Introduction to Computer Science via Scheme

Luke Hunsberger

© 2017 Luke Hunsberger
All rights reserved

Spring 2017
## Contents

1 Introduction 7

1 Non-Destructive Programming in Scheme 11

2 Primitive Data Expressions 13
   2.1 Numbers ................................................................. 13
   2.2 Booleans ................................................................. 14
   2.3 The Empty List (or Null) ............................................. 15
   2.4 Symbols ................................................................. 15
   2.5 Summary ................................................................. 15

3 Evaluation 17
   3.1 Applying the Evaluation Function to Numbers, Booleans, or the Empty List 17
   3.2 Evaluating Symbols .................................................... 18
   3.3 Summary ................................................................. 19

4 Introduction to DrScheme 21
   4.1 Some Sample Interactions ........................................... 22
   4.2 Summary ................................................................. 23

5 Primitive (Built-in) Functions 25
   5.1 Built-in Functions for Arithmetic .................................. 25
   5.2 Contracts ................................................................. 26
   5.3 Built-in Functions for Integer Arithmetic ......................... 26
   5.4 The Built-in eval Function ........................................... 27
   5.5 The void Datum and the Built-in void Function .................... 27
   5.6 Summary ................................................................. 28

6 Non-Empty Lists 29
   6.1 The Syntax and Semantics for Non-Empty Lists ................... 29
   6.2 Evaluating Non-Empty Lists: the Default Case ..................... 30
      6.2.1 A More Formal Description of the Default Rule ............. 33
   6.3 Summary ................................................................. 35

7 Special Forms 37
   7.1 The define Special Form .............................................. 38
      7.1.1 The Syntax of the define Special Form ....................... 38
      7.1.2 The Semantics of the define Special Form ................... 38
   7.2 The quote Special Form .............................................. 41
      7.2.1 The Syntax of the quote Special Form ......................... 41
      7.2.2 The Semantics of the quote Special Form ..................... 41
      7.2.3 Alternate Syntax for quote Special Forms ..................... 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3 Summary</td>
<td>43</td>
</tr>
<tr>
<td>8 Predicates</td>
<td>45</td>
</tr>
<tr>
<td>8.1 Type-Checker Predicates</td>
<td>45</td>
</tr>
<tr>
<td>8.2 Comparison Predicates</td>
<td>47</td>
</tr>
<tr>
<td>8.3 Summary</td>
<td>48</td>
</tr>
<tr>
<td>9 Defining Functions</td>
<td>51</td>
</tr>
<tr>
<td>9.1 Defining Functions vs. Applying Them to Inputs</td>
<td>51</td>
</tr>
<tr>
<td>9.2 The \texttt{lambda} Special Form</td>
<td>52</td>
</tr>
<tr>
<td>9.3 The Syntax and Semantics of Lambda Expressions</td>
<td>54</td>
</tr>
<tr>
<td>9.3.1 The Syntax of a Lambda Expression</td>
<td>54</td>
</tr>
<tr>
<td>9.3.2 The Semantics of a Lambda Expression</td>
<td>55</td>
</tr>
<tr>
<td>9.3.3 Applying a \texttt{lambda} Function to Input Values</td>
<td>56</td>
</tr>
<tr>
<td>9.4 Summary</td>
<td>60</td>
</tr>
<tr>
<td>10 Strings, \texttt{printf}, \texttt{load}, \texttt{Run}, and Comments</td>
<td>63</td>
</tr>
<tr>
<td>10.1 Strings</td>
<td>63</td>
</tr>
<tr>
<td>10.2 The \texttt{printf} Function</td>
<td>64</td>
</tr>
<tr>
<td>10.3 The \texttt{load} Function and the \texttt{Run} Button</td>
<td>68</td>
</tr>
<tr>
<td>10.4 Comments</td>
<td>69</td>
</tr>
<tr>
<td>10.5 Summary</td>
<td>70</td>
</tr>
<tr>
<td>11 Conditional Expressions</td>
<td>73</td>
</tr>
<tr>
<td>11.1 The \texttt{if} Special Form</td>
<td>73</td>
</tr>
<tr>
<td>11.2 The Boolean Operators: \texttt{and}, \texttt{or}, and \texttt{not}</td>
<td>77</td>
</tr>
<tr>
<td>11.2.1 The \texttt{not} Function</td>
<td>77</td>
</tr>
<tr>
<td>11.2.2 The \texttt{and} Special Form</td>
<td>78</td>
</tr>
<tr>
<td>11.2.3 The \texttt{or} Special Form</td>
<td>80</td>
</tr>
<tr>
<td>11.3 The \texttt{cond} Special Form</td>
<td>81</td>
</tr>
<tr>
<td>11.4 Defining Predicates using Boolean Operators instead of Conditional Expressions</td>
<td>85</td>
</tr>
<tr>
<td>11.5 Simplifying Conditional and Boolean Expressions</td>
<td>87</td>
</tr>
<tr>
<td>11.6 Summary</td>
<td>89</td>
</tr>
<tr>
<td>12 Recursion</td>
<td>95</td>
</tr>
<tr>
<td>12.1 Defining Recursive Functions in Scheme</td>
<td>95</td>
</tr>
<tr>
<td>12.2 Tail Recursion</td>
<td>104</td>
</tr>
<tr>
<td>12.3 Accumulators</td>
<td>113</td>
</tr>
<tr>
<td>12.4 Wrapper Functions</td>
<td>121</td>
</tr>
<tr>
<td>12.5 Summary</td>
<td>124</td>
</tr>
<tr>
<td>13 Defining Local Variables with the \texttt{let}, \texttt{let*} and \texttt{letrec} Special Forms</td>
<td>131</td>
</tr>
<tr>
<td>13.1 The \texttt{let} Special Form</td>
<td>131</td>
</tr>
<tr>
<td>13.1.1 The Syntax of the \texttt{let} Special Form</td>
<td>131</td>
</tr>
<tr>
<td>13.1.2 The Semantics of the \texttt{let} Special Form</td>
<td>132</td>
</tr>
<tr>
<td>13.1.3 Deriving the \texttt{let} Special Form from the \texttt{lambda} Special Form</td>
<td>136</td>
</tr>
<tr>
<td>13.2 The \texttt{let*} Special Form</td>
<td>144</td>
</tr>
<tr>
<td>13.2.1 The Syntax of the \texttt{let*} Special Form</td>
<td>144</td>
</tr>
<tr>
<td>13.2.2 The Semantics of the \texttt{let*} Special Form</td>
<td>145</td>
</tr>
<tr>
<td>13.2.3 Deriving a Single \texttt{let*} Expression from Nested \texttt{let} Expressions</td>
<td>146</td>
</tr>
<tr>
<td>13.3 The \texttt{letrec} Special Form</td>
<td>147</td>
</tr>
<tr>
<td>13.3.1 The Syntax of the \texttt{letrec} Special Form</td>
<td>148</td>
</tr>
<tr>
<td>13.3.2 The Semantics of the \texttt{letrec} Special Form</td>
<td>148</td>
</tr>
<tr>
<td>13.4 Summary</td>
<td>151</td>
</tr>
</tbody>
</table>
14 Lists and List-Based Recursion 153
14.1 The Built-in Functions: first, rest and cons .............................................. 153
14.2 List-based Recursion ..................................................................................... 157
14.3 Recursively Generating Lists as Output Values ............................................. 171
14.4 Tail Recursion, Accumulators, and Wrapper Functions Revisited .................. 179
14.5 Sorting Algorithms ......................................................................................... 188
  14.5.1 The Insertion-Sort Algorithm ................................................................. 188
  14.5.2 The Merge-Sort Algorithm .................................................................... 193
  14.5.3 Comparing the Performance of Insertion Sort and Merge Sort .............. 196
14.6 The Underlying Structure of Non-Empty Lists ............................................. 198
  14.6.1 Data Structures ..................................................................................... 198
  14.6.2 Cons Cells (a.k.a. Pairs) ...................................................................... 198
14.7 Hierarchical/Deep/Nested Lists ..................................................................... 203

II Destructive Programming in Scheme 211
15 Vectors 213
  15.1 Vector Expressions ..................................................................................... 214
  15.2 Constructing Vectors .................................................................................. 215
    15.2.1 The Built-in vector Function ............................................................... 215
    15.2.2 The make-vector Function .................................................................. 216
  15.3 Accessing Information Stored in a Vector ................................................... 216
    15.3.1 The vector-ref Function ...................................................................... 216
    15.3.2 The vector-length Function ................................................................. 217
  15.4 Recursively Processing Vectors ................................................................... 218
    15.4.1 The dotimes Special Form .................................................................. 219
  15.5 Destructively Modifying a Vector .................................................................. 224

16 Data Structures 237
  16.1 The define-struct Special Form ................................................................... 241

17 The Model-View-Controller Paradigm 247
  17.1 Implementing a Simple Game using the MVC Paradigm .............................. 247

Appendices 257
A Guide to Your CS Account 259
  A.1 The File System .......................................................................................... 259
  A.2 Using Terminal to Explore and Augment the File System ............................ 260
  A.3 Submitting Programming Assignments ...................................................... 261
    A.3.1 Printing Text Files ............................................................................... 263
    A.3.2 Submitting your Files Electronically ................................................... 264

B Labs 265
  B.1 Lab 1: Your first CMPU-101 lab session! ..................................................... 266
## 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 \textit{sphere-volume} function to the value 2 ..................................................................... 58
12.1 DrScheme’s evaluation of (\texttt{facty} 3) ................................................................................................. 101
12.2 DrScheme’s evaluation of (\texttt{sum-to-n-acc} 3 0) ........................................................................... 116
14.1 The non-empty list, \((3 4 6)\), as a single cons cell—with very particular contents ................................. 200
14.2 An alternative depiction of the non-empty list, \((3 4 6)\), as a chain of cons cells .............................. 201
15.1 A scattered chain of cons cells representing the list \((32 7 #f xy w)\) ............................................... 214
15.2 Accessing the elements of a vector by their numerical indices ............................................................. 214
16.1 Partitioning a vector of cards ............................................................................................................... 242
17.1 States and transitions in the game of \textit{dice-dice} ...................................................................................... 248
A.1 A portion of a typical file system ........................................................................................................... 260
A.2 The organization of files and directories for your work in CMPU-101 .................................................. 260
B.1 Using the \texttt{cd} command to navigate through a directory tree in the \texttt{hunsberg} account ............ 268
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 15, 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:

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

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

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

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

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

As programmers, we only get to type the numerical expressions (i.e., character sequences); however, behind the scenes, the computer is 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:
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.

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

(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.

\[ \text{hello} \rightarrow \text{the symbol } \text{hello} \]
\[ * \rightarrow \text{the asterisk symbol} \]

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: \( 342, -81, 34/9, 21.832, \text{etc.} \)

Booleans: \#t and \#f.

The empty list: \( () \).

Symbols: \( x, \text{miles-per-gallon}, \text{dollarsPerGallon}, *, +, /, \text{etc.} \)

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

| Input Datum | Evaluation Function | Output Datum |

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, 15 and 16.

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.
Evaluation Function

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.

Input Datum \quad \Longrightarrow \quad Output Datum

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

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

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

\[
\text{the number zero } \quad \Longrightarrow \quad \text{the number zero}
\]
\[
\text{the boolean true } \quad \Longrightarrow \quad \text{the boolean true}
\]
\[
\text{the empty list } \quad \Longrightarrow \quad \text{the empty list}
\]

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:

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

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 \text{my-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 \( my-empty-list \) \( \implies \) 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:

the symbol \( foo \) \( \implies \) 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 (\( \Rightarrow \)) to represent the process of evaluation. For example, if the Global Environment contains an entry associating the symbol \( x \) with the number \( eighty-six \), this fact can be represented by:

the symbol \( x \) \( \Rightarrow \) the number \( eighty-six \)

It is important to remember that:

1. each expression—which is a character sequence—denotes a Scheme datum; and
2. each Scheme datum evaluates to a (usually different) Scheme datum.

For example:

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

Introduction to DrScheme

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

- 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 \rightarrow [ \text{the number three} \implies \text{the number three} ] \rightarrow 3
#t \rightarrow [ \text{the boolean true} \implies \text{the boolean true} ] \rightarrow #t
() \rightarrow [ \text{the empty list} \implies \text{the empty list} ] \rightarrow ()
```

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>Here is a contract for the built-in addition function:</td>
</tr>
<tr>
<td>Name: +</td>
</tr>
<tr>
<td>Inputs: x₁, x₂, ..., xₙ; any number of numerical inputs</td>
</tr>
<tr>
<td>Output: The sum, x₁ + x₂ + ... + xₙ</td>
</tr>
<tr>
<td>Side Effects: 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:

| Name: − |
| Inputs: x₁, x₂, ..., xₙ; any number of numerical inputs |
| Output: The value, x₁ − x₂ − x₃ ... − xₙ |
| Side Effects: None |

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: quotient provides the answer; remainder provides the remainder. The contracts for these functions are given below:

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

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

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

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

5.4 The Built-in eval Function

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

> eval
#<procedure:eval>

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

* Like numbers, booleans 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:

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

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 Arithmetic: quotient, remainder, integer?
- 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>#t</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3/5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void</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:

\[
\begin{align*}
(3 \ 4) & \longrightarrow \text{a list containing the number three and the number four} \\
(+ \ 3 \ 4) & \longrightarrow \text{a list containing the + symbol, the number three, and the number four} \\
(eval \ (+ \ 3 \ 4)) & \longrightarrow \text{a list containing:} \\
& \text{(1) the symbol eval, and} \\
& \text{(2) a subsidiary list containing the + symbol, the number three, and the number four}
\end{align*}
\]
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

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

(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.

**Example 6.2.1**

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

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

In each of these examples, the expression entered by the programmer is a legal Scheme expression that denotes a Scheme list. (You should convince yourself of this.) In addition, the evaluation of each list appears to result in an arithmetic computation—in fact, the kind of arithmetic computations you’ve seen in math classes over the years. In each case, the list is being evaluated according to the Default Rule.
Example 6.2.2

Consider the expression \((+ 2 3)\), which denotes a list containing three items: the + symbol, the number two, and the number three. The 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:

- the + symbol \(\Rightarrow\) the addition function
- the number two \(\Rightarrow\) the number two
- the number three \(\Rightarrow\) the number three

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

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

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

where the evaluation step is explained by:

First Step of Default Rule:

\[+ \text{ symbol } \Rightarrow \text{addition function}\]
\[\text{number two } \Rightarrow \text{number two}\]
\[\text{number three } \Rightarrow \text{number three}\]

Second Step of Default Rule:

addition function applied to two and three yields output of five

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

Example 6.2.3

Although the Default Rule is not trivial, there are several advantages to it. First, it 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:

\[
\begin{align*}
> & (+ 2 (* 3 10)) \\
& 32
\end{align*}
\]
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, \( K_1, K_2, \ldots, K_n \):
  
  \[ D_1 \Rightarrow K_1 \]
  
  \[ D_2 \Rightarrow K_2 \]
  
  \[ \ldots \]
  
  \[ D_n \Rightarrow K_n \]

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

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

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

As indicated by the parenthetical comments, it is possible for some things to go wrong in the process of evaluating a non-empty list. For example, the function \( K_1 \) might expect a different number of inputs than are present in the rest of the original list. Or the attempt to evaluate one of the data \( D_i \) might be undefined. Or the application of the function \( K_1 \) to the 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:

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

These Scheme data evaluate to the following:

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

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

```scheme
> (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:

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

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

- `eval symbol` ➞ the evaluation function
- `+ symbol` ➞ the addition function

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

\[ (+ 3 \times 4 10) \]
\[ 73 \]

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:

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

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

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

\[(define \ C_1 \ C_2)\]

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.

\(C_1 \rightarrow s\) and \(C_2 \rightarrow e\)

Therefore:

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

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

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

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

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

7.1.2 The Semantics of the define Special Form

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

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

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

\(\star\) 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:

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

Evaluation Function

no output!

side effect

Global Environment
symbol value

(1) New entry inserted into Global Environment

(2) e \Rightarrow E

(3) E becomes value for s

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

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

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

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

Of course, something has happened!

**Example 7.1.1**

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

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

Side Effect: New Global Environment Entry: 

| the symbol x | the number six |

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

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

\textbf{Example 7.1.2: Confirming the semantics of define}

The following admittedly unusual interactions confirm the semantics of the \texttt{define} special form.

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

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

(1) A new entry, \texttt{w\hspace{1em}void}, is inserted into the Global Environment.

(2) The expression \texttt{w} is evaluated, yielding the value \texttt{void}: \texttt{w ⇒ void}. (That’s what’s currently stored in the Global Environment as the value for \texttt{w}!)

(3) That value (i.e., \texttt{void}) is inserted as the value for \texttt{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 \texttt{w}. Afterward, when we ask DrScheme to evaluate \texttt{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

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

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

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

7.2.2 The Semantics of the quote Special Form

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

If: \(C \rightarrow D\)

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

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

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

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

---

Example 7.2.1

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

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

---

\(^1\)In fact, the keyword symbol is never evaluated in a special form of any kind. The purpose of the keyword symbol is simply to indicate that the given list is a special form, thereby requiring a special mode of evaluation.
In the first example, \((\text{quote } \text{x})\) denotes a list containing the \text{quote} symbol and the symbol \text{x}. That list is the Input Datum. The result of evaluating that list is the symbol \text{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:

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

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

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

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, \(\mathcal{C}'\), to describe \(D\) to us; however, \(\mathcal{C}'\) must nonetheless denote \(D\). (An example of this will be given shortly.)

### Example 7.2.2

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

\[
> \ (\text{define } \text{x } (+ \ 1 \ 2 \ 3))
> \ \text{x}
6
> \ (\text{quote } \text{x})
\text{x}
\]

### 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.

\[
> \ (\text{define my-list } (\text{quote } (1 \ 2 \ 3)))
> \ \text{my-list}
(1 \ 2 \ 3)
\]

### 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 \(\mathcal{C}\) is any Scheme expression denoting some datum \(D\), then the expressions, \((\text{quote } \mathcal{C})\) and \(\mathcal{C}'\), 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.

> 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?`.

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

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

### Example 8.1.2

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

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

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

### Example 8.1.3

The following Interactions Window session illustrates the behavior of the type-checker predicates.

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

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

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

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

DrScheme also provides a comparison predicate called eq? that is more general that the = predicate. Whereas the = predicate only works on numerical input, the eq? predicate can be used to test the equality of inputs that can be any combination of numbers, booleans, symbols or the empty list. Here’s a contract for the eq? predicate.

Name:  eq?

Inputs:  
- $d_1$, a number, boolean, symbol, or the empty list
- $d_2$, a number, boolean, symbol, or the empty list

Output:  
- #t if $d_1$ and $d_2$ are the same; #f otherwise.

---

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.

> (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:

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

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

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

The list? predicate works for any kind of list: empty or non-empty. The null? predicate works only for the empty list. The procedure? predicate works for functions. The comparison predicates include the standard functions for comparing numbers (e.g., less-than and greater-than-or-equal-to), as well as the more general eq? predicate that works on any combination of numbers, booleans, symbols, or the empty list.
Built-in Functions Introduced in this Chapter

Type-checker Predicates: number?, symbol?, boolean?, list?, null?, procedure?, void?.

