rightarrow \quad \text{Output Datum}
\]

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

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

Here are some more examples illustrating the trivial behavior of the evaluation function when applied to numbers, booleans or the empty list:

\[
\begin{align*}
\text{the number zero} & \Rightarrow \text{the number zero} \\
\text{the boolean true} & \Rightarrow \text{the boolean true} \\
\text{the empty list} & \Rightarrow \text{the empty list}
\end{align*}
\]

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 most 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 and the empty list. In particular, symbols typically do not evaluate to themselves; instead, they evaluate to the value associated with them. (Keep reading!)

The evaluation of a symbol is based on table lookup. In particular, the evaluation function may be thought of as having a private table (or little black book) called the Global Environment.\(^1\) The Global Environment contains a bunch of entries. Each entry pairs a symbol (which is a Scheme datum) with its corresponding value (which also is a Scheme datum). To evaluate a symbol, the evaluation function simply looks up the value associated with that symbol in the Global Environment (i.e., in its little black book). For example, if the Global Environment contains an entry that associates the number two with the symbol \(xyz\), then the result of applying the evaluation function to the symbol \(xyz\) will be the number two:

\[
\text{the symbol } xyz \quad \Rightarrow \quad \text{the number two}
\]

The Scheme datum associated with a symbol in the Global Environment can be of any type. Thus, it might be that the boolean \(true\) is associated with the symbol \(pq\). Similarly, the empty list might be associated with the symbol \(my\text{-empty-list}\).

---

\(^1\)Since the Global Environment is a private appendage of the evaluation function, it is not an official Scheme datum and, thus, is not available for direct inspection.
the symbol \( pq \) \( \implies \) the boolean \( true \)

the symbol \( my\text{-}empty\text{-}list \) \( \implies \) the empty list

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

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

On the other hand, if a symbol does not have a corresponding entry in the Global Environment, then it is not possible to evaluate that symbol. In other words, the result of applying the evaluation function to a symbol having no entry in the Global Environment is undefined. A little later on, we’ll see how to insert new entries into the Global Environment, thereby enabling us to create and use variables of our own.

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 (usually 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, and the empty list evaluate to themselves (i.e., the evaluation function works like the identity function for instances of those data types). However, a symbol is evaluated differently: by looking for a corresponding entry in the Global Environment.

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

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

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 (usually different) Scheme datum.

For example:

\[
\times \implies \text{ the symbol } \ x \implies \text{ the number } \ eighty\text{-}six
\]
Chapter 4

Introduction to DrScheme

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

- Enter a typewritten character sequence into the Interactions Window (the lower window-pane in DrScheme’s window).
- The datum denoted by that character sequence is evaluated (i.e., fed into the evaluation function as input), generating an 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 we type into the Interactions Window; and those that DrScheme displays to us in response. We never get to “touch” the Scheme data denoted by our character sequences. (What would it mean to touch a number anyway?) For this reason, it is extremely important that we maintain an accurate mental model of what’s going on in that simulated world. In other words, we need to have an accurate understanding of Scheme’s computational model.

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

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

This process is illustrated below.

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

1The DrScheme software is freely available from drscheme.org.
4.1 Some Sample Interactions

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 manipulated by the Scheme computer. For example, the following interactions demonstrate that numbers, booleans and the empty list all evaluate to themselves:

```
> 3
3
> #t
#t
> ()
()
```

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 following line is that generated by DrScheme in response. Thus, the above example shows three 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:

```
3 → [ the number three ⇒ the number three ] → 3
#t → [ the boolean true ⇒ the boolean true ] → #t
() → [ the empty list ⇒ the empty list ] → ()
```

Furthermore, we can confirm that several different character sequences can be used to denote the number zero:

```
> 0
0
> 000
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.

To generate more interesting examples, we need a few more building blocks.

### 4.2 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 (usually different) character sequence. However, something very important happens in between:

1. the input character sequence \( C_{\text{in}} \) denotes some Scheme datum \( S_{\text{in}} \);
2. DrScheme evaluates \( S_{\text{in}} \), yielding some datum \( S_{\text{out}} \); and
3. DrScheme displays a character sequence representing \( S_{\text{out}} \).

This process is concisely summarized by:

\[
C_{\text{in}} \rightarrow [ S_{\text{in}} \Rightarrow S_{\text{out}} ] \rightarrow C_{\text{out}}
\]

where the stuff between the square brackets is invisible to us. Since such important computations are happening behind the scenes, it is important that we, as programmers, have an accurate mental model of what Scheme is doing.
Chapter 5

Primitive (Built-in) Functions

For convenience, Scheme includes a variety of primitive (or built-in) functions. Examples include the addition function, the subtraction function, and the multiplication function.

* Each primitive function is a Scheme datum, just like numbers and booleans.

In view of this, you might be wondering what character sequences in Scheme denote primitive functions. That is a legitimate question. However, the answer may surprise you:

* There are no Scheme expressions that denote primitive Scheme functions!

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

* For each built-in function, there is an entry in the Global Environment that associates that function with some symbol. Therefore, the evaluation of that symbol can be used to gain access to the corresponding function.

5.1 Built-in Functions for Arithmetic

For example, the Global Environment contains entries such that each of the following evaluations holds:

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

Thus, a Scheme programmer can refer to each primitive function indirectly, by specifying its name. That these entries do indeed exist in the Global Environment can be confirmed by DrScheme, as illustrated below:

```scheme
> +
#<procedure:+>

> -
#<procedure:->

> *
#<procedure:*>

> /
#<procedure:/>
```

The behind-the-scenes work involved in these interactions can be summarized as follows:
+ → [ the + symbol ⇒ the addition function ] → #<procedure:+>
- → [ the - symbol ⇒ the subtraction function ] → #<procedure:->
* → [ the * symbol ⇒ the multiplication function ] → #<procedure:*>
/ → [ the / symbol ⇒ the division function ] → #<procedure:/>

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 (a.k.a. a procedure)—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.

<table>
<thead>
<tr>
<th>Example 5.2.1: Contracts for some built-in functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name:</strong> +</td>
</tr>
<tr>
<td><strong>Inputs:</strong> $x_1, x_2, \ldots, x_n$; any number of numerical inputs</td>
</tr>
<tr>
<td><strong>Output:</strong> The sum, $x_1 + x_2 + \ldots + x_n$</td>
</tr>
<tr>
<td><strong>Side Effects:</strong> None</td>
</tr>
</tbody>
</table>

Notice that the contract describes what the output value should be, but it does not go into the underlying details about how that output value is actually computed. Similar remarks apply to the following contract for the built-in subtraction function:

<table>
<thead>
<tr>
<th>Name: -</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs: $x_1, x_2, \ldots, x_n$; any number of numerical inputs</td>
</tr>
<tr>
<td>Output: The value, $x_1 - x_2 - x_3 \ldots - x_n$</td>
</tr>
<tr>
<td>Side Effects: None</td>
</tr>
</tbody>
</table>

* 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 not to have any.

Later on, we’ll discuss other entries that populate the Global Environment. We’ll also learn how to insert new entries into the Global Environment. Once we discover how to create Scheme functions of our own design, this will enable us to give our new functions names, simply by placing appropriate entries into the Global Environment. Although we now know that DrScheme provides a variety of built-in functions, we shall have to wait until the next chapter to see how to apply these functions to inputs (i.e., to make them do something).

### 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: \(\text{quotient}\) provides the answer; \(\text{remainder}\) provides the remainder. The contracts for these functions are given below:

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

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

### 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 and the empty list, Scheme 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! A demonstration of this fact will be given in the next chapter.

### 5.5 The `void` Datum and the Built-in `void` Function

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 \(f\) 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.

Although the `void` datum is a primitive datum, there is no corresponding primitive data expression that we can type into the Interactions Window that denotes the `void` datum. However, there is a built-in function, called `void`, that generates the `void` datum as its output. Here is its contract:

```
;; VOID -- built-in
;; -------------------------------
;; INPUTS: None
;; OUTPUT: The void datum (representing "no value")
;; 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:

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

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. In addition, there is a built-in function called `void` whose output is the one-and-only `void` datum.

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

| Basic Arithmetic: | +, -, *, / |
| Integer Division: | `quotient`, `remainder` |
| Evaluation Function: | `eval` |
| Generating the `void` datum: | `void` |

**Problems**

**Problem 5.1**

Each of the following Scheme expressions denotes/represents some kind of Scheme datum. For each, state the data type (e.g., `number`, `boolean`, `symbol`, `list`, `function`, etc.) of the Scheme datum it represents. In addition, specify the data type of the Scheme datum it evaluates to. The first one is done for you as an illustration.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Represents a datum of this type</th>
<th>Evaluates to a datum of this type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#t</code></td>
<td><code>boolean</code></td>
<td><code>boolean</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>/</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>84</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>3/5</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>void</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6

Non-Empty Lists

Previously, we have only seen examples of primitive data—namely, numbers, booleans, the empty list, 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, this chapter presents an example of non-primitive data—that is, data that does have parts that we, as Scheme programmers, can access. In particular, this chapter presents non-empty lists.

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

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

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

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

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

6.1 The Syntax and Semantics for Non-Empty Lists

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

- `(3 4)` → a list containing the number three and the number four
- `( + 3 4 )` → a list containing the + symbol, the number three, and the number four
- `(eval (+ 3 4))` → 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

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

(i.e., each $E_i$ is a Scheme expression that denotes a Scheme datum, $D_i$), then the character sequence

$$(E_1 E_2 \ldots E_n)$$

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

For example, the character sequences $+,$ 3 and 4 are legal Scheme expressions that respectively denote the $+$ symbol, the number 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 Case

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.

<table>
<thead>
<tr>
<th>Example 6.2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>We begin with some examples that confirm that something new is happening when DrScheme evaluates non-empty lists.</td>
</tr>
</tbody>
</table>

```
> (+ 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 first step in evaluating this list 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. This first step is summarized below:

\[\begin{align*}
\text{the + symbol} & \implies \text{the addition function} \\
\text{the number two} & \implies \text{the number two} \\
\text{the number three} & \implies \text{the number three}
\end{align*}\]

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:

\[\begin{align*}
\text{the addition function} & \\
\text{the number two} & \\
\text{the number three} & \implies \text{the number five}
\end{align*}\]

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

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

where the evaluation step is explained by:

**First Step of Default Rule:**

\[\begin{align*}
+ \text{ symbol} & \implies \text{addition function} \\
\text{number two} & \implies \text{number two} \\
\text{number three} & \implies \text{number three}
\end{align*}\]

**Second Step of Default Rule:**

\[\text{addition function applied to two and three yields output of five}\]

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 only has two steps, and they are always the same. Second, it can be used on arbitrarily complex lists without requiring any modifications. For example, recall the interaction:

\[
> (+ 2 (* 3 10))
\]

32
If we follow the rules we already know, we will see that nothing new is needed to explain this interaction.
First, the character sequence \((+ \ 2 \ (* \ 3 \ 10))\) is a legal Scheme expression that denotes a list. The
denoted list contains three items: the \(+\) symbol, the number two, and a subsidiary list. The subsidiary list
contains three items: the \(*\) symbol, the number three, and the number ten. (You should convince yourself
of all of this before proceeding.) Okay, so far so good: we have seen that our input expression denotes a
particular list. That list, which happens to be a list of lists, shall be the Input Datum for the evaluation
function.
To evaluate this list, we need to use the Default Rule. The first step of the Default Rule requires us to
evaluate each item in the list:
\[
\begin{align*}
\text{the } + \text{ symbol} & \Rightarrow \text{ the addition function} \\
\text{the number two} & \Rightarrow \text{ the number two} \\
\text{the subsidiary list} & \Rightarrow \text{ oops!}
\end{align*}
\]
Before we can complete the first step of the Default Rule, we must evaluate the subsidiary list (i.e., the list
containing the \(*\) symbol, the number three, and the number ten. Okay, so we pause for a moment and then
proceed.
To evaluate the subsidiary list, we need to use the Default Rule. The first step of the Default Rule requires
us to evaluate each item in the list:
\[
\begin{align*}
\text{the } * \text{ symbol} & \Rightarrow \text{ the multiplication function} \\
\text{the number three} & \Rightarrow \text{ the number three} \\
\text{the number ten} & \Rightarrow \text{ the number ten}
\end{align*}
\]
The second step of the Default Rule requires us to apply the first item (i.e., the function) to the rest of the
items. In other words, we need to apply the multiplication function to the numbers three and ten. The result
is the number thirty.
Now that we know that the subsidiary list evaluates to thirty, we can pick up from where we left off when
evaluating the original list. The first step of the Default Rule (for evaluating the original list) requires us
to evaluate each item in the list:
\[
\begin{align*}
\text{the } + \text{ symbol} & \Rightarrow \text{ the addition function} \\
\text{the number two} & \Rightarrow \text{ the number two} \\
\text{the subsidiary list} & \Rightarrow \text{ oops!}
\end{align*}
\]
The second step of the Default Rule then requires us to apply the first item (i.e., the addition function) to the rest of the items (i.e., the numbers two and thirty). The result is the number thirty-two. And that is the Output Datum that results from evaluating the original list! Phew! Of course, DrScheme reports this result using the character sequence 32.

6.2.1 A More Formal Description of the Default Rule

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

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

- 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, then DrScheme will report an error.) The resulting output will be some datum, \( P \).

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

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 arguments \( K_2, \ldots, K_n \) might be undefined because, for example, the function expects numbers and it gets something else. In any of these cases, the result is undefined and DrScheme would report an error. Thus, none of the following lists can be evaluated:

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

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

**Example 6.2.4**

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

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

These Scheme data evaluate to the following:

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

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

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

** When using the Default Rule to evaluate a non-empty list, the only side effects that can be generated are those generated by the function that is applied in Step Two. If the function applied in Step Two has no side effects, then neither will the evaluation of the non-empty list. In other words, the Default Rule does not directly generate any side effects, but Step Two might indirectly lead to some side effects.

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**

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

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

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

- eval symbol $\Rightarrow$ the evaluation function
- + symbol $\Rightarrow$ the addition function

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

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

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

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:

\[
> (+ 3 (* 4 10)) \\
73
\]

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.)

Problems

<table>
<thead>
<tr>
<th>Problem 6.1</th>
</tr>
</thead>
</table>

Each of the following Scheme expressions denotes/represents some kind of Scheme datum. For each, state the data type (e.g., number, boolean, symbol, list, function, etc.) of the Scheme datum it represents. In addition, specify the data type of the Scheme datum it evaluates to. The first one is done for you as an illustration.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Represents a datum of this type</th>
<th>Evaluates to a datum of this type</th>
</tr>
</thead>
<tbody>
<tr>
<td>#t</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>(* 4 6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(eval 3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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. And, once you get the hang of it, learning the syntax and semantics for each new kind of special form gets easier and easier.

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

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

The define special form is signaled by the define keyword.

7.1.1 The Syntax of the define Special Form

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

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

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

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

Therefore:

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

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

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

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

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

7.1.2 The Semantics of the define Special Form

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

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

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 5.5 that the void datum is used to represent “no value”, and that DrScheme does not display anything in the Interactions Window when void is the result.)

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, \([ s \quad \text{void}]\), 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 \( E: \quad e \Rightarrow E \).
3. Insert \( E \) as the value for \( s \) in the Global Environment: \([ s \quad E] \).
Input Datum
List containing:
define symbol
The s symbol
The datum e

Evaluation Function

Global Environmentsymbol value

side effect

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

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

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

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

\begin{verbatim}
> (define x 6)
> (define y 3)
> (define z 34)
> 
\end{verbatim}

Of course, something \textit{has} happened!

\begin{exam}

\textit{Typing the character sequence, \texttt{(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 \texttt{x} in the Global Environment, as illustrated below.}

\begin{align*}
\texttt{x} & \quad \rightarrow \quad \text{the symbol } \texttt{x} \\
\texttt{(+ 1 2 3)} & \quad \rightarrow \quad [\text{a list containing the + symbol and the numbers one, two and three}] \quad \Rightarrow \quad \text{the number six}
\end{align*}

\textbf{Side Effect: New Global Environment Entry:}  \quad \begin{tabular}{|c|c|}
\hline
the symbol & \texttt{x} \\
\hline
the number & \texttt{six} \\
\hline
\end{tabular}

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

\begin{verbatim}
> x
\end{verbatim}

\end{exam}
Notice that the first attempt to evaluate the symbol \( x \) resulted in an error; however, after the \texttt{define} special form, attempts to evaluate \( x \) result in 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.

\begin{quote}
\textbf{Example 7.1.2: Confirming the semantics of \texttt{define}}
\end{quote}\\
The following admittedly unusual interactions confirm the semantics of the \texttt{define} special form.

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

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

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

Of course, in this case, the third step is redundant, since \( \texttt{void} \) is already there as the value for \( w \).

Afterward, when we ask DrScheme to evaluate \( w \), it does so, coming up with the answer \( \texttt{void} \). However, since \( \texttt{void} \) 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 \texttt{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 \textit{Run} button to erase what you’ve done and thereby restore that symbol’s status as a keyword.
7.2 The quote Special Form

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

7.2.1 The Syntax of the quote Special Form

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

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

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

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

7.2.2 The Semantics of the quote Special Form

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

If: \( \mathcal{C} \rightarrow D \)

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

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

\star \{ A list containing the quote symbol and } D \text{ evaluates to } D .

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

Example 7.2.1

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

> (quote x)
\( x \)

> (quote (1 2 3))
\( (1 2 3) \)

> (quote (+ 2 3))
```

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

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

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

In the second example, \((\text{quote } (1 \ 2 \ 3))\) denotes a list containing the \text{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 \text{quote} special form comes in handy for such cases.

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

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

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

### Example 7.2.2

Notice the difference between the evaluations of \(x\) and \((\text{quote } x)\) below:

\[
\begin{align*}
> & \ (\text{define } x \ (+ \ 1 \ 2 \ 3)) \\
> & \ x \\
> & \ 6 \\
> & \ (\text{quote } x) \\
> & \ x
\end{align*}
\]

### Example 7.2.3

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

\[
\begin{align*}
> & \ (\text{define } \text{my-list} \ (\text{quote } (1 \ 2 \ 3))) \\
> & \ \text{my-list} \\
> & \ (1 \ 2 \ 3)
\end{align*}
\]

### 7.2.3 Alternate Syntax for \text{quote} Special Forms

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

**Example 7.2.4**

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

```
> (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. Notice that 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.

```
> (quote (quote x))
’x
> ’’x
’x
```

### 7.3 Summary

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

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

* The quote special form is used to shield a datum from evaluation; it has no side effects.

The define special form enables us to use symbols as variables (i.e., names for pieces of data). Later on, when you create functions of your own design, you will typically use the define special form to give them names. In turn, this will enable you to apply your new functions to any desired inputs simply by asking DrScheme (and the Default Rule) to evaluate an appropriate non-empty list.

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

Problems

**Problem 7.1**

When the Default Rule is used to evaluate a non-empty list, the first step is to evaluate each element in the list. However, special forms are not evaluated by the Default Rule. As a result, it can happen that some of the elements in a special form are evaluated, while others are not. For this problem, summarize the following information about the `define` and `quote` special forms:

1. how many input elements there are—not including the keyword symbol;
2. which input elements get evaluated and which do not;
3. whether there is an output value and, if so, how it is computed; and
4. whether there is a side effect and, if so, what it is.

**Problem 7.2**

For each statement below, decide which of the words in parentheses apply:

- Evaluation of a `define` special form (always, never, sometimes) causes a side effect.
- Evaluation of a `quote` special form (always, never, sometimes) causes a side effect.
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 a compound data type called list. For each one of these data types, Scheme includes a primitive function called a type-checker predicate. When a type-checker predicate is applied to some Scheme datum, it outputs true if that datum belongs to the indicated data type; otherwise, it outputs false. Thus, the type-checker predicate associated with the number data type outputs true whenever the input belongs to the number data type. Similarly, the type-checker predicate associated with the list data type outputs true whenever the input datum belongs to the list data type. And so on.

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

Example 8.1.1

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

```scheme
> number?
 #<procedure:number?>
> symbol?
 #<procedure:symbol?>
> boolean?
 #<procedure:boolean?>
> list?
 #<procedure:list?>
> null?
 #<procedure:null?>
> procedure?
 #<procedure:procedure?>
> void?
 #<procedure:void?>
```
Notice that the symbols mirror the names of the corresponding data types, except that the symbol associated with the type-checker predicate for functions is `procedure?`, not `function?`.\footnote{This text uses `function` and `procedure` interchangeably; however, the term `function` seems better suited given that Scheme is typically referred to as a \textit{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 \texttt{true} if that input datum is of the appropriate data type.

\begin{example}
Here’s a contract for the built-in `number?` type-checker predicate:

\begin{itemize}
  \item \textbf{Name:} \texttt{number?}
  \item \textbf{Input:} \texttt{d}, any Scheme datum
  \item \textbf{Output:} \texttt{#t} if \texttt{d} is a number; otherwise, \texttt{#f}
\end{itemize}

The contracts for the other type-checker predicates are similar.
\end{example}

\begin{example}
The following Interactions Window session illustrates the behavior of the type-checker predicates.

\begin{verbatim}
> (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
\end{verbatim}
\end{example}
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 \( \text{procedure? } + \) evaluates to the addition function, whereas the ‘+ expression in \( \text{procedure? } ‘+ \) 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, \( \text{void} \) evaluates to void, whereas 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.\(^1\) To enable us to refer to such predicates, each is associated with a particular symbol in the Global Environment.

\[
\begin{align*}
> & \text{ greater than} \\
>= & \text{ greater than or equal to} \\
= & \text{ equal to} \\
< & \text{ less than} \\
<= & \text{ less than or equal to}
\end{align*}
\]

Each of these predicates, when applied to two numeric inputs, generates the expected boolean output, as illustrated below.\(^2\)

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

DrScheme also provides a comparison predicate called \( \text{eq?} \) that is more general that the \( = \) predicate. Whereas the \( = \) predicate only works on numerical input, the \( \text{eq?} \) 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 \( \text{eq?} \) predicate.

- **Name:** \( \text{eq?} \)
- **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.

---

\(^1\)In other contexts, these predicates are commonly called relational operators.

\(^2\)These predicates can also be applied to more than two inputs; however, we shall postpone discussion of such things until Chapter 12.
Example 8.2.2

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

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

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

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

```scheme
> (number? 3)
#t
> (number? '(a 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?.
Comparison Predicates: <, <=, =, >=, > (these work only on numbers).
eq? (this works on numbers, booleans, symbols or the empty list).

Problems

Problem 8.1

Each of the following Scheme expressions denotes/represents some kind of Scheme datum. For each, state the data type (e.g., number, boolean, symbol, list, function, etc.) of the Scheme datum it represents. In addition, specify the data type of the Scheme datum it evaluates to. The first one is done for you as an illustration.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Represents a datum of this type</th>
<th>Evaluates to a datum of this type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*) 4 6</td>
<td>list</td>
<td>number</td>
</tr>
<tr>
<td>'cs101</td>
<td>list</td>
<td>number</td>
</tr>
<tr>
<td>(&gt; 4 3)</td>
<td>number</td>
<td>number</td>
</tr>
<tr>
<td>(number? 'x)</td>
<td>symbol</td>
<td>symbol</td>
</tr>
<tr>
<td>(symbol? 'x)</td>
<td>symbol</td>
<td>symbol</td>
</tr>
<tr>
<td>(symbol? eval)</td>
<td>symbol</td>
<td>symbol</td>
</tr>
</tbody>
</table>

Problem 8.2

Write down a contract for the built-in >= function.

Problem 8.3

Explain the steps taken by DrScheme in the following interactions:

```
> +
#<procedure:+>
> (define addn +)
> (addn 4 5)
9
> (define function? procedure?)
> (function? +)
#t
```

Problem 8.4

Explain the output values generated by the following sequence of interactions.

```
> (define myvar 'sunday)
> (define yourvar 'monday)
> (eq? myvar 'sunday)
#t
> (eq? myvar myvar)
```
#t
> (eq? myvar 'myvar)
#f
> (eq? myvar yourvar)
#f
> (eq? yourvar 'monday)
#t
> (eq? yourvar 'sunday)
#f
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, \( x \), which is used to refer to potential input values. In addition, the expression, \( x^2 \), on the righthand side of the equation indicates how to compute the corresponding output value for any given value of \( x \). (The expression on the righthand side is sometimes referred to as the body of the function.) For example, if we wanted to know the output value generated by \( f \) when given 3 as its input, we could get the answer by first substituting the value 3 for \( x \) in the expression, \( x^2 \), yielding \( 3^2 \). Evaluating the expression, \( 3^2 \), would then yield the desired output value, 9. Similarly, if we wanted to know the output value generated by \( f \) when given the input value 4, we would first substitute the value 4 for \( x \) in the expression, \( x^2 \), yielding \( 4^2 \), which evaluates to 16.

Example 9.1.2

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

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

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

**Example 9.1.3**

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

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

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

### 9.2 The lambda Special Form

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

* The use of the lambda symbol in a lambda special form comes from 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 defines everything about a function except its name.

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

Recall the mathematical definition of the squaring function:

$$f(x) = x^2$$

This mathematical definition does three things:

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

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

```
(lambda (x) (* x x))
```

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

\[
> (\lambda \ (x) \ (\times \ x))
#<procedure>
\]

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

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

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

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

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

\[
> \text{square}
#<procedure:square>
\]

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

\[
> (\text{square} \ 3)
9
> (\text{square} \ 4)
16
> (\text{square} \ -8)
64
\]

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

Example 9.2.2

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

\[
> ((\lambda \ (x) \ (\times \ x)) \ 4)
16
\]

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:

\[
> \text{(define rect-area (lambda (w h) (* w h)))}
> \text{(rect-area 2 3)}
6
> \text{(rect-area 3 8)}
24
> \text{(define cone-volume (lambda (r h) (* 1/3 3.14159 r r h)))}
> \text{(cone-volume 3 2)}
18.849539999999998
> \text{(cone-volume 10 1)}
104.71966666666665
\]

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

---

### 9.3 The Syntax and Semantics of Lambda Expressions

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

#### 9.3.1 The Syntax of a Lambda Expression

A lambda expression has the following syntax:

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

where:

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

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

#### Example 9.3.1

The following are examples of well-formed lambda expressions:

- \((\text{lambda} () 44)\)
- \((\text{lambda} (x) (* x x))\)
• (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).

Assuming that
• each $C_i$ denotes a Scheme symbol, $s_i$;
• the symbols, $s_1, s_2, ..., s_n$, are distinct; and
• $B$ denotes some Scheme datum $D$,

then a lambda expression of the form

$$(\text{lambda } (C_1 \ C_2 \ ... \ C_n) \ B)$$

denotes a Scheme list whose elements are as follows:

• the lambda symbol;
• a list containing $n$ distinct symbols, $s_1, s_2, ..., s_n$; and
• the Scheme datum, $D$

This list is referred to as a lambda special form.

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 correctly 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 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.

9.3.3 Applying a lambda Function to Input Values

Example 9.3.3: Applying the Squaring Function

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

- A local environment is set up containing a single entry which associates the value 4 with the symbol \(x\).
- The expression, \((\times \ x)\), is evaluated with respect to the newly created local environment. This means that any occurrence of the symbol \(\times\) is evaluated giving preference to entries in the local environment over the Global Environment. The evaluation of \((\times \ x)\) therefore yields the result 16, because \(x\) evaluates to 4 in the local environment, and \(\times\) evaluates to the built-in multiplication function in the Global Environment.
- That value, 16, is taken to be the output value that results from applying the lambda function to the input value 4.

This process is illustrated in Fig. 9.1.
Example 9.3.4: Computing the Volume of a Sphere

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

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

\[
\begin{align*}
> & \quad \text{(define pi 3.14159)} \\
> & \quad \text{(define sphere-volume (lambda (r) (* 4/3 pi r r r)))} \\
> & \quad \text{(sphere-volume 1)} \\
& \quad \quad 4.188786666666666 \\
> & \quad \text{(sphere-volume 2)} \\
& \quad \quad 33.51029333333333
\end{align*}
\]

Consider the evaluation of the expression, \( \text{(sphere-volume 2)} \). It involves the following steps:

- First, a local environment is created containing a single entry that associates the symbol \( r \) with the input value 2.

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

- Finally, the value \( 33.51029333333333 \) is reported as the output value generated by applying the \( \text{sphere-volume} \) function to the input value 2.

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

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

The evaluation of \( \text{(sphere-volume 2)} \) is illustrated in Fig. 9.2.

The following Interactions Window session (continuing from the one given above) illustrates that the existence of a global variable named \( r \) has no effect on the local variable that also happens to be named \( r \). In contrast, changing the value of the global variable, \( \text{pi} \), has disastrous effects! (That is one of many reasons why the use of global variables should be very carefully restricted!)

\[
\begin{align*}
> & \quad \text{(define r 55)} \\
> & \quad \text{(sphere-volume 1)} \\
& \quad \quad 4.188786666666666 \\
> & \quad \text{(sphere-volume 2)} \\
& \quad \quad 33.51029333333333 \\
> & \quad \text{(define pi 100)} \quad ;; \text{YIKES!!} \\
> & \quad \text{(sphere-volume 1)} \quad ;; \text{YIKES!!} \\
& \quad \quad 400/3
\end{align*}
\]

Incidentally, any character sequence beginning with a semi-colon is ignored by DrScheme. (Such character sequences are called comments.) Thus, for example, the sequence, \( ;; \text{YIKES!!} \), has no effect on the evaluation of the expression, \( \text{(sphere-volume 1)} \), above. (Comments are addressed in more detail in Chapter 10.)
Example 9.3.5: More Complex Input Expressions

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

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

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

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

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

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

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

- The `square` symbol evaluates to the squaring function;
- The expression, 10, evaluates to 10; and
- The squaring function is applied to 10, yielding the output value 100.
Example 9.3.6

Here’s an example of a function that takes more than one input argument (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 frequently use a lambda special form to define a function without giving it a name. For example, we might use an expression such as \( \text{lambda (x) (* x x)} \) to represent a squaring function. However, we cannot replace the parameter \( x \) in the lambda expression by arbitrary Scheme expressions. For example, \( \text{lambda (3) (* 3 3)} \) is malformed in Scheme. But we can see a similarity to the common mathematical notation for applying functions to inputs as follows:

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

The corresponding mathematical equations/expressions would be:

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

Example 9.3.7: A Lambda Expression with a Bigger Body

The following example 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; furthermore, the value of the last expression serves as the output value.

⋆ This function is kind of silly since the values of the first four expressions in its body are simply 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, we have not seen functions that have side effects. In fact, it is a very good idea to steer clear of creating functions that cause side effects as much as possible. However, we shall make a few important exceptions, as will be discussed very soon.

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 evaluation of a lambda special form always generates a function. For example, the evaluation of (lambda (x) (* x x)) generates 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 \texttt{(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 \texttt{10}, the input parameter \texttt{x} receives the value \texttt{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 \texttt{s} that must be evaluated is evaluated by looking first for a corresponding entry in the local environment; if no entry for \texttt{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 \texttt{(* x x)} in the body of the squaring function, \texttt{x} evaluates to \texttt{10}, courtesy of the local environment, whereas \texttt{*} 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 \texttt{100}, obtained by evaluating \texttt{(* x x)}, is taken to be the output value for the application of the squaring function to the input value \texttt{10}.

The parameter lists in a \texttt{lambda} special form may specify zero or more parameters, each represented by a Scheme symbol. And the body of a \texttt{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 create functions of our own design.

### Problems

**Problem 9.1**

\[
\text{Consider the following Interactions Window session:}
\]

\begin{verbatim}
> (define pie 3.14159)
> (define funk (lambda (x) (* x pie)))
> (funk 10)
31.4159
\end{verbatim}

\textit{Accurately describe the process that DrScheme goes through in evaluating these three expressions to generate the result, 31.4159. Strive to be complete, while also being concise.}

Chapter 10

Strings, printf, load, Run, and Comments

This chapter introduces the following practicalities:

- Scheme’s string data type. From a conceptual perspective, it would be nice to postpone our discussion of strings; however, from a practical perspective, we cannot do that. Strings are simply too useful for testing, debugging and so on.

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

- The built-in load function. This function causes the Scheme expressions in a specified file to be evaluated in the Interactions Window. 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.

- The Run button. This button is located at the top-right of DrScheme’s main window. When pressed, it causes the Scheme expressions in the Definitions Window to be evaluated, just as if they had been typed into a fresh Interactions Window.

- 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 Strings

Syntactically, strings in Scheme are character sequences delimited by double-quotes. For example, "hi" and "Howdy!" are character sequences that denote string data. The following Interactions Window session demonstrates that the evaluation function behaves like the identity function when applied to string data (i.e., strings evaluate to themselves).

```
> "hi"
"hi"
> "Howdy!"
"Howdy!"
```

Scheme also includes a type-checker predicate, called string?, for the string data type, as demonstrated below.

```
> (string? "abc")
#t
> (string? '("a" "b" "c"))
```
#f
> (string? #t)
#f

## 10.2 The printf Function

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

- The display of textual information by the `printf` function is an example of a harmless side effect.
- The `printf` function does not generate any Scheme output value. (Well, actually, it outputs `void`.)

---

<table>
<thead>
<tr>
<th>Example 10.2.1</th>
</tr>
</thead>
</table>

The simplest way of using the `printf` function is to give it a single string as its only input. The `printf` function will not generate any output value; it will only cause the string to be displayed (without the double quotes) in the Interactions Window, as a side effect, 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, it is not a Scheme output value.

---

* 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.

**Escape sequences.** In the above example, the `printf` function effectively copied the input string into the Interactions Window verbatim. However, the `printf` function sometimes deviates from this simple behavior. In particular, as the `printf` function walks through the input string, it reacts to a few special character sequences in special ways. For example, it reacts to the character sequence, `~\%`, by moving to a new line in the Interactions Window (i.e., it interprets `~\` as a newline character). It also interprets `\n` as a newline character. In addition, whenever it encounters the character sequence `~A` in the input string, the `printf` function treats it as a placeholder for a piece of data to be displayed, as discussed in Example 10.2.2 below. Because the character sequences `~\`, `\n`, and `~A` are not interpreted literally, but involve the `printf` function escaping from a literal interpretation, they are frequently called escape sequences. (And the characters `~` and `\` that introduce escape sequences are sometimes called escape characters.) Although the `printf` function can deal with a variety of other escape sequences, these are the only ones that we’ll need for this course. Their use enables the `printf` function to generate nicely formatted text in the Interactions Window. For this reason, the input string is frequently called a format string—which explains the `f` in `printf`.

In summary, the `printf` function causes the format string (i.e., its first argument) 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 a carriage return in the Interactions Window; and
- each instance of the escape sequence, `~A`, is replaced by a character sequence representing the value of the corresponding input expression.
Notice that if the format string contains \( n \) instances of \(^\sim\)A, then there must be \( n \) input expressions following the format string, as follows:

\[
(\text{printf format-string } expr_1 \ldots expr_n)
\]

---

**Example 10.2.2: Formatted printing with printf**

The following Interactions Window session illustrates the use of the escape sequences \(^\sim\)%, \(^\sim\)\n and \(^\sim\)A 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: \^\%A, second thing: \^\%A\^\%\( + 2 3 \) \( + 6 7 \)"
First thing: 5, second thing: 42
> (printf "Line One!\^\% Line Two!!!\^\% Line Three!!!\^\%"
Line One!
   Line Two!!!
   Line Three!!!
> (printf "First ===> \^\%A, Second ===> \^\%A, Third ===> \^\%A\^\%\( + 4 2 \) \(- 9 6.3 \) \(* 4 100 \)"
First ===> 6, Second ===> 2.7, Third ===> 400
> (printf "A symbol: \^\%A, a string: \^\%A, a boolean: \^\%A\^\%\( 'I-am-a-symbol
   'I-am-a-symbol
   "I am a String!"
   (> 4 2))
A symbol: I-am-a-symbol, a string: I am a String!, a boolean: #t
```

The last of the above interactions illustrates a peculiarity of the printf function: when displaying instances of the string data type, it does not display the double-quotes. That’s true of the formatting string, as well as any elements of lists that happen to be strings (e.g., "I am a String!", above).

---

**Example 10.2.3: 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: \"A\"\%" (void))
Show us void: #<void>
```

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

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

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

**Example 10.2.4**

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

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

* In this class, we will be exploring 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 allow the use of the printf function, which has a harmless, but very useful side effect—namely, displaying information in the Interactions Window.

**Example 10.2.5: 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.
The `tester` function takes any Scheme datum as its input. As a side effect, it first 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.

Problems

**Problem 10.1**

Write down a contract for the `tester` function using the form below:

- **Name:**
- **Input:**
- **Output:**
- **Side Effects:**

**Problem 10.2**

DrScheme uses the Default Rule to evaluate the list denoted by `(tester '(+ 1 2))`. As indicated above, the result of evaluating this list is the number 3. Carefully describe the process DrScheme goes through to generate this result. (You may wish to review Examples ?? and ?? to recall how DrScheme applies a function to inputs.) In particular, what value is associated with the input parameter `datum` in the local environment? What value is passed to the `printf` function called in the body of the `tester` function? And what steps does DrScheme go through to evaluate the expression `(eval datum)` in the body of the `tester` function?
**Problem 10.3**

*How would you change the definition of the tester function so that it printed out the result of evaluating the Scheme datum instead of returning it as the output value? In this case, the tester function would return the "no value" datum.*

---

**10.3 The load Function and the Run Button**

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:** None
- **Side Effect:** Evaluates all of the Scheme expressions in the file named filename.

**Example 10.3.1**

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

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

(define x 34)
```

*Then the following Interactions Window session could ensue:*

```scheme
> 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 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.
10.4 Comments

In Scheme, the semi-colon character is used to signal comments, as illustrated by the following example.

Example 10.4.1

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

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

Evaluating the above code in the Interactions Window would have the same result as evaluating the following, uncommented code:

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

(tester '(+ 2 3))
(tester (+ 2 3))
```

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

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

Example 10.4.2: A contract for the squaring function

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

My personal convention is to use upper-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 10.4.3: A contract for the tester function

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

```
;; TESTER
;; -------------------------------------------
;; INPUT: DATUM, any Scheme datum
;; OUTPUT: The result of evaluating (the value of) DATUM
;; SIDE EFFECT: Displays (the value of) DATUM *before* evaluating it
(define tester
  (lambda (datum)
    ;; Display (the value of) DATUM
    (printf "˜A ==> " datum)
    ;; Evaluate (the value of) DATUM
    (eval datum)))
```

* To avoid being overly cumbersome, contracts may intentionally blur the distinction between the names of input parameters—which are symbols—and their values—which can be anything.

Example 10.4.4: Revised contract for tester

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

```
;; TESTER
;; -------------------------------------------
;; INPUT: DATUM, any Scheme datum
;; OUTPUT: The result of evaluating DATUM
;; SIDE EFFECT: Displays DATUM *before* evaluating it
(define tester
  (lambda (datum)
    ;; Display DATUM
    (printf "˜A ==> " datum)
    ;; Evaluate DATUM
    (eval datum)))
```

10.5 Summary

This chapter introduced the string data type, the built-in printf function, the built-in load function, DrScheme’s Run button, and comments.
Almost any character sequence that begins and ends with double quotes denotes a string datum in Scheme. (The exceptions (e.g., "hi\") involve escape sequences (e.g., \") that effectively capture the final double quote. They need not concern us.) For example, "the brown dog\n" and "i am a fox" both denote strings in Scheme.

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