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>list</td>
<td>number</td>
</tr>
<tr>
<td>(number? 'x)</td>
<td>boolean</td>
<td>number</td>
</tr>
<tr>
<td>(symbol? 'x)</td>
<td>boolean</td>
<td>number</td>
</tr>
<tr>
<td>(symbol? eval)</td>
<td>symbol</td>
<td>number</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)
```
\begin{verbatim}
#t
> (eq? myvar 'myvar)
#f
> (eq? myvar yourvar)
#f
> (eq? yourvar 'monday)
#t
> (eq? yourvar 'sunday)
#f
\end{verbatim}
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 \texttt{lambda} Special Form

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

* The use of the \texttt{lambda} symbol in a \texttt{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 \texttt{lambda} special form is a list whose first element is a keyword symbol—in this case, the symbol \texttt{lambda}. The second element in a \texttt{lambda} special form is used to specify the input parameter(s) for the function being defined. The rest of the elements in the \texttt{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 \texttt{define} special form is used to assign names to things in Scheme. Furthermore, a single function could have several different names. Thus:

* The \texttt{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 \texttt{lambda} special form. For example, the following \texttt{lambda} expression can be used to specify a squaring function in Scheme:

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

This \texttt{lambda} expression denotes a \texttt{lambda} special form (i.e., a Scheme list whose first element happens to be the \texttt{lambda} symbol). Like any special form, a \texttt{lambda} special form has its own, special rule for being evaluated. For now, suffice it to say that:
The evaluation of a lambda special form always results in a function.

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

\[
> \ (\text{lambda} \ (x) \ (* \ x \ x))
#<procedure>
\]

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

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

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

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

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, square—is not evaluated; however, the given expression—in this case, \((\text{lambda} \ (x) \ (* \ x \ x))\)—is evaluated. Thus, the value associated with the square symbol is the function that results from evaluating the given lambda special form, as demonstrated below:

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

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 square symbol evaluates to the function that we defined earlier, which is then applied to the desired input value.

Example 9.2.2

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

\[
> \ (((\text{lambda} \ (x) \ (* \ x \ x)) \ 4)
16
\]

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

Example 9.2.3

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

\[
\begin{align*}
> & \ (\text{define rect-area} \ (\text{lambda} \ (w \ h) \ (* \ w \ h))) \\
> & \ (\text{rect-area} \ 2 \ 3) \\
> & \ 6 \\
> & \ (\text{rect-area} \ 3 \ 8) \\
> & \ 24 \\
> & \ (\text{define cone-volume} \ (\text{lambda} \ (r \ h) \ (* \ 1/3 \ 3.14159 \ r \ r \ h))) \\
> & \ (\text{cone-volume} \ 3 \ 2) \\
> & \ 18.84953999999998 \\
> & \ (\text{cone-volume} \ 10 \ 1) \\
> & \ 104.71966666666665
\end{align*}
\]

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

9.3 The Syntax and Semantics of Lambda Expressions

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

9.3.1 The Syntax of a Lambda Expression

A lambda expression has the following syntax:

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

where:

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

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

Example 9.3.1

The following are examples of well-formed lambda expressions:

- \((\text{lambda} \ () \ 44)\)
- \((\text{lambda} \ (x) \ (* \ x \ x))\)
• (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, $s_1, s_2, \ldots, s_n$, are distinct; and
• $B$ denotes some Scheme datum $D$,

then a lambda expression of the form

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

denotes a Scheme list whose elements are as follows:
• the lambda symbol;
• a list containing $n$ distinct symbols, $s_1, s_2, \ldots, s_n$; and
• the Scheme datum, $D$

This list is referred to as a lambda special form.

By now, you should be getting used to the fact that a piece of syntax, such as $(\text{lambda } (x) (* x x))$, denotes a Scheme datum—in this case, a Scheme list containing the lambda symbol and two subsidiary lists. Although it is important to be able to 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 $(\text{lambda } (x) (* x x))$ evaluates to a function, when we really mean that the list denoted by the expression $(\text{lambda } (x) (* x x))$ evaluates to a function.
The Evaluation of a lambda Special Form

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

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

* Although evaluating a lambda special form only creates the corresponding function, it is necessary to describe what that function would do if it ever were applied to input values.

9.3.3 Applying a lambda Function to Input Values

Example 9.3.3: Applying the Squaring Function

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

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

This process is illustrated in Fig. 9.1.
Introduction to Computer Science via Scheme

Example 9.3.4: Computing the Volume of a Sphere

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

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

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

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

- **Finally**, the value 33.51029333333333 is reported as the output value generated by applying the \( \text{sphere-volume} \) 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 \( \ast \) and \( \pi \) came from the Global Environment.

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

The evaluation of \((\text{sphere-volume} \, 2)\) 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, \( \pi \), has disastrous effects! (That is one of many reasons why the use of global variables should be very carefully restricted!)

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

Incidentally, any character sequence beginning with a semi-colon is ignored by DrScheme. (Such character sequences are called comments.) Thus, for example, the sequence, ;; 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:

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

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

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

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

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

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

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

Here’s an example of a function that takes more than one input 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 `(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, `(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:

\[ f(x) = x^2 \]
\[ f(2 + 3) = 25 \]
\[ f(f(10)) = 10000 \]

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
     ()))
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 `(square 10)`, which is carried out by the Default Rule for evaluating non-empty lists.

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

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

**Special Forms Introduced in this Chapter**

- `lambda` Used to create functions of our own design.

**Problems**

<table>
<thead>
<tr>
<th>Problem 9.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consider the following Interactions Window session:</td>
</tr>
<tr>
<td>&gt; (define pie 3.14159)</td>
</tr>
<tr>
<td>&gt; (define funk (lambda (x) (* x pie)))</td>
</tr>
<tr>
<td>&gt; (funk 10)</td>
</tr>
<tr>
<td>31.4159</td>
</tr>
</tbody>
</table>

*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).

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

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

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

---

Example 10.2.1

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.

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

Note that the textual information displayed by DrScheme in each case is side-effect printing, 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 \% and \n 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 \( \tilde{A} \), then there must be \( n \) input expressions following the format string, as follows:

\[
\text{(printf format-string expr}_1\ldots\text{expr}_n)\]

---

**Example 10.2.2: Formatted printing with printf**

The following Interactions Window session illustrates the use of the escape sequences \( \tilde{\%} \), \( \tilde{\text{n}} \) and \( \tilde{A} \) 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: \tilde{A}, second thing: \tilde{\%}% (+ 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 ===> \tilde{A}, Second ===> \tilde{A}, Third ===> \tilde{\%}%
(+ 4 2) (- 9 6.3) (* 4 100)"
First ===> 6, second ===> 2.7, Third ===> 400
> (printf "A symbol: \tilde{A}, a string: \tilde{\text{A}}, a boolean: \tilde{\%}%
'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:

```scheme
> (printf "Show us void: " (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.

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

```scheme
> (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:

```
> 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\%" 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 x² − y²; otherwise, evaluate y² − x².” 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 ((x² < y³) 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:

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

where:

- *condExpr* is a condition (i.e., an expression that evaluates to #t or #f); and

- *thenExpr* and *elseExpr* are any Scheme expressions.
Example 11.1.1

The following expressions are examples of the if special form:

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

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

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

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

Example 11.1.2

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

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

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

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

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

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

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

\[
\begin{align*}
;; \ \text{HOW-BIG} \\
;; \ \text{-----------------------------------} \\
;; \ \text{INPUT: NUM, a number} \\
;; \ \text{OUTPUT: The symbol SMALL, if NUM is less than 10;} \\
;; \ \quad \text{Otherwise, the symbol BIG.}
\end{align*}
\]

\(\text{(define how-big}\)
(lambda (num)
  (if (< num 10)
      'small
      'big)))

The following interactions demonstrate its behavior:

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

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

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

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

Example 11.1.4

The following are legal instances of the if special form:

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

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

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

Example 11.1.5

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

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

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

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:

```
> (maxx 2 3)
3
> (maxx 5 1)
5
> (maxx 4 4)
4
```

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:

```scheme
;; MAXX
;; -------------------------------------------
;; INPUTS: X, Y, two numbers
;; OUTPUT: The maximum of X and Y
```

And some tester expressions:

```scheme
(tester '(maxx 2 3))
(tester '(maxx 5 1))
(tester '(maxx 4 4))
```

**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:

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

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

```scheme
> (banana-msg 3)
I ate 3 bananas! ← these are not output values!!
> (banana-msg 1)
```
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 \texttt{#f} is interpreted as boolean \textit{true}). The only catch is that the non-strict version of the \texttt{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 \textit{true}, all will be well.

\begin{example}
\textbf{Example 11.2.3}

The following Interactions Window session demonstrates the behavior of \texttt{and} with non-strict truth values:

\begin{quote}
\begin{itemize}
  \item \texttt{> (and 3 4)} \hfill \texttt{←} the output is 4, which counts as \textit{true}
  \item \texttt{> (and (* 3 4) (* 8 8))} \hfill \texttt{←} the output is 64, which counts as \textit{true}
  \item \texttt{> (and (* 3 4) (= 9 7))} \hfill \texttt{←} the output is boolean false
\end{itemize}
\end{quote}

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

\begin{in-class-problem}
\textbf{In-Class Problem 11.2.2}

Define a function, called \texttt{my-and}, satisfies the following contract:

\begin{quote}
\begin{verbatim}
;; 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)
\end{verbatim}
\end{quote}

\textit{Implement this function using the \texttt{if} special form; do not use the \texttt{and} special form.}

\begin{itemize}
  \item \textit{Because \texttt{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, \texttt{my-and} would be applied to the inputs 5 and 30, not the lists (+ 2 3) and (* 5 6).}
\end{itemize}
\end{in-class-problem}

More than two input expressions for the \texttt{and} special form. The \texttt{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 \texttt{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.

\begin{example}
\textbf{Example 11.2.4}

\begin{quote}
\begin{verbatim}
> (and #t #t #t #t)
#t
> (and #t #t #f #t #t)
#f
> (and (> 3 2) (= 9 9) (<= 5 20))
\end{verbatim}
\end{quote}
\end{example}
#t
> (and 1 2 3 4 5) ← the output value is 5, which counts as true
5
> (and 1 2 #f 4 5)
#f

Notice that if the input expressions are strict (i.e., expressions that evaluate to booleans), then the and expression will evaluate to a boolean. However, if one or more of the input expressions is non-boolean, then the and expression might evaluate to a non-boolean value.

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

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

In-Class Problem 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:

;;; 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:

```
> (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

Below, an equivalent function, called letter-grade-v2, is defined that uses a cond expression in-
stead 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)
    (#t '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 \textit{cond}_i \text{ is a (strict or non-strict) condition;}

(by convention) the last condition, \textit{cond}_n, is either \texttt{#t} or \texttt{else}; and

each \textit{expr}_i \text{ is some Scheme expression.}

The value of such a \textit{cond} expression is determined as follows:

Each condition, \textit{cond}_i, is evaluated in turn until one is found that evaluates to \texttt{#t}—or something that counts as true.

The value of the \textit{cond} expression is the value of the corresponding expression, \textit{expr}_i.

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

\textbf{The \textit{cond} special form, even more generally!} Recall that the body of a \texttt{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 \textit{cond} expression, each condition, \textit{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 \texttt{lambda} expression, it is the value of the last subsidiary expression that serves as the value of the \textit{cond} expression.

\begin{example}

\textit{Example 11.3.3}

\textit{Below, a function, \texttt{cond-effects}, is defined whose body contains a \textit{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.}

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

\end{example}
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)`) 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.

<table>
<thead>
<tr>
<th>Example 11.4.1: Defining a predicate using conditional expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 <code>cafe-open?</code>, that satisfies the following contract:</td>
</tr>
<tr>
<td>```scheme</td>
</tr>
<tr>
<td>;; CAFE-OPEN?</td>
</tr>
</tbody>
</table>
| ```
;; INPUTS: DAY, a symbol, one of SUN, MON, TUE, ..., FRI, SAT
;; AM-OR-PM, a symbol, either AM or PM
;; HOUR, an integer from 1 to 12, inclusive
;; MINUTES, an integer from 0 to 59, inclusive
;; OUTPUT: #t, if the inputs specify a time at which the
;; CAFE CMPU-101 is open; #f, otherwise.

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

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

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

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

(define cafe-open?
(lambda (day am-or-pm hour minutes)
(cond
;; Case 1: Open after 11:30 pm on Wednesdays
((and (eq? day 'wed)
      (eq? am-or-pm 'pm)
      (= hour 11)
      (> minutes 30))
   #t)
;; Case 2: Open all day on Thursdays
((eq? day 'thu)
 #t)
;; Case 3: Open Friday mornings *before* 9
;; (including times such as 12:25 a.m.)
((and (eq? day 'fri)
      (eq? time-of-day 'am)
      (or (< hour 9) (= hour 12)))
   #t)
;; Case 4: Open Friday mornings between 9 and 9:15
((and (eq? day 'fri)
      (eq? time-of-day 'am)
      (= hour 9)
      (<= minutes 15))
   #t)
;; Case 5: Closed at all other times
   (else
    #f))))
Note that the cases in this cond expression can be built up incrementally. For example, we could have started with just case 1 and the else case. When those were working, we could’ve inserted case 2, testing to make sure the new case was working before inserting case 3, and so on, until all cases were working.

Example 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}.

\begin{definition}
\begin{center}
\textbf{Definition 11.1: Equivalent boolean conditions}
\end{center}
\end{definition}

\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.}

\begin{example}
\begin{center}
\textbf{Example 11.5.1: Simplifying if expressions involving boolean conditions}
\end{center}
\end{example}

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

\begin{verbatim}
> (define x 32)
> (define y 4)
> (if (> x y) \#t \#f)
\#t
> (> x y)
\#t
> (define y 1000)
> (if (> x y) \#t \#f)
\#f
> (> x y)
\#f
\end{verbatim}

\begin{itemize}
\item More generally, if \textit{boolCond} is any boolean condition, then the following simplification yields an equivalent expression:

\begin{verbatim}
(if boolCond \#t \#f) \rightarrow boolCond
\end{verbatim}

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

\begin{verbatim}
(if boolCond \#f \#t) \rightarrow (not boolCond)
\end{verbatim}
\end{itemize}

So, if you ever find yourself writing an 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}.

\begin{example}
\begin{center}
\textbf{Example 11.5.2: Simplifying if expressions: non-strict truth values}
\end{center}
\end{example}

\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:

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

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:

\[
\begin{align*}
\text{not} & \quad (\text{not} \; (\text{and} \; e_1 \; e_2)) \quad \sim \quad (\text{or} \; (\text{not} \; e_1) \; (\text{not} \; e_2)) \\
\text{not} & \quad (\text{not} \; (\text{or} \; e_1 \; e_2)) \quad \sim \quad (\text{and} \; (\text{not} \; e_1) \; (\text{not} \; e_2))
\end{align*}
\]

### 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:

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

### 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 data-type-of, that satisfies the following contract. Note that it returns a symbol as its output.

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

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

> (data-type-of #t)
boolean
> (data-type-of ())
list
> (data-type-of 45)
number

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

(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")

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

Problem 11.6

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

(implies #t #t) ===> #t
(implies #f #t) ===> #t
(implies #t #f) ===> #f
(implies #f #f) ===> #t
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 ≤ mil-hours < 24` and `0 ≤ 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:

  ```scheme
  > (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 ≤ mil-hours < 24`. It should generate as its output, the corresponding number of hours according to the 12-hour civilian clock. For example:

  ```scheme
  (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:

\[
(10\% \text{ of } $10,000) + (15\% \text{ of } $20,000) + (25\% \text{ of } $70,000)
\]

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.

```scheme
;; BOOKSTORE-OPEN?
;; ---------------------------------------
;; INPUTS: DAY, a symbol, one of SUN, MON, TUE, ..., FRI, SAT
;; HOUR, an integer from 1 to 12, inclusive
```
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:

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

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

**Definition 12.2: Recursive function**

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

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

* A recursive function typically includes a conditional 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:
96  Introduction to Computer Science via Scheme  © 2017 Luke Hunsberger  Spring 2017

- 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:

```scheme
> (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:

```scheme
;; 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 \texttt{facty}. This function differs only in that it contains some \texttt{printf} expressions that will help us to trace what happens when an expression such as \texttt{(facty 3)} is evaluated:

\begin{verbatim}
;; FACTY
;; ---------------------------------------------
;; INPUT: N, a positive integer
;; OUTPUT: The factorial of N (i.e., N*(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
     (else (printf "Recursive Case (n = \A)\n" n)
           (* n (facty (- n 1)))))))
\end{verbatim}

Notice that the \texttt{printf} expressions do not affect the output of the function; they only cause some useful side-effect printing to occur.

\textbf{Evaluating (facty 3).} Consider DrScheme’s evaluation of the expression, \texttt{(facty 3)}. This is a function-call expression whose evaluation is governed by the Default Rule. Thus, the symbol \texttt{facty} and the number \texttt{3} must both be evaluated. The symbol \texttt{facty} evaluates to the function we just defined; and the number \texttt{3} evaluates to itself. Next, the \texttt{facty} function is applied to the input \texttt{3}. The application of the \texttt{facty} function to the input \texttt{3} is depicted at the top of Fig. 12.1. First, a local environment is created with an entry associating the input parameter \texttt{n} with the value \texttt{3}. Next, the expression in the body of the \texttt{facty} function, shown below, is evaluated with respect to that local environment.

\begin{verbatim}
(cond
  ;; Base Case: n = 1
  ((= n 1) (printf "Base Case (n = 1)\n") 1)
  ;; Recursive Case: n > 1
  (else (printf "Recursive Case (n = \A)\n" n)
        (* n (facty (- n 1))))))
\end{verbatim}
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, \((\ast n (\text{facty} (- n 1)))\), must be evaluated—according to the Default Rule. The \(*\) symbol evaluates to the multiplication function, \( n \) evaluates to 3, and \((\text{facty} (- n 1))\) evaluates to \#f. Gosh, we need a new paragraph!

The expression, \((\text{facty} (- n 1))\), is evaluated according to the Default Rule. First, the \text{facty} symbol evaluates to the \text{facty} function; and \((- n 1)\) evaluates to 2 (since \( n \) has the value 3). Next, the \text{facty} function must be applied to the input value 2, as depicted in the second box in Fig. 12.1.

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

The application of the \text{facty} function to the value 2, depicted in the second function-call box in the figure, is similar to the application of the \text{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.

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

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 \texttt{printf} expression causes the line, Recursive Case \((n = 2)\), to be displayed in the Interactions Window; and the evaluation of the expression, \((\ast n (\text{facty} (- n 1)))\), leads to yet another recursive function call—this time the application of the \text{facty} function to the input value 1, as illustrated in the third box in Fig. 12.1.

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

The application of the \text{facty} 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 \texttt{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 \texttt{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 \text{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, \((\text{facty} (- n 1))\), that was being evaluated in the middle function-call box (where \( n \) has the value 2). Now that that the value of \((\text{facty} (- n 1))\) is in hand, the evaluation of the expression, \((\ast n (\text{facty} (- n 1)))\), 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, \((\text{facty} (- n 1))\), that was being evaluated in the top function-call box (where \( n \) has the value 3). Now that that the value of \((\text{facty} (- n 1))\) is in hand, the evaluation of the expression, \((\ast n (\text{facty} (- n 1)))\), 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 `(facty 3)` is evaluated in the Interactions Window:

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

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

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 `sum-to-n`, that satisfies the following contract:

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

**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
;; -----------------------------------------
```
\[ \textbf{facty} \]

\begin{figure}
\centering
\begin{tikzpicture}[scale=0.8, auto, ->, node distance=2cm, semithick]
  \node (n3) [rectangle, draw] {\textbf{facty}};
  \node (n1) [rectangle, draw, below of=n3, xshift=1cm] {n \ 3};
  \node (n2) [rectangle, draw, below of=n1] {n \ 2};
  \node (n4) [rectangle, draw, below of=n2] {n \ 1};

  \path (n3) edge (n1)
        (n1) edge node [above] {3} (n3)
        (n1) edge node [below] {\textit{Recursive Case}} (n2)
        (n2) edge node [above] {6} (n3)
        (n2) edge node [below] {\textit{Recursive Case}} (n4)
        (n3) edge node [above] {6} (n4)
        (n4) edge node [below] {1} (n3)
        (n4) edge node [above] {1} (n2);

  \end{tikzpicture}
\end{figure}

\textbf{Figure 12.1: DrScheme’s evaluation of} \((\textbf{facty} \ 3)\)
;;; 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  p\text{th}  power (i.e.,  x^p ), as illustrated below.