```scheme
> (printf "One: \A, Two: \A, Three: \A\n%" 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:

```scheme
> (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 contained in the Definitions 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 his/her 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 display text in the Interactions Window
- **load** To load the contents of a file
Chapter 11

Conditional Expressions

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

In Scheme, the if and cond special forms are provided to facilitate the writing of conditional expressions. From the programmer’s perspective, a single if special form can be used to, in effect, make a binary decision (i.e., a decision to evaluate one datum or another), as in: “If $x > y$, evaluate $x^2 - y^2$; otherwise, evaluate $y^2 - x^2$.” Multiple if special forms can be strung together to, in effect, make an $n$-ary decision (i.e., a decision to evaluate one datum selected from $n$ 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 ….” Since stringing together multiple if special forms to make $n$-ary decisions can get quite cumbersome, Scheme provides the cond special form, which has a simpler syntax for conditional expressions associated with $n$-ary decisions.

The conditions in a conditional expression can be simple or complicated. For example, compare “$x > y$” versus “($x > y$) or (($x^2 < y^3$) and ($x + y < 10$))”. In Scheme, more complicated conditions can be composed using the boolean operators, and, or and not. For efficiency reasons, and and or are implemented as special forms, whereas not is provided as a built-in function.

The evaluation of the if, cond, and and or special forms is lazy in the sense that only the computations needed to ascertain the final value are actually performed.

11.1 The if Special Form

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

The syntax of an if special form is as follows:

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

where:

- $\text{condExpr}$ is a condition (i.e., an expression that evaluates to #t or #f); and
- $\text{thenExpr}$ and $\text{elseExpr}$ are any Scheme expressions.
Example 11.1.1

The following expressions are examples of the if special form:

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

The semantics of the if special form stipulates that it is evaluated thusly:

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

Notice that the condition, \textit{condExpr}, is always evaluated; however, after that, \textit{one and only one} of the remaining expressions, \textit{thenExpr} or \textit{elseExpr}, is evaluated. We say that the evaluation of the if special form is lazy, in the sense that it only evaluates the expressions needed to compute the value of the entire if expression.

Example 11.1.2

The following Interactions Window session demonstrates the evaluation of the if special forms seen earlier.

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

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

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

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

\[
\begin{align*}
&;; \ \text{HOW-BIG} \\
&;; \ \text{------------------------------------} \\
&;; \ \text{INPUT: NUM, a number} \\
&;; \ \text{OUTPUT: The symbol SMALL, if NUM is less than 10;} \\
&;; \ \quad \text{Otherwise, the symbol BIG.} \\
&(\text{define how-big}
\end{align*}
\]
The following interactions demonstrate its behavior:

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

Notice that the result of evaluating the condition, \((< \text{num} \ 10)\), depends on the value of \text{num} in the local environment, which is not known at the time the function is specified by the programmer; instead, the value of \text{num} is only known when the function \text{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 \text{if} special form can be quite useful.

### The non-strict version of the \text{if} special form.

In the strict version of the \text{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.

#### Example 11.1.4

The following are legal instances of the \text{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 \text{if} special form is governed by the following rule:

★ When interpreting the value of the condition, anything other than \#'f counts as boolean \text{true} (i.e., \#'f is the only Scheme datum that counts as boolean \text{false}).

#### Example 11.1.5

The following Interactions Window session demonstrates the evaluation of the non-strict \text{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

\text{is not}

\text{true}, \text{the \text{if} expression}

\text{will choose the second}

\text{option.}
counts as boolean false, the conditions in these examples all count as boolean true. Thus, in each case, the then expression is evaluated—and the value of the then expression is the value of the entire if expression.

Problems

Problem 11.1: The maxx function

Define a function, called maxx, that takes two numbers as its inputs. It should return the maximum of the two numbers, as illustrated below:

\[
\begin{align*}
\texttt{> (maxx 2 3)} & \quad \text{3} \\
\texttt{> (maxx 5 1)} & \quad \text{5} \\
\texttt{> (maxx 4 4)} & \quad \text{4}
\end{align*}
\]

As you may have guessed, there is a built-in max function. However, you are not allowed to use it for this problem! Instead, use if to determine which input number is bigger. The operation of the maxx function could be described thusly: if \(x\) is bigger than \(y\), then the output should be \(x\); otherwise, it should be \(y\). Here's the contract:

\[
\begin{align*}
;; \ \text{MAXX} \\
;; \ \text{-------------------------------------------} \\
;; \ \text{INPUTS: } \ X, \ Y, \ \text{two numbers} \\
;; \ \text{OUTPUT: } \ \text{The maximum of } X \ \text{and } Y
\end{align*}
\]

And some tester expressions:

\[
\begin{align*}
\texttt{(tester '(maxx 2 3))} & \quad \text{3} \\
\texttt{(tester '(maxx 5 1))} & \quad \text{5} \\
\texttt{(tester '(maxx 4 4))} & \quad \text{4}
\end{align*}
\]

Problem 11.2: Printing out a message about bananas!

Here’s the contract for the banana-msg function. Note that it does not generate any output value; however, it does cause some side-effect printing to occur in the Interactions Window:

\[
\begin{align*}
;; \ \text{BANANA-MSG} \\
;; \ \text{-----------------------------} \\
;; \ \text{INPUT: } \ \text{NUM, an integer} \\
;; \ \text{OUTPUT: } \ \text{don’t care} \\
;; \ \text{SIDE EFFECT: } \ \text{Print out a message in the Interactions Window} \\
;; \ \text{such as: } \ \text{I ate NUM bananas!}, \ \text{except that NUM should be} \\
;; \ \text{replaced by its value. Also, if you only ate one banana,} \\
;; \ \text{then banana should not be pluralized!}
\end{align*}
\]

Here are some examples of its use (in the Interactions Window):

\[
\begin{align*}
\texttt{> (banana-msg 3)} & \quad \text{I ate 3 bananas!} \\
\texttt{> (banana-msg 1)} & \quad \text{← these are not output values!!}
\end{align*}
\]
I ate 1 banana! ← they are side-effect printing!!
> (banana-msg 0)
I ate 0 bananas!

Here are some tester expressions to copy into your Definitions Window:

(tester '(banana-msg -2))
(tester '(banana-msg 0))
(tester '(banana-msg 1))
(tester '(banana-msg 3))

Note: There are many ways to solve this problem. See the description of the printf function in Section 10.2.

11.2 The Boolean Operators: and, or and not

This section introduces the boolean operators, and, or and not. The first two are implemented as special forms in Scheme; in contrast, not is a built-in function. The reasons are discussed below.

11.2.1 The 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 11.2.1

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

In each of these examples, the non-boolean input is interpreted as boolean true. Thus, the output is #f.

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
In-Class Problem 11.2.1

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

11.2.2 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 11.2.2

The following Interactions Window session demonstrates the behavior of and:

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

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

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

Example 11.2.3

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

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 11.2.2

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, 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 11.2.4

> (and #t #t #t #t) #t
> (and #t #t #f #t) #f
> (and (> 3 2) (= 9 9) (<= 5 20))
> (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.

11.2.3 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 11.2.5

> (or #f #f #f #f)
#f
> (or #f #f #t #f)
#t
> (or #t #t #t #t)
#t
> (or (= 9 8) (> 7 9) (<= 4 2)) ← each input evaluates to #f . . .
#f
> (or #f #f 3 #f #f 5)
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 11.2.3

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 compare two symbols (e.g., tue and the value of the input parameter day).
In-Class Problem 11.2.4

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 or PM). In particular, define a function that satisfies the following contract:

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

Here are a few examples of the desired behavior:

```
> (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 equality among symbols (e.g., AM, PM, etc.); use `=` to test equality among numbers (e.g., hours).

Problems

Problem 11.3

For each statement below, decide which of the words in parentheses apply:

- Evaluation of an `if` special form (always, never, sometimes) causes a side effect.
- Evaluation of an `and` special form (always, never, sometimes) causes a side effect.
- Evaluation of an `or` special form (always, never, sometimes) causes a side effect.
- Evaluation of an `if` special form always requires evaluating exactly (one, two, all) of its inputs.
- Evaluation of an `and` special form always requires evaluating at least (one, two, all) of its inputs.
- Evaluation of an `or` special form always requires evaluating at least (one, two, all) of its inputs.

11.3 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 11.3.1: Nested if expressions

Consider the following letter-grade function:

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

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

The following interactions illustrate its behavior:

```scheme
> (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. Thus, DrScheme evaluates the condition, (>= num 90). Since num has the value 86, this condition 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 the condition, (>= num 80). Since num has the value 86, this expression 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).

Example 11.3.2: Using cond instead of nested if expressions

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.

```scheme
(define letter-grade-v2
  (lambda (num)
    (cond ((>= num 90) 'A) ((>= num 80) 'B) ((>= num 70) 'C) (else 'D))))
```
That this 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 cond expression is evaluated as follows. The conditions are evaluated, in turn, until one is found that evaluates to true (or something that counts as true). The value of the cond expression is the value of the expression following the first condition that evaluated to true (or something that counts as true). In this case, the first condition, (>= num 90), evaluates to #f. Similarly, the second condition, (>= num 80), evaluates to #f. However, the third condition, (>= num 70), evaluates to #t. Thus, the value of the entire cond expression is whatever the expression, ’C, evaluates to. Since ’C evaluates to C, that is the value of the entire cond expression.

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 (i.e., the value of ’D).

* The last condition in a cond expression should always 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.

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

```
(cond
  (cond1 expr1)
  (cond2 expr2)
  ...
  (condn exprn)
)
```

where:
• each \( cond_i \) is a (strict or non-strict) condition;
• (by convention) the last condition, \( cond_n \), is either \#t or else; and
• each \( expr_i \) is some Scheme expression.

The value of such a \( cond \) expression is determined as follows:

Each condition, \( cond_i \), 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 expression, \( expr_i \).

Like the if, and and or special forms, 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, \( cond_i \), evaluates to true, then no subsequent conditions will be evaluated. In addition, only one expression, \( expr_i \), is evaluated; all others are ignored.

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 that serves as the value of the \( cond \) expression.

**Example 11.3.3**

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.

```scheme
(define cond-effects
  (lambda (num)
    (cond
      ;; ----------------------------
      ((>= num 90)
        (printf "Oh my gosh! You did great!!!\n")
        'A)
      ;; ----------------------------
      ((>= num 80)
        (printf "Well, you know, a B is pretty good!!\n")
        (printf "Nothing to be ashamed of at all!!\n")
        'B)
      ;; ----------------------------
      ((>= num 70)
        (printf "According to Vassar, a C is considered average!!!\n")
        (printf "Thus, your grade, ~A, is average!!\n" num)
        'C)
      ;; ----------------------------
      (else
        (printf "Hmmm... Hard to find much positive to say here.\n")
        (printf "Maybe there’s been a mistake...\n")
        (printf "But until we find it, your grade stands\n")
    )
  )
)
```
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)
According to Vassar, a C is considered 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.

### 11.4 Defining Predicates using Boolean Operators instead of Conditional Expressions

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 11.4.1: Defining a predicate using conditional expressions**

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

```scheme
;; CAFE-OPEN?
```
;; INPUTS: DAY, a symbol, one of SUN, MON, TUE, ..., FRI, SAT
;; AM-OR-PM, a symbol, either AM or PM
;; HOUR, an integer from 1 to 12, inclusive
;; MINUTES, an integer from 0 to 59, inclusive
;; OUTPUT: #t, if the inputs specify a time at which the
;; CAFE CMPU-101 is open; #f, otherwise.

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

Here are some examples of the desired behavior of this function:

> (cafe-open? 'tue 'am 10 30)
#f
> (cafe-open? 'wed 'am 11 45)
#f
> (cafe-open? 'wed 'pm 11 45)
#t
> (cafe-open? 'thu 'am 12 15)
#t

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

(define cafe-open?
  (lambda (day am-or-pm hour minutes)
    (cond
      ;; Case 1: Open after 11:30 pm on Wednesdays
      ((and (eq? day 'wed)
             (eq? am-or-pm 'pm)
             (= hour 11)
             (> minutes 30))
       #t)
      ;; Case 2: Open all day on Thursdays
      ((eq? day 'thu)
       #t)
      ;; Case 3: Open Friday mornings *before* 9
      ;; (including times such as 12:25 a.m.)
      ((and (eq? day 'fri)
             (eq? time-of-day 'am)
             (< 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 11.4.2: 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.

11.5 Simplifying Conditional and Boolean Expressions

Conditional and boolean expressions can be combined in many ways to enable Scheme functions to make finely tuned decisions amongst any number of cases. Although conditional and boolean 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 \textit{true}, while only \#f counts as \textit{false}.

\textbf{Definition 11.1: Equivalent boolean conditions}

\textit{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, whenever the evaluation of boolOne and boolTwo with respect to some environment \(E\) generates the respective boolean values, \(B_1\) and \(B_2\), \(B_1\) and \(B_2\) must be the same.}

\textbf{Example 11.5.1: Simplifying if expressions involving boolean conditions}

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

\begin{verbatim}
> (define x 32) ← Setting up an environment where \(x > y\)
> (define y 4)
> (if (> x y) \#t \#f) \#t
> (> x y) \#t
> (define y 1000) ← Changing the environment so that \(x < y\)
> (if (> x y) \#t \#f) \#f
> (> x y) \#f
\end{verbatim}

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

\(\text{(if boolCond \#t \#f)} \leadsto \text{boolCond}\)

\(*\) In addition, it is not hard to verify that the following are also equivalent:

\(\text{(if boolCond \#f \#t)} \leadsto \text{(not boolCond)}\)

So, if you ever find yourself writing an \textit{if} expression whose \textit{then} and \textit{else} clauses are some combination of \#t and \#f, consider making one of the above simplifications.

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

\textbf{Example 11.5.2: Simplifying if expressions: non-strict truth values}

\begin{verbatim}
> (if 'happy \#t \#f) \#t
> 'happy happy
> (if 'sad \#f \#t)
\end{verbatim}
In the first case, the if expression evaluates to #t, whereas ‘happy evaluates to happy—which counts as true. In the second case, both expressions evaluate to #f, the only expression that counts as false.

Finally, 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:

$$
\begin{align*}
(\text{and } e_1 (\text{and } e_2 e_3)) & \sim (\text{and } e_1 e_2 e_3) \\
(\text{or } e_1 (\text{or } e_2 e_3)) & \sim (\text{or } e_1 e_2 e_3)
\end{align*}
$$

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

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

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

$$
\begin{align*}
(\text{not } (\text{and } e_1 e_2)) & \sim (\text{or } (\text{not } e_1) (\text{not } e_2)) \\
(\text{not } (\text{or } e_1 e_2)) & \sim (\text{and } (\text{not } e_1) (\text{not } e_2))
\end{align*}
$$

### 11.6 Summary

This chapter introduced the if special form for making binary decisions; the boolean operators and, or and not that can be combined to form complex boolean expressions; and the cond special form for making decisions among any number of cases.

★ This chapter introduced non-strict truth values. In particular, anything other than #f counts as boolean true. Equivalently, only #f counts as boolean false.

★ if, and, or, not and cond all accommodate non-strict truth values.

An if special form includes a boolean condition, a then expression, and an else expression. The evaluation of an if special form starts by evaluating the boolean condition. If it evaluates to (non-strict) true, then the then expression is evaluated, and its value is taken to be the value of the entire if expression. However, if the boolean condition evaluates to #f, then the else expression 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.

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 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 \( n \)th case is the first case whose condition evaluates to (non-strict) true, then the expressions in the body of that case are evaluated; and all subsequent cases are ignored.

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

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

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

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

### Special Forms Introduced in this Chapter

- **if** For making binary decisions
- **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

## Problems

**Problem 11.4**

Define a function, called quadrant, that satisfies the following contract:

```scheme
;; QUADRANT
;; ---------------------------
;; INPUTS: X, Y, two numbers
```
;; OUTPUT: A number specifying the quadrant to which the point
;; (X,Y) belongs in the XY-plane; or 0 if it lies on an axis.

Recall that the first quadrant is where both \( x \) and \( y \) are positive; the second quadrant is where \( x \) is negative and \( y \) is positive; the third quadrant is where both \( x \) and \( y \) are negative; and the fourth quadrant is where \( x \) is positive and \( y \) is negative.

Hint: Use cond.

Be sure to test your function on a variety of inputs (for all four quadrants and various places on the axes, including the origin).

Problem 11.5

Define a function, called \texttt{data-type-of}, that satisfies the following contract. Note that it returns a symbol as its output.

\begin{verbatim}
;; DATA-TYPE-OF
;; ------------------------
;; INPUT: DATUM, anything
;; OUTPUT: A SYMBOL representing the data type of DATUM,
;; one of: NUMBER, BOOLEAN, LIST, SYMBOL, STRING, etc.
\end{verbatim}

Note: You don’t need to handle every possible data type. Returning the symbol \texttt{unknown} is okay if you get tired. Here are some examples of its behavior in the Interactions Window:

\begin{verbatim}
> (data-type-of #t)
boolean
> (data-type-of ())
list
> (data-type-of 45)
number
\end{verbatim}

And here are some tester expressions to copy into your Definitions Window:

\begin{verbatim}
(tester '(data-type-of 3))
(tester '(data-type-of #t))
(tester '(data-type-of '(+ 2 3)))
(tester '(data-type-of (+ 2 3)))
(tester '(data-type-of "abc"))
\end{verbatim}

Hint: Use the built-in type-checker predicates from Chapter 8.

Problem 11.6

The \texttt{implies} function takes two boolean inputs, and generates a boolean output, as illustrated below:

\begin{verbatim}
(implies #t #t) ===> #t
(implies #f #t) ===> #t
(implies #t #f) ===> #f
(implies #f #f) ===> #t
\end{verbatim}
Write a contract for the `implies` function, and then implement it in Scheme. For this problem, you are allowed to use `if`, but you are not allowed to use `and`, `or`, `not` or `cond`.

- There are only four different combinations of inputs for this function; however, you can include more complicated input expressions, such as: `(implies (> 3 2) (< 4 5))`. And since anything other than `#f` counts as true, you can even do things like: `(implies 'hi 'there)`.

Problem 11.7

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.

(a) Define a function, called `time-of-day`, that takes two numerical inputs, `mil-hours` and `minutes`, where `mil-hours` represents the number of hours according to the 24-hour military clock, and `minutes` represents the number of minutes.

- You may assume that `mil-hours` and `minutes` are integers such that: 
  
  \[0 \leq \text{mil-hours} < 24\]  
  \[0 \leq \text{minutes} < 60\]

- `time-of-day` should return a symbol as its output. In particular, it should return one of the following: midnight, am, noon or pm, as appropriate. For example:

  > (time-of-day 15 39)
  
  pm

  > (time-of-day 12 0)
  
  noon

  Note that the `pm` and `noon` are output values that are symbols; they are not side-effect printing.

- HINT: You may use `if` or `cond` in the body of your function. In either case, be sure to include comments that briefly describe each case that you’re handling.

- NOTE: 12:00 midnight is neither a.m. nor p.m. Rather, it is a boundary between a.m. and p.m. Similar remarks apply to 12:00 noon. So `(time-of-day 12 0)` should return noon, not am or pm. Similarly `(time-of-day 0 0)` should return midnight, not am or pm. But `(time-of-day 0 15)` should return am, since 00:15 in military time corresponds to 12:15 a.m. in civilian time.

- Be sure to write a contract for your function!

(b) Define a function, called `mil-to-civil-hrs`, that takes a single numerical input, `mil-hours`, where \[0 \leq \text{mil-hours} < 24\]. It should generate as its output, the corresponding number of hours according to the 12-hour civilian clock. For example:

  `(mil-to-civil-hrs 19) ===> 7`

  Because 19:00 on the military clock corresponds to 7:00 p.m. on the civilian clock.

  Hint: Use a `cond` in the body of your function.

  Hint: Be careful about 0.

  Be sure to include a contract for your function!

(c) Define a function, called `print-civil-time-from-mil`. It should take two numerical inputs, `mil-hours` and `minutes`, as in part (a). However, this function, unlike the above functions, should not generate any Scheme datum as output. Instead, it should have the side effect of displaying the time in the 12-hour civilian format, as the following interactions window session illustrates:
> (print-civil-time-from-mil 15 39)
3:39 pm

* In this example, 3:39 pm is side-effect printing, generated using the built-in printf function. There is no output value.

* Use the time-of-day and mil-to-civil-hrs functions as helpers. You shouldn’t need to re-implement the computations done by time-of-day or mil-to-civil-hrs.

* Be sure to include a contract for your function!

(Optional) If the number of minutes is small, you will have to work a little harder to make sure that a leading zero is displayed. Consider 3:6 pm versus 3:06 pm.

> (print-civil-time-from-mil 15 6)
3:06 pm

There are many examples that demonstrate the use of the built-in printf function in the amst-helper.txt file available in each lab and assignment directory.

Problem 11.8

Define a function, compute-tax, that takes a single input, income. It should return the amount of tax owed on that income, as determined by the following tax brackets:

Income below $10,000 is taxed at 10%.

Income between $10,000 and $30,000 (only the amount after the first $10,000) is taxed at 15%.

Income above $30,000 (only the amount after the first $30,000) is taxed at 25%.

⇒ If you make more than $10,000, the first $10,000 of your income is taxed at 10%; only the amount above $10,000 is taxed at the higher rates. Similarly, if you make more than $30,000, the first $10,000 of income is taxed at 10%, the next $20,000 (i.e., the amount between $10,000 and $30,000) is taxed at 15%, and the rest is taxed at 25%. Thus, if you earn $100,000, your taxes will not be $25,000 (i.e., 25% of $100,000), instead, they will be:

(10% of $10,000) + (15% of $20,000) + (25% of $70,000)

which equals: $1,000 + $3,000 + $17,500 = $21,500.

Be sure to include tester expressions that test a representative set of cases. You may use if or cond for this problem. And, as always, be sure to include a contract for your function.

Problem 11.9

The CMPU-101 bookstore is open on Saturdays from 11:45 a.m. to 12:15 p.m., inclusive, and all day Tuesday, except from 12:01 p.m. to 12:59 p.m., inclusive.

(a) Define a function, called bookstore-open?, that satisfies the following contract. Use a cond special form to structure the body of this function.

;;; BOOKSTORE-OPEN?
;;; ---------------------------------------
;;; INPUTS: DAY, a symbol, one of SUN, MON, TUE, ..., FRI, SAT
;;; HOUR, an integer from 1 to 12, inclusive
MINUTES, an integer from 0 to 59, inclusive
AM-OR-PM, a symbol, either AM or PM
OUTPUT: #t, if the inputs specify one of the following:
Saturday from 11:45 am to 12:15 pm, inclusive.
All day Tuesday, except from 12:01 pm to 12:59 pm,
inclusive; #f otherwise.

Here are some examples of its behavior:

> (bookstore-open? 'sat 11 30 'am)
#f
> (bookstore-open? 'sat 11 50 'am)
#t
> (bookstore-open? 'sat 12 12 'pm)
#t
> (bookstore-open? 'sat 12 49 'pm)
#f

(b) Same as above, except this time use the boolean operators, and, or and not, instead of conditional expressions, as described in Section 11.4.

Problem 11.10

The following predicate is defined using a cond special form:

(define office-open?
 (lambda (day am-or-pm)
  (cond
   ;; Case 1: Closed on Fridays
   ((eq? day 'fri) #f)
   ;; Case 2: Open Wed afternoons
   ((and (eq? day 'wed)
         (eq? am-or-pm 'pm)) #t)
   ;; Case 3: Closed on Tuesday mornings
   ((and (eq? day 'tue)
         (eq? am-or-pm 'am)) #f)
   ;; Case 4: Open all other times
   (else #t))))

Your job is to define a predicate, called office-open?-alt, that works just like office-open?, except that it is defined using and, or and not, instead of the conditional expressions, if or cond.

⋆ Careful! Some of the cases above output #t, while others output #f.
Chapter 12
Recursion

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); instead, we simply 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 $\text{expr}$ is a Scheme expression that denotes a non-empty list, $L$, whose evaluation is governed by the Default Rule. Then we say that $\text{expr}$ 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 $\text{expr}$ calls $f$.

Thus, for example, the expression, $(+ 2 3)$, is a function-call expression that calls the built-in addition function. Similarly, $(\text{symbol? } 'x)$ is a function-call expression that calls the built-in symbol? function. In contrast, the expressions, $(\text{define myVar } 3)$ and $(\lambda (x) (* x x))$, 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 statement that tests some stopping condition (or base case). If the stopping condition evaluates to (non-strict) 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.

Thus, no new Scheme constructs are required to support recursion.

---

**Example 12.1.1: The factorial function**

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

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

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

- **Base Case** (\( n = 1 \)): \( 1! = 1 \) (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.)