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

* Hint: Use recursion, similar to how it is used in facty. For example, note that  2^0 = 2 \cdot (2^0).  
* 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, `(facty 5)`, effectively requires evaluating `(facty 4)`, `(facty 3)`, `(facty 2)` and `(facty 1)`. Similarly, evaluating `(facty 100)` would involve a sequence of nearly one hundred recursive function calls. For functions such as `facty`, the evaluation of each recursive function call is suspended pending the evaluation of all of the subsidiary function calls. Keeping track of all of these suspended evaluations requires storing relevant information somewhere in the computer’s memory. Thus, if the value of `n` gets large enough, DrScheme’s evaluation of `(facty n)` would eventually cause problems. In particular, at some point, the operating system would refuse to grant DrScheme more memory to hold the needed information.

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

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

---

**Example 12.2.1: Printing Dashes**

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

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

(define print-n-dashes
  (lambda (n)
    (cond
      ;; Base Case: n <= 0
      ((<= n 0) (newline))
      ;; Recursive Case: n > 0
      (#t
        ;; Print one dash
        (printf "-
")
        ;; 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.

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

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 \text{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:

\* 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.

\* 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, \((\text{print-n-dashes} \ 5)\), \((\text{print-n-dashes} \ 4)\), \ldots, \((\text{print-n-dashes} \ 0)\), 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 \( \text{expr} \) is a recursive function-call expression somewhere within \( B \). We say that \( \text{expr} \) is a tail-recursive function-call expression within \( B \) if, whenever evaluating \( B \) requires evaluating \( \text{expr} \), it is necessarily the case that the last step in evaluating \( B \) is the evaluation of \( \text{expr} \) and, thus, the value of \( B \) is identical to the value of \( \text{expr} \). If every recursive function-call expression in the body of \( f \) is tail-recursive, then \( f \) is called a tail-recursive function.

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

Example 12.2.2

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

```
(define rec-func
  (lambda (...
    (cond
      ;; Base Case
      (... ...)
      ;; Recursive Case
      (... ... ... (rec-func ...))))
)
)
```

The recursive function-call expression must not be a subsidiary expression within some larger expression within the recursive case; it must be the entirety of the last (i.e., tail) expression. If that is the case, then whenever the recursive case applies, the value for the entire \text{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 \text{print-n-dashes} function from Example 12.2.

In contrast, consider the definition of the \text{facty} function, seen earlier:

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

Notice that the last expression in the recursive case of the \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{facy}, is not tail recursive.

\footnote{This is actually the \texttt{facy-v2} function, but the same points apply to all versions of the \texttt{facy} function seen earlier.}

In-Class Problem 12.2.1

Define a function that satisfies the following contract:

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

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{verbatim}
(define tester-alt
  (lambda (datum)
    (printf "A ==>" datum)
    (newline)
    (eval datum)
    (newline)))
\end{verbatim}

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

> (print-thing-n-times 'Hi 5)
HiHiHiHiHi
> (print-thing-n-times '--- 3)
---------

Be sure to include a contract for your function.

Problem 12.6

Define a function, called print-down-to-zero, that takes a non-negative integer 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 n down to zero in the Interactions Window, as illustrated below.

> (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

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 print-n-dashes function from Example 12.2 into your Definitions Window. Recall that print-n-dashes does not generate any Scheme output value; instead, it has the side effect of displaying a row of n dashes in the Interactions Window, as illustrated below:

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

(Printing Rectangles) For this part, you must define a function called 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:

> (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:

;;; 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:

;;; 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, \( \text{FROM}, \text{FROM+1}, \text{FROM+2}, \ldots, \text{TO-2}, \text{TO-1}, \text{TO} \).

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 \texttt{facty} from Example 12.1.1:

\[
> \text{(plotter facty 0 4)}
\]

*  
*  
**  
*****  
************************

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

\[
> \text{(plotter abs -3 3)}
\]

***  
**  
*  
*  
**  
***

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

\[
> \text{(plotter (lambda (x) (* x x)) 1 5)}
\]

*  
****  
******  
*********  
****************************

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

\begin{center}
Problem 12.11
\end{center}

\[\Rightarrow\] This problem assumes that you have already defined the \texttt{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 \texttt{fancy-plotter} that satisfies the following contract:

\[
; ; \text{FANCY- PLOTTER}
; ; \text{-------------}
; ; \text{INPUTS: \hspace{1em} FUNC, a function that takes numerical input}
; ; \hspace{1em} \text{FROM, a starting number (integer)}
\]
;; TO, a stopping number (integer)
;; OUTPUT: None
;; SIDE EFFECTS: This function plots the function values for FUNC
;; for each input in the range from FROM to TO. Each input
;; value will generate one line of printing in the Interactions
;; Window. For example, if (FUNC FROM) ==> 5, then this
;; function will display a line of five + signs; if (FUNC FROM)
;; ==> -5, then this function will display a line of five -
;; signs; if (FUNC FROM) ==> 0, then this function will simply
;; display a zero.

Here are some examples, one of which uses the built-in sin function:

> (fancy-plotter (lambda (x) (* x x x)) -3 3)

+---------------------------
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>--------\</td>
</tr>
<tr>
<td>---+\</td>
</tr>
<tr>
<td>0+\</td>
</tr>
<tr>
<td>+++++++\</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
</tbody>
</table>

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

+---------------------------
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>--------\</td>
</tr>
<tr>
<td>---+\</td>
</tr>
<tr>
<td>0+\</td>
</tr>
<tr>
<td>+++++++\</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
<tr>
<td>+++++++++++++++++++++++</td>
</tr>
</tbody>
</table>


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 \((\text{facty} \; 4)\) (i.e., \(4!\)) resulted in the recursive function calls, \((\text{facty} \; 3), (\text{facty} \; 2)\) and \((\text{facty} \; 1)\), whose values were \(3!, 2!\) and \(1!\), respectively. This section explores a slightly different way of organizing recursive computations using \textit{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.

\textbf{Example 12.3.1: Computing sums of the form, \(0 + 1 + 2 + \ldots + n\) without accumulators}

\begin{verbatim}
;; 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=\n) ...\n")
       (+ n (sum-to-n (- n 1))))))
\end{verbatim}

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

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

Example 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 \(\text{printf}\) expressions serve only to display useful information; they do not affect the output value.

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

(define sum-to-n-acc
  (lambda (n acc)
    (cond
      ;; Base Case: n = 0
      (= n 0)
      (printf "Base Case (n=\(^\text{A}\)=\(^\text{A}\)\)\^0" acc)
      ;; Return the accumulator!
      acc)
      ;; Recursive Case: n > 0
      (#t
       (printf "Recursive Case (n=\(^\text{A}\), acc=\(^\text{A}\))\(^0\)" 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 \(acc\) when calling this function. Thus, to compute the sum, \(0 + 1 + 2 + 3\), using this function, we would evaluate the expression, (sum-to-n-acc 3 0). Notice that the initial accumulator has a value of 0, which is akin to our basket being initially empty. Here’s what the evaluation of (sum-to-n-acc 3 0) looks like in the Interactions Window:

> (sum-to-n-acc 3 0)
Recursive Case (n=3, acc=0)
Recursive Case (n=2, acc=3)
Recursive Case (n=1, acc=5)
Base Case (n=0, acc=6)
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:

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

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 \( \text{sum-to-n-acc} 3 0 \). 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 \( \text{sum-to-n-acc} \) 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 \( \text{facy-acc} \):

\[
\begin{align*}
\text{;; FACTY-ACC} \\
\text{;; } \text{-----------------------------------------------} \\
\text{;; INPUTS: N, a positive integer} \\
\text{;; ACC, a number} \\
\text{;; OUTPUT: When called with ACC=1 the output is N!} \\
\text{;; } \text{(i.e., the factorial of N).} \\
\text{;; More generally, the output is: \( ACC \times N! \).} \\
\end{align*}
\]

\[
\text{(define facy-acc}
\text{ (lambda (n acc}
\text{ (cond}
\text{ ;; Base Case: n = 1}
\text{ (= n 1)}
\text{ (printf "Base Case (n=\(N\), acc="A")" acc)}
\text{ ;; Return the accumulator!}
\text{ acc))}
\text{))}
\]
Figure 12.2: DrScheme’s evaluation of (sum-to-n-acc 3 0)
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:

```
> (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
```

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:

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

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

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

---

**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 \(n\):

```
;;;; SUM-SQUARES-ACC
;;;; ----------------------------------------------------
;;;; INPUTS: N, a non-negative integer
;;;; ACC, a number (accumulator)
;;;; OUTPUT: If the accumulator is 0, then the output
;;;; is equal to the sum \(0\times0 + 1\times1 + 2\times2 + \ldots + N\times N\).
```
More generally, the output is the sum:

\[
\text{ACC} + 0^0 + 1^1 + 2^2 + \ldots + N^N.
\]

(define sum-squares-acc
  (lambda (n acc)
    (cond
      ;; Base Case: n <= 0
      (\(<= n 0\)
        (printf "Base Case: n=\(\wedge A\), acc=\(\wedge A\)^%n acc
        acc)
      ;; Recursive Case: n > 0
      (#t
        (printf "Recursive Case: n=\(\wedge A\), acc=\(\wedge A\)^%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:

\[
> (\text{sum-squares-acc 3 0}) \leftarrow 3^2 + 2^2 + 1^2 + 0^2 = 14
\]

Recursive Case: n=3, acc=0
Recursive Case: n=2, acc=9
Recursive Case: n=1, acc=13
Base Case: n=0, acc=14
14

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

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

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

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

Example 12.3.5: Approximating \(\pi\)

Mathematicians tell us that the value of \(\pi\) can be approximated using sums of the form shown below:

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

where \(n\) is some positive odd number. Furthermore, as the value of \(n\) increases, the approximation becomes better and better. This example defines a function, called \(\text{approx-pi-acc}\), that processes terms in the above sum from left to right, using several inputs to keep track of relevant information along the way. On successive recursive function calls, the input \(\text{from}\) will be 1, then 3, then 5, etc. It will be used to identify the particular term in the sum that is currently being processed. The input \(\text{sign}\) will alternate between 1.
and −1 and, thus, keeps track of the sign of the current term. The input \( n \) will not change on successive recursive function calls. It is used as a fixed upper bound that indicates the last term in the sum. And the input \( \text{acc} \) is used to accumulate the desired sum. Here is the contract:

```scheme
;; APPROX-PI-ACC
;; ---------------------------
;; INPUTS: FROM, a positive odd number that specifies the term
;; that is currently being processed
;; SIGN, either +1 or -1, the sign of the term currently
;; being processed
;; N, a positive odd number that specifies the last term
;; in the sum
;; ACC, an accumulator
;; OUTPUT: When called with FROM=1, SIGN=1, and ACC=0, computes
;; the following estimate of the value of PI:
;; 4 * (1 - 1/3 + 1^5/5 - 1/7 + ... (+/-) 1/n)
```

```scheme
(define approx-pi-acc
 (lambda (from sign n acc)
   (cond
    ;; Base Case: FROM > N (i.e., we’ve gone too far!)
    (>(from n)
      ;; Multiply the accumulator by 4:
      acc)
    ;; Recursive Case: FROM <= N
    (else
      ;; Tail-recursive function call with adjusted inputs
      (approx-pi-acc
        (+ from 2.0) ;; increment by 2
        (* sign -1) ;; alternate between 1 and -1
        n ;; fixed upper bound
        (+ acc (/ sign n)) ;; accumulate current term
      )))))
```

Notice how the accumulator is multiplied by 4 in the base case. In addition, \( \text{from} \) is incremented by 2.0 to ensure that the computations are done using floating-point numbers instead of fractions. (To see the difference, try testing the function with \( \text{from} \) incremented by 2 instead of 2.0.) Here are some examples of its use:

```scheme
> (approx-pi-acc 1 1 3 0) ;; = 4*(1 - 1/3)
2.666666666666667
> (approx-pi-acc 1 1 5 0) ;; = 4*(1 - 1/3 + 1/5)
3.466666666666667
> (approx-pi-acc 1 1 101 0) ;; = 4*(1 - 1/3 + ... + 1/101)
3.1611986129970506
> (approx-pi-acc 1 1 10001 0) ;; = 4*(1 - 1/3 + ... + 1/10001)
3.1417926135957908
> (approx-pi-acc 1 1 1000001 0) ;; = 4*(1 - 1/3 + ... + 1/1000001)
3.1415946535856922
```

Notice how big the input \( n \) must be to get even modestly accurate approximations of \( \pi \).
Example 12.3.6: Approximating $e$

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

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

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

- the input parameter $n$, which indicates the last term in the sum, will stay the same across all recursive function calls;
- the input parameter `indy` will take on the values, 0, 1, 2, ..., $n$, on successive recursive function calls, and will be used to identify the current term;
- the input parameter `curr-denom` (i.e., current denominator) will accumulate the factorials that comprise the various denominators that appear in the sum (i.e., 1, 1, 2, 6, 24, ..., $n!$); and
- the input parameter `acc` will accumulate the desired sum; it will take on the values 1, 2, 2.5, 2.66666666666, ....

;; APPROX-E-ACC
;; ----------------------------------------------------------------
;; INPUTS: N, non-negative integer (indicates last term)
;; INDY, non-negative integer (indicates current term)
;; CURR-DENOM, positive integer (current denominator)
;; ACC, accumulates desired sum
;; OUTPUT: When called with INDY=0, CURR-DENOM=1, and ACC=0,
;; the output is the following approximation of $e$:
;; $\frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{N!}$

(define approx-e-acc
  (lambda (n indy curr-denom acc)
    ;; Print out the values of the input parameters first...
    (printf "n=\n, indy=\n, curr-denom=\n, acc=\n\n" n indy curr-denom acc)
    (cond
      ;; Base Case: INDY > N (we’re done!)
      ((> indy n)
       ;; Return the accumulator!
       acc)
      ;; Recursive Case: INDY <= N
      (#t
       ;; Tail-recursive function call with adjusted inputs
       (approx-e-acc
        n ;; n doesn’t change
        (+ 1 indy) ;; increment indy
        (* (+ 1 indy) curr-denom) ;; update current denom
        (+ acc (/ 1.0 curr-denom))))))))
To get the desired results, the various input parameters must be properly initialized. In particular, the initial values for indy, curr-denom and acc must be 0, 1 and 0, respectively. Thus, the sum
\[ 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} \]
can be computed by evaluating \((\text{approx-e-acc } 4 \ 0 \ 1 \ 0)\), as illustrated below:

```
> (approx-e-acc 4 0 1 0)
n=4, indy=0, curr-denom=1, acc=0
n=4, indy=1, curr-denom=1, acc=1.0
n=4, indy=2, curr-denom=2, acc=2.0
n=4, indy=3, curr-denom=6, acc=2.5
n=4, indy=4, curr-denom=24, acc=2.6666666666666665
n=4, indy=5, curr-denom=120, acc=2.708333333333333
2.708333333333333
```

Notice that the \(n\) parameter stays fixed at 4 across all the recursive function calls. This parameter is used only to signal the end of the sum. The parameter \(indy\) takes on the integer values from 0 to 5. It identifies the current term. Thus, when \(indy\) is greater than \(n\), no further accumulation of terms is necessary. The parameter \(curr-denom\) represents the denominator of the term currently being worked on; thus, it takes on the values of the successive factorials: \(0!, 1!, 2!, 3!, \ldots\). Notice that these factorials are not computed from scratch each time; instead, the value of \(curr-denom\) is simply multiplied by \((+ \text{indy } 1)\) to generate the next factorial. Finally, the parameter \(acc\) accumulates the desired sum. By the time the base case is reached (i.e., when \(indy > n\)), the accumulator holds the desired answer. Thus, the accumulator is simply returned as the output value for this function.

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

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

### 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 facty-acc

The following defines a wrapper function, facty-wr, for the accumulator-based function, facty-acc, defined earlier. Notice that the wrapper function simply calls facty-acc with the accumulator appropriately initialized to 1.