```scheme
;; FACTY-V1
;; ----------------------------------
;; INPUT: N, a positive integer
;; OUTPUT: The factorial of N (i.e., N*(N-1)*...*3*2*1)
(define facty-v1
  (lambda (n)
    (if (= n 1)
      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—in this case, the lambda expression—is evaluated. Like any lambda expression, the one above evaluates to a function. However:

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

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

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

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

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

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

Before delving deeper into why facty-v1 works, observe that we can define an equivalent function, facty-v2, using a cond expression, as follows:

;; 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. Once again, this function appears to correctly compute the factorial of its input:
Finally, we can define another equivalent version of the factorial function, this time called facty. This function differs only in that it contains some printf expressions that will help us to trace what happens when an expression such as (facty 3) is evaluated:

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

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

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

(cond
  ;; Base Case: n = 1
  (= n 1)
  (printf "Base Case (n = 1)\n")
  1)
  ;; Recursive Case: n > 1
  #t
  (printf "Recursive Case (n = ~A)\n" n)
  (* n (facty (n 1))))))
Since the value of \( n \) is 3 in the local environment, the condition, \((= n 1)\), evaluates to \#f. Thus, we skip to the second condition, \#t, which of course evaluates to \#t. Thus, the expressions associated with the recursive case are evaluated in turn. The first expression causes the line, Recursive Case \((n = 3)\), to be displayed in the Interactions Window. Then, the second expression, \((* n (facty (- n 1)))\), must be evaluated—according to the Default Rule. The \(*\) symbol evaluates to the multiplication function, \( n \) evaluates to 3, and \((facty (- n 1))\) evaluates to \(*\). Gosh, we need a new paragraph! The expression, \((facty (- n 1))\), is evaluated according to the Default Rule. First, the \(facty\) symbol evaluates to the \(facty\) function; and \((- n 1)\) evaluates to 2 (since \( n \) has the value 3). Next, the \(facty\) 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, \((* (facty (- n 1)))\), in the top function-call box cannot continue until the subsidiary expression, \((facty (- n 1))\), is 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 \(facty\) function to the value 2, depicted in the second function-call box in the figure, is similar to the application of the \(facty\) 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 expressions associated with the recursive case are evaluated. The evaluation of the \(printf\) expression causes the line, Recursive Case \((n = 2)\), to be displayed in the Interactions Window; and the evaluation of the expression, \((* n (facty (- n 1)))\), leads to yet another recursive function call—this time the application of the \(facty\) function to the input value 1, as illustrated in the third box in Fig. 12.1.

* Once again, the evaluation of the expression, \((* n (facty (- n 1)))\), in the second box cannot continue until the output value for the third box has been generated. In other words, the evaluation of the expression in the second box must be suspended, pending the outcome of the third box.

The application of the \(facty\) 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 \(cond\) expression in the body of the function is evaluated. This time, however, the condition \((= n 1)\) evaluates to \#t; thus, the base case expressions are evaluated. Evaluating the \(printf\) expression causes the line, Base Case \((n = 1)\), to be displayed in the Interactions Window. Next, the expression, 1, evaluates to itself, yielding the output value for the application of the \(facty\) function to the value 1 (i.e., the output value for the third box). This output value, 1, is the value of the expression, \((facty (- n 1))\), that was being evaluated in the middle function-call box (where \( n \) has the value 2). Now that that the value of \((facty (- n 1))\) is in hand, the evaluation of the expression, \((* n (facty (- n 1)))\), in the middle box can continue. To wit, the multiplication function is applied to 2 and 1, yielding the output value 2 for the middle function-call box.

This output value, 2, is the value of the expression, \((facty (- n 1))\), that was being evaluated in the top function-call box (where \( n \) has the value 3). Now that that the value of \((facty (- n 1))\) is in hand, the evaluation of the expression, \((* n (facty (- n 1)))\), 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!
Here is what it looks like when \((\text{facty } 3)\) is evaluated in the Interactions Window:

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

**Example 12.1.1** illustrates many of the features that are frequently found in recursive functions.

- The body of the function contains a conditional expression that enables a stopping condition—commonly called a **base case**—to be recognized. If that stopping condition evaluates to \#t (or any non-strict true), then no more recursive function calls are made.

- 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 expression in the body of the function is identical in each recursive function call, it is evaluated with respect to a different local environment. In other words, the evaluation of the 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 function, called `summer`, that satisfies the following contract:

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

**Example 12.1.3: 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:

```scheme
;; SUM-SQUARES
;; ---------------------
```
Figure 12.1: DrScheme’s evaluation of \texttt{(facty 3)}
;; 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

Problems

Problem 12.1

Define a function, called power, that takes two inputs: x, any real number, and p any non-negative integer. It should return as its output the value of x raised to the pth power (i.e., \(x^p\)), as illustrated below.

> (power 2 3) ← 2^3 = 2·2·2 = 8
8
> (power 3 2) ← 3^2 = 3·3 = 9
9
> (power 2 5) ← 2^5 = 2·2·2·2·2 = 32
32

* Hint: Use recursion, similar to how it is used in facty. For example, note that 2^9 = 2·(2^8).

* Be sure to include a contract for your function.

Problem 12.2

Define a function, called sum-recips, that computes sums of the following form:

\[
\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}
\]

where n is some positive integer. Here are some sample interactions:

> (sum-recips 1)
1
> (sum-recips 2)
3/2
> (sum-recips 3)
11/6

And the corresponding tester expressions:

(tester '(sum-recips 1))
(tester '(sum-recips 2))
(tester '(sum-recips 3))

Insert some more tester expressions of your own. If you want to encourage DrScheme to display numbers in “floating point” form (e.g., 1.5 instead of 3/2), just use 1.0 in your base case, instead of 1.

Consider the following:

> (/ 3 2)
3/2
> (/ 3.0 2)
1.5

Be sure to include a contract for your function!

Problem 12.3

Define a function, called alt-sum, that takes a positive integer n as its only input. It should return as its output the following sum:

\[ 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots \pm \frac{1}{n} \]

where the sign of each term is negative if the denominator is even, and positive if the denominator is odd. Here are some examples:

> (alt-sum 1)
1
> (alt-sum 2)
0.5
> (alt-sum 3)
0.8333333333333333

For example, (alt-sum 3) returns the sum, \(1 - \frac{1}{2} + \frac{1}{3} = \frac{5}{6}\). To ensure that you get the desired “floating point” notation, you can use expressions such as (/ 1.0 n) instead of (/ 1 n), as illustrated below:

> (/ 1 5)
1/5
> (/ 1.0 5)
0.2

This is especially relevant for cases where n is large. The value of (alt-sum 100) in fractional notation would be very cumbersome!

* There are built-in functions called even? and odd? that return #t if their input is an even (or odd) number, as illustrated below.
12.2 Tail Recursion

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

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

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

### Example 12.2.1: Printing Dashes

Consider the \text{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: \(n \leq 0\)
      ((<= n 0) (newline))
      ;; Recursive Case: \(n > 0\)
      (#t
        ;; Print one dash
        (printf "-")
        ;; Let the recursive func call print the rest of the dashes
        (print-n-dashes (- n 1))))))
```
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.

\[
\begin{align*}
> & \ (\text{print-n-dashes} \ 5) \\
& \ ------ \\
> & \ (\text{print-n-dashes} \ 12) \\
& \ -----------
\end{align*}
\]

Consider the evaluation of the expression, \((\text{print-n-dashes} \ 5)\). According to the Default Rule for evaluating non-empty lists, evaluating this list requires applying the \text{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 printf 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, \((\text{printf} \ "-" )\), causes a single dash to be displayed in the Interactions Window. Evaluating the second expression, \((\text{print-n-dashes} \ (- n \ 1))\), requires making a recursive function call.

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

\( \star \) When the value of the recursive function-call expression, \((\text{print-n-dashes} \ (- n \ 1))\), is known, it will be the output value for the original expression, \((\text{print-n-dashes} \ 5)\). 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 \text{print-n-dashes} to the value \( 4 \), we can simply re-use the function-call box we already have. This will require us 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.

\( \star \) 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 \text{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 \text{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 ask you to draw a sequence of function-call boxes for all of the expressions, \((\text{print-n-dashes} \ 5)\), \((\text{print-n-dashes} \ 4)\), ..., \((\text{print-n-dashes} \ 0)\), 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 \text{print-n-dashes} function is an example of a tail-recursive function. But what exactly do we mean by tail recursive?
Definition 12.3: Tail-recursive function

Suppose that $f$ is a function, $B$ is its body, and $expr$ is a recursive function-call expression somewhere within $B$. We say that $expr$ is a tail-recursive function-call expression within $B$ if, whenever evaluating $B$ requires evaluating $expr$, it is necessarily the case that the last step in evaluating $B$ is the evaluation of $expr$ and, thus, the value of $B$ is identical to the value of $expr$. If every recursive function-call expression in the body of $f$ is tail-recursive, then $f$ is called a tail-recursive function.

Okay, the above definition is correct and completely general, but it may be a little hard to process. The following example considers a less general, but quite common case of a tail-recursive function—one that exhibits the characteristic features, and covers the print-n-dashes from Example 12.2.

Example 12.2.2

Suppose that $rec-func$ is a recursive function whose body $B$ consists of a single $cond$ expression. Suppose further that this $cond$ has only two cases: a base case and a recursive case. The only way that $rec-func$ can be tail recursive is if, as shown below, the recursive function-call expression, ($rec-func$ ...), is the last (i.e., tail) expression within the recursive case.

\[
\begin{align*}
(define \text{rec-func} & \quad (lambda (\ldots)) \\
& \quad (cond \\
& \quad \quad ;; \text{Base Case} \\
& \quad \quad (\ldots \\
& \quad \quad \ldots \\
& \quad \quad ) \\
& \quad \quad ;; \text{Recursive Case} \\
& \quad \quad (\ldots \\
& \quad \quad \ldots \\
& \quad \quad (\text{rec-func} \ldots) \\
& \quad \quad )))
\end{align*}
\]

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. 12.2, this function is tail recursive; as is the print-n-dashes function from Example 12.2.

In contrast, consider the definition of the $facty$ function, seen earlier:

\[
\begin{align*}
(define \text{facty} & \quad (lambda (n)) \\
& \quad (cond \\
& \quad \quad ;; \text{Base Case: } n = 1 \\
& \quad \quad ((= n 1) \\
& \quad \quad 1) \\
& \quad \quad ;; \text{Recursive Case: } n > 1 \\
& \quad \quad (#t \\
& \quad \quad (* n (\text{facty} (- n 1)))))) \\
\end{align*}
\]

Notice that the last expression in the recursive case of the $\text{cond}$ is ($* n (\text{facty} (- n 1))$). This expression includes the recursive function-call expression, ($\text{facty} (- n 1)$), 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 expres-
sion 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, \texttt{facty}, is not tail recursive.

\[ \text{This is actually the \texttt{facty-v2} function, but the same points apply to all versions of the \texttt{facty} function seen earlier.} \]

**In-Class Problem 12.2.1**

Define a function that satisfies the following contract:

\[
;; \hspace{1em} \text{PRINT-FUNC-VALS} \\
;; \hspace{1em} \text{--------------------------------------------------------} \\
;; \hspace{1em} \text{INPUTS: FUNC, a function that expects a single numerical input} \\
;; \hspace{1em} \hspace{1em} FROM, a starting input \\
;; \hspace{1em} \hspace{1em} TO, an ending input \\
;; \hspace{1em} \hspace{1em} \text{OUTPUT: None} \\
;; \hspace{1em} \hspace{1em} \text{SIDE EFFECT: Prints the values of FUNC when applied to} \\
;; \hspace{1em} \hspace{1em} \hspace{1em} the successive inputs from FROM to TO.}
\]

Tail-recursive functions like \texttt{print-n-dashes} do not generate interesting output values; instead, their pri-
mary 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 in-
put parameters. Frequently, those additional input parameters are called \textit{accumulators} because they are used to
incrementally accumulate values of interest. Section 12.3 addresses accumulator-based tail-recursive functions.

**Problems**

The following set of problems involve functions that do not generate any output value, but instead cause side-effect
printing to occur. For such functions, the following \texttt{tester-alt} function will generate nicer looking test results
in the Interactions Window.

\[
\begin{array}{l}
\text{(define tester-alt} \\
\text{  (lambda (datum) } \\
\text{    (printf "A =>" datum) } \\
\text{    (newline) } \\
\text{    (eval datum) } \\
\text{    (newline)))})
\end{array}
\]

It is the same as the \texttt{tester} function seen earlier, except that it makes sure that any side-effect printing caused
by evaluating the expression \texttt{(eval datum)} starts on a new line. To enable use of this function, copy-and-paste
the above definition into your Definitions Window.

**Problem 12.4**

Why would it be difficult to implement the \texttt{print-n-dashes} function from Example 12.2 using \texttt{if}
instead of \texttt{cond}?
Problem 12.5

Define a function, called \textit{print-thing-n-times}, that takes two inputs: \textit{thing} and \textit{n}, where \textit{thing} can be anything, and \textit{n} is a non-negative integer. It should not generate an output value; instead, it should have the side effect of printing out \textit{thing} \textit{n} times in the Interactions Window, as illustrated below:

\begin{verbatim}
> (print-thing-n-times 'Hi 5)
HiHiHiHiHi
> (print-thing-n-times '--- 3)
---------
\end{verbatim}

Be sure to include a contract for your function.

Problem 12.6

Define a function, called \textit{print-down-to-zero}, that takes a non-negative integer \textit{n} as its only input. It should not generate any output value; instead, it should have the side effect of printing out the values from \textit{n} down to zero in the Interactions Window, as illustrated below.

\begin{verbatim}
> (print-down-to-zero 5)
5 4 3 2 1 0
> (print-down-to-zero 22)
22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\end{verbatim}

Be sure to include a contract for your function.

Problem 12.7: Printing rectangles and squares

Copy-and-paste the contract and function definition for the \textit{print-n-dashes} function from Example 12.2 into your Definitions Window. Recall that \textit{print-n-dashes} does not generate any Scheme output value; instead, it has the side effect of displaying a row of \textit{n} dashes in the Interactions Window, as illustrated below.

\begin{verbatim}
> (print-n-dashes 5)
-----
> (print-n-dashes 12)
--------
\end{verbatim}

(Printing Rectangles) For this part, you must define a function called \textit{print-rectangle}, that takes two inputs, both of which are non-negative integers. This function should not generate any output value. Instead, it should have the following side effect: It should display a rectangular pattern of dashes whose number of rows and number of columns correspond to the two numerical inputs, as illustrated below.

\begin{verbatim}
> (print-rectangle 5 2)
--
--
--
--
--
> (print-rectangle 2 5)
-----
Here are some hints:

- Pay attention to which input specifies the number of rows, and which specifies the number of columns.
- Use recursion.
- Use print-n-dashes as a helper function to print individual rows.

(Printing Squares) Define a function, called print-square, that takes a single non-negative integer as its only input. It should not generate any output value, but instead should print out a square pattern of dashes in the Interactions Window, whose number of rows and columns is specified by the single numerical input, as illustrated below.

```scheme
> (print-square 3)
---
---
---
> (print-square 4)
----
----
----
----
```

Hint: Use the print-rectangle function as a helper. Your print-square function should not be complicated!

---

Problem 12.8: Printing upside-down triangles

Copy-and-paste the contract and definition for the print-n-dashes function from Example 12.2 into your Definitions Window. Then define a function, called print-upside-down-triangle, that satisfies the following contract:

```scheme
;; PRINT-UPSIDE-DOWN-TRIANGLE
;; ------------------------------------
;; INPUT: NUM-ROWS, a non-negative integer
;; OUTPUT: Nothing
;; SIDE EFFECT: Prints an upside-down triangle in the
;; Interactions Window consisting of NUM-ROWS rows.
```

Here are some examples of its behavior:

```scheme
> (print-upside-down-triangle 3)
---
--
-
> (print-upside-down-triangle 5)
------
----
---
--
```
Note that this is very similar to printing a rectangle (cf. Problem 12.7), except that the width of the row decreases with each recursive function call.

**Problem 12.9: Printing rightside-up triangles**

Copy-and-paste the contract and definition for the `print-n-dashes` function from Example 12.2 into your Definitions Window. Then define a function, called `print-rightside-up-triangle`, that satisfies the following contract:

```scheme
;; PRINT-RIGHTSIDE-UP-TRIANGLE
;; ------------------------------------
;; INPUTS: NUM-ROWS, a non-negative integer
;; CURR-WIDTH, the width of the current row
;; OUTPUT: Nothing
;; SIDE EFFECT: Prints a rightside-up triangle in the Interactions Window consisting of NUM-ROWS rows.
```

Here are some examples illustrating its behavior:

```
> (print-rightside-up-triangle 3 1)
--
---

> (print-rightside-up-triangle 5 1)
--
---
----
-----
```

Notice that this function is called with `curr-width` equal to 1, because that’s the width of the first row to be printed.

**Problem 12.10**

Suppose that `func` is a function whose output values are within some small non-negative range, say, from 0 to 50. For example, suppose that `(func 3)` evaluates to 25. That output value could be represented graphically by a horizontal line containing 25 asterisks. Similarly, if `(func 4)` evaluates to 16, then the next line of printing could show 16 asterisks. Your job is to define a function, called `plotter`, that plots the output values of a given function over a specified range of inputs. Here’s the contract:

```scheme
;; PLOTTER
;; ----------------------------------------------------------------
;; INPUTS: FUNC, a function that expects a numerical input
;; FROM, a starting input value (an integer)
;; TO, an ending input value (an integer)
;; OUTPUT: None
;; SIDE EFFECT: Displays the output vaules of FUNC for each
```
input in the range, FROM, FROM+1, FROM+2, ..., TO-2, TO-1, TO.
For each input value, the corresponding output value is
displayed by the appropriate number of asterisks printed on a
single line of the Interactions Window.

Here is an example that uses facty from Example 12.1.1:

```
> (plotter facty 0 4)
*  
*  
** 
*****
************************
```

And here is an example using abs, a built-in function that computes the absolute value of its input. (Notice what happens when abs is given an input of zero.)

```
> (plotter abs -3 3)
***  
**  
*  
*  
**  
***  
```

Finally, here’s an example where that uses lambda to create a squaring function on the spot, without bothering to give it a name!

```
> (plotter (lambda (x) (* x x)) 1 5)
*  
*****  
******
********
**********
```

Although you can run the above examples in the Interactions Window, you should also put the corresponding alt-tester expressions in your Definitions Window. Insert additional alt-tester expressions to demonstrate that your plotter function works as desired.

### Problem 12.11

⇒ This problem assumes that you have already defined the print-thing-n-times function from Problem 12.5. That function can be used to print a line of + signs, or a line of – signs, in the Interactions Window.

Define a tail-recursive function called fancy-plotter that satisfies the following contract:

```
;; FANCY- PLOTTER
;; ----------------------------------------
;; INPUTS: FUNC, a function that takes numerical input
;; FROM, a starting number (integer)
```
Here are some examples, one of which uses the built-in \texttt{sin} function:

\begin{verbatim}
> (fancy-plotter (lambda (x) (* x x x)) -3 3)
---------------------------
--------
-
0
+
+++++++ 
xxxxxxxxxxxxxxxxxxxxxxxx

> (fancy-plotter (lambda (x) (* 20 (sin (/ x 4)))) -20 20)
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
0
+++++++ 
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx

\end{verbatim}
12.3 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 $3!$, $2!$, and $1!$, respectively. This section explores a slightly different way of organizing recursive computations using accumulators.

* An accumulator is nothing more than an input parameter that is used, in effect, to incrementally accumulate the result of a desired computation.

As each recursive function call is made, the value of the accumulator gets closer and closer to the desired output value, until finally, when the base case is reached, the accumulator holds the desired answer. Accumulator-based recursive functions are typically tail recursive. This section explores the use of accumulators in tail-recursive functions.

Example 12.3.1: 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)\n")
     0)
     ;; Recursive Case: n > 0
     (#t
      (printf "Recursive Case (n=\nA) \ldots\n" n)
      (+ n (sum-to-n (- n 1))))))
```

As in prior examples, the printf expressions serve only to display information about the recursive function calls; they do not affect the output value, as illustrated below.
> (sum-to-n 3) ;; compute 0 + 1 + 2 + 3
Recursive Case (n=3) ...
Recursive Case (n=2) ...
Recursive Case (n=1) ...
Base Case (n=0)
6

Notice that the evaluation of \((\text{sum-to-n} \ 3)\) involved a sequence of function calls—namely: \((\text{sum-to-n} \ 3), (\text{sum-to-n} \ 2), (\text{sum-to-n} \ 1)\) and \((\text{sum-to-n} \ 0)\).

Example 12.3.2: Computing sums of the form, \(0 + 1 + 2 + \ldots + n\) with an accumulator

Below, we define a function, \(\text{sum-to-n-acc}\), that solves the same problem using an extra input parameter, called an accumulator. The accumulator is like a basket that starts out empty, but incrementally accumulates stuff: when the base case is reached, the accumulator (i.e., the basket) holds the desired answer. Once again, the \printf\ expressions serve only to display useful information; they do not affect the output value.

;;; SUM-TO-N-ACC
;;; --------------------------------------------------
;;; INPUTS: N, a non-negative integer
;;; ACC, a number (an accumulator)
;;; OUTPUT: When called with ACC=0, the output is the value
;;; 0 + 1 + 2 + \ldots + N.
;;; More generally, the output is the value of
;;; ACC + 0 + 1 + 2 + \ldots + N.

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

Since the function, \(\text{sum-to-n-acc}\), includes an extra input parameter, we need to supply the values for both \(n\) and \(\text{acc}\) when calling this function. Thus, to compute the sum, \(0 + 1 + 2 + 3\), using this function, we would evaluate the expression, \((\text{sum-to-n-acc} \ 3 \ 0)\). 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 \((\text{sum-to-n-acc} \ 3 \ 0)\) 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)
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:

$$\text{(sum-to-n-acc } (- n 1) (+ \text{ acc } n))$$

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. 12.2 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 12.3.3: 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="A)"" acc)
      ;; Return the accumulator!
      acc)
\begin{figure}
\centering
\begin{tikzpicture}
\node[draw] (n) {\texttt{n} \quad 3};
\node[draw, below=of n] (acc) {\texttt{acc} \quad 0};
\node[draw, below=of acc] (nacc) {\texttt{nacc} \quad 0};
\node[draw, below=of nacc] (sum-to-n-acc) {\texttt{sum-to-n-acc} \quad (+ acc n)};
\node[draw, below=of sum-to-n-acc] (rec) {\texttt{sum-to-n-rec} \quad (+ acc n)};
\node[draw, below=of rec] (base) {\texttt{sum-to-n-rec} \quad (+ acc 0)};
\draw[->] (n) -- (acc);\node[above=of n, anchor=west] (r1) {\texttt{;; Recursive Case}};
\draw[->] (acc) -- (nacc);\node[above=of acc, anchor=west] (r2) {\texttt{;; Recursive Case}};
\draw[->] (nacc) -- (sum-to-n-acc);\node[above=of nacc, anchor=west] (r3) {\texttt{(sum-to-n-acc (- n 1))}};
\draw[->] (sum-to-n-acc) -- (rec);\node[above=of sum-to-n-acc, anchor=west] (r4) {\texttt{(sum-to-n-acc (- n 1))}};
\draw[->] (rec) -- (base);\node[above=of rec, anchor=west] (r5) {\texttt{(sum-to-n-acc (- n 1))}};
\node[above=of rec, anchor=west] (r6) {\texttt{;; Base Case}};
\node[above=of base, anchor=west] (r7) {\texttt{acc}};
\end{tikzpicture}
\caption{DrScheme’s evaluation of \texttt{(sum-to-n-acc 3 0)}}
\end{figure}
An expression of the form, \((\text{facty-acc } n 1)\), 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, \(\text{facty-acc}\), is tail recursive, as evidenced by the fact that the recursive function-call expression, \((\text{facty-acc } (- n 1) (* n acc))\), 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.

For \(\text{facty-acc}\), the current accumulator, \(acc\), is multiplied by \(n\) to generate the value of the accumulator for the recursive function call. Since \(\text{facty-acc}\) 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 \(\text{facty-acc}\) to compute \(4!\), we would evaluate an expression such as \((\text{facty-acc } 4 1)\), as illustrated below:

\[> (\text{facty-acc } 4 1)\]

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, \(\text{facty-acc}\), is more general, in that it allows the accumulator to have values other than 1:

\[\star \text{ The output value for } (\text{facty-acc } n acc) \text{ is equal to the factorial of } n \text{ multiplied by } acc.\]

Notice that if \(acc\) equals 1, then the output value is indeed \(n!\). However, if \(acc\) is something other than 1, then the value is \(n! \times acc\).

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

---

**Example 12.3.4: Summing squares**: \(1^2 + 2^2 + \ldots + n^2\)

Here’s a tail-recursive function for computing the sums of squares from 1 to \(n\):
;; More generally, the output is the sum:
;; \( ACC + 0 \times 0 + 1 \times 1 + 2 \times 2 + \ldots + N \times N. \)

(define sum-squares-acc
  (lambda (n acc)
    (cond
     ;; Base Case: \( n \leq 0 \)
     ((<= n 0)
      (printf "Base Case: n=\~A, acc=\~A\~%" n acc)
      acc)
     ;; Recursive Case: \( n > 0 \)
     (#t
      (printf "Recursive Case: n=\~A, acc=\~A\~%" n acc)
      (sum-squares-acc (- n 1) (+ acc (* n n))))))

Notice that the function is clearly tail recursive, since the recursive function-call expression, by itself, is the last expression in the recursive case. (It is not a subsidiary expression within some larger computation.) Notice, too, that the accumulator is initially 0. Finally, notice that the value of the accumulator for the recursive function call is the original accumulator plus \( n^2 \). In other words, each recursive function call involves accumulating a squared term.

Here’s the result of evaluating the expression, \((\text{sum-squares-acc } 3 \ 0)\), in the Interactions Window:

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

Notice that by the time the base case is reached, the accumulator holds the desired answer—in this case, 14—for the original computation. You should convince yourself that 14 is the output value for each of the recursive function calls shown above.

Although the function returns the desired output value when the accumulator is 0, the following is a more general characterization of this function’s behavior:

* An expression of the form, \((\text{sum-squares-acc } n \ acc)\), evaluates to \( 0^2 + 1^2 + \ldots + n^2 + acc \).

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 12.3.5: 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, \( \text{exp-acc} \), 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 represents the maximum exponent to be encountered in the sum, will
stay the same across all recursive function calls;

- the input parameter `curr-exp` (i.e., the current exponent) will take on the values `0, 1, 2, ..., n`;
- the input parameter `curr-denom` (i.e., current denominator) will accumulate the factorials that comprise the various denominators in the terms in the sum (i.e., it will take on the values `1, 1, 2, 6, 24, ..., n!`); and
- the input parameter `acc` will accumulate the desired sum (i.e., it will take on the values `1, 2, 2.5, 2.6666666666666666, ...`).

;; EXP-ACC
;; ---------------------------------------------------------------
;; INPUTS: N, number (non-negative integer, maximum exponent)
;; CURR-EXP, number (non-negative int., current exponent)
;; CURR-DENOM, number (accumulator for curr. denom.)
;; ACC, number (accumulator for desired answer)
;; OUTPUT: When called with CURR-EXP=0, CURR-DENOM=1, and ACC=0, the output is the value of the sum:
;; 1 + 1/(1!) + 1/(2!) + 1/(3!) + ... + 1/(N!)
(define exp-acc
  (lambda (n curr-exp curr-denom acc)
    ;; Print out the values of the input parameters first...
    (printf "n=\%, curr-exp=\%, curr-denom=\%, acc=\%\" n curr-exp curr-denom acc)
    (cond
      ;; Base Case: curr-exp > n
      ((> curr-exp n)
        ;; Return the accumulator!
        acc)
      ;; Recursive Case: curr-exp <= n
      (#t
        ;; Make recursive function call with updated inputs
        (exp-acc n ; n doesn’t change
          (+ 1 curr-exp) ; update curr exp
          (+ (+ 1 curr-exp) curr-denom) ; update curr denom
          (+ acc (/ 1.0 curr-denom)) ; update acc
          ))))))

To get the desired results, the various input parameters must be properly initialized. In particular, the initial values for `curr-exp`, `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 `(exp-acc 4 0 1 0)`, as illustrated below:

\[
\begin{align*}
\text{n=4, curr-exp=0, curr-denom=1, acc=0} \\
\text{n=4, curr-exp=1, curr-denom=1, acc=1.0} \\
\text{n=4, curr-exp=2, curr-denom=2, acc=2.0} \\
\text{n=4, curr-exp=3, curr-denom=6, acc=2.5} \\
\text{n=4, curr-exp=4, curr-denom=24, acc=2.6666666666666665} \\
\text{n=4, curr-exp=5, curr-denom=120, acc=2.7083333333333333} \\
\text{2.7083333333333333}
\end{align*}
\]
Notice that the \( n \) parameter stays fixed at 4 across all the recursive function calls. This parameter is used only to distinguish the base case from the recursive case. The parameter \( \text{curr-exp} \) represents the exponent of the term currently being worked on; thus, it starts at 0 and increments by 1 each time. The parameter \( \text{curr-denom} \) represents the denominator of the term currently being worked on; thus, it takes on the values of successive factorials: \( 0!, 1!, 2!, 3!, \ldots \). The parameter \( \text{acc} \) accumulates the desired sum. By the time the base case is reached (i.e., when \( \text{curr-exp} > n \) in the last line), the accumulator holds the desired answer. Thus, the accumulator is simply returned as the output value for this function.

If the \( \text{printf} \) 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:

\[
\begin{align*}
> & \ (\text{exp-acc} \ 20 \ 0 \ 1 \ 0) \\
& 2.7182818284590455 \\
\end{align*}
\]

12.4 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. A wrapper function is a function 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 12.4.1: A wrapper for \texttt{facy-acc}

The following defines a wrapper function, \texttt{facy-wr}, for the accumulator-based function, \texttt{facy-acc}, defined earlier. Notice that the wrapper function simply calls \texttt{facy-acc} with the accumulator appropriate initialized to 1.

\[
\begin{align*}
; ; & \ \text{FACTY-WR} \\
; ; & \ \text{---------------------------------------} \\
; ; & \ \text{INPUT: N, a non-negative integer} \\
; ; & \ \text{OUTPUT: The factorial of N (i.e., N!)} \\
\text{(define facy-wr}
\text{ (lambda (n)}
  \text{  ; Just call accumulator-based helper with ACC=1}
  \text{    (facy-acc n 1))}
\text{)}
\end{align*}
\]

The following Interactions Window session demonstrates how the wrapper function shields the user from the accumulator. In fact, the user of \texttt{facy-wr} may not even be aware that an accumulator is being used at all.

\[
\begin{align*}
> & \ (\text{facy-wr} \ 3) \\
& 6 \\
> & \ (\text{facy-wr} \ 4) \\
& 24 \\
> & \ (\text{facy-wr} \ 5) \\
& 120 \\
\end{align*}
\]
Example 12.4.2: A wrapper for exp-acc

The function, exp-acc, seen earlier, involved several accumulators. The following wrapper function, exp-wr, shields the user from having to know the appropriate initial values for these accumulators:

;;; EXP-WR
;;; ---------------------------------------
;;; INPUT: N, a non-negative integer
;;; OUTPUT: The value of the sum:
;;; 1/1! + 1/2! + 1/3! + ... + 1/N!

(define exp-wr
  (lambda (n)
    (exp-acc n 0 1 0)))

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

> (exp-wr 4)
2.708333333333333
> (exp-wr 5)
2.7166666666666663
> (exp-wr 6)
2.7180555555555554
> (exp-wr 100)
2.7182818284590455

Notice that the user of exp-wr may not even be aware that accumulators are being used!

12.5 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.

\[
\begin{align*}
g(5) &= 1^2 + 2^2 + 3^2 + 4^2 + 5^2 \\
     &= (1^2 + 2^2 + 3^2 + 4^2) + 5^2 \\
     &= g(4) + 5^2
\end{align*}
\]

In turn, this suggests that \( g(n) = g(n - 1) + n^2 \) for each \( n > 1 \), which leads to the following solution in Scheme:

\[
\text{(define sum-squares (lambda (n)}
  \text{(cond)}
  \quad ;; Base Case: N <= 1
  \quad (\text{if} \ n \text{ } 1)
  \quad ;; Recursive Case: N > 1
  \quad (else
    \text{(+ (sum-squares (- n 1)) (* n n))))}
\]

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.

\[
\text{(define print-n-dashes (lambda (n))}
  \text{(cond)}
  \quad ;; Base Case: N <= 0
  \quad (\text{if} \ n \text{ } 0)
  \quad (\text{newline})
  \quad ;; Recursive Case: N > 0
  \quad (else
    \text{(printf "-"))}
  \quad \text{(print-n-dashes (- n 1)))})
\]

Notice that, if the recursive case is followed, the last expression in that case, \((\text{print-n-dashes (- n 1))}\), 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.

\[
\text{(define facty-acc (lambda (n acc))}
  \text{(cond)}
  \quad ;; Base Case: N <= 1
\]
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 \((\star n acc)\). 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 \texttt{facty-acc} function would take care of calling \texttt{facty-acc} with an initial value of \(1\) for \(acc\).

### Built-in Functions Introduced in this Chapter

- \texttt{even?}: Returns \#t if its input is an even number
- \texttt{odd?}: Returns \#t if its input is an odd number
- \texttt{sin}: Returns the sine of its input
- \texttt{log}: Returns the natural logarithm of its input

### Problems

**Problem 12.12**

For this problem, you will implement a \texttt{print-checkerboard} function that displays a checkerboard pattern in the Interactions Window.

(a) Define a tail-recursive function called \texttt{print-checkerboard-acc} that takes four inputs: \texttt{num-rows}, \texttt{num-cols}, \texttt{curr-row} and \texttt{curr-col}. \texttt{num-rows} and \texttt{num-cols} specify the overall size of the checkerboard; \texttt{curr-row} and \texttt{curr-col} specify the location of the next square to be printed.

When called with appropriate initial values for \texttt{curr-row} and \texttt{curr-col}, this function should cause a \texttt{num-rows-by-num-cols} checkerboard pattern to be printed in the Interactions Window.

- The values of \texttt{num-rows} and \texttt{num-cols} should not change across the various recursive function calls, but the values of \texttt{curr-row} and \texttt{curr-col} will change.
- If the current square is somewhere in the middle of the board, this function should print just that one square. It should then let the recursive function call print the rest of the checkerboard. (How should the values of \texttt{curr-row} and \texttt{curr-col} be updated in this case?)
- If the sum of \texttt{curr-row} and \texttt{curr-col} is even, then print one kind of square (e.g., X); if their sum is odd, then print the other kind of square (e.g., _). (You may use the built-in functions, \texttt{even?} and \texttt{odd?}, to test whether a given number is even or odd.)
- How do you recognize that you have already finished printing out the entire checkerboard (i.e., you’ve hit the base case)?
- How do you recognize that you have finished printing out the current row? How should the values of \texttt{curr-row} and \texttt{curr-col} be updated in that case?
- Summary of cases to consider:
  * You’re in the middle of a row.
(b) Define a wrapper function called \textit{print-checkerboard} that takes two inputs, \textit{num-rows} and \textit{num-cols}. It should cause a \textit{num-rows}-by-\textit{num-cols} checkerboard pattern to be displayed in the Interactions Window, as illustrated below:

\begin{verbatim}
> (print-checkerboard 3 6)
X _ X _ X _
_ X _ X _ X
X _ X _ X _
\end{verbatim}

Note that this function should just call the function from part (a) with appropriate inputs. You may wish to use the \textit{tester-alt} function when writing test cases in the Definitions Window.

Problem 12.13

The following functions use the built-in \textit{quotient} and \textit{remainder} functions to access the individual digits in the base-ten representation of a number. For the purposes of this problem, the rightmost digit in a number will be considered to be in position zero, the next rightmost digit in position one, and so on. For example, the 3 in 9999399 will be considered to be in position 2.

(a) Define a function called \textit{nth-rightmost-digit} that satisfies the following contract:

\begin{verbatim}
;; NTH-RIGHTMOST-DIGIT
;; ---------------------------------------------------
;; INPUTS: NUM, a non-negative integer
;; N, a non-negative integer
;; OUTPUT: The Nth rightmost digit of NUM, where N=0 refers
to the rightmost digit.
\end{verbatim}

Here are some examples of the desired behavior:

\begin{verbatim}
> (nth-rightmost-digit 92845 0) 5
> (nth-rightmost-digit 92845 2) 8
\end{verbatim}

\textit{Hint:} Consider how dividing a number by ten, using the built-in \textit{quotient} and \textit{remainder} functions, can effectively “peel off” the rightmost digit of the number.

\begin{verbatim}
> (quotient 345678 10) 34567
> (remainder 345678 10) 8
\end{verbatim}

(b) Define a tail-recursive, accumulator-based function called \textit{num-occurs-acc} that satisfies the following contract:

\begin{verbatim}
;; NUM-OCCURS-ACC
;; -------------------------------------
;; INPUTS: DIGIT, an integer from 0 to 9, inclusive
\end{verbatim}
;; NUM, a non-negative integer
;; ACC, an accumulator
;; OUTPUT: When called with ACC = 0, the output is the number
;; of occurrences of DIGIT in the decimal repr’n of NUM.
;; Example: (num-occurs-acc 3 32123334 0) ==> 4

After you have done so, then define the following “wrapper” function:

;; NUM-OCCURS-WR -- wrapper function for NUM-OCCURS-ACC
;; --------------------------------------------------------
;; INPUTS: DIGIT, an integer from 0 to 9, inclusive
;; NUM, a non-negative integer
;; OUTPUT: The number of occurrences of DIGIT in the decimal
;; representation of NUM.

(define num-occurs-wr
  (lambda (digit num)
    (num-occurs-acc digit num 0)))

Here are some examples of the desired behavior:

> (num-occurs-wr 3 12312344444443)
3
> (num-occurs-wr 0 100)
2
> (num-occurs-wr 5 1234)
0

Hint 1: Use the built-in quotient and remainder functions. In the context of this problem, consider the following examples:

> (quotient 345678 10)
34567
> (remainder 345678 10)
8

So, dividing by ten each time allows you to effectively “peel off” the rightmost digit.

Hint 2: The base case should be when you have exactly one digit left (i.e., when num ≤ ten).

Problem 12.14: Approximating the natural logarithm function

Mathematicians tell us that the natural logarithm function can be approximated using certain kinds of sums. In particular, for any real number \( x \) ∈ \((-1, 1]\), and for any “sufficiently large” positive integer \( n \), the value \( \log(1 + x) \) is well approximated by the following sum:

\[
x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots \pm \frac{x^n}{n}
\]

For example, the value \( \log(1.5) \), where \( x = 0.5 \), is well approximated by the sum:

\[
0.5 - \frac{(0.5)^2}{2} + \frac{(0.5)^3}{3} - \frac{(0.5)^4}{4}
\]
Admittedly, these facts are probably not obvious! But that’s okay, since this is not a math class! We can just accept what the mathematicians tell us and go about the business of computing these kinds of sums. So... for this problem, define an accumulator-based function, called log-acc that satisfies the following contract. Your function should be tail-recursive. Also, you may want to use the built-in even? and odd? functions, seen previously. The power function, from Problem 12.1, should be helpful.