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

(define facty-wr
```
(lambda (n)
    ;; Just call accumulator-based helper with ACC=1
    (facty-acc n 1)))

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

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

Example 12.4.2: A wrapper for approx-pi-acc

The function, approx-pi-acc, from Example 12.3.5, uses several inputs to keep track of relevant parts of the computation over the course of all of the recursive function calls. The following wrapper function, approx-pi-wr, shields the user from having to know the appropriate initial values for these additional inputs:

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

(define approx-pi-wr
   (lambda (n)
       (approx-pi-acc 1 1 n 0)))

Here are some examples of its use:

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

Example 12.4.3: A wrapper for approx-e-acc

The function, approx-e-acc, from Example 12.3.6, involved several accumulators. The following wrapper function, approx-e-wr, shields the user from having to know the appropriate initial values for these accumulators:

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

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

;;; APPROX-E-WR -- wrapper function for APPROX-E-ACC
;;; ________________________________________________
;;; INPUT: N, a non-negative integer
;;; OUTPUT: The value of the sum:
;;; 1/1! + 1/2! + 1/3! + ... + 1/N!
(define approx-e-wr
  (lambda (n)
    (approx-e-acc n 0 1 0)))

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

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

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

Example 12.4.4: A wrapper function for input validation

The facty-v1 function defined in Example 12.1.2 presumes that its input will be a positive integer. If it is applied to any other kind of input, bad things can happen. For example, if it is applied to a negative number, the facty-v1 function will go into an infinite loop, each recursive call moving further away from the base case. And if it is applied to a non-numeric input, it will generate an error (e.g., because the built-in = function cannot be applied to non-numeric input). To avoid these sorts of problems, we can provide a wrapper function for facty-v1 that checks whether the input is valid before applying facty-v1 to it. Here is its contract and definition, followed by some examples of its use. (The wrapper function makes use of the built-in integer? function, seen previously in Section 5.3, whose output is #t if and only if its input is an integer.)

;; FACTY-V1-WRAPPER
;; -------------------------------------------------------
;; INPUT: DATUM, anything
;; OUTPUT: If DATUM is a positive integer, the output
;;         is the factorial of DATUM; otherwise, the
;;         output is void.
;; SIDE EFFECT: If DATUM is not a positive integer, it
;;              prints out an error message

(define facty-v1-wrapper
  (lambda (n)
    (cond
     ;; Good case: N is a positive integer
     ((and (integer? n)
           (> n 0))
      (facty-v1 n))
     ;; Bad case: N is something else
     (else
      (printf "ERROR: Input must be a positive integer!\""))))))

> (facty-v1-wrapper 5)
120
> (facty-v1-wrapper -3)
ERROR: Input must be a positive integer!
> (facty-v1-wrapper 4.32)
ERROR: Input must be a positive integer!
> (facty-v1-wrapper ’xyz)
ERROR: Input must be a positive integer!

Although the process of input validation could be taken care of in the facty-v1 function itself, that would not be a good idea because it would occur on every recursive function call! It is much better to do the input validation once, in the wrapper function.

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:

```scheme
(define facty
  (lambda (n)
    (cond
      ;; Base Case: N <= 1
      ((<= n 1) 1)
      ;; Recursive Case: N > 1
      (else (* n (facty (- n 1)))))))
```

The `cond` special form is used to distinguish the base case from the recursive case. The recursive case involves applying `facty` not to $n$, but to $(- n 1)$. As a result, the chain of recursive function calls will eventually involve applying `facty` to 1, at which point the recursion stops.

The above function `facty` is not tail recursive since the recursive function call, `(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.

$$
g(5) = 1^2 + 2^2 + 3^2 + 4^2 + 5^2
= (1^2 + 2^2 + 3^2 + 4^2) + 5^2
= g(4) + 5^2
$$

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

```scheme
(define sum-squares
  (lambda (n)
    (cond
      ;; Base Case: N <= 1
      )
```
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.

\[
\begin{align*}
\text{(define print-n-dashes}
\text{(lambda (n))}
\text{ (cond}
\text{ ;; Base Case: } N \leq 0
\text{ ((<= n 0)}
\text{ (newline))}
\text{ ;; Recursive Case: } N > 0
\text{ (else}
\text{ (printf ",-\")}
\text{ (print-n-dashes (- n 1)))})
\end{align*}
\]

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.

\[
\begin{align*}
\text{(define facty-acc}
\text{(lambda (n acc))}
\text{ (cond}
\text{ ;; Base Case: } N \leq 1
\text{ ((<= n 1)}
\text{ acc)}
\text{ ;; Recursive Case: } N > 1
\text{ (else}
\text{ (facty-acc (- n 1) (* n acc))))})
\end{align*}
\]

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 \((\times n \text{ acc})\). In the base case, the accumulator is returned as the output value, since it has, by that time, accumulated the desired answer.

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

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

Problems

**Problem 12.12**

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

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

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

- The values of `num-rows` and `num-cols` should not change across the various recursive function calls, but the values of `curr-row` and `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 `curr-row` and `curr-col` be updated in this case?)*
- If the sum of `curr-row` and `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, `even?` and `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 `curr-row` and `curr-col` be updated in that case?

(b) Define a wrapper function called print-checkerboard that takes two inputs, `num-rows` and `num-cols`. It should cause a `num-rows`-by-`num-cols` checkerboard pattern to be displayed in the Interactions Window, as illustrated below:

```
> (print-checkerboard 3 6)
X _ X _ X _
_ X _ X _ X
X _ X _ X _
```

Note that this function should just call the function from part (a) with appropriate inputs. You may wish to use the tester-alt function when writing test cases in the Definitions Window.
Problem 12.13

The following functions use the built-in quotient and 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 \texttt{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}

Hint: Consider how dividing a number by ten, using the built-in quotient and 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 \texttt{num-occurs-acc} that satisfies the following contract:

\begin{verbatim}
;;; NUM-OCCURS-ACC
;;; --------------------------------------------
;;; INPUTS: DIGIT, an integer from 0 to 9, inclusive
;;; 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) \( \Rightarrow \) 4
\end{verbatim}

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

\begin{verbatim}
(define num-occurs-wr
  (lambda (digit num)
    (num-occurs-acc digit num 0)))
\end{verbatim}
Here are some examples of the desired behavior:

```scheme
> (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:

```scheme
> (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 \( x \in (-1, 1] \), 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 `approx-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.

```scheme
;; APPROX-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 `approx-log-acc`, define the following wrapper function:

```scheme
(define APPROX-LOG-WR
  (lambda (x to)
    ;; Call the acc-based helper function with FROM=1 and ACC=0
    (approx-log-acc x 1 to 0.0)))
```
Notice that \((\text{approx-log-wr } 0.5 4)\) 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 \((\text{approx-log-wr } 0.5 4)\) will evaluate to a good approximation of \(\log(1 + 0.5)\), since \(1 + 0.5 = 1.5\).

\[
\begin{align*}
\text{> (approx-log-wr 0.5 10)} & \quad \leftarrow x = 0.5 \\
 & 0.4054346478174603 \\
\text{> (log 1.5)} & \quad \leftarrow 1 + x = 1.5 \\
 & 0.4054651081081644
\end{align*}
\]

Although \(\text{approx-log-wr}\) 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 \((\text{approx-log-wr } 1 n)\) 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 \((\text{approx-log-wr } x n)\) 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:

\[
egin{align*}
1 + 10 + 10^2 + 10^3 &= 1 + 10 + 100 + 1000 = 1111 \\
1 + 2 + 2^2 + 2^3 + 2^4 &= 1 + 2 + 4 + 8 + 16 = 31 \\
1 + 3 + 3^2 + 3^3 &= 1 + 3 + 9 + 27 = 40
\end{align*}
\]

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

\[
1 + x + x^2 + x^3 + \ldots + x^n
\]

where terms of the form \(x^k\) stand for “\(x\) raised to the \(k\)th power”. Your job is to define a function, called \(\text{geom}\), 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 \(\text{geom-helper-acc}\), 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:

\[
\begin{array}{cccccc}
\hline
\text{k} & 0 & 1 & 2 & 3 & 4 & 5 & \leftarrow \text{We’ll stop here, since } k > 4 \\
\text{x-to-the-k} & 1 & 2 & 4 & 8 & 16 & \text{32} \\
\text{acc} & 0 & 1 & 3 & 7 & 15 & 31 & \leftarrow \text{That’s the desired answer!!} \\
\hline
\end{array}
\]
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:

$1 + 0 \Rightarrow 1; \quad 2 + 1 \Rightarrow 3; \quad 4 + 3 \Rightarrow 7; \quad 8 + 7 \Rightarrow 15; \quad 16 + 15 \Rightarrow 31.$

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.

---

**Problem 12.16: Approximating the arctangent function**

Mathematicians tell us that the arctangent function can be approximated using sums of the following form:

$$\arctan(x) = 1 - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots \pm \frac{x^n}{n}$$

where $n$ is any odd positive integer. Define a function, called approx-arctan-acc, that uses the following inputs to keep track of relevant information over the course of the recursive function calls:

- **x**: Fixed value
- **from**: Positive odd integer, numerator of current term
- **sign**: Alternates between 1 and $-1$
- **curr-power**: Current power of $x$ (computed incrementally)
- **n**: Fixed value, indicates last term in the sum
- **acc**: An accumulator

What initial values should be given to the inputs from, sign, curr-power and acc?
Chapter 13

Defining Local Variables with the let, let* and letrec Special Forms

This chapter introduces the let special form along with its more general variants, let* and letrec. The purpose of a let special form is to 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 let special form. The value of the last expression in its body is taken to be the value of the entire let expression. Once a let expression has been evaluated, its local environment typically vanishes.¹

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

13.1 The let Special Form

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

13.1.1 The Syntax of the let Special Form

The syntax of the let special form is as follows:

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

¹There 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/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:

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

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

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

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/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: \texttt{\textbar}A, y: \texttt{\textbar}A, z: \texttt{\textbar}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 this let expression.

The following example introduces a destructive built-in function called \texttt{random} that has many uses, one of which is to demonstrate the need for the 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.

\begin{example}{The built-in \texttt{random} function}

\textit{Scheme includes a built-in function called \texttt{random} that can be used to generate pseudo-random numbers. Unlike all of the functions that we have seen so far in this book, the \texttt{random} function has the unusual property that successive applications of it to the same input can generate different output values! This can happen because the computations it performs to generate its output depend on the values of secret global variables that it destructively modifies. 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 \texttt{random} function satisfies the following contract.

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

Here are some examples demonstrating its behavior:

\begin{verbatim}
> (random 2)  \rightarrow  output will be 0 or 1
0
> (random 2)
1
> (random 2)
0
> (random 6)  \rightarrow  output will be in \{0, 1, 2, 3, 4, 5\}
4
> (random 6)
3
> (random 6)
5
\end{verbatim}

In general, when called with an input \(n\), the \texttt{random} function returns one of the \(n\) numbers in the set \{0, 1, 2, \ldots, n-1\}.\end{example}
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 \texttt{let} expression, followed by the evaluation of an equivalent expression involving the application of a \texttt{lambda} function to some inputs.

\begin{verbatim}
> (let ((x (+ 2 3))
       (y (* 3 4)))
   (printf "x: \texttt{~A}, y: \texttt{~A}\%" x y)
   (+ x y))
 x: 5, y: 12
17
> ((lambda (x y)
    (printf "x: \texttt{~A}, y: \texttt{~A}\%" x y)
    (+ x y))
    (+ 2 3)
    (* 3 4))
 x: 5, y: 12
17
\end{verbatim}

⇒ \textit{The semantics for the evaluation of the first expression is identical to the semantics for the evaluation of the second expression!}

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

The evaluation of the second expression is governed by the Default Rule for evaluating non-empty lists. The first entry in the list is a \texttt{lambda} expression. It evaluates to a function. The other entries, (+ 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 \texttt{x} and \texttt{y} are associated with the values 5 and 12, respectively. Then the body of the \texttt{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, \texttt{z}. It then evaluates a \texttt{let} expression and an equivalent expression involving the application of a \texttt{lambda} function.

\begin{verbatim}
> (define z 1000)
> (let ((x 3)
       (y 4))
   (* x y z))
12000
> ((lambda (x y)
    (* x y z))
    3
    4)
12000
\end{verbatim}

Once again, the evaluation of the two expressions is the same. In particular, each involves a local environment containing entries for \texttt{x} and \texttt{y}, with the respective values 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 \texttt{x} and \texttt{y} are drawn from the local environment, whereas the value for \texttt{z} is drawn from the Global Environment. In each case, the value of the entire expression is 12000.
In general, a `let` expression of the form,

\[
(\text{let} \ ((\text{var}_1 \ \text{val}_1) \\
\text{\hspace{1em}}(\text{var}_2 \ \text{val}_2) \\
\text{\hspace{2em}}\cdots \\
\text{\hspace{3em}}(\text{var}_n \ \text{val}_n)) \\
\text{expr}_1 \\
\text{\hspace{1em}}\text{expr}_2 \\
\text{\hspace{2em}}\cdots \\
\text{\hspace{3em}}\text{expr}_k)
\]

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

\[
((\lambda \ (\text{var}_1 \ldots \text{var}_n) \\
\text{\hspace{1em}}\text{expr}_1 \\
\text{\hspace{2em}}\text{expr}_2 \\
\text{\hspace{2em}}\cdots \\
\text{\hspace{3em}}\text{expr}_k) \\
\text{val}_1 \ldots \text{val}_n)
\]

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

**Problems**

### Problem 13.1

This problem has two parts. The first part can be implemented without using `let`; the second part is best implemented using `let`.

(a) Define a function called `print-n-tosses` that satisfies the following contract:

```
;; 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.
```

Here are some examples:

```
> (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
```

(b) Define a function called `print-and-sum-n-tosses` that satisfies the following contract.

```
;; 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.
```
Here are some sample interactions:

```scheme
> (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 `sum-the-even-tosses`, that satisfies the following contract:

```scheme
;; SUM-THE-EVEN-TOSSES
;; ---------------------------------------------
;; INPUTS:  N, a non-negative integer
;; OUTPUT: The sum of the even tosses out of N random tosses
;; SIDE EFFECT: Print out the tosses along the way.
```

Here is an example of its use:

```scheme
> (sum-the-even-tosses 5)  
3 6 2 1 5  
8  
> (sum-the-even-tosses 7)  
2 4 1 6 6 3 2  
20
```

**Problem 13.3**

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

```scheme
;; 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 13.4: 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 max-streak-in-n-flips, that satisfies the following contract:

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

Here are some examples of its use:

(max-streak-in-n-flips 10) ===> T H H H T T T H H ... 4
(max-streak-in-n-flips 10) ===> T H T H T T T T H ... 3
(max-streak-in-n-flips 15) ===> T T H T H T T H T H H T T H ... 3

In the first sequence, the longest streak involves four consecutive Hs. In the second and third sequences, the longest streak involves three consecutive Hs.

⋆ Begin by defining a separate helper function, called max-streak-in-n-flips-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, max-streak-in-n-flips.
Consider the following sequence of coin flips:

H T H T T H T H H H T T H T ...

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

**Problem 13.5**

Define a function, `num-tosses-until-repeat`, that satisfies the following contract:

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

Here are some examples that illustrate the desired behavior:

```
> (num-tosses-until-repeat)
3 5 1 3 4 2 3 6 1 3 2 2 --> 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:

```scheme
;; 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 13.6

Define a function, called `num-tosses-until-three`, that satisfies the following contract:

```
;; NUM-TOSSES-UNTIL-THREE
;; -----------------------------------------------
;; INPUTS: None
;; OUTPUT: An integer representing the number of random
;; tosses of a 6-sided die that were required before three
;; CONSECUTIVE tosses came out the same.
;; SIDE EFFECTS: Prints out the tosses along the way
```

Here are some examples of it in action:

```
> (num-tosses-until-three)
5 2 1 4 4 4 →→ We got three in a row! ← side-effect printing
6
> (num-tosses-until-three)
1 1 4 6 6 3 6 5 3 4 5 1 5 2 4 6 2 1 3 1 3 5 2 1 2 5 6 1 6 3 4 3
1 4 1 4 6 5 3 4 4 6 2 4 2 2 5 3 5 5 5 →→ We got three in a row!
51
```

Hint: Define a helper function, called `num-tosses-until-three-helper`, that does most of the work:

```
;; NUM-TOSSES-UNTIL-THREE-HELPER
;; -----------------------------------------------
;; INPUTS: NUM-TOSSES-SO-FAR, an integer
;; PREV-TOSS-1, PREV-TOSS-2, the most recent tosses
;; (or #f if just getting started)
;; OUTPUT: When called with NUM-TOSSES-SO-FAR = 0, and
;; PREV-TOSS-1 = PREV-TOSS-2 = #f, outputs the number of
;; random tosses of a 6-sided die before three consecutive
;; tosses came out the same.
;; SIDE EFFECTS: Prints out the tosses along the way.
```

Problem 13.7

Define a function, called `toss-until-doubles`, that satisfies the following contract:

```
;; TOSS-UNTIL-DUORES
;; -----------------------------------------------
;; 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
```
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 13.8

Define a function, called toss-three-dice-until-beat-target, that satisfies the following contract:

;;; 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

Problem 13.9

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
You’ll notice that the only difference is the asterisk in the name of the special form: \texttt{let*} instead of \texttt{let}.

### 13.2.2 The Semantics of the \texttt{let*} Special Form

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

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

\[\Rightarrow\] 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, \ldots, \texttt{var}_i.

- Then the expressions in the body of the \texttt{let*} are evaluated, in turn.
- The value of the last expression in the body of the \texttt{let*} serves as the value of the entire \texttt{let*} expression.

#### Example 13.2.1

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

```scheme
> (let* ((x 4)
         (y (+ x 2))
         (z (* x y))
         (w (+ x y z)))
   (printf "x: ~A, y: ~A, z: ~A, w: ~A\%" x y z w)

x: 4, y: 6, z: 24, w: 34
68
```

Notice that the expression, \((+ x 2)\), that is used to compute the value for \texttt{y} refers to the local variable \texttt{x}. Similarly, the expression, \((* x y)\), that is used to compute the value for \texttt{z} refers to both \texttt{x} and \texttt{y}. Finally, the expression, \((+ x y z)\), 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 DrScheme to complain.

```scheme
> (let ((x 4)
         (y (+ x 2))
         (z (* x y))
         (w (+ x y z)))
   (printf "x: ~A, y: ~A, z: ~A, w: ~A\%" x y z w)

... reference to undefined identifier: x
```

The reason is due to the difference in the way \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.
13.2.3 Deriving a Single \texttt{let*} Expression from Nested \texttt{let} Expressions

In general, a \texttt{let*} expression of the form,

\begin{verbatim}
(let* ((var1 val1)
       (var2 val2)
       ...
       (var\_n val\_n))
  expr\_1
  expr\_2
  ...
  expr\_k)
\end{verbatim}

is equivalent to \textit{n} nested \texttt{let} expressions:

\begin{verbatim}
(let ((var1 val1))
  (let ((var2 val2))
    ...
    (let ((var\_n val\_n))
      expr\_1
      expr\_2
      ...
      expr\_k)))
\end{verbatim}

The following example demonstrates the equivalence.

\textbf{Example 13.2.2}

The following Interactions Window session evaluates a \texttt{let*} expression and the equivalent nested \texttt{let} expression:

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

Notice that the outermost \texttt{let} expression (i.e., the one that specifies the local variable \texttt{x}) has a body that consists of a single \texttt{let} expression (i.e., the one that specifies the local variable \texttt{y}). Because the \texttt{let} expression for \texttt{y} is evaluated with respect to the local environment containing an entry for \texttt{x}, it is okay for the value expression, \texttt{(+ x 2)}, to refer to \texttt{x}. Similar remarks apply to the remaining variables.

In general, \texttt{let*} provides a simpler syntax than the equivalent set of nested \texttt{let} expressions. Thus, if you ever need to do incremental computations where the value of each local variable depends of the values of the preceding local variables, then you should consider using \texttt{let*}. 
Problems

Problem 13.10: 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 letrec Special Form

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

13.3.2 The Semantics of the letrec Special Form

In sharp contrast to how the let and let* special forms are evaluated, the evaluation of a letrec special form begins by creating the entire local environment, complete with entries for all of the local variables, before evaluating any of the value expressions. Because none of the value expressions have yet been evaluated, each local variable is initially given the dummy value, #<undefined>. However, since all of the local variables have corresponding entries in the local environment before any of the value expressions are evaluated, each value expression can refer to any or all of the local variables, whether they have values or not!