;;; LOG-ACC
;;; --------------------------------
;;; INPUTS: X, any number such that -1 < X <= 1
;;; FROM, the index of the "current term" in the sum
;;; TO, the index of the "last term" in the sum
;;; ACC, an additive accumulator
;;; OUTPUT: When called with FROM=1, ACC=0, and TO=N the output is the sum: X - X*X/2 + X*X*X/3 - ... (+/-)XˆN/N

After you have defined log-acc, define the following wrapper function:

(define log-wr
  (lambda (x to)
    ;; Call the acc-based helper function with FROM=1 and ACC=0
    (log-acc x 1 to 0.0)))

Notice that (log-wr 0.5 4) should compute the sum seen earlier that is supposed to be a good approximation of \(\log(1.5)\). To verify these sorts of examples, you can use Scheme’s built-in log function, but keep in mind that the above sums are good approximations for \(\log(1 + x)\), not for \(\log(x)\). So, for example, the expression (log-wr 0.5 4) will evaluate to a good approximation of \(\log(1.5)\), since \(1 + 0.5 = 1.5\).

> (log-wr 0.5 10) ←- \(x = 0.5\)
0.4054346478174603
> (log 1.5) ←- \(1 + x = 1.5\)
0.4054651081081644

Although log-wr will be good at estimating values of \(\log(1 + x)\) for small values of \(x\), it doesn’t do so well as the value of \(x\) approaches 1. Compare the values returned by (log-wr 1 n) for various values of \(n\) against the value of (log 2). How big does \(n\) have to be before the answer is correct to within 3 decimal places? Be sure to include a variety of tester expressions in your definitions file to see how well expressions of the form (log-wr \(x\) \(n\)) approximate \(\log(1 + x)\) for various values of \(x\) and \(n\).

Problem 12.15: Computing geometric sums

This problem concerns the computation of sums such as those shown below:

1 + 10 + 10² + 10³ = 1 + 10 + 100 + 1000 = 1111
1 + 2 + 2² + 2³ + 2⁴ = 1 + 2 + 4 + 8 + 16 = 31
1 + 3 + 3² + 3³ = 1 + 3 + 9 + 27 = 40

More generally, for any number \(x\) and any non-negative integer \(n\), the following expression is called a geometric sum:

\[1 + x + x² + x³ + \ldots + x^n\]

where terms of the form \(x^k\) stand for “\(x\) raised to the \(k\)th power”. Your job is to define a function, called geom, that takes two inputs, \(x\) and \(n\), and whose output is the value of the corresponding geometric sum, as shown above.
Now, there are lots of ways to do this problem. Here, we are going to focus on a way that involves defining an accumulator-based tail-recursive helper function, called `geom-helper-acc`, that takes the following additional inputs:

- k, a counter that goes from 0 up to n
- x-to-the-k, a variable that takes on the values, 1, x, x^2, x^3, etc.
- acc, a variable that accumulates the desired sum

(When I say that k is a counter “that goes from 0 up to n”, I really mean that the value of the input k on successive recursive function calls increases by one each time.)

Consider the case where x = 2 and n = 4. The sum we want to compute is: 1 + 2 + 2^2 + 2^3 + 2^4, which happens to be equal to 31. Here are the successive values we want the variables, k, x-to-the-k and acc to take on during successive recursive function calls:

<table>
<thead>
<tr>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-to-the-k</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>acc</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>15</td>
<td>31</td>
</tr>
</tbody>
</table>

← We’ll stop here, since k > 4
← That’s the desired answer!!

As suggested earlier, the value of k increases by one for each successive recursive function call; x-to-the-k is multiplied by x each time; and acc accumulates the most recent value of x-to-the-k. In particular, the value of acc in one column is the sum of the values of x-to-the-k and acc from the preceding column:

1 + 0 ⇒ 1; 2 + 1 ⇒ 3; 4 + 3 ⇒ 7; 8 + 7 ⇒ 15; 16 + 15 ⇒ 31.

Okay, you are now ready to define the accumulator-based, tail-recursive helper function, `geom-helper-acc`. It should take the following inputs: x, n, k, x-to-the-k and acc.

* For the base case... when should this function stop?

* For the recursive case... make a tail-recursive function call with appropriately adjusted inputs.

Afterward, you can define `geom` as a wrapper function that simply calls the above helper function with appropriate initial values for its five inputs. Here’s how it should work in the end:

```
> (geom-sum 10 3)
1111
> (geom-sum 2 4)
31
```

As always, be sure to include contracts for each function you define.
Chapter 13

Defining Local Variables with the \texttt{let}, \texttt{let*} and \texttt{letrec} Special Forms

This chapter introduces the \texttt{let} special form along with its more general variants, \texttt{let*} and \texttt{letrec}. The purpose of a \texttt{let} special form is to set up a local environment that is populated with local variables, just like the local environments that exist within function-call boxes. That local environment provides a temporary context for the evaluation of the expressions in the body of the \texttt{let} special form. The value of the last expression in its body is taken to be the value of the entire \texttt{let} expression. Once a \texttt{let} expression has been evaluated, its local environment typically vanishes.\footnote{As will be seen, the \texttt{let*} special form can do everything that a \texttt{let} can do, plus a little bit more. Similarly, a \texttt{letrec} special form can do everything that a \texttt{let*} can do, plus a little bit more. Thus, the \texttt{let} special form is the most basic of the three.}

As will be seen, the \texttt{let*} special form can do everything that a \texttt{let} can do, plus a little bit more. Similarly, a \texttt{letrec} special form can do everything that a \texttt{let*} can do, plus a little bit more. Thus, the \texttt{let} special form is the most basic of the three.

13.1 The \texttt{let} Special Form

The purpose of the \texttt{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 \texttt{let}. A \texttt{let} special form is often used to store the value of some lengthy computation in a local variable, after which that value can be accessed as many times as needed without having to re-do the lengthy computation over and over again. For example, suppose it takes a year to compute some desired numerical value. You wouldn’t want to have to re-do that year-long computation each time you wanted to print out that value. It would be much more efficient to store the computed value in a local variable and then refer to that stored value as often as desired. Furthermore, it is not desirable to overpopulate the Global Environment with values that may only be needed for a brief time. It is preferable to create local variables to store values for only as long as they are needed.

13.1.1 The Syntax of the \texttt{let} Special Form

The syntax of the \texttt{let} special form is as follows:

\begin{verbatim}
(let ((var1 val1)
     (var2 val2)
     ...
     (varn valn))
expr1
expr2
...
exprk)
\end{verbatim}

\footnote{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.}
where:

- \( \text{var}_1, \ldots, \text{var}_n \) are character sequences representing \( n \) distinct Scheme symbols, where \( n \geq 0 \);
- \( \text{val}_1, \ldots, \text{val}_n \) are \( n \) Scheme expressions of any kind; and
- \( \text{expr}_1, \ldots, \text{expr}_k \) are \( k \) Scheme expressions of any kind, where \( k \geq 1 \).

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

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

### Example 13.1.1: Some legal \text{let} expressions

The following expressions are all legal \text{let} expressions:

\[
\begin{align*}
\text{(let () #t)} \\
\text{(let ((x (+ 2 3)))} \\
&\quad (* x x)) \\
\text{(let ((x (+ 2 3))} \\
&\quad (y 3) \\
&\quad (z (* 2 2)))} \\
&\quad (\text{printf } "x: \%A, y: \%A, z: \%A\%\" x y z) \\
&\quad (+ x y z))
\end{align*}
\]

The first \text{let} expression includes no \text{var}/\text{val} pairs, as indicated by the empty list. Its body consists of the single expression, \#t. The second \text{let} expression includes a single \text{var}/\text{val} pair: \((x (+ 2 3))\). Its body consists of the single expression, \((* x x)\). The third \text{let} expression includes three \text{var}/\text{val} pairs: \((x (+ 2 3)), (y 3)\) and \((z (* 2 2))\). Its body consists of two expressions: a \text{printf} expression and \((+ x y z)\).

### 13.1.2 The Semantics of the \text{let} Special Form

Like any special form expression, a \text{let} special form expression denotes a list. The more interesting part of the semantics of a \text{let} special form is how it is evaluated. A \text{let} expression of the form

\[
\begin{align*}
\text{(let ((\text{var}_1 \text{val}_1) \\
&\quad (\text{var}_2 \text{val}_2) \\
&\quad \ldots \\
&\quad (\text{var}_n \text{val}_n))} \\
&\quad \text{expr}_1 \\
&\quad \text{expr}_2 \\
&\quad \ldots \\
&\quad \text{expr}_k)
\end{align*}
\]

is evaluated as follows.

- First, the expressions, \( \text{val}_1, \ldots, \text{val}_n \), are evaluated.
- Second, a local environment is created containing \( n \) entries—one for each of the \text{var}/\text{val} pairs in the \text{let} expression. In particular, each symbol \( \text{var}_i \) is associated with the result of evaluating the corresponding \text{val}_i expression.
• Third, the expressions, $expr_1, \ldots, expr_k$, in the body of the let special form are evaluated, in turn, with respect to that newly created local environment. Thus, in the process of evaluating these expressions, if any symbol $var_i$ ever needs to be evaluated, its value is drawn from the newly created local environment. For other symbols, the parent environment—which is often the Global Environment—is used.

• The value of the last expression, $expr_k$, is the value of the entire let expression.

**Example 13.1.2: Evaluating 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: ~A, y: ~A, z: ~A" 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, the local environment contains a single entry that associates the value 5 with the symbol x. Notice the plethora of parentheses required to represent a list containing a single entry that is itself a list! Furthermore, the second entry in that subsidiary list is itself a list! The body of the let consists of the single expression, (* x x), which evaluates to 25 in this context. Notice that 25 is reported as the value of the entire let expression.

In the third expression, the local environment contains three entries: one associating the value 5 with x, one associating the value 3 with y, and one associating the value 4 with z. The body contains two expressions. The printf expression causes information to be displayed in the Interactions Window; the expression (+ x y z) is then evaluated, resulting in the value 12, which is reported as the value for the entire let expression.

**Example 13.1.3: Local vs. Global**

The following Interactions Window session demonstrates that the local environment supercedes the Global Environment when evaluating expressions in the body of a let special form.

```
> (define x 1000)
> (define y 100)
> (define z 10)
> (+ x y z)
1110
> (let ((x 3)
```
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.

The following example introduces a **destructive** built-in function called `random` that has many uses, one of which is to demonstrate the need for the `let` special form. I know... this part of the book is supposed to only deal with non-destructive functions. But, this one exception is too much fun to postpone any further.

### Example 13.1.4: 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 13.1.6, below).

The `random` function satisfies the following contract.

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

Here are some examples demonstrating its behavior:

```scheme
> (random 2) ← output will be 0 or 1
0
> (random 2)
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, ..., \( n-1 \}\).
There’s an entire field of Computer Science that deals with so-called randomized algorithms (i.e., algorithms whose computations depend on pseudo-random generators). Randomized algorithms can often be surprisingly efficient.

Example 13.1.5: 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}
;; Note: Since (RANDOM 6) generates a number in {0,1,2,3,4,5},
;; we must add one to simulate the toss of a die.

(define toss-die
  (lambda ()
    (+ 1 (random 6)))))

Here are some examples of their use:

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

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

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

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

---

**Example 13.1.6: Using let to store a randomly generated value**

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

```scheme
> (printf "My toss: \^A\%" (toss-die))
3
> (printf "The square of my toss: \^A\%" (* (toss-die) (toss-die)))
10
```

Why? Because each time DrScheme evaluates `(toss-die)`, it may generate a different value. To get the desired behavior, you need some way of storing the value of a single toss, so that you may then refer to it as often as you like. In short, you need a `let` special form, as illustrated below:

```scheme
> (let ((toss (toss-die)))
  (printf "My toss: \^A\%" toss)
  (printf "The square of my toss: \^A\%" (* toss toss))
  (* 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.)

---

### 13.1.3 Deriving the let Special Form from the lambda Special Form

If you’re thinking that the evaluation of a `let` special form seems awfully close to the evaluation of a function call, you’re right. In fact, each `let` special form expression is simply a convenient abbreviation for an expression in which a `lambda` function is applied to some input values. Before going into all the details, we give some examples illustrating the equivalence of expressions involving `let` and `lambda`. 

---
Example 13.1.7

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

```scheme
> (let ((x (+ 2 3))
    (y (* 3 4)))
  (printf "x: \dollar A, y: \dollar A" x y)
  (+ x y))
x: 5, y: 12
17
> ((lambda (x y)
    (printf "x: \dollar A, y: \dollar A" x y)
    (+ x y))
  (+ 2 3)
  (* 3 4))
x: 5, y: 12
17
```

⇒ The semantics for the evaluation of the first expression is identical to the semantics for the evaluation of the second expression!

In particular, for the let expression, a local environment is set up in which the symbol x is associated with the value 5 and the symbol y is associated with the value 12. After that, the two expressions in the body of the let are evaluated with respect to that local environment yielding some side-effect printing and an output value of 17.

The evaluation of the second expression is governed by the Default Rule for evaluating non-empty lists. The first entry in the list is a lambda expression. It evaluates to a function. The other entries, (+ 2 3) and (* 3 4), evaluate to the numbers 5 and 12, respectively. When that function is applied to those inputs, a local environment is set up in which x and y are associated with the values 5 and 12, respectively. Then the body of the lambda is evaluated, yielding side-effect printing and the output value 17.

Example 13.1.8

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

```scheme
> (define z 1000)
> (let ((x 3)
    (y 4))
  (* x y z))
12000
> ((lambda (x y)
    (* x y z))
  3
  4)
12000
```

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

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

is equivalent to the following expression involving the application of a \texttt{lambda} function:

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

The reason we have \texttt{let} expressions is that they have a friendlier syntax for the cases where you want to create a local environment and then evaluate some expressions with respect to that local environment.

\textbf{Problems}

\begin{tcolorbox}
\textbf{Problem 13.1}

\textit{This problem has two parts. The first part can be implemented without using \texttt{let}; the second part is best implemented using \texttt{let}.}

(a) Define a function called \texttt{print-n-tosses} that satisfies the following contract:

\begin{verbatim}
;; PRINT-N-TOSSES
;; ----------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: None
;; SIDE EFFECT: Prints the results of N random tosses of a six-sided die in the Interactions Window.
\end{verbatim}

Here are some examples:

\begin{verbatim}
> (print-n-tosses 10)
5 2 6 4 5 6 3 2 3 1
> (print-n-tosses 10)
5 2 6 1 5 6 2 5 5 3
> (print-n-tosses 10)
6 2 6 4 5 3 1 2 5 3
\end{verbatim}

(b) Define a function called \texttt{print-and-sum-n-tosses} that satisfies the following contract.

\begin{verbatim}
;; PRINT-AND-SUM-N-TOSSES
;; -------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: The sum of N random tosses of a six-sided die.
;; SIDE EFFECT: Prints out the tosses along the way.
\end{verbatim}
\end{tcolorbox}
Here are some sample interactions:

```
> (print-and-sum-n-tosses 5)
2 2 6 6 2 : 18
> (print-and-sum-n-tosses 5)
3 3 5 3 5 : 19
> (print-and-sum-n-tosses 5)
3 1 5 1 2 : 12
```

Note that the numbers to the left of the "::" are side-effect printing, whereas the numbers to the right of the "::" are output values.

Hints: Define an accumulator-based, tail-recursive function called `print-and-sum-n-tosses-acc` that takes an accumulator `acc` as an extra input. (You may wish to review Section 12.3.) In the recursive case, store the current toss in a local variable before printing it and making the tail-recursive function call. Afterward, define `print-and-sum-n-tosses` as a wrapper function that calls `print-and-sum-n-tosses-acc` with appropriate inputs. (You may wish to review Section 12.4.)

---

**Problem 13.2**

Define a function called `num-occurs-in-n-tosses` that satisfies the following contract.

```
;; NUM-OCCURS-IN-N-TOSSES
;; ---------------------------------------------------------------
;; INPUTS: TARGET, an integer from 1 to 6, inclusive
;; N, a non-negative integer
;; OUTPUT: Reports the number of times the TARGET number showed up when tossing a six-sided die N times.
;; SIDE EFFECT: Prints out the random tosses along the way.
```

Here are some examples of it in action:

```
> (num-occurs-in-n-tosses 3 20)
4 3 3 1 6 3 5 6 4 1 6 5 5 4 3 5 3 3 5 2 ... 6
> (num-occurs-in-n-tosses 3 20)
4 6 5 1 6 5 3 2 3 4 2 4 2 4 4 5 3 6 3 5 ... 4
> (num-occurs-in-n-tosses 3 20)
5 4 5 1 4 2 4 3 5 3 1 1 2 5 5 1 6 4 2 3 ... 3
```

Notice that the numbers to the left of the dot-dot-dots are side-effect printing, whereas the numbers to the right of the dot-dot-dots are output values.

Hint: In the recursive case, use a `let` special form to store the value of the toss of a die. Then print it out and decide whether you hit the target number or not.

Note: You may choose to implement this function using tail recursion or not, as you wish. If using tail recursion, you should name your tail-recursive helper function `num-occurs-in-n-tosses-acc`. It will need an extra input—an accumulator—that accumulates the number of occurrences of the target number over all the tosses. After your accumulator-based helper function is working properly, you should then define a wrapper function, called `num-occurs-in-n-tosses`, that simply calls the accumulator-based function with appropriate inputs. Of course, you may wish to implement both versions!
Problem 13.3: Flipping coins

When flipping coins, any occurrence of \( n \) consecutive coin flips that come out the same (i.e., all \( H \) or all \( T \)) may be called a streak of length \( n \). For this problem, you must define a function, called \( \text{max-streak-in-n-flips} \), that satisfies the following contract:

\[
\text{;; MAX-STREAK-IN-N-FLIPS}
\text{;; ----------------------------------------------}
\text{;; INPUT: N, a non-negative integer}
\text{;; OUTPUT: The length of the longest streak of consecutive}
\text{;; coin flips (whether all H or all T) that occur in a}
\text{;; sequence of N random coin flips.}
\text{;; SIDE EFFECT: Prints out the coin flips along the way.}
\]

Here are some examples of its use:

\[
\begin{align*}
\text{(max-streak-in-n-flips 10)} & \implies T H H H T T H \ldots \\
& \quad \implies 4 \\
\text{(max-streak-in-n-flips 10)} & \implies T H T H T T T H \ldots \\
& \quad \implies 3 \\
\text{(max-streak-in-n-flips 15)} & \implies T T H T H T H T H H H T T H \ldots \\
& \quad \implies 3
\end{align*}
\]

In the first sequence, the longest streak involves four consecutive \( H \)s. In the second and third sequences, the longest streak involves three consecutive \( H \)s.

\* Begin by defining a separate helper function, called \( \text{max-consec-same-acc} \) that takes additional inputs to keep track of things such as: the value of the most recent coin flip, the number of consecutive coin flips that have just come out the same, and the maximum number of consecutive coin flips that have been seen since you started flipping coins. Once you get things working properly, you should then define your wrapper function, \( \text{max-streak-in-n-flips} \).

Consider the following sequence of coin flips:

\[
H T H T T T H H H H T T H \ldots
\]

What information would you need to keep track of along the way to solve this problem?

Problem 13.4

Define a function, \( \text{num-tosses-until-repeat} \), that satisfies the following contract:

\[
\text{;; NUM-TOSSES-UNTIL-REPEAT}
\text{;; ---------------------------------------------}
\text{;; INPUTS: None}
\text{;; OUTPUT: An integer specifying the number of random tosses of a}
\text{;; 6-sided die until two CONSECUTIVE tosses come out the same.}
\text{;; SIDE EFFECTS: Prints out the tosses along the way.}
\]

Here are some examples that illustrate the desired behavior:

\[
> \text{(num-tosses-until-repeat)}
3 5 1 3 4 2 3 6 1 3 2 2 \implies \text{Got a repeat!}
\]
12
> (num-tosses-until-repeat)
2 4 4 --> Got a repeat!
3
> (num-tosses-until-repeat)
1 6 1 4 4 --> Got a repeat!
5

Note that, in each case, the first line of text is side-effect printing, while the second line displays the output value.

Hint: Define a helper function, num-tosses-until-repeat-helper, that does most of the work. It should satisfy the following contract:

;; NUM-TOSSES-UNTIL-REPEAT-HELPER
;; -------------------------------------------
;; INPUTS: NUM-TOSSES-SO-FAR, a non-negative integer
;; MOST-RECENT-TOSS, a non-negative integer
;; OUTPUT & SIDE-EFFECTS similar to NUM-TOSSES-UNTIL-REPEAT

Note that num-tosses-so-far keeps track of how many tosses have been made so far. (It accumulates the number of tosses so far.) In addition, note that most-recent-toss keeps track of the value of the most recently tossed die so that a new toss can be compared to the most recent toss.

In the body of num-tosses-until-repeat-helper, you should toss one die and store its value in a local variable. Then print it out. Then compare this new toss to the most recent toss. If they are the same, then stop; otherwise, keep going—with adjusted inputs.

The num-tosses-until-repeat “wrapper” function should let the helper function do most (or all) of the work.

Problem 13.5

Define a function, called toss-until-doubles, that satisfies the following contract:

;; TOSS-UNTIL-DUBBLES
;; -----------------------------------
;; INPUTS: None
;; OUTPUT: The sum of the first occurrence of "doubles"
;; SIDE EFFECT: Tosses a pair of dice, printing out the results (and their sum), until doubles are encountered!

Here are some examples of its use:

> (toss-until-doubles) ==> TOSSES: 5, 2; sum = 7
TOSSES: 1, 2; sum = 3
TOSSES: 3, 6; sum = 9
TOSSES: 2, 2; sum = 4
HEY! We got doubles!!
4
> (toss-until-doubles) ==> TOSSES: 5, 6; sum = 11
TOSSES: 2, 6; sum = 8
In the first example, each pair of tosses is printed out, along with their sum, until doubles are found. (The 2 and 2 count as doubles.) Then, the message “HEY! We got doubles!” is printed out. Finally, 4 (i.e., the sum of the recently tossed doubles) is returned as the output value; it is not displayed as side-effect printing. Similarly, in the second example, each pair of tosses is printed out until the 1 and 1 occurrence of doubles is found. In that case, 2 is the output value, not side-effect printing.

**Problem 13.6**

Define a function, called `toss-three-dice-until-beat-target`, that satisfies the following contract:

```scheme
;; TOSS-THREE-DICE-UNTIL-BEAT-TARGET
;; --------------------------------------------------
;; INPUT: TARGET, an integer LESS THAN 18
;; SIDE EFFECT: Simulates the repeated tossing of three dice,
;; printing out the tosses and their sum, until the sum is
;; GREATER than TARGET
;; OUTPUT: The sum of the three dice that beat the TARGET.

; Here are some examples of its behavior:

> (toss-three-dice-until-beat-target 12)
6 + 2 + 3 = 11
3 + 1 + 2 = 6
5 + 6 + 3 = 14
14
> (toss-three-dice-until-beat-target 12)
5 + 3 + 6 = 14
14
> (toss-three-dice-until-beat-target 14)
5 + 4 + 5 = 14
6 + 3 + 6 = 15
15
> (toss-three-dice-until-beat-target 14)
1 + 1 + 5 = 7
5 + 5 + 1 = 11
4 + 1 + 3 = 8
3 + 5 + 4 = 12
1 + 3 + 1 = 5
1 + 2 + 4 = 7
2 + 2 + 2 = 6
5 + 2 + 3 = 10
5 + 5 + 3 = 13
5 + 1 + 1 = 7
4 + 6 + 6 = 16
16
```
Problem 13.7

Define a function, called `toss-until-total-beats-target`, that satisfies the following contract:

```
;; TOSS-UNTIL-TOTAL-BEATS-TARGET
;; ------------------------------------------------------------
;; INPUT: TARGET, an integer
;; SIDE EFFECT: Simulates the tossing of a die, printing out
;; all tosses along the way, until the sum of all dice tossed
;; is greater than TARGET.
;; OUTPUT: The total of the 3 dice that finally beat the target.
```

Here are some examples of its behavior:

```
> (toss-until-total-beats-target 10)
5 4 4 ... 13
> (toss-until-total-beats-target 10)
4 5 6 ... 15
> (toss-until-total-beats-target 20)
1 3 4 1 4 5 ... 22
> (toss-until-total-beats-target 20)
5 1 6 3 4 3 ... 22
```

13.2 The let* Special Form

The syntax of the let* special form is nearly identical to that of the let special form. (The only difference is the presence of the * in let*.) However, the semantics is substantially different. In particular, the local environment is populated incrementally, as each var/val pair is processed. This difference allows a certain kind of incremental computation that turns out to be quite useful. When a let special form is evaluated, each val_i is evaluated with respect to the parent environment and, thus, none of the val_i expressions can depend on any of the variables in the nascent local environment. In contrast, when a let* special form is evaluated, each val_i is evaluated with respect to the portion of the local environment that has been created so far. As a result, the expression val_i may depend on the values of the local variables var_1, ..., var_i-1 that precede it in the let* expression.

13.2.1 The Syntax of the let* Special Form

Each let* expression has the following form:

```
(let* ((var_1 val_1)
       (var_2 val_2)
       ...
       (var_n val_n))
  expr_1
  expr_2
  ...
  expr_k)
```

You’ll notice that the only difference is the asterisk in the name of the special form: let* instead of let.

13.2.2 The Semantics of the let* Special Form

A let* special form is evaluated as follows:
• An empty local environment is created.
• Each \texttt{var}/\texttt{val} pair is processed, in turn. In particular, an entry is created in the local environment that associates the value of \texttt{val}_i with the symbol \texttt{var}_i.

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

• Then the expressions in the body of the \texttt{let*} are evaluated, in turn.
• The value of the last expression in the body of the \texttt{let*} serves as the value of the entire \texttt{let*} expression.

\begin{example}

\textit{Example 13.2.1}

\begin{verbatim}
> (let* ((x 4)
       (y (+ x 2))
       (z (* x y))
       (w (+ x y z)))
    (printf "x: \texttt{\textasciicircum}A, y: \texttt{\textasciicirci}r, z: \texttt{\textasciicirci}r, w: \texttt{\textasciicirci}r\textasciitilde" x y z w)
(+ x y z w))
x: 4, y: 6, z: 24, w: 34
68
\end{verbatim}

Notice that the expression, \((+ \times 2)\), that is used to compute the value for \texttt{y} refers to the local variable \texttt{x}. Similarly, the expression, \((\times \times \times)\), that is used to compute the value for \texttt{z} refers to both \texttt{x} and \texttt{y}. Finally, the expression, \((+ \times \times \times)\), that is used to compute the value for \texttt{w} refers to \texttt{x}, \texttt{y} and \texttt{z}. Trying to do this with a \texttt{let} expression causes \textit{DrScheme} to complain.

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

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

\end{example}

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

In general, a \texttt{let*} expression of the form,

\[
(\texttt{let*} \ (\langle \texttt{var}_1 \ \texttt{val}_1 \rangle \ \langle \texttt{var}_2 \ \texttt{val}_2 \rangle \ \ldots \ \langle \texttt{var}_n \ \texttt{val}_n \rangle)
\]


\[
\texttt{expr}_1
\]
is equivalent to \( n \) nested `let` expressions:

\[
\begin{align*}
\text{expr}_2 \\
\vdots \\
\text{expr}_k \\
\end{align*}
\]

\[
\begin{align*}
&\text{(let } ((\text{var}_1 \ \text{val}_1)) \\
&\quad \text{(let } ((\text{var}_2 \ \text{val}_2)) \\
&\quad \quad \vdots \\
&\quad \quad \text{(let } ((\text{var}_n \ \text{val}_n)) \\
&\quad \quad \quad \text{expr}_1 \\
&\quad \quad \quad \text{expr}_2 \\
&\quad \quad \quad \quad \vdots \\
&\quad \quad \quad \text{expr}_k \\
&\quad \text{...} \\
&\text{...}) \text{...})
\end{align*}
\]

The following example demonstrates the equivalence.

**Example 13.2.2**

The following Interactions Window session evaluates a `let*` expression and the equivalent nested `let` expression:

```
> (let* ((x 4)
\quad (y (+ x 2))
\quad (z (* x y))
\quad (w (+ x y z))
\quad (printf "x: \%A, y: \%A, z: \%A, w: \%A\n" x y z w)
\quad (+ x y z w))
\quad x: 4, y: 6, z: 24, w: 34
\quad 68
> (let ((x 4))
\quad (let ((y (+ x 2)))
\quad \quad (let ((z (* x y)))
\quad \quad \quad (let ((w (+ x y z)))
\quad \quad \quad \quad (printf "x: \%A, y: \%A, z: \%A, w: \%A\n" x y z w)
\quad \quad \quad \quad (+ x y z w))))
\quad x: 4, y: 6, z: 24, w: 34
\quad 68
```

Notice that the outermost `let` expression (i.e., the one that specifies the local variable `x`) has a body that consists of a single `let` expression (i.e., the one that specifies the local variable `y`). Because the `let` expression for `y` is evaluated with respect to the local environment containing an entry for `x`, it is okay for the value expression, `(+ x 2)`, to refer to `x`. Similar remarks apply to the remaining variables.

In general, `let*` provides a simpler syntax than the equivalent set of nested `let` expressions. Thus, if you ever need to do incremental computations where the value of each local variable depends of the values of the preceding local variables, then you should consider using `let*`. 
Problems

**Problem 13.8: Using let* to create a fuel report**

Define a function, called `fuel-report`, that satisfies the following contract:

```
;; FUEL-REPORT
;; -----------------------------------
;; INPUTS: STARTING-MILES, non-negative number representing
;; the starting reading of the odometer of a car
;; ENDING-MILES, non-negative number representing
;; the ending reading of the odometer of a car
;; COST-PER-GALLON, cost of gas purchased
;; NUM-GALLONS, number of gallons purchased
;; OUTPUT: none
;; SIDE EFFECT: Prints out a fuel report including the number
;; of miles traveled, the miles per gallon, the amount of
;; money spent (in dollars), and the cost per mile (in
;; dollars per mile).
;; NOTE: miles-per-gallon = num-miles-traveled / num-gallons
;; dollars-spent = cost-per-gallon * num-gallons
;; cost-per-mile = num-dollars-spent / num-miles-traveled
```

Here are some examples of its desired behavior:

```
> (fuel-report 0 100 5.0 10)
Miles traveled: 100, miles-per-gallon: 10
Dollars spent: 50.0, cost-per-mile: 0.5
> (fuel-report 25 75 4.0 3.0)
Miles traveled: 50, miles-per-gallon: 16.666666666666668
Dollars spent: 12.0, cost-per-mile: 0.24
```

Note that it does not generate any output; all of the text is side-effect printing.

The purpose of this problem is to practice using the let* special form to simplify a sequence of computations. Thus, you should use a single let* to create a sequence of local variables with the following names: `miles-traveled`, `miles-per-gallon`, `dollars-spent` and `cost-per-mile`. Note that the value of each variable depends only on the values of variables defined before it. For example, the value of `miles-per-gallon` depends only on `miles-traveled` and `num-gallons`. Similarly, the value of `miles-traveled` depends only on the inputs `starting-miles` and `ending-miles`.

### 13.3 The letrec Special Form

The letrec special form is provided to enable the specification of local recursive functions, something that cannot be done by let or let*. The specification of a local recursive function within a letrec special form is quite similar to the specification of a global recursive function within a define special form; however, the syntax of a letrec expression is much closer to that of let and let*. A common use of letrec is to embed an accumulator-based, tail-recursive helper function within the body of its wrapper function. In this way, the existence of the helper function (and access to it) can be hidden from the general programming public. As usual, in such scenarios, the wrapper function takes care of supplying appropriate inputs to the helper function, freeing the user to think about other things.
13.3.1 The Syntax of the \texttt{letrec} Special Form

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

13.3.2 The Semantics of the \texttt{letrec} Special Form

In sharp contrast to how the \texttt{let} and \texttt{let*} special forms are evaluated, the evaluation of a \texttt{letrec} special form begins by creating the entire local environment, complete with entries for all of the local variables, before evaluating any of the value expressions. Because none of the value expressions have yet been evaluated, each local variable is initially given the dummy value, \texttt{#<undefined>}. However, since all of the local variables have corresponding entries in the local environment before any of the value expressions are evaluated, each value expression can refer to any or all of the local variables, whether they have values or not!

Example 13.3.1

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

\begin{verbatim}
> (let ((x y)
      (y x))
  (printf "x:" A C y x)
ERROR: reference to undefined identifier: y
> (let* ((x y)
      (y x))
  (printf "x:" A C y x)
ERROR: reference to undefined identifier: y
> (letrec ((x y)
      (y x))
  (printf "x:" A C y x))
  x:#<undefined>, y:#<undefined>
> 
\end{verbatim}

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

Example 13.3.2: Using \texttt{letrec} to create a local recursive function

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

\begin{verbatim}
> (let ((factyOne (lambda (n)
    (if (<= n 1)
    1
    ;; The following reference to factyOne
    ;; causes an error!!
    (* n (factyOne (- n 1)))))
  (factyOne 4))
ERROR: reference to undefined identifier: factyOne
\end{verbatim}
> (let* ((factyTwo (lambda (n)
                (if (<= n 1)
                    1
                    ;; The following reference to factyTwo
                    ;; causes an error!!
                    (* n (factyTwo (- n 1))))))
        (factyTwo 4))
ERROR: reference to undefined identifier: factyTwo
> (letrec ((factyThree (lambda (n)
                  (if (<= n 1)
                      1
                      ;; No problems here! :) 
                      (* n (factyThree (- n 1))))))
        (factyThree 4))
24

As the comments indicate, the reason that the let expression causes an error is that the lambda expression is evaluated before an entry for factyOne has been created in the local environment. It’s not that evaluating the lambda expression requires evaluating factyOne—it doesn’t. But, to create the function corresponding to the given lambda expression, DrScheme needs to know where to look for the value of factyOne should that function ever be called. And that information is unavailable because there is no entry anywhere for any variable named factyOne.

Similar remarks apply to the let* expression because although a val, expression inside a let* can refer to previously defined local variables, it cannot refer to the current variable, var,—because no entry exists yet for var. (For this reason, a let* that includes only one var/val pair is equivalent to a let.) However, for the letrec expression, there are no problems. DrScheme sees where to find the value for factyThree should that function ever be called, because the entry has already been created in the local environment. Although the current value of factyThree in the local environment is #<undefined>, that is not a problem, since DrScheme does not yet need to evaluate factyThree—it just needs to know where to look in the future, whenever that function gets called.

Although this example is also illustrative, it seems kind of silly to create a function like factyThree to use it only once. The following example highlights are more common, useful way of using letrec.

Example 13.3.3: Using letrec to create a local recursive function within a wrapper function

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

> (define facty
  (lambda (n)
    ;; Body of FACTY starts here
    (letrec ((facty-acc (lambda (m acc)
                       ;; Body of FACTY-ACC starts here
                       (if (<= m 1)
                           acc
                           (facty-acc (- m 1) (* m acc))))))
     ;; Body of LETREC starts here
     (facty-acc n 1))))
\begin{verbatim}
> (facty 4)
24
> (facty 5)
120
\end{verbatim}

This kind of application of \texttt{letrec} is commonly used to hide the existence of a recursive helper function from users who may not understand what inputs to give it, or may not want to be bothered with thinking about what inputs to give it. The helper function only exists for use by the parent function; it is not visible to the general programming public. The parent function (\texttt{facty}) takes care of supplying the helper function (\texttt{facty-acc}) with appropriate inputs.

⋆ Take care when defining local recursive helper functions. For example, note the difference between the input \texttt{n} to \texttt{facty} and the input \texttt{m} to \texttt{facty-acc}. On successive recursive function calls, \texttt{m} takes on different values, while \texttt{n} never changes.

\section*{Problems}

\textbf{Problem 13.9}

\textit{Mimicking the structure of \texttt{facty} and \texttt{facty-acc} from Example 13.3.3, define a function called \texttt{sum-cubes} that uses \texttt{letrec*} to define an accumulator-based, tail-recursive local helper function called \texttt{sum-cubes-acc}. The \texttt{sum-cubes} function should satisfy the following contract:}

\begin{verbatim}
;; SUM-CUBES
;; ----------------------------------------------
;; INPUTS: N, a positive integer
;; OUTPUT: The sum: 1*1*1 + 2*2*2 + ... + N*N*N
\end{verbatim}

Here are some examples of its desired behavior:

\begin{verbatim}
> (sum-cubes 3)
36
> (sum-cubes 4)
100
\end{verbatim}

\textbf{Problem 13.10}

\textit{Same as Problem 13.1b, except that you should use \texttt{letrec} to define the recursive helper function as a local function.}

\textbf{Problem 13.11}

\textit{Same as Problem 13.2, except that you should use \texttt{letrec} to define the recursive helper function as a local function.}

\textbf{Problem 13.12}

\textit{Same as Problem 13.4, except that you should use \texttt{letrec} to define the recursive helper function as a local function.}
13.4 Summary

Special Forms Introduced in this Chapter

- `let` Create local environment
- `let*` Create local environment, supports incremental computations
- `letrec` Create local environment, supports recursive function definitions

Built-in Functions Introduced in this Chapter

- `random` Pseudo-random number generator (an impure function)
Appendices
Appendix A

Guide to Your CS Account

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

A.1 The File System

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

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

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

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

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

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

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

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

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

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

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

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

### A.3 Submitting Programming Assignments

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

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

#### Before Printing or Electronically Submitting your Files

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

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

- Your definitions window should be nicely formatted. See the code-from-class postings on the course website for examples of nicely formatted code. Or look at the posted solutions to any lab or assignment. In particular:
  ⭐ Make sure that your definitions file begins with a block of comments like this:

    ```
    ;; ===========================================
    ;; CMPU-101, Spring 2017
    ;; Asmt. or Lab Info
    ;; Your Name
    ;; ===========================================
    ```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Save your definitions window one last time.

Hit the Run button one last time.

Save your interactions as plain text! (Use the Save Other and Save Interactions as Text... menu items in DrScheme.)

Double-check that your interactions begin with a nice block of text generated by the header function. The top of your interactions should have the following form:
Introduction to Computer Science via Scheme  © 2017 Luke Hunsberger  Spring 2017  155

CMPU-101, Spring 2017
Asmt. or Lab Info
Your Name

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

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

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

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

A.3.1 Printing Text Files

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

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

Example A.3.1

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

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

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

enscript -P Asprey hun-lab2.txt

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

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

enscript -P Asprey myfile.txt

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

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

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

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

    submit101  AsmtSubmissionName  YourAsmtDir

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

Example A.3.2

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

    submit101  h-asmt3  asmt3

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

    cd ..

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

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

Labs
B.1 Lab 1: Your first CMPU-101 lab session!

The purpose of this lab is to demonstrate the basics of navigating your Computer Science account, creating files and directories, saving them, and so on.

- During this lab, you will see some Scheme expressions that you won’t fully understand until you read later chapters. For now, just think of them as fillers that illustrate where certain kinds of things go within a program file.

- If you get stuck anywhere along the line, please ask for help!

You will access your CS account through computers that are running the Linux operating system. The following instructions introduce the basic Linux commands that you will use from within your CS account to download files, create files, organize your files, and so on, for all future labs and programming assignments.

Part One: Logging into your CS account

Sit down at one of the computers. Log into your account using the following information:

Username: The same as the first part of your Vassar email address.

Password: Look at the whiteboard!

Once the “Desktop” appears on-screen, click on the System menu in the lower-left corner of the screen. Select System Tools and then one of the options for a Terminal window. A Terminal window should appear on-screen. (If you can’t find the appropriate menu item to open up a Terminal window, ask for help.)

The Terminal window acts a lot like DrScheme’s Interactions Window that you have seen in class. In the Interactions Window, you type a Scheme expression at the prompt, followed by hitting the Enter key. In response, the Scheme datum denoted by that Scheme expression is evaluated and, usually, some information is displayed.

In the Terminal window, you type Linux commands at the prompt. When you hit the Enter key, the Terminal program tries to execute the command you entered. Of course, if you enter something wrong, it may complain vigorously.

One of the main jobs of commands entered into the Terminal window is to enable you to navigate the files and directories not only in your account, but also the entire file system for all of the CS accounts.

Before proceeding, be sure to read Chapter A through Section A.2.

Table B.1 lists a sequence of Linux commands, along with explanations for each. For each command shown, type the command into the Terminal window, and then hit the Enter key. You should enter the commands one at a time, in the order shown. If you get mixed up, just go back to the first command—or ask for help.

After entering the entire sequence of commands listed in Table B.1, you should end up in a newly created directory, called lab1, within your CS account. The full pathname for this lab1 directory should be displayed as `/Desktop/my101/labs/lab1` or `/home/yourAcctName/Desktop/my101/labs/lab1`. (The character `~` is frequently used as a convenient abbreviation for your home directory.)

- If you get stuck and want to return to your home directory, just type: cd ~. Alternatively, you could just type cd without any inputs because it assumes you want to go to your home directory by default.