Example 13.3.1

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

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

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

Example 13.3.2: Using letrec to create a local recursive function

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

```
> (let ((factyOne (lambda (n)
                   (if (<= n 1)
                       1
                       ;; The following reference to factyOne
                       ;; causes an error!!
                       (* n (factyOne (- n 1)))))
       (factyOne 4))
ERROR: reference to undefined identifier: factyOne
> (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 vali expression inside a let* can refer to previously defined local variables, it cannot refer to the current variable, vari—because no entry exists yet for vari. (For this reason, a let* that includes only one vari/vali 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 a 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))))

> (facty 4)
24
> (facty 5)
120

This kind of application of letrec is commonly used to hide the existence of a recursive helper function from users who may not understand what inputs to give it, or may not want to be bothered with thinking about what inputs to give it. The helper function only exists for use by the parent function; it is not visible to the general programming public. The parent function (facty) takes care of supplying the helper function (facty-acc) with appropriate inputs.

* Take care when defining local recursive helper functions. For example, note the difference between the input \( n \) to facty and the input \( m \) to facty-acc. On successive recursive function calls, \( m \) takes on different values, while \( n \) never changes.

Problems

Problem 13.11

Mimicking the structure of facty and facty-acc from Example 13.3.3, define a function called sum-cubes that uses letrec to define an accumulator-based, tail-recursive local helper function called sum-cubes-acc. The sum-cubes function should satisfy the following contract:

```scheme
;; SUM-CUBES
;; ---------------------------------------------------------
;; INPUTS: N, a positive integer
;; OUTPUT: The sum: 1\*1\*1 + 2\*2\*2 + ... + N\*N\*N
```

Here are some examples of its desired behavior:

> (sum-cubes 3)
36
> (sum-cubes 4)
100

Problem 13.12

Same as Problem 13.1b, except that you should use letrec to define the recursive helper function as a local function.

Problem 13.13

Same as Problem 13.3, except that you should use letrec to define the recursive helper function as a local function.

Problem 13.14

Same as Problem 13.5, except that you should use 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)
Chapter 14

Lists and List-Based Recursion

Previous chapters have highlighted the many important roles that non-empty lists play in Scheme’s computational model. For example, the Default Rule for evaluating non-empty lists can be used to apply functions to inputs, the define special form can be used to assign values to variables, the quote special form can be used to shield a datum from evaluation, and so on. In contrast, this chapter focuses on lists as *containers of data*. When viewing lists as containers of data, we typically don’t want them to be evaluated. In addition, to do any meaningful computations involving lists (e.g., to sort a list of numbers or recursively walk through a list of data), we need to be able to access the individual elements. Finally, we will often want to be able to construct lists incrementally, for example, by attaching a new element to the front of a list.

Scheme provides the following built-in functions to facilitate the use of lists as containers of data:

- **first** to access the *first* element of a list
- **rest** to access the *rest* of a list
- **cons** to construct a new list by attaching a new element to the front of an existing list

These few functions, together with the `null?` type-checker predicate from Chapter 8, will enable us to design functions that can recursively process the elements in a list.

We shall see that list-based recursion is quite similar to numerical recursion. Whereas numerical recursion is driven by the size of a numerical input, list-based recursion is driven by some feature of a list—usually whether that list is empty or not. In list-based recursion, there is a base case—usually signaled by the empty list (analogous to $n = 0$); and there is a recursive case—usually signaled by a non-empty list (analogous to $n > 0$). And, just as a numerical-recursive function can typically process numerical inputs of any size, a list-based recursive function can typically process lists containing any number of elements.

### 14.1 The Built-in Functions: first, rest and cons

This section describes the built-in functions, first, rest and cons, that Scheme provides to enable us to access parts of lists, and to attach new elements to pre-existing lists.

The *first and rest accessor functions*. The **first** and **rest** functions are called *accessor functions* because they enable us to access certain parts of a non-empty list. The contracts for these built-in functions are given below.

```scheme
;; first input: LISTY, a non-empty list
;; output: The FIRST element of LISTY
```
**Example 14.1.1**

The following Interactions Window session demonstrates the use of the `first` and `rest` accessor functions to access the parts of a non-empty list.

```scheme
> (first '(a b c d e))
a
> (rest '(a b c d e))
(b c d e)
> (first '(64))
64
> (rest '(64))
() ← the rest is a list, even if it is empty
```

**Example 14.1.2: Accessing other elements of a non-empty list**

We can combine the `first` and `rest` functions to access any individual element of a list, as follows:

```scheme
> (first (rest '(a b c d e)))   ← access second element
b
> (first (rest (rest '(a b c d e))))  ← access third element
c
> (first (rest (rest (rest '(a b c d e))))) ← access fourth element
d
```

Rather than re-typing these sorts of cumbersome expressions to access various elements of a list, we can define functions to simplify the process, as illustrated below:

```scheme
;; SEKUND/THURD/FORTH
;; --------------------------------
;; INPUT:  LISTS, a list containing at least two elements
;; OUTPUT:  The second/third/fourth element of LISTS

(define sekund
  (lambda (listy)
    (first (rest listy)))))

(define thurd
  (lambda (listy)
    (first (rest (rest listy))))))

(define forth
  (lambda (listy)
    (first (rest (rest (rest listy)))))))
```
The following interactions demonstrate the use of these functions:

> (sekund '(a b c d e))
b
> (thurd '(yes #t 383 () why))
383
> (forth '(my bonnie lies over the ocean))
over

Although we could continue in this fashion, defining additional accessor functions called fiftth, sicksth, and so on, we shall soon discover that there is a much easier way to access any desired element of a list: using recursion! In the meantime, you should know that Scheme provides a slew of built-in functions for accessing individual elements of a list in the manner seen above. They are called second, third, fourth, etc. As you may have guessed, the existence of these built-in functions is the reason that I gave names such as sekund, thurd and forth to the functions defined above.

In-Class Problem 14.1.1: Checking for a one-element list

Define a function, called one-elt-list?, that satisfies the following contract:

;;; ONE-ELT-LIST?
;;; ---------------------------------------------
;;; INPUTS: LISTY, any list
;;; OUTPUT: #t if LISTY contains *exactly* one element;
;;; #f otherwise.

Here are some examples of the desired behavior:

> (one-elt-list? ())
#f
> (one-elt-list? '(xyz))
#t
> (one-elt-list? '(a b c d))
#f

Hint: Use some of these: null?, first, rest.

Using cons to construct a new list. The built-in cons function constructs a new list by attaching a new element onto the front of an existing list. Here is its contract:

;;; CONS -- built-in function
;;; ---------------------------------------------
;;; INPUTS: FST, any Scheme datum
;;; RST, a list (either empty or non-empty)
;;; OUTPUT: A new list whose FIRST element is FST, and the REST of whose elements are RST.

* When using the cons function to construct a new list, the second input must be a list!
Example 14.1.3

The following Interactions Window session demonstrates the use of the cons function.

> (cons 8 '(a b c))
(8 a b c)
> (cons 'john '(paul george ringo))
(john paul george ringo)
> (cons 64 ()) ← the second input must be a list, even if it is empty
(64)
> (define my-list '(a b c))
> (define new-list (cons 'x my-list))
> new-list
(x a b c)
> my-list
(a b c)

The last example shows that the cons function is non-destructive. The new list (x a b c) formed by attaching x to the front of my-list does not change my-list.

In-Class Problem 14.1.2: Using cons to create short lists

Define functions, called list-one and list-two, that satisfy the following contracts:

;; LIST-ONE
;; -----------------------------------------------------------
;; INPUT: DATUM, anything
;; OUTPUT: A list that contains DATUM as its only element

;; LIST-TWO
;; -----------------------------------------------------------
;; INPUTS: ONE, TWO, anything
;; OUTPUT: A list whose first element is ONE, and whose second element is TWO

Here are examples of the desired behavior:

> (list-one 'a)
(a)
> (define listy '(a b c))
> (define symby 'xyz)
> (list-one listy)
((a b c))
> 'listy ← quote produces different results!
listy
> (list-one symby)
(xyz)
> 'symby ← quote produces different results!
symby
> (list-two 'a 'b)
(a b)
> (list-two listy symby)
((a b c) xyz)
There is a built-in function, called list, that takes any number of inputs. It returns as its output a list containing those inputs, as illustrated below:

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

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

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

### 14.2 List-based Recursion

Chapter 12 introduced recursive functions for which the recursion was driven by the size of a number. For example, in the factorial function (cf. Example 12.1.1), \( f(4) \) was computed by multiplying 4 by \( f(3) \), where \( f(3) \) was computed by multiplying 3 by \( f(2) \), where \( f(2) \) was computed by multiplying 2 by \( f(1) \), and where \( f(1) = 1 \) terminated the recursion. The relevant sequence of computations is shown below:

\[
\begin{align*}
f(4) &= 4 \cdot f(3) \\
&= 4 \cdot (3 \cdot f(2)) \\
&= 4 \cdot (3 \cdot (2 \cdot f(1))) \\
&= 4 \cdot (3 \cdot (2 \cdot 1)) \\
&= 4 \cdot (3 \cdot 2) \\
&= 4 \cdot 6 \\
&= 24
\end{align*}
\]

More generally, for any \( n > 1 \), the factorial of \( n \) can be computed by making a sequence of \( n - 1 \) recursive function calls, terminating in the base case, where \( f(1) = 1 \). Of course, numerical recursion can take many forms. For example, the input \( n \) might start out at 0 and increase by 3 on each recursive function call until some stopping value (e.g., 90) is reached. Or the value of \( n \) might be multiplied by some value at each recursive function call.

This section introduces list-based recursion. In list-based recursion the recursion is driven not by the size of a number, but by some feature of a list. In many cases, the relevant feature is simply whether a certain list is empty or not: if the list is empty, we’re in the base case; otherwise, we’re in the recursive case. For example, if a typical recursive function is applied to a list containing, say, five elements, then, because that list is non-empty, a recursive function call will be made on the rest of that list (i.e., a list containing four elements). And because that list is non-empty, another recursive function call will be made, this time on the rest of that list (i.e., a list containing three elements). The sequence of recursive function calls will eventually lead to the function being applied to the empty list, at which point the base case will terminate the recursion. This common kind of list-based recursion is explored in the following example.

**Example 14.2.1**

**Suppose we are given the following contract for a function called mult-all:**

```scheme
;; MULT-ALL
```
;;  -------------------------------------------------------------------------------------------------------
;;  INPUT:  LISTY, a list of numbers
;;  OUTPUT: The product of all the elements of LISTY

Here are some examples of the desired behavior:

> (mult-all '(2 3 4 10))
240
> (mult-all '(10 2 4))
80

This function can be defined recursively since:

\[
\text{(the product of all of the elements of a non-empty list)} = \begin{cases} 
\text{(the first element of the list)} \\
\times \\
\text{(the product of the rest of the elements of the list)}
\end{cases}
\]

For example:

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

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

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

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

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

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

\[
\begin{align*}
\text{(mult-all '(2 3 4 10))} & \quad \text{\textit{Recursive Case}} \\
& \Rightarrow (* 2 \text{(mult-all '(3 4 10)))} \quad \text{\textit{Recursive Case}} \\
& \Rightarrow (* 2 (* 3 \text{(mult-all '(4 10)))}) \quad \text{\textit{Recursive Case}} \\
& \Rightarrow (* 2 (* 3 (* 4 \text{(mult-all '(10)))))) \quad \text{\textit{Recursive Case}} \\
& \Rightarrow (* 2 (* 3 (* 4 (* 10 \text{(mult-all ()))))))) \quad \text{\textit{Base Case}} \\
& \Rightarrow (* 2 (* 3 (* 4 (* 10 1)))) \\
& \Rightarrow (* 2 (* 3 (* 4 10))) \\
& \Rightarrow (* 2 (* 3 40)) \\
& \Rightarrow (* 2 120) \\
& \Rightarrow 240
\end{align*}
\]
As long as the list in question is non-empty, the recursive case evaluates an expression of the form
\((* \text{(first some-list)} \text{(mult-all (rest some-list))})\). However, when the list in question is empty, the base case is reached, terminating the recursion. These sorts of considerations lead to the following solution:

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

**Example 14.2.2: Summing the numbers in a list**

The following defines a \text{sum-all} function that sums the numbers in the input list. Its structure is similar to that of the \text{mult-all} function.

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

\[
> \text{(sum-all '(1 2 3 4))}
10
> \text{(sum-all '(1 10 100 1000))}
1111
> \text{(sum-all '(2 5 3 8 1))}
\]
In-Class Problem 14.2.1

Define a function, called \texttt{add-squares}, that satisfies the following contract:

\begin{verbatim}
;;; ADD-SQUARES
;;; -----------------------------------------
;;; INPUT: LISTY, a list of numbers
;;; OUTPUT: The sum of the squares of the numbers in LISTY
\end{verbatim}

Here are some examples of the desired behavior:

\begin{verbatim}
> (add-squares '(2 3 10))  ←  2^2 + 3^2 + 10^2 = 4 + 9 + 100 = 113
113
> (add-squares '(1 0 5 2))  ←  1^2 + 0^2 + 5^2 + 2^2 = 1 + 0 + 25 + 4 = 30
30
\end{verbatim}

In-Class Problem 14.2.2: Computing the length of a list

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

\begin{verbatim}
;;; LENGTHY
;;; -------------------------------------------------------------
;;; INPUT: LISTY, any list
;;; OUTPUT: The number of elements of LISTY (i.e., its length)
\end{verbatim}

Here are some examples of the desired behavior:

\begin{verbatim}
> (lengthy '(a b c d e)) 5
> (lengthy '(#t () 22 xyz)) 4
\end{verbatim}

Hints: Use list-based recursion. What’s the relationship between the length of \texttt{listy} and the length of \texttt{(rest listy)}? And how many elements are in the empty list?

Incidentally, now that you know how to define a function to compute the length of a list, it’s time to tell you that there is a built-in function, called \texttt{length}, that does just that!

In-Class Problem 14.2.3: Accessing the \texttt{N}th element of a list

Define a function, called \texttt{fetch-nth-element}, that satisfies the following contract:

\begin{verbatim}
;;; FETCH-NTH-ELEMENT
;;; ------------------------------------------
;;; INPUTS: LISTY, a list
;;; \texttt{N}, a non-negative integer treated as an "index"
\end{verbatim}
;; OUTPUT: Returns the Nth element of LISTY
;; (or #f if LISTY doesn’t have an Nth element)
;; NOTE: The elements of LISTY are indexed starting at 0.

Thus, for example, a is considered to be the zeroth element of the list (a b c d e), while c is considered to be the element with index 2. Thus, the elements in a list containing five elements will have indices ranging from 0 to 4, inclusive. Here are some examples of the behavior of the fetch-nth-element function:

> (fetch-nth-element '(a b c d e) 0)
a
> (fetch-nth-element '(a b c d e) 2)
c
> (fetch-nth-element '(a b c d e) 8)
#f

Incidentally, now that you know how to implement the fetch-nth-element function, I can tell you that there is a built-in function, called list-ref, that does the same thing. Like fetch-nth-element, the list-ref function treats the first element of a list as having index 0.

Example 14.2.3

Suppose we want to define a function called is-elt-of? that satisfies the following contract:

;; IS-ELT-OF?
;; ------------------------
;; INPUTS: ITEM, anything
;; LISTY, a list of stuff
;; OUTPUT: #t (or something that counts as true) if ITEM
;; appears as an element of LISTY -- as judged by EQ?
;; #f otherwise.

Here are examples of the desired behavior:

> (is-elt-of? 3 '(3 4 5))
#t
> (is-elt-of? 3 '(1 2 3 4 5))
#t
> (is-elt-of? 'x '(a b a b a))
#f

Consider the first example, where ITEM is 3, and LISTY is (3 4 5). In this case, it is clear that ITEM appears in LISTY because it appears as the first element. (Notice that this is a kind of base case since, once we find an occurrence of ITEM in LISTY, there is no need to continue looking any further.) On the other hand, in the second example, where ITEM is 3, and LISTY is (1 2 3 4 5), it is true that ITEM appears in LISTY because, as a sequence of recursive functions call might discover, ITEM appears somewhere in the rest of LISTY. Finally, in the third example, where ITEM is x, and LISTY is (a b a b a), we could imagine a sequence of recursive function calls that never discover an occurrence of x, eventually leading to the base case: (is-elt-of? ’x ()), which must evaluate to #f, since nothing can appear as an element of the empty list.

In view of these considerations, we are led to the following solution:
(define is-elt-of?  
  (lambda (item listy)  
    (cond  
      ;; Base Case 1: LISTY is EMPTY  
      ((null? listy)  
        ;; No occurrence of ITEM in the empty list  
        #f)  
      ;; Base Case 2: ITEM appears as first element of LISTY  
      ((eq? item (first listy))  
        ;; We found ITEM in LISTY!  
        #t)  
      ;; Recursive Case: Haven’t found ITEM in LISTY yet  
      (else  
        ;; Keep looking  
        (is-elt-of? item (rest listy))))))  

Notice that we must check whether LISTY is empty before trying to use first or rest, since those accessor functions can only be used on non-empty lists.

Example 14.2.4: The built-in member function

Now that you know how to define the is-elt-of? function, I can tell you that there is a built-in function, called member, that does the same thing! The only difference is that the value returned by member, in cases where it finds ITEM in LISTY, is the portion of LISTY that starts from the first occurrence of ITEM, as illustrated below:

> (member 3 '(1 2 3 4 5))  
(3 4 5)  
> (member 'x '(a b c d e f x y z))  
(x y z)

Recall that anything other than than #f counts as true. So, expressions such as the following are handled appropriately:

> (if (member 3 '(1 2 3 4 5)) 'say_yes 'say_no)  
say_yes

In this case, the condition evaluated to the list (3 4 5), which counts as true, so the if special form evaluated the expression 'say_yes, generating the output value say_yes. For this reason, it does no harm for member to return something that counts as true. Furthermore, in some cases, you might be glad to have access to the list returned by member as its output.

Example 14.2.5: An alternative implementation of is-elt-of?

Recall from Section 11.4 that, when defining a predicate (i.e., a function that returns a boolean value), one can often write the body of the function using the boolean operators, and, or and not, instead of the conditional expressions, if or cond. Recall further that:

* When defining a predicate using only the boolean operators, the body of the predicate should specify the conditions under which the predicate should output the value #t (or something that counts as true).
Regarding `(is-elt-of? item listy)`, we know that it will evaluate to `#f` if `listy` is empty; therefore, it can only evaluate to `#t` if `listy` is non-empty. However, that is not enough. In addition, we need to find `item` somewhere in `listy`. What are the possibilities? Well, `item` can appear either as the first element of `listy`, or somewhere in the rest of `listy`. These considerations lead to the following alternative definition of the `is-elt-of?` function. To distinguish the two versions, we call this one `is-elt-of-alt?`.

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

Try using this function in the Interactions Window to confirm that it works as advertised.

---

**In-Class Problem 14.2.4: Is a list of numbers in increasing order?**

Define a function, called `incr?`, that satisfies the following contract:

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

Here are some examples illustrating its behavior:

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

* What’s the best way of checking whether the input list contains exactly one element?  

Try writing one version of `incr?` that uses `if` or `cond`, and another that uses `and`, `or` and `not`.

---

**Example 14.2.6: Printing a histogram**

The goal for this exercise is to define a function, called `print-histy`, that satisfies the following contract:
;; PRINT-HISTY
;; ---------------------------------------------
;; INPUT: LISTY, a list of non-negative integers
;; OUTPUT: None
;; SIDE EFFECT: Displays a histogram in the Interactions Window
;; based on the numbers in LISTY. In particular, for each
;; number in LISTY, prints one row of that many asterisks.

Here are some examples of the desired behavior:

> (print-histy '(3 2 8 4 6))
***
**
******
****
******
> (print-histy '(1 2 3 4))
*
**
***
****

Consider the first example: (print-histy '(3 2 8 4 6)). The beauty of recursive programming is that we can write a function that explicitly does only a small part of the job, while leaving most of the work to the recursive function call. For example, to print out the desired histogram, we can just print out the first row of 3 asterisks, and then let the recursive function call take care of printing the rest of the histogram, based on the rest of the list (i.e., (2 8 4 6)). Of course, in the base case, when the list is empty, we're all done!

(define print-histy
  (lambda (listy)
    (cond
     ;; Base Case: LISTY is empty
     ((null? listy)
      ;; Use the built-in VOID function to do ... nothing!
      (void))
     ;; Recursive Case: LISTY is non-empty
     (else
      ;; Use a helper function to print one row of the histogram
      (print-n-stars (first listy))
      ;; Then print out the rest of the histogram
      (print-histy (rest listy))))))

Notice that since there's nothing to do in the base case, we just use the built-in void function to do ... nothing! (Recall from Section 5.5, the void function actually outputs the special void value which DrScheme interprets as “no output”.)

Here's the helper function, which is a slight re-write of the print-n-dashes function from Example 12.2.1:

(define print-n-stars
  (lambda (n)
    (cond
     ((<= n 0)
      (newline))
     (else
      ;; Print n stars
      (newline
        (do ((i 0 (+ i 1))) ((= i n)) (write #\*))
      newline)))))

Notice that since there's nothing to do in the base case, we just use the built-in void function to do ... nothing!
Problems

Problem 14.1

Define a function, called all-numbers?, that satisfies the following contract:

;; ALL-NUMBERS?
;; -----------------------------------------------
;; INPUT: LISTORY, a list
;; OUTPUT: #t (or something that counts as true) if all the
;; items in LISTORY are numbers; #f otherwise

Here are some examples:

> (all-numbers? '(1 2 3 4))
#t
> (all-numbers? '(1 2 a b #t c 4))
#f

Problem 14.2

Define a function, called index-of, that satisfies the following contract:

;; INDEX-OF
;; -----------------------------------------------
;; INPUTS: ITEM, anything
;; LISTORY, a list of stuff
;; OUTPUT: The index of the *first* occurrence of ITEM in LISTORY
;; or #f if ITEM doesn’t appear in LISTORY.
;; NOTE: Indices start at 0.

Here are some examples:

> (index-of 'a '(a b c d e a a b))
0
> (index-of 'c '(a b c d e c e f))
2
> (index-of 'g '(a b c d e f))
#f

Hint: Use the built-in eq? function to test the equality of two pieces of data.
Problem 14.3

Define a function, called first-symbol, that satisfies the following contract:

;; FIRST-SYMBOL
;; -----------------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: The first symbol that appears in LISTY;
;; or #f, if no symbols appear in LISTY.

Here are some examples:

> (first-symbol '(3 #t x y #f))
x
> (first-symbol '(1 2 3))
#f

Hint: Use the built-in type-checker predicate, symbol?.

Problem 14.4

Define a function, called has-symbol?, that satisfies the following contract:

;; HAS-SYMBOL?
;; -----------------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: #t if LISTY contains at least one symbol

Here are some examples:

> (has-symbol? '(1 2 3))
#f
> (has-symbol? '(1 2 3 4 x 5 6))
#t

(Optional) Define a version of the has-symbol? function that uses some combination of and, or and not, instead of if or cond. In that case, the body of the predicate should specify the condition under which this function should return true.

Problem 14.5

Define a function, called max-elt, that satisfies the following contract:

;; MAX-ELT
;; -------------------------------
;; INPUT: LISTY, a non-empty list of numbers
;; OUTPUT: The MAXIMUM number in LISTY

Here are some examples:

> (max-elt '(6 7 71 3 4))
71
> (max-elt '(8))
8

Hint: Notice that the base case should be a list that contains exactly one element. What is the easiest way to test for that? (Warning! Do not use the length function for that purpose! Think about why!)

Problem 14.6

Recall the built-in predicate, even?. It takes a number as its only input and returns #t if that number is even; otherwise it returns #f. Now, if some number N is even (i.e., if (even? N) ⇒ #t), then we say that N “satisfies” the even? predicate (i.e., makes it return #t as its output). So, for example, the number 6 satisfies the even? predicate, but does not satisfy the odd? predicate. Similarly, the number 7 satisfies the odd? predicate, but not the even? predicate.

For this problem, define a function, called contains-a-satisfier?, that satisfies the following contract:

;; CONTAINS-A-SATISFIER?
;; --------------------------------------------------------------
;; INPUTS: PRED, a predicate (e.g., EVEN?) that takes a single input
;; LISTY, a list of suitable inputs for PRED
;; OUTPUT: #t if LISTY contains at least one element that "satisfies" PRED; #f otherwise.

Here are some examples:

> (contains-a-satisfier? even? '(1 2 3 4 5))
#t
> (contains-a-satisfier? even? '(1 3 5 7 9))
#f

The first example evaluates to #t because the input list contains the number 2, which is even. The second example evaluates to #f because the input list does not contain any even numbers.

⋆ Note that you can make lots of tester expressions using any of the type-checker predicates that we have seen in class (e.g., number?, symbol?, null?, etc.), as well as: even? and odd?. However, predicates such as <, >, <=, =, etc., which expect two inputs, would not work here.

⋆ If the input list is non-empty, check what happens when PRED is applied to (FIRST LISTY), and react accordingly.

Problem 14.7

Define a function, called n-elt-list?, that satisfies the following contract:

;; N-ELT-LIST?
;; ---------------------------------------------------------------
;; INPUTS: N, a non-negative integer
;; LISTY, a list
;; OUTPUT: #t if LISTY contains exactly N elements; #f otherwise.
Here are some examples of its use:

```scheme
> (n-elt-list? 5 '(a b c d e))
#t
> (n-elt-list? 5 '(a b c))
#f
> (n-elt-list? 5 '(a b c d e f g))
#f
```

Implement two versions of this function: one that uses cond, and one that uses only and, or and not.

* Do not use the length function! If a list contains a billion elements, we don’t want the length function to walk all the way through its one billion elements just to find out whether or not it is a 4-element list!

Problem 14.8: Testing whether a list is sorted

Recall that the incr? predicate, from In-Class Problem 14.2.4, returned #t if its input list was sorted into non-decreasing order. For this problem, you will define a more general predicate, called sorted?, that takes an extra input, called comparer. The sorted? predicate returns #t if its input list is sorted according to the comparer predicate. For example, if the comparer predicate is the less-than function, then the behavior of sorted? is the same as that of incr?. However, other choices of the comparer predicate lead different behavior. Here is the contract for sorted?, followed by some examples of its desired behavior:

```scheme
;; SORTED?
;; ------------------------------------------------------------------------
;; INPUTS: LISTY, a non-empty list of stuff
;; COMPARER, a predicate that returns #t
;; if its two inputs are in some desired order
;; OUTPUT: #t, if the elements of LISTY are sorted into the
;; order determined by COMPARER; #f, otherwise.
```

```scheme
> (sorted? '(1 2 3 5 8) <) ← Equivalent to using incr?
#t
> (sorted? '(1 2 3 5 5 8) >)
#f
> (string<=? "beard" "bread") ← string<=? is built-in; it outputs #t if its two inputs are in alphabetic order.
#t
> (sorted? '("bead" "beard" "bread" "broom") string<=?)
#t
> (sorted? '("bead" "bread" "beard" "broom") string<=?)
#f
```

The examples involving strings and the built-in string<=? predicate illustrate the flexibility of the sorted? predicate.

Problem 14.9: Computing dot products

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

```scheme
;; DOTTY
```
;; -- -----------------------------------------------
;; INPUT: LISTY, LISTZ, two lists of numbers, having
;; the same length
;; OUTPUT: The "dot product" of LISTY and LISTZ. In other
;; words, if LISTY = (y1 y2 ... yn) and LISTZ = (z1 z2 ... zn),
;; then the output is the sum: y1*z1 + y2*z2 + ... + yn*zn.

Here are some examples:

> (dotty '(5 4 3) '(100 10 1))
543 ← (5*100) + (4*10) + (3*1)

> (dotty '(2 4) '(9 7))
46 ← (2*9) + (4*7)

> (dotty '(1 -2 1) '(2 3 4))
0 ← (1*2) + ((-2)*3) + (1*4)

Hint: Even though there are two lists as input, the recursive processing is very similar to other examples we have done, especially since this function assumes that the input lists have the same number of elements.

---

**Problem 14.10**

Define a predicate, called **dominates?** , that satisfies the following contract:

;; DOMINATES?
;; -----------------------------------------------
;; INPUTS: LISTY, LISTZ, two lists of numbers having
;; the same length
;; OUTPUT: #t if each element of LISTY is greater than or
;; equal to the corresponding element of LISTZ

Here are some examples:

> (dominates? '(10 10 12 15) '(2 5 3 1))
#t

> (dominates? '(10 10 12 15) '(2 5 18 6))
#f

---

**Problem 14.11**

Define a function, called **first-pair**, that satisfies the following contract:

;; FIRST-PAIR
;; -----------------------------------------------
;; INPUT: LISTY, a list of stuff
;; OUTPUT: The first item that appears twice consecutively in
;; LISTY (as judged by EQ?); otherwise, #f.

Here are some examples:

> (first-pair '(1 2 3 4 4 5 3 3 3))
4
> (first-pair '(a b f d d r c c c a))
d
> (first-pair '(a b c a b c))
#f

Hints: Note that a list containing zero or one elements cannot have any consecutive elements. Define a helper function that is the same as first-pair, except that it takes an extra input, called prev, that keeps track of the most recently seen item.

Problem 14.12: Checking whether two lists are “equal”

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

;; LIST-EQUAL?
;; ---------------------------
;; INPUTS: LISTY, LISTZ, any two lists
;; OUTPUT: #t if LISTY and LISTZ contain the same
;; elements, in the same order, where equality of
;; elements is judged by the EQ? predicate;
;; #f, otherwise

Here are some examples:

> (list-equal? '(a b c) '(a b c))
#t
> (list-equal? '(1 2 3 4) (cons 1 (cons 2 (cons 3 (cons 4 ())))) )
#t
> (list-equal? '(1 2 3) '(1 2 3 4 5))
#f

Hint: Walk through the lists in parallel, checking equality of their corresponding elements, until one or both lists run out of elements.

After you’ve implemented this function, you may wish to know that there is a built-in function, called equal?, that does almost the same thing. As demonstrated below, the equal? predicate also works on hierarchical lists, whereas list-equal? does not.

> (equal? '(a b c) '(a b c))
#t
> (equal? '(a (b (c) d)) '(a (b (c) d)))
#t
> (list-equal? '(a (b (c) d)) '(a (b (c) d)))
#f

The reason for the difference is that list-equal? uses eq? to test the equality of corresponding elements, but eq? is not sophisticated enough to judge equality of lists. (Try it.) Hierarchical lists will be covered in Section 14.7, below.
14.3 Recursively Generating Lists as Output Values

So far, we have seen examples of recursive functions where the recursion is driven by a list, and the output has been a number, a boolean, or \texttt{void}—along with some side-effect printing. This section addresses list-based recursion where the output value is a list that has been incrementally generated by the recursive function calls. The incremental generation of lists is accomplished using the built-in \texttt{cons} function, introduced in Section 14.1.

<table>
<thead>
<tr>
<th>Example 14.3.1: Doubling all the elements of a list</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Suppose we want to define a function, called \texttt{double-all}, that satisfies the following contract:</em></td>
</tr>
<tr>
<td>;; DOUBLE-ALL</td>
</tr>
<tr>
<td>;; ----------------------------------------------</td>
</tr>
<tr>
<td>;; INPUT: LISTY, a list of numbers</td>
</tr>
<tr>
<td>;; OUTPUT: A list of numbers, each of whose elements</td>
</tr>
<tr>
<td>;; is twice the corresponding element in LISTY.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Here are some examples of the desired behavior:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; (double-all '(3 2 10 13))</td>
</tr>
<tr>
<td>(6 4 20 26)</td>
</tr>
<tr>
<td>&gt; (double-all '(5 3 8))</td>
</tr>
<tr>
<td>(10 6 16)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Let’s apply some recursive thinking to the first example: (double-all ’(3 2 10 13)). We can generate the desired output list (6 4 20 26) as follows.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Consider the following pieces of the desired output list, (6 4 20 26):</td>
</tr>
<tr>
<td>• Its first element: 6</td>
</tr>
<tr>
<td>• The rest of its elements: (4 20 26)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(2) Fetch the corresponding pieces of the input list, (3 2 10 13):</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Its first element: 3</td>
</tr>
<tr>
<td>• The rest of the list: (2 10 13)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(3) Do the following to the corresponding pieces of the input list:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Double the first element: (* 2 3) ⇒ 6</td>
</tr>
<tr>
<td>• Use a recursive function call to double the rest of the elements:</td>
</tr>
<tr>
<td>(double-all ’(2 10 13)) ⇒ (4 20 26)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(4) Use the above pieces to construct the desired output list using \texttt{cons}:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• (cons 6 ’(4 20 26)) ⇒ (6 4 20 26)</td>
</tr>
</tbody>
</table>

We can more concisely describe the process outlined above, as follows. If \texttt{listy} is a non-empty list, the element-wise doubling of \texttt{listy} can be obtained by the following expression:

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

Before jumping to the completed function definition, we need to determine what should happen in the base case, where the input list is empty. There are two things to consider:
• The list obtained by doubling each element of the empty list is ... the empty list:
  (double-all ()) ⇒ ()

• When the input list is a one-element list, the recursive rule described above looks like this:
  (double-all '(4)) ⇒ (cons (* 2 4) (double-all ()))
  ⇒ (cons 8 ())
  ⇒ (8)

Therefore, whether we consider the base case in isolation—what should double-all do to the empty list based on the contract?—or we consider the base case as the terminating case of a sequence of recursive function calls, we conclude that (double-all ()) should evaluate to ()

Here’s the finished product:

(define double-all
  (lambda (listy)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
        ;; The double-all of () is ...
        ())
      ;; Recursive Case: LISTY is non-empty
      (else
        ;; Double the first element and attach it to the
        ;; double-all of the rest of the list
        (cons (* 2 (first listy))
          (double-all (rest listy))))))))

Example 14.3.2: Applying a given function to each element of a list

Recall the facty function seen in Example 12.1.1. It takes a single number as its input, and returns the factorial of that number as its output:

> (facty 3)
6
> (facty 5)
120

For this exercise, we want to define a function called mappy that takes two inputs: (1) a function func that, like facty, can be applied to a single input, and (2) a list listy, each of whose elements is a suitable input for func. The expression (mappy func listy) should generate as its output the list whose elements are obtained by applying func, in turn, to each of the elements of listy. Here are some examples:

> (mappy facty '(3 4 5 6))
(6 24 120 720)
> (mappy even? '(1 2 3 4 5 6))
(#f #t #f #t #f #t)
> (mappy abs '(1 -1 2 -2 3 -3)) ← abs computes the absolute value
(1 1 2 2 3 3)

As in Example 14.3.1, we analyze this problem by thinking recursively, using a concrete example:

(mappy facty '(3 4 5 6)) ⇒ (6 24 120 720)
(1) The parts of the desired output list:
- Its first element: 6
- The rest of its elements: (24 120 720)

(2) The corresponding parts of the input list:
- Its first element: 3
- The rest of its elements: (4 5 6)

(3) Do the following to the pieces of the input list:
- Apply facty to the first element: \(\text{facty } 3 \Rightarrow 6\)
- Let a recursive function call apply facty to the rest of the elements:
  \((\text{mappy facty '(4 5 6)}) \Rightarrow (24 120 720)\)

(4) Use the cons function to combine the above pieces:
- \((\text{cons } 6 ' (24 120 720)) \Rightarrow (6 24 120 720)\)

The above analysis suggests that for a non-empty list \(\text{listy}\), the following expression will evaluate to the desired result:
\[
(\text{mappy func listy}) \Rightarrow (\text{cons (func (first listy)})
\hspace{1cm} (\text{mappy func (rest listy)}))
\]

In addition, you should convince yourself that, as in Example 14.3.1, the base case, \((\text{mappy func ()})\), should evaluate to \((\text{)}\). Here is the completed solution.

```scheme
;;; MAPPY
;;; -------------------------------------------------------
;;; INPUTS: FUNC, a function that takes a single input
;;; LISTY, a list of suitable inputs for FUNC
;;; OUTPUT: A list whose elements are obtained by applying
;;; FUNC to each of the elements of LISTY, in turn.