- If you want to “back up” to the “parent” of your working directory, use the following command (with two periods): cd ..

Examples of using the cd command to navigate through a directory tree are shown in Fig. B.1. The figure presumes that the account name—and hence the name of the home directory—is hunsberg.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description of what it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pwd</code></td>
<td>Display the <em>working directory</em> (i.e., the directory you are in right now). You are probably in your account’s home directory, which may be displayed as: /home/<em>yourAcctName.</em></td>
</tr>
<tr>
<td><code>cd ~</code></td>
<td>Move (conceptually) to your “home” directory (i.e., set the working directory to be your home directory). (This is probably not necessary since you were probably already in your home directory.) Note: <code>cd</code> stands for “change directory”; and <code>~</code> is a convenient shorthand for your home directory.</td>
</tr>
<tr>
<td><code>ls</code></td>
<td>List the contents of your working directory. It should contain a subdirectory called <em>Desktop</em>.</td>
</tr>
<tr>
<td><code>cd Desktop</code></td>
<td>Change the working directory to be your <em>Desktop</em> directory.</td>
</tr>
<tr>
<td><code>ls</code></td>
<td>List the contents of the working directory—which should be the <em>Desktop</em> directory at this point. Note that most of the contents of the <em>Desktop</em> directory have a corresponding icon on the Desktop!</td>
</tr>
<tr>
<td><code>mkdir my101</code></td>
<td>Create a new directory called <em>my101</em>. Note that because <em>my101</em> is <em>not</em> a full pathname, the new directory will created within the working directory—in this case, the <em>Desktop</em> directory. And since the new directory is located within the <em>Desktop</em> directory, an icon will automatically appear on the Desktop!</td>
</tr>
<tr>
<td><code>ls</code></td>
<td>This should show you that the <em>Desktop</em> directory now contains a subdirectory called <em>my101</em>.</td>
</tr>
<tr>
<td><code>cd my101</code></td>
<td>Travel into the <em>my101</em> directory.</td>
</tr>
<tr>
<td><code>pwd</code></td>
<td>Show that you are now in the <em>my101</em> sub-directory. It will probably be displayed before the prompt as ~/Desktop/my101 or /home/<em>yourAcctName</em>/Desktop/my101.</td>
</tr>
<tr>
<td><code>ls</code></td>
<td>Show the (non-existent) contents of your new <em>my101</em> directory.</td>
</tr>
<tr>
<td><code>mkdir labs</code></td>
<td>Create a new directory called <em>labs</em> inside the <em>my101</em> directory. (It is created within the <em>my101</em> directory because that is the working directory—i.e., the directory where you are right now.)</td>
</tr>
<tr>
<td><code>ls</code></td>
<td>Show that you indeed have created <em>labs</em>.</td>
</tr>
<tr>
<td><code>cd labs</code></td>
<td>Move into the <em>labs</em> directory.</td>
</tr>
<tr>
<td><code>pwd</code></td>
<td>Show that you are indeed there.</td>
</tr>
<tr>
<td><code>ls</code></td>
<td>Show that the <em>labs</em> directory is currently empty.</td>
</tr>
<tr>
<td><code>mkdir lab1</code></td>
<td>Create a sub-directory called <em>lab1</em>.</td>
</tr>
<tr>
<td><code>ls</code></td>
<td>Show that <em>lab1</em> is there.</td>
</tr>
<tr>
<td><code>cd lab1</code></td>
<td>Move into the <em>lab1</em> directory.</td>
</tr>
<tr>
<td><code>pwd</code></td>
<td>Show where you are.</td>
</tr>
</tbody>
</table>

Table B.1: A sequence of Linux commands to create the directory structure for Lab 1
Part Two: Downloading Files

Okay, you have now created a directory called `lab1`, which is where you will put all of the files needed for this lab. There are two ways to get the desired files into your `lab1` directory.

1. **Fast and easy.** In your Terminal window, type the following command, exactly as shown below—which assumes that you have created the folders/directories named `my101`, `labs` and `lab1`, as discussed above:

   ```bash
   ```

   This says to copy all of the files from the `lab1` directory at the specified location within the `cs101` course account into the `lab1` directory you recently created within your account. (The `cs101` account is the owner of the first `lab1` directory; you are the owner of the second `lab1` directory.) In the above command, `~/cs101` denotes the home directory of the `cs101` account, and `*` is used to specify that all files within the first `lab1` directory should be copied into your `lab1` directory.

   **Note.** If you are already in your `~/Desktop/my101/labs/lab1/` directory, then you can get the same result using the following, shorter command:

   ```bash
   ```

   In this command, the period represents your working directory. So this command will copy all of the files from the `lab1` directory owned by the `cs101` account, into your working directory which, hopefully, is your `lab1` directory.

2. **Very slow—but perhaps more familiar.** Use a web browser to fetch the needed files from the course website, and then move them into your newly created `lab1` directory, as follows. First, click on the GLOBE icon in your task bar to open up a web browser. Then enter the following URL into the address bar of the browser window:

   ```
   ```
On our course web page, scroll down slightly until you see the link for “Lab Files”. Click on it. Then click on the link for Lab 1. You should see the following list of Scheme files:

- lab1-defns.txt
- asmt-helper.txt

Download each of these files by taking the following steps:

- Click on the file name.
- Under the File menu of your browser, select Save Page As... (or Save File As...).

Tell the browser that (for fun) you want to save the file into your Desktop directory. Do this by first selecting your home directory (whose name is the same as your account name) and then your Desktop sub-directory. (If you don’t see how to do this, ask for help.) When successful, there should be a new icon on your Desktop for each file.

After downloading the files from the course’s Lab 1 page, issue the following Linux commands in the Terminal window:

```
cd ~/Desktop  
ls
mv lab1-defns.txt asmt-helper.txt my101/labs/lab1/
```

The last command uses the MoVe command: mv. It moves the files, lab1-defns.txt and asmt-helper.txt, from the working directory into the my101/labs/lab1 directory. Note that my101/labs/lab1 is not a full pathname (since it does not begin with a forward slash). Instead, it is a relative pathname, which specifies only the portion of the full pathname starting from the working directory. In this case:

- the working directory’s full pathname is: /home/yourAcctName/Desktop
- the relative pathname is: my101/labs/lab1
- and lab1’s full pathname is: /home/yourAcctName/Desktop/my101/labs/lab1

Notice that lab1’s full pathname is constructed by concatenating the working directory’s full pathname and the relative pathname. (Ask for help if you have trouble seeing this.) For the above mv command to work, your working directory must be your Desktop directory, which contains the my101 sub-directory. Because the my101 directory is contained within the Desktop directory, we may say that the my101 directory is visible from the Desktop directory.

Incidentally, since you just moved the two downloaded files from your Desktop directory into the lab1 directory, those files are no longer on your Desktop. However, their icons might still be there. To update your Desktop display, hit the F5 key on the keyboard. The icons should disappear.

Finally, use the cd command to move into the lab1 directory, and then the ls command to list its contents:

```
cd my101/labs/lab1
ls
```

If you don’t see the two downloaded files, ask for help.

*Phew!* The good news is: The above commands are almost all of the Linux commands we are going to need for the entire semester.
Part Three: Firing Up DrScheme

Use the `pwd` command to verify that you are currently in your `lab1` directory. Use the `ls` command to verify that you have successfully downloaded the desired files and moved them into your `lab1` directory. Then—while still in your `lab1` directory—type the following command into the Terminal window to start up the DrScheme program:

```
drscheme&
```

If you forget to type the `&` character, then the Terminal window will freeze until the DrScheme program is closed/finished. If you include the `&` character, you can continue to use the Terminal window while the DrScheme program is running.

⇒ Since this is the first time that you have opened DrScheme, you will need to “choose” the “Full Swindle” language, as follows. First, in the DrScheme menu bar, click on the Language menu item, and then select Choose Language. Then, in the pop-up window, under Swindle, select Full Swindle.

**Opening a file in DrScheme.** Under the File menu of DrScheme, select the Open item. When prompted, select the file `lab1-defns.txt`. The contents of the file should appear in DrScheme’s Program Definitions window pane. Normally, the Program Definitions window pane occupies the top half of DrScheme’s main window, with the Interactions Window in the bottom half; however, until you click the Run button, the Program Definitions window may be all you see.

Under the File menu of DrScheme, select the Save Other menu item, and then, after that, choose Save Definitions As Text.... When prompted, type in: `yourName-lab1-defns.txt`. Make sure that it is saved within your `lab1` directory. You can check this by using the `ls` command in the Terminal window.

**Programming in DrScheme.** Have a look at the contents of the `yourName-lab1-defns.txt` file. In addition to a variety of comments (the lines that start with semi-colons), it contains a bunch of (possibly strange-looking) Scheme expressions, such as:

```
(load "asmt-helper.txt")
(header "myName" "Lab 1")
(problem 0)
(problem 1)
```

As will be seen—particularly in Chapter 10—these expressions will enable us to generate nicely formatted text in the Interactions Window that is suitable for printing and submitting! For now, the following brief descriptions will suffice. (Remember, the point of this lab is to demonstrate the mechanics of using your CS account, not to delve into the meaning of these kinds of Scheme expressions.)

The `load` expression causes the expressions contained within the file `asmt-helper.txt` to be loaded into the Interactions Window, just as if you had typed those expressions, by hand, into the Interactions Window, one after the other. The `asmt-helper.txt` file defines several useful functions, including: `tester`, `header` and `problem`. The `tester` function can be used to facilitate testing Scheme functions; and the `header` and `problem` functions enable the display of nicely formatted headings within the Interactions Window.

The following sequence of actions will demonstrate what these functions do.

Click on the Run button at the top-right of DrScheme’s window. The Run button loads the contents of the Program Definitions Window into the Interactions Window, so that you don’t have to manually enter (and re-enter) them. You should see some stuff printed out in the Interactions Window. Scroll through the Interactions Window results to get an idea of which expressions in the Definitions Window gave rise to the expressions you see in the Interactions Window.

Find the expression, `(header "myName" "Lab 1")`, in the Program Definitions Window. Change the characters `myName` to something that more accurately reflects your name. (Keep the double-quotes.)
Find the expression, (problem 0), in the Program Definitions Window. Notice that, following that expression, there are several expressions that involve the tester function. You can see the corresponding results in the Interactions Window when you hit the Run button. Put in a few more tester expressions. Include expressions whose evaluation you are unsure of. Predict (to yourself) what the result will be, then hit the Run button to see what DrScheme does.

From time to time, be sure to hit the Save button, located near the top of the DrScheme window, so that your program file (i.e., the contents of the Definitions Window) is saved. (The Save button is only visible if you have made some changes to the contents of the Definitions Window since the last time the contents were saved.)

Below the expression, (problem 1), in the Definitions Window, you should see an expression indicating that you haven’t yet defined the distance-fallen function. Delete that expression and replace it with the following:

```
(define distance-fallen
  (lambda (num-seconds)
    (* 16 num-seconds num-seconds)))
```

This expression defines a function named distance-fallen that takes a single input, called num-seconds, which represents the number of seconds since an object was dropped. (Chapter 9 shows how to define functions in Scheme using the define and lambda special forms.) The output value generated by this function is the corresponding distance (in feet) that the object would have fallen in that number of seconds.

* After entering the above expression, be sure to hit the Save button. Then hit the Run button, which causes the contents of your Definitions Window to be loaded into DrScheme.

Enter expressions such as (distance-fallen 3) directly into the Interactions Window to see if your function is working properly. (Chapter 6 addresses the application of functions to inputs using the Default Rule for evaluating non-empty lists.) Since $16 \cdot 3 \cdot 3 = 144$, that is the result that DrScheme should report.

Better yet, you can automate the process of testing by typing several tester expressions in the Definitions Window. (Make sure that you type the tester expressions after the function-definition expression.) For example, you could type expressions such as:

```
(tester '(distance-fallen 3))
(tester '(distance-fallen 8))
```

(The reason for the quote mark is discussed in Chapter 10.) After putting several such expressions into your Definitions Window, and hitting the Save button, hit the Run button to see the results.

**Saving your interactions!** When you are confident that everything is working properly, click the Run button one last time. The Interactions Window should now contain nicely formatted results.

* It is important to save the contents of your Interactions Window exactly as described below. Otherwise, you may end up with a file containing a bunch of garbled text.

Under the File menu, select Save Other, and then select Save Interactions as Text.... When prompted, enter a name for your file, such as: `yourNameInteractions.txt`. Be sure it is saved into your lab1 directory.

That’s it for this lab! The next lab will demonstrate how to print out your files and submit them electronically.
Lab 2

The main point of this lab is to continue demonstrating the technicalities involved in doing a lab (or programming assignment) using your CS computer account and DrScheme. The instructions presume that you have already completed Lab 1. In particular, they presume that you already have a directory called labs whose full pathname is ~/Desktop/my101/labs. You will see how to print your definitions and interactions files and submit your files electronically. Once this lab is completed, you will have seen everything you will need for the rest of the semester concerning the mechanics of doing a lab or programming assignment; you will then be able to turn your full attention to programming in Scheme!

First, log in to your CS account and then use sequence of commands listed below to:

1. cd into your labs directory: cd ~/Desktop/my101/labs
2. create a new lab2 directory within your labs directory: mkdir lab2
3. cd into your new lab2 directory: cd lab2
4. copy the relevant files from the course website’s lab2 directory into your working directory (i.e., your lab2 directory):
   
   
   (Recall that, in Linux commands, the period stands for your working directory.)

5. list the contents of your lab2 directory: ls
   It should now contain a file called lab2-defns-template.txt.

6. rename the lab2-defns-template.txt file as follows, but replacing yourName with some version of your own name:
   
   mv lab2-defns-template.txt yourName-lab2-defns.txt

7. launch the DrScheme program: drscheme&

Next, within DrScheme, open the file you just renamed. The following instructions will refer to this file as your definitions file.

For this lab, you will define two Scheme functions, maxx and banana-msg, as described in Problems 11.1 and 11.2 from Chapter 11. For each function, you are given a contract (i.e., a block of comments describing the name of the function, its inputs, its output, and its side effects—if any). For each function, copy-and-paste the contract into your definitions file, inserting the contract immediately below the corresponding (problem ...) expression. Then, below the contract, insert an appropriate Scheme expression, involving the define and lambda special forms, that effectively defines the desired function. (See Chapter 9 for examples.)

For convenience, you are given a few tester expressions for each function. In each case, after you have defined your function, you should try evaluating the various tester expressions in the Interactions Window to see if your function does what it is supposed to do. And you should insert the tester expressions into your definitions file—immediately following the corresponding function definition. In addition, come up with a few more tester expressions to confirm that your function works on other examples too.

When you are confident that your functions are working properly, ask me or a coach to come over to check your work.

* Then, read through the guidelines for printing and electronically submitting your work in the Appendix, Section A.3.

For practice, print out your definitions and interactions files. (Normally, you would not need to print out anything for a lab, but you will need to print out files for assignments.) Finally, electronically submit your lab2 directory using the submit101 command, as described in Section A.3:

submit101 h-lab2 ~/Desktop/my101/labs/lab2

If you are unsure whether it worked, ask for help.
Index

accumulators, 113
booleans, 13
built-in functions
  *: multiplication function, 25
  +: addition function, 25
  -: subtraction function, 25
  /: division function, 25
  <: less than, 47
  <=: less than or equal to, 47
  ==: equal to, 47
  >: greater than, 47
  >=: greater than or equal to, 47
  abs: computes absolute value, 111
  boolean?: type checker, 45
  eval: evaluation function, 27
  even?: is its input even?, 103, 123
  list?: type checker, 45
  load: for loading files, 68
  log: natural logarithm, 126
  not: flip boolean value, 77
  null?: type checker, 45
  number?: type checker, 45
  odd?: is its input odd?, 103, 123
  printf: formatted printing, 64
  procedure?: type checker, 45
  quotient: integer division, 26, 34, 124, 125
  random: pseudo-random number gen., 132
  remainder: integer division, 26, 34, 124, 125
  sin: trig function, 112
  symbol?: type checker, 45
  void, 27
  void?: type checker, 45, 65
character sequences, 13, 14
comments, 69
computational model, 8, 21, 34
contracts, 26
  as comments in a Scheme program, 69
data type, 13
  booleans, 14
  functions, 25
  non-empty lists, 29
  null, 15
  numbers, 13
  strings, 63
  symbols, 15
  void, 27
debug, 8
Default Rule for evaluating non-empty lists, 30, 33
  applying functions to inputs, 31, 34
  generating errors, 33
  on lists of lists, 31
denotation
  single arrow notation (→), 14, 22
destructive functions, 9
DrScheme, 21
  interactions window, 21, 22
  prompt, 22
  the Run button, 68
elements
  of non-empty lists, 29
empty list, 15
escape sequences, 64
evaluation, 17, 21
  double arrow notation (⇒), 18, 22
  of booleans, 17
  of functions, 27
  of non-empty lists, 30
  of numbers, 17
  of special forms, 37
  of strings, 63
  of symbols, 18
  of the empty list, 17
evaluation function, 17
  as a built-in function, 27
expressions, 13
  primitive data expressions, 13
floating point numbers, 103
function-call expression, 95
functions, 8, 17, 51
  built-in, 25
  defining vs. applying, 51
  factorial, 96
  function application using the Default Rule, 31
  primitive, 25
  recursive, 95
tail-recursive, 106

Global Environment, 18
and built-in functions, 25

identity function, 17

keyword symbols, 37, see also special forms
else, 83
redefining, 40

Lambda Calculus, 8
lambda function
applying to inputs, 56
lazy evaluation, 73
of the and special form, 78, 89
of the cond special form, 84
of the if special form, 74
of the or special form, 89

local environment, 56, 129
local variable, 129
local vs. global environments, 57, 131

nested if expressions, 82
equivalence in terms of cond, 82
non-destructive functions, 8
non-empty lists, 29
syntax and semantics, 29
non-strict truth values
with and, 78, 80
with if, 75
with not, 77
with or, 80

numbers, 13

predicates, 45
defined using boolean operators, 85
type-checker predicates, 45

program, 13

recursion, 95
accumulators, 113
base case, 95
recursive case, 96
tail recursion, 104
wrapper functions, 120

sample functions
alt-sum, 103
banana-msg, 76
bookstore-open?, 93
cafe-open?, 85
cafe-open?-alt, 87
civil-to-mil-hours, 81
compute-tax, 93
cond-effects, 84
cone-volume, 54
data-type-of, 91
discriminant, 59
exp-acc, 119
exp-wr, 121
f, 59
facty, 98, 106, 146
facty-acc, 115, 146
facty-v1, 96
facty-v2, 97
fancy-plotter, 111
funky, 59
gem, 126
gem-helper-acc, 127
how-big, 74
implies, 91
in-class, 80
letter-grade, 82
letter-grade-v2, 82
log-acc, 126
max-streak-in-n-flips, 138
maxx, 76
mil-to-civil-hrs, 92
my-and, 79
my-not, 78
nth-rightmost-digit, 124
num-occurs, 125
num-occurs-acc, 124
num-occurs-in-n-tosses, 137
num-tosses-until-repeat, 138
open-office?, 94
open-office?-alt, 94
plotter, 110
power, 102
print-and-sum-n-tosses, 136
print-checkerboard, 123
print-civil-time-from-mil, 92
print-down-to-zero, 108
print-func-vals, 107
print-n-dashes, 104
print-n-tosses, 136
print-rightside-up-triangle, 110
print-square, 109
print-thing-n-times, 108
print-upside-down-triangle, 109
quadrant, 90
rect-angle, 54
sphere-volume, 57
square, 53, 58, 60
sum-cubes, 147
sum-recips, 102
sum-squares, 100
sum-squares-acc, 117
sum-to-n, 113
sum-to-n-acc, 114
summer, 100
tester, 66
    and quote, 67
tester-alt, 107
time-of-day, 92
toss-die, 133
toss-three-dice-until-beat-target, 140
toss-until-doubles, 139
toss-until-total-beats-target, 141
verbose-func, 66
semantics, 7
side effect, of the printf function, 64
side effects, 8, 60
    of define special form, 38
    of printf, 64
special forms, 37
    and, 78
cond, 81
define, 38
    semantics of, 38
    side effect of, 38
    syntax of, 38
if, 73
lambda, 52
    semantics of, 55
    syntax of, 54
let, 129
    in terms of lambda, 134
    semantics of, 130
    syntax of, 129
let*, 129, 141
    equivalence to nested lets, 142
    semantics of, 141
    syntax of, 141
letrec, 129, 144
    semantics of, 145
    syntax of, 145
or, 80
quote, 41
    alternate syntax (’), 42
    semantics of, 41
    syntax of, 41
syntax, 7

truth values, 13

#<undefined>, 145