(define mappy
  (lambda (func listy)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
        ;; Applying FUNC to each element of the empty list
        ;; yields ... the empty list
        ())
      ;; Recursive Case: LISTY is non-empty
      (else
        ;; Apply FUNC to the FIRST element of LISTY, and then
        ;; use CONS to attach the result to the front of the
        ;; list obtained from the recursive function call on
        ;; the REST of LISTY.
        (cons (func (first listy))
          (mappy func (rest listy)))))))
```

Incidentally, now that you know how to implement the \text{mappy} function, I can tell you that there is a built-in
function, called map, that does the same thing. The following example illustrates how the map function can be used to facilitate testing.

**Example 14.3.3: Using map to facilitate testing**

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

```
> (tester '(map square '(1 2 3 4 10 25)))
(map (square '(1 2 3 4 10 25))) ==> (1 4 9 16 100 625)
```

**In-Class Problem 14.3.1: Removing items from a list**

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

```scheme
;; REMOVER
;; ________________________________
;; INPUTS: ITEM, anything
;; LISTY, a list
;; OUTPUT: A list that contains all of the elements of
;; LISTY, except any occurrences of ITEM.
```

Here are some examples:

```
> (remover 3 '(1 2 3 4 5 4 3 2 1))
(1 2 4 5 4 2 1)
> (remover 'a '(a b r a c a d a b r a))
(b r c d b r)
```

Incidentally, now that you know how to implement the remover function, I can tell you that there is a built-in function, called remove, that does the same thing, except that it only removes the first occurrence of item from listy.

The following example implements a function that takes two input lists, but uses only one to drive the recursion.

**In-Class Problem 14.3.2: Concatenating two lists**

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

```scheme
;; CONC
;; ________________________________
;; INPUTS: LISTY, LISTZ, two lists
;; OUTPUT: A list containing all of the elements of LISTY
;; followed by all of the elements of LISTZ.
```

Here are some examples of the desired behavior:

```
> (conc '(1 2 3 4) '(a b c))
(1 2 3 4 a b c)
> (conc '(a b c) '(1 2 3 4))
(a b c 1 2 3 4)
```
Hints: Let listy drive the recursion. What is the output when listy is empty?

Now that you know how to implement the conc function, I can tell you that there is a built-in function called append that does the same thing!

The preceding examples showed how a recursive function can be used to incrementally generate a new list as its output. In each case, some input list was driving the recursion. However, as the following examples show, functions whose recursion is driven by the size of a number can also be used to incrementally generate output lists.

Example 14.3.4

The goal of this exercise is to define a function, called list-down-to-zero, that satisfies the following contract:

;;; LIST-DOWN-TO-ZERO
;;; ---------------------------------------------------
;;; INPUT: N, a non-negative integer
;;; OUTPUT: A list of the form (N N-1 N-2 ... 2 1 0)

Here are some examples of the desired behavior:

> (list-down-to-zero 5)
(5 4 3 2 1 0)
> (list-down-to-zero 8)
(8 7 6 5 4 3 2 1 0)

Thinking recursively about the first example, we note that the list from 5 down to 0 can be constructed by attaching the number 5 to the front of the list from 4 down to 0. More generally, for any non-negative number n:

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

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

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

In-Class Problem 14.3.3

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

;;; LIST-UP-TO-N
;;; ---------------------------------------------------
;;; INPUTS: FROM, a non-negative integer (starting point)
### In-Class Problem 14.3.4

**Define a function, called `random-flips`, that satisfies the following contract:**

```scheme
;; RANDOM-FLIPS
;; -------------------------------------
;; INPUTS: N, a non-negative integer
;; OUTPUT: A list containing N random flips of a coin,
;; where each flip is either H or T
```

**Here are some examples of the desired behavior:**

```scheme
> (random-flips 8)
(H H T H T T T H)
> (random-flips 5)
(T H T T H)
```

**Hint:** Use the `flip-coin` function from Example 13.1.5 as a helper. Fill in the blanks: A list of \( n \) random coin flips can be generated by attaching ________________ to the front of a list of ________________ random coin flips.

### Problems

#### Problem 14.13: Removing from a list items that have some property

**Define a function, called `remove-if`, that satisfies the following contract:**

```scheme
;; REMOVE-IF
;; -------------------------------------
;; INPUTS: PRED, a predicate that expects one input
;; LISTY, a list of elements, each of which is a
;; suitable input for PRED
;; OUTPUT: A list that contains all of the elements of LISTY,
;; except those for which PRED returns #t.
```

**Here are some examples:**

```scheme
> (remove-if even? '(1 2 3 4 5 6))
```
> (remove-if odd? '(1 2 3 4 5 6))
(2 4 6)

*Hint:* There can be two recursive cases, one where (first listy) “satisfies” pred, the other where (first listy) does not. In the first case, you do not want to include (first listy) in the answer list; in the second case, you do want to include it.

---

**Problem 14.14: Replacing items in a list**

Define a function, called `replace`, that satisfies the following contract:

```
;; REPLACE
;; ----------------------------------
;; INPUTS: OLD, anything
;; NEW, anything
;; LISTY, a list of stuff
;; OUTPUT: A list just like LISTY except that each occurrence of
;; OLD in LISTY has been replaced by an occurrence of
;; NEW in the output. (Equality of two items should
;; be determined by the EQ? predicate.)
```

Here are some examples:

> (replace 'x 'y '(a x b x c x))
(a y b y c y)
> (replace 0 1 '(0 1 1 0 0 0 1))
(1 1 1 1 1 1 1)

---

**Problem 14.15**

Define a function, called `every-other-one`, that satisfies the following contract.

```
;; EVERY-OTHER-ONE
;; ----------------------------------
;; INPUT: LISTY, a list
;; OUTPUT: A list containing every other element of LISTY.
;; Note: The output list should contain roughly half the
;; elements of LISTY; and their occurrences in the output list
;; should be in the same order as their occurrences in LISTY.
```

Here are some examples of its behavior:

> (every-other-one '(a b c d e f g))
(a c e g)
> (every-other-one '(a b c d e f))
(a c e)
> (every-other-one '(0 1 0 1 0 1 0 1))
(0 0 0 0)
Problem 14.16: Fetching the first \( N \) elements of a list

Define a function, called first-n-elts, that satisfies the following contract:

```
;; FIRST-N-ELTS
;; ---------------------------------------
;; INPUTS: N, a non-negative integer
;; LISTY, a list that contains at least N elements
;; OUTPUT: A list containing the first N elements of LISTY, in the same order as their order in LISTY.
```

Here are some examples of its use:

```
> (first-n-elts 3 '(a b c d e f g))
(a b c)
> (first-n-elts 5 '(a b c d e f g))
(a b c d e)
```

Note: You need not deal with the case where listy has fewer than \( n \) elements.

Problem 14.17

Define a function called stutter that satisfies the following contract:

```
;; STUTTER
;; ---------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: A list that contains twice as many elements as LISTY. In particular, each element of LISTY should appear twice consecutively in the output.
```

Here are some more examples:

```
> (stutter '(life is fun))
(life life is is fun fun)
> (stutter '(i went home yesterday))
i i went went home home yesterday yesterday
```

Hint: In the recursive case, use cons twice!

Problem 14.18

Define a function, called consec-sums, that satisfies the following contract:

```
;; CONSEC-SUMS
;; ---------------------------------------
;; INPUT: LISTY, a list of at least two numbers
;; OUTPUT: A list containing the sums of consecutive items from LISTY
```

Here are some examples:

```
> (consec-sums '(1 20 300 4000))
```
(21 320 4300)
> (consec-sums '(50 40 30 20 10))
(90 70 50 30)

Notice that the output list contains one fewer element than the input list.

Problem 14.19: Generating a list of random tosses of a die

Define a function, called random-tosses, that satisfies the following contract:

;;; RANDOM-TOSSES
;;; -------------------------------------------------------
;;; INPUTS: NUM, a non-negative integer
;;; OUTPUT: A list containing NUM elements, each element
;;; obtained by randomly tossing a 6-sided die.

Here are some examples:

> (random-tosses 10)
(4 1 4 2 1 6 1 5 5 6)
> (random-tosses 10)
(6 2 2 3 3 6 5 6 3 2)
> (random-tosses 10)
(5 2 1 4 6 3 3 1 2 5)

14.4 Tail Recursion, Accumulators, and Wrapper Functions Revisited

Sections 12.2 through 12.4 introduced the concepts of tail recursion, accumulators, and wrapper functions, respectively. As will be seen in this section, these concepts apply equally well to list-based recursion and the incremental generation of lists as output values.

Recall from Defn. 12.2 that a recursive function-call expression is tail recursive if, whenever its evaluation is needed as part of evaluating the parent function’s body, its evaluation is the last step in that process. And a recursive function is tail-recursive if each of its recursive function-call expressions is tail recursive.

Checking the functions implemented in Examples 14.2.1 through 14.3.4 reveals that mult-all, double-all, mappy and list-down-to-zero are not tail recursive, while is-elt-of?, is-elt-of?-alt and print-histy are tail recursive. The following examples define tail-recursive versions of mult-all, list-down-to-zero and double-all, respectively called mult-all-acc, list-down-to-zero-acc and double-all-acc. As the names indicate, each of these tail-recursive functions will take an additional input that serves to accumulate the desired answer. For mult-all-acc, the extra input will incrementally accumulate the product of the numbers in the input list, much as the accumulator in facty-acc (cf. Example 12.3.3) accumulated the factorial of its input. For list-down-to-zero-acc and double-all-acc, the extra input will incrementally accumulate a list: in particular, each tail-recursive function call will include a call to the cons function to attach a new element to the front of some list. As in Section 12.4, for each accumulator-based, tail-recursive function we shall define an accompanying wrapper function that takes care of providing appropriate initial values for any additional inputs.
Recall that the mult-all function computes the product of all of the numbers in a given list. The mult-all-acc function will work similarly, except that it will take an extra input, called acc, that will accumulate the desired product. In particular, as we walk through the given list of numbers, as each number is encountered, it will be multiplied into the accumulator. As with facty-acc from Example 12.3.3, the initial value of acc will be 1 (i.e., the multiplicative identity).

It can often help to consider a concrete example. Therefore, suppose that we want to use mult-all-acc to compute the product of the numbers in the list (3 7 2 4). We start with acc equal to 1. Imagine the computation proceeding as follows, where the first input to mult-all-acc is the list of numbers, and the second input is the accumulator:

\[
\begin{align*}
(mult-all-acc '(3 7 2 4) 1) & \Rightarrow (mult-all-acc '(7 2 4) 3) & \text{rec. case: “accumulate” a factor of 3} \\
& \Rightarrow (mult-all-acc '(2 4) 21) & \text{rec. case: “accumulate” a factor of 7} \\
& \Rightarrow (mult-all-acc '(4) 42) & \text{rec. case: “accumulate” a factor of 2} \\
& \Rightarrow (mult-all-acc () 168) & \text{base case: accumulator has the answer!}
\end{align*}
\]

Notice that the inputs for each recursive function call are:

- the rest of the current list, and
- the product of the first element of the current list and the current accumulator.

Thus, by the time the base case (i.e., the empty list) is reached, the accumulator has the desired product: \(3 \cdot 7 \cdot 2 \cdot 4 = 168\). Here is the completed solution:

```scheme
;; MULT-ALL-ACC
;;---------------------------------------------------------------
;; INPUTS: LISTY, a list of numbers
;; ACC, a number (accumulator of desired product)
;; OUTPUT: When called with ACC=1, the output is the product
;; of all of the numbers in LISTY. More generally, the output
;; is the product of ACC and all of the numbers of LISTY
(define mult-all-acc
  (lambda (listy acc)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy) acc)
      ;; The accumulated product
      ;; Recursive Case: LISTY is non-empty
      (else
       ;; Tail-recursive function call on adjusted inputs:
       ;; Note: ACC "accumulates" (first listy)
       (mult-all-acc (rest listy) (+ (first listy) acc))))))
```

As is often the case, describing the output for accumulator-based functions can be challenging in the general case (e.g., above, when ACC is something other than 1). Here is the accompanying wrapper function:
MULT-ALL-WR
----------------------------------------
INPUT: LISTY, a list of numbers
OUTPUT: The product of the numbers in LISTY

(define mult-all-wr
  (lambda (listy)
    ;; Call the tail-recursive helper with ACC=1:
    (mult-all-acc listy 1)))

Notice that the contract for mult-all-wr is the same as that for mult-all—except for the name of
the function. That is, the two functions are equivalent.

Example 14.4.2: Tail-recursive function: list-down-to-zero-acc

Recall that the list-down-to-zero function takes a non-negative integer \( n \) as its only input, and generates as its output a list of the form \((n \ n-1 \ n-2 \ \ldots \ 2 \ 1 \ 0)\). The list-down-to-zero-acc function will work similarly, except that it will incrementally accumulate the desired list in an extra input, acc. As in the double-all and mappy functions (cf. Examples 14.3.1 and 14.3.2, respectively) the list-accumulator will start out as the empty list.

Consider the example where the numerical input \( n \) is 3, and we want to generate the list \((3 \ 2 \ 1 \ 0)\). As in list-down-to-zero, the value of \( n \) will decrease by one on each recursive function call, but the accumulator will be adjusted by using the cons function to attach \( n \) to the front of the accumulator, as illustrated in the following sequence of evaluations:

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

Whoops! While this would be fine for generating a list from 0 to \( n \), that is not what we were aiming for! This example illustrates a common issue that arises when using list accumulators:

* When using an accumulator to incrementally generate a list, the order of the elements in the accumulator ends up being the reverse of the order in which they were attached!

There are two ways to fix this problem: (1) define a function to reverse the elements of a list; or (2) arrange to process the desired elements in the opposite order. Below, we take the second approach. Later on, we’ll define a function for reversing the elements of a list.

For the list-down-to-zero-acc function, we can arrange to visit the numbers in the order from 0 up to \( n \) by including yet another input, called curr (for current number), whose value shall start out at 0 and increment by one on each recursive function call. Since 0 will be the first number to be attached to the accumulator, it will end up being the last number in the generated list, as desired. So the inputs to list-down-to-zero-acc will be \( n \), acc and curr. In this version, the value of \( n \) will be the same for each recursive function call. That is, \( n \) serves as an upper bound on the value of curr. When that upper bound is reached, the recursion will terminate, as illustrated below:
(list-down-to-zero-acc 3 () 0)
⇒ (list-down-to-zero-acc 3 '(0) 1) ← attach 0 to front of acc
⇒ (list-down-to-zero-acc 3 '(1 0) 2) ← attach 1 to front of acc
⇒ (list-down-to-zero-acc 3 '(2 1 0) 3) ← attach 2 to front of acc
⇒ (list-down-to-zero-acc 3 '(3 2 1 0) 4) ← attach 3 to front of acc
⇒ (3 2 1 0) ← acc has the answer!

Notice that in this version of list-down-to-zero-acc, the base case is signaled by curr being greater than n—in this example, when 4 > 3. Here is the completed solution:

;;; LIST-DOWN-TO-ZERO-ACC
;;; ---------------------------------------------------------
;;; INPUTS: N, a non-negative integer
;;; ACC, a list accumulator
;;; CURR, a non-negative integer
;;; OUTPUT: When called with ACC=() and CURR=0, the output
;;; is the list (N N-1 N-2 ... 2 1 0). More generally,
;;; the output is the "concatenation" of the lists
;;; (N N-1 N-2 ... CURR) and ACC.

(define list-down-to-zero-acc
  (lambda (n acc curr)
    (cond
      ;; Base Case: CURR > N
      ((> curr n)
        ;; The accumulator has the desired list
        acc)
    ;; Recursive Case: CURR <= N
      (else
      ;; Tail-recursive function call with adjusted inputs:
        (list-down-to-zero-acc n (cons curr acc) (+ curr 1)))))

(You should convince yourself that the "more generally" part of the contract is correct.) Here is the associated wrapper function:

;;; LIST-DOWN-TO-ZERO-WR
;;; ---------------------------------------------------------
;;; INPUT: N, a non-negative integer
;;; OUTPUT: The list (N N-1 N-2 ... 2 1 0)

(define list-down-to-zero-wr
  (lambda (n)
    ;; Call the tail-recursive helper with ACC=() and CURR=0:
    (list-down-to-zero-acc n () 0))))

Before introducing the double-all-acc function, which also uses a list accumulator and, so, suffers from the same problem seen earlier regarding the order of accumulated elements, we first introduce the transfer-all and reversey functions. The latter function can be used to reverse the elements in a list.
Example 14.4.3: The **transfer-all** and **reversey** functions

The goal for this exercise is to define a function, called **transfer-all**, that satisfies the following contract:

`; ; TRANSFER-ALL`
`; ; ----------------------------------------------------------`
`; ; INPUTS: LISTY, LISTZ, two lists`
`; ; OUTPUT: The list obtained by "popping" each element in`
`; ; turn off of the front of LISTY and "pushing" it onto`
`; ; the front of LISTZ.

Here are some examples of the desired behavior:

> (transfer-all '(a b c) '(1 2))
(c b a 1 2)
> (transfer-all '(1 2) '(a b c))
(2 1 a b c)

Notice that the elements from the first list appear in the reverse order in the output list. Here is a sample sequence of evaluations corresponding to the first example above:

(transfer-all '(a b c) '(1 2))
⇒ (transfer-all '(b c) '(a 1 2)) ← attach a to front of second list
⇒ (transfer-all '(c) '(b a 1 2)) ← attach b to front of second list
⇒ (transfer-all () '(c b a 1 2)) ← attach c to front of second list
⇒ (c b a 1 2) ← base case!

As the above example illustrates, the first list (i.e., **listy**) is driving the recursion, and the second list (i.e., **listz**) is acting like an accumulator. When **listy** is empty, the accumulator **listz** contains the desired answer. Here is the completed function definition:

```
(define transfer-all
  (lambda (listy listz)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy) listz)
      ;; Recursive Case: LISTY is non-empty
      (else
       ;; Tail-recursive function call with adjusted inputs
       (transfer-all (rest listy) (cons (first listy) listz))))))
```

Next, we define a “wrapper” for **transfer-all** which we shall call **reversey**, for reasons that will soon become apparent.

`; ; REVERSEY -- wrapper for TRANSFER-ALL`
`; ; -----------------------------------------------`
`; ; INPUT: LISTY, a list`
`; ; OUTPUT: A list that contains the same elements as`
`; ; LISTY, but in the opposite order.
Here are some examples that illustrate that reversey does indeed generate the reversal of its input:

> (reversey '(a b c d))
(d c b a)
> (reversey '(1 2 3 4 5 6))
(6 5 4 3 2 1)

Incidentally, now that you know how to implement the reversey function, I can tell you that there is a built-in function called reverse that does the same thing!

Example 14.4.4: Not all ways of reversing a list are equal!

This example considers an alternative approach to reversing a list, one based on repeated concatenation. Although this approach leads to a function that correctly reverses a list, it turns out to be very inefficient. First, since it is not tail recursive, it can use an awful lot of the computer’s memory when reversing long lists. Second, by repeatedly concatenating long lists, it takes a lot longer to reverse a list than the reversey function seen earlier. To illustrate the inefficiency of this approach, both functions, konk and bad-reverse, defined below, print out some information each time they are called. The konk function concatenates two lists; bad-reverse uses konk as a helper function.

(define konk
  (lambda (listy listz)
    (printf "KONK: LISTY: \(A, LISTZ: \(A\)%" listy listz)
    (cond
      ;; Base Case: LISTY is empty
      (null? listy)
      ;; The concatenation of () and LISTZ is:
      listz)
      ;; Recursive Case: LISTY is non-empty
      (else
       ;; Attach (FIRST LISTY) onto the concatenation
       ;; of (REST LISTY) and LISTZ
       (cons (first listy)
            (konk (rest listy) listz)))))))

;; BAD-REVERSE
;; -------------------------
;; INPUT: LISTY, any list
;; OUTPUT: A list containing the same elements as
;; LISTS, but in the opposite order.

(define bad-reverse
  (lambda (listy)
    (printf "$BAD-REVERSE: LISTY: ~A~%" listy)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
       ()))
      ;; Recursive Case: LISTY is non-empty
      (else
       ;; Recursive function call reverses the REST of LISTY.
       ;; So, we need to attach (first listy) at the end.
       ;; Unfortunately this involves walking through the
       ;; potentially long list returned by the rec. func. call.
       (konk (bad-reverse (rest listy))
             (cons (first listy) ()))))))

To get an idea of how inefficient bad-reverse is, try evaluating the following expression in the Interactions Window: (bad-reverse '(a b c d e)).

---

Example 14.4.5: The double-all-acc function

The goal of this problem is to define a tail-recursive function that doubles all of the elements of a given list of numbers. Because we shall use a list accumulator, the doubled numbers in the accumulated list will come out in the wrong order. But we shall just use the built-in reverse function to reverse the order of the accumulated list before returning it as the output. Here is the completed function definition:

;; DOUBLE-ALL-ACC
;;----------------------------------
;; INPUTS: LISTY, a list of numbers
;; ACC, a list accumulator
;; OUTPUT: When called with ACC=(), the output is
;; a list just like LISTY, except that each element
;; has been doubled.

(define double-all-acc
  (lambda (listy acc)
    (cond
      ;; Base Case: LISTY is empty
      ((null? listy)
       ;; REVERSE the accumulator!
       (reverse acc))
      ;; Recursive Case: LISTY is non-empty
      (else
       ;; Tail-recursive function call with adjusted inputs
       (double-all-acc (rest listy)
                       (cons (* 2 (first listy)) acc))))))

As this example illustrates, the previously identified issue with list accumulators (i.e., that the accumulated elements come out in the opposite order) is easily resolved using the reverse function at the very last
Problems

Problem 14.20

Define a version of the list-down-to-zero-acc function from Example 14.4.2 that accumulates the desired list in the wrong order, but then uses the built-in reverse function to reverse the accumulated list in the base case. Here's the contract:

;; LIST-DOWN-TO-ZERO-ACC-V2
;; ----------------------------------------------
;; INPUTS: N, a non-negative integer
;; ACC, a list accumulator
;; OUTPUT: When called with ACC=(), the output
;; is the list (N N-1 N-2 ... 2 1 0). More generally,
;; the output is the "concatenation" of the lists
;; (N N-1 N-2 ... CURR) and ACC.

Here are some examples of the desired behavior:

> (list-down-to-zero-acc-v2 5 ())
(5 4 3 2 1 0)
> (list-down-to-zero-acc-v2 3 ())
(3 2 1 0)

Then define a wrapper function, list-down-to-zero-wr-v2, that only takes a single input, n.

Problem 14.21

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

;; LIST-FROM-ZERO-TO-N
;; ----------------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: A list of the form (0 1 2 ... N)

Here are some examples:

> (list-from-zero-to-n 5)
(0 1 2 3 4 5)
> (list-from-zero-to-n 15)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

Use a helper function that accumulates the desired list.

Hint: Recall Example 14.4.2.
Problem 14.22: Concatenating lists using `transfer-all` and `reverse`

Define an alternative implementation of the `conc` function from In-Class Problem 14.3.2 that lets the `transfer-all` and `reverse` functions (from Example 14.4.3) do all of the work. For example, one way to concatenate the lists `(1 2 3)` and `(a b c d e)` is to first reverse `(1 2 3)` and then transfer all of its elements onto the front of `(a b c d e)`.

Problem 14.23: Removing duplicate elements from a list

Define a function that satisfies the following contract:

```scheme
;; REM-DUPES
;; -----------------------------------------------------
;; INPUTS: LISTY, any list
;; OUTPUT: A list that contains the same elements as
;;          LISTY, but without any duplicates.
```

The order of the elements in the output list does not really matter, but try to preserve as much of the order of elements in `listy` as possible. Here are some examples:

```scheme
> (rem-dupes '(1 1 1 1 1))
(1)
> (rem-dupes '(a b r a c a d a b r a))
(a b r c d)
> (rem-dupes '(1 2 3 1 2 3 1 2 3 4 3 2 1))
(1 2 3 4)
```

Use an accumulator-based helper function, `rem-dupes-acc`, that satisfies the following contract.

```scheme
;; REM-DUPES-ACC
;; -----------------------------------------------------
;; INPUTS: LISTY, any list
;; ACC, a list accumulator
;; OUTPUT: When called with ACC=(), the output is a
;;         list that contains the same elements as
;;         LISTY, but without any duplicates.
```

Hint: In the recursive case, use the built-in `member` function to decide whether or not `(first listy)` already appears in the accumulator. Use that information to decide whether or not to accumulate it now.

Problem 14.24

Suppose you have a list of dice, such as `(6 3 6 2 6)`. And suppose that in the game you are playing, you are allowed to re-roll some of the dice. If you are trying to get as many sixes as possible, you might want to re-roll the three and the two, but not the sixes. For this problem, you will define a function called `roll-some` that would allow you to do this. For the above example, the function call would look like this:

```scheme
(roll-some '(6 3 6 2 6) '(#f #t #f #t #f))
```

where each `#f` means “Don’t roll that die!”; and each `#t` means “Do roll that die!” The contract is given below, followed by some examples.
**14.5 Sorting Algorithms**

This section introduces two algorithms for sorting a list of numbers: the *insertion-sort* algorithm, and the *merge-sort* algorithm. After defining Scheme functions that implement these algorithms, they are compared by running them on long lists of randomly generated numbers. In what follows, we shall assume that the goal is to sort lists of numbers into *non-decreasing* order, as illustrated below:

Before sorting: \((3 \ 2 \ 1 \ 4 \ 3 \ 2 \ 3 \ 3 \ 6 \ 1 \ 0 \ 5)\)

After sorting: \((0 \ 1 \ 1 \ 2 \ 2 \ 3 \ 3 \ 3 \ 4 \ 5 \ 6)\)

Notice that for any consecutive elements, \(x\) and \(y\), in the sorted list, the following holds: \(x \leq y\).

### 14.5.1 The Insertion-Sort Algorithm

The insertion-sort algorithm uses a helper function, called *insert*, that inserts a number into an *already-sorted* list, such that the resulting list is still sorted. Here is its contract, followed by some examples of the desired behavior:

```scheme
;; INSERT
;; -----------------------------------------------
;; INPUTS: NUM, a number
;; SORTED, a list of numbers that are already sorted
;; into non-decreasing order
;; OUTPUT: The list obtained by inserting NUM into SORTED while
;; preserving the non-decreasing ordering
```

```scheme
> (insert 3 '(5 8 9 10 11))   ← 3 goes at the front of the sorted list
(3 5 8 9 10 11)
```
> (insert 3 '(0 1 1 2)) ←− 3 goes at the end of the sorted list
(0 1 1 2 3)
> (insert 3 '(1 2 4 5 6)) ←− 3 goes somewhere in the middle
(1 2 3 4 5 6)
> (insert 3 '(1 2 3 4 4 4 9 12)) ←− Same as above, except that there’s another 3
(1 2 2 3 3 4 4 4 9 12)

Intuitively, the insert function walks through the already-sorted list until it finds the proper place for the given number. (What distinguishes the “proper place” for the given number?) We’ll have more to say about how the insert function might do this—in fact, we’ll define the insert function from scratch—but, for now, we’ll just take the insert function as given.

As indicated earlier, the insertion-sort algorithm takes a (usually unsorted) list of numbers as its only input. Its goal is to generate as its output a list containing the same elements, but sorted into non-decreasing order. Here is its contract:

;; INSERTION-SORT
;; ---------------------------------------------------------
;; INPUTS: LISTY, a list of numbers
;; OUTPUT: A list containing the same elements as LISTY,
;; but sorted into non-decreasing order

It can be implemented using list-based recursion, as follows. First, as a base case, consider that the empty list is already sorted.¹ Next, for the recursive case (i.e., when its input is a non-empty list), the insertion-sort algorithm applies the following recursive rule:

(insertion-sort listy) ⇒ (insert (first listy)
(insertion-sort (rest listy)))

According to its contract, the recursive call on the rest of listy should generate a sorted list containing all of the elements of (rest listy).² Therefore, to generate the desired output (i.e., a sorted list that contains all of the elements of listy), it only remains to find out where (first listy) should be inserted into that sorted rest of listy. And that is precisely what the call to the insert helper function does. Here is the completed definition of the insertion-sort function:

(define insertion-sort
(lambda (listy)
(cond
  ;; Base Case: LISTY is empty
  ((null? listy)
   (null? listy))
  ;; The empty list is already sorted
  ()
  ;; Recursive Case: LISTY is non-empty
  (else
   (insert (first listy)
            (insertion-sort (rest listy))))))

¹A one-element list is also already sorted, but we stick with the empty list as the base case to simplify the code slightly.
²In general, when defining recursive functions, we assume that the recursive function call will generate the right answer. After all, it will be evaluated using the same function that we are currently defining! This sort of assumption—which, at first, may seem crazy—is justified by mathematical induction.
Example 14.5.1: Applying insertion-sort to a sample list

Suppose that listy is the list (3 2 5 1 6). Then the recursive function call on the rest of listy would be, in effect,

(insertion-sort '(2 5 1 6))

Assuming that the recursive function call does the right thing, it should generate as its output the sorted list (1 2 5 6). Therefore, in this case, the above-mentioned recursive rule would, in effect, lead to the following sequence:

(insertion-sort '(3 2 5 1 6))
⇒ (insert 3 (insertion-sort '(2 5 1 6)))
⇒ (insert 3 '(1 2 5 6))
⇒ '(1 2 3 5 6)

And if we were to consider the details of each recursive function call, we would, in effect, end up with the following sequence of evaluations, using the abbreviations, i for insert, and isort for insertion-sort:

(isort '(3 2 5 1 6))
⇒ (i 3 (isort '(2 5 1 6)))
⇒ (i 3 (i 2 (isort '(5 1 6))))
⇒ (i 3 (i 2 (i 5 (isort '(1 6)))))
⇒ (i 3 (i 2 (i 5 (i 1 (isort '(6)))))
⇒ (i 3 (i 2 (i 5 (i 1 (i 6 (isort ()))))))
⇒ (i 3 (i 2 (i 5 (i 1 (i 6 ())))))
⇒ (i 3 (i 2 (i 5 (i 1 '6))))
⇒ (i 3 (i 2 (i 5 '(1 6))))
⇒ (i 3 (i 2 '(1 5 6)))
⇒ (i 3 '(1 2 5 6))
⇒ '(1 2 3 5 6)

In-Class Problem 14.5.1: The insert helper function

Define the insert function to satisfy the contract given earlier.

Hints: Use recursion to walk through sorted until you find the proper place for num. How will you recognize the proper place for num? Consider (first listy) and num. Finally, what should you do if sorted is empty?

In-Class Problem 14.5.2: Generating long lists of random numbers

Define a function, called list-of-n-random-numbers, that satisfies the following contract:

;; LIST-OF-N-RANDOM-NUMBERS
;; ----------------------------------------------
;; INPUT: N, a positive integer
;; OUTPUT: A list containing N numbers, each randomly generated
;;         from the set {0, 1, 2, ..., 99999}

Here are some examples of the desired behavior:

> (list-of-n-random-numbers 10)
(18980 44224 94176 57470 23568 47609 70753 77870 98756 11729)
> (list-of-n-random-numbers 5)
(68856 3578 85898 27820 87029)

Hint: In the recursive case, use the built-in random function with an appropriate input. This function can be used to randomly generate lists of numbers for insertion-sort to sort, as illustrated below:

> (let* ((list-o-randies (list-of-n-random-numbers 5))
        (sorted (insertion-sort list-o-randies)))
  (printf "BEFORE: " list-o-randies)
  (printf "AFTER: " sorted)
BEFORE: (68502 79284 50452 31764 48239)
AFTER: (31764 48239 50452 68502 79284)

> (let* ((list-o-randies (list-of-n-random-numbers 5))
        (sorted (insertion-sort list-o-randies)))
  (printf "BEFORE: " list-o-randies)
  (printf "AFTER: " sorted)
BEFORE: (51897 96352 87874 82047 17760)
AFTER: (17760 51897 82047 87874 96352)

Of course, it will be more interesting to see how long it takes insertion-sort to sort really long lists of numbers (e.g., lists having thousands of elements). In such cases, you wouldn’t want to print out the before and after lists!

To avoid excessive memory usage, it is better to implement accumulator-based tail-recursive versions of the insert and insertion-sort functions.

In-Class Problem 14.5.3: Accumulator-based tail-recursive version of the insert function

For this problem, the goal is to define an accumulator-based tail-recursive version of the insert function, called insert-acc. Recall that the insert function aims to insert a given number num into its proper place in an already-sorted list, sorted. The main idea behind the accumulator-based tail-recursive approach is to walk through sorted, accumulating all of its elements that are smaller than num, as illustrated below:

<table>
<thead>
<tr>
<th>(insert-acc</th>
<th>num</th>
<th>sorted</th>
<th>acc</th>
<th>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(insert-acc</td>
<td>5</td>
<td>'(1 2 4 6 12 15)</td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>(insert-acc</td>
<td>5</td>
<td>'(2 4 6 12 15)</td>
<td>'(1)</td>
<td></td>
</tr>
<tr>
<td>(insert-acc</td>
<td>5</td>
<td>'(4 6 12 15)</td>
<td>'(2 1)</td>
<td></td>
</tr>
<tr>
<td>(insert-acc</td>
<td>5</td>
<td>'(6 12 15)</td>
<td>'(4 2 1)</td>
<td></td>
</tr>
</tbody>
</table>

Notice that when all of the numbers smaller than num have been accumulated, the proper place for num has been found (i.e., the base case has been reached). The only thing that remains is to assemble the pieces into the final sorted list. In the above example, the desired list is (1 2 4 5 6 12 15), which can be built as follows:

(1) Use cons to attach num to the front of sorted, yielding (5 6 12 15).
(2) Use transfer-all (from Example 14.4.3) to transfer all of the elements of acc onto the result
of Step 1, yielding \((1 \ 2 \ 4 \ 5 \ 6 \ 12 \ 15)\).

Using the approach outlined above, define the \texttt{insert-acc} to satisfy the following contract:

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

Here are some examples of its use:

\begin{verbatim}
> (insert-acc 5 '(1 2 4 6 12 15) ())
(1 2 4 5 6 12 15)
> (insert-acc 3 '(1 1 2 2 3 3 4 4 5 5) ())
(1 1 2 2 3 3 3 4 4 5 5)
\end{verbatim}

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

\begin{verbatim}
;; INSERT-WR -- wrapper function for INSERT-ACC
;; -----------------------------------------------
;; INPUT: NUM, a number
;; SORTED, a list of numbers that are already sorted
;; into non-decreasing order
;; OUTPUT: A list containing NUM and all the numbers in SORTED,
;; all sorted into non-decreasing order.
\end{verbatim}

\begin{verbatim}
> (insert-wr 5 '(1 2 4 6 12 15))
(1 2 4 5 6 12 15)
> (insert-wr 3 '(1 1 2 2 3 3 4 4 5 5))
(1 1 2 2 3 3 3 4 4 5 5)
\end{verbatim}

\textbf{In-Class Problem 14.5.4: Tail-recursive version of \texttt{insertion-sort}}

For this problem, we seek a tail-recursive version of the \texttt{insertion-sort} algorithm. For convenience, we call it \texttt{isort-acc}. The following sequence of recursive function calls illustrates the approach, which uses an extra accumulator argument to accumulate the sorted list. At each step the first element of the unsorted list is inserted into its proper place in the sorted list:
14.5.2 The Merge-Sort Algorithm

The *merge-sort* algorithm, like the *insertion-sort* algorithm, takes a (typically unsorted) list of numbers as its input, and generates a sorted version of that list as its output. Here is its contract:

```scheme
;; MERGE-SORT
;; ---------------------------------------------------------
;; INPUTS: LISTY, a list of numbers
;; OUTPUT: A list containing the same elements as LISTY,
;; but sorted into non-decreasing order
```

However, the merge-sort algorithm takes a very different approach to sorting lists, as follows. First, its base case handles the case where the *listy* is a *one-element list* which, of course, must already be sorted. Second, when *listy* is non-empty, it uses recursion, as follows:

1. **Split** *listy* into two lists, *lefty* and *righty*, of roughly the same size;

2. Use the *merge-sort* function to sort *lefty*, yielding a sorted list, *sorted-lefty*; and use *merge-sort* to sort *righty*, yielding a sorted list, *sorted-righty*; and

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

As indicated above, the *merge-sort* function uses two helper functions: *split* and *merge*. These helpers will be defined shortly. For now, we will assume that they are available, and that they satisfy the following contracts:

```scheme
;; SPLIT
;; --------------------------------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: A list of the form (LEFTY RIGHTY) where LEFTY and RIGHTY are two subsidiary lists such that the elements of LISTY have been allocated as evenly as possible to LEFTY and RIGHTY, but with no regard to their order.

;; MERGE
;; ---------------------------------------------------------------
;; INPUT: SORTED-ONE, SORTED-TWO, two lists of numbers that are already sorted into non-decreasing order.
;; OUTPUT: A single list that contains all of the elements of SORTED-ONE and SORTED-TWO, sorted into non-decreasing order.
```
Example 14.5.2: The \texttt{split} and \texttt{merge} helper functions

Here are some examples of the behavior of the \texttt{split} and \texttt{merge} helper functions:

\begin{verbatim}
> (split '(5 3 1 2 8 4 9 4))  \leftarrow \text{Input has an even number of elements}
(4 4 2 3) (9 8 1 5))
> (split '(5 3 1 2 7))     \leftarrow \text{Input has an odd number of elements}
((7 1 5) (2 3))
> (merge '(1 3 5 7) '(2 4 6 8))
(1 2 3 4 5 6 7 8)
> (merge '(1 1 2 3 3 3 5 9) '(2 3 3 4 8 8 9)
(1 1 2 2 3 3 3 3 3 4 5 8 8 9 9)
\end{verbatim}

In the case of the \texttt{split} function, notice that the order of the elements in the input list and the two subsidiary lists in the output do not matter at all. The reason is that \texttt{split} will typically be applied to unsorted lists—so the order of the elements doesn’t matter. Also, if the input list has an even number of elements, then the two lists in the output will have the same number of elements; otherwise, one of the output lists will have the odd element. For the \texttt{merge} function, the two input lists must already be sorted, but they may have duplicate elements, and the two input lists need not have the same number of elements.

Example 14.5.3: Applying \texttt{merge-sort} to a sample list

Here, we consider the application of the \texttt{merge-sort} function to the input list \((8 2 5 9 3 4 6 1)\). As described previously, there are three steps to the recursive case:

1. Split listy into two lists, lefty and righty, of roughly the same size. Here:
   
   \begin{verbatim}
   lefty = (6 3 5 8)
   righty = (1 4 9 2)
   \end{verbatim}

2. Use the \texttt{merge-sort} function to sort lefty, yielding a sorted list, sorted-lefty; and use \texttt{merge-sort} to sort righty, yielding a sorted list, sorted-righty. Here:
   
   \begin{verbatim}
   sorted-lefty = (3 5 6 8)
   sorted-righty = (1 2 4 9)
   \end{verbatim}

3. Merge the two sorted lists, sorted-lefty and sorted-righty, into a single sorted list, which will be the desired output. Here:
   
   \begin{verbatim}
   (merge '(3 5 6 8) '(1 2 4 9)) \Rightarrow (1 2 3 4 5 6 8 9).
   \end{verbatim}

Here is the completed definition of the \texttt{merge-sort} function:

\begin{verbatim}
(define merge-sort
  (lambda (listy)
    (cond
      ;; Base Case: LISTY has exactly one element
      ((null? (rest listy)) listy)
      ;; A one-element list is already sorted
      ;; Recursive Case: LISTY has at least two elements
      (else
        (let* (;; LIST-O-LISTS has the form (LEFTY RIGHTY)
               (list-o-lists (split listy))
               ;; Access the two subsidiary lists in LIST-O-LISTS
               (lefty (list-o-lists 0))
               (righty (list-o-lists 1)))
          (let ((sorted-lefty (merge-sort lefty))
                 (sorted-righty (merge-sort righty)))
            (merge sorted-lefty sorted-righty))))))
\end{verbatim}
(lefty (first list-o-lists))
(righty (second list-o-lists))
;; Recursively sort LEFTY and RIGHTY
(sorted-lefty (merge-sort lefty))
(sorted-righty (merge-sort righty)))
;; Body of the LET*: MERGE the two sorted lists
(merge sorted-lefty sorted-righty)))))

Notice that most of the work is done in the variable-declaration part of the let* special form. The body of the let* just applies the merge function to the two sorted lists.

Now it is time to define the split and merge helper functions needed by merge-sort.

---

### In-Class Problem 14.5.5: The split helper function

Define the split helper function to satisfy the contract seen earlier. Here are some hints:

1. Define an accumulator-based helper function, called split-acc, that includes two extra inputs, lefty and righty. These will serve as accumulators for the two subsidiary lists.

2. In the base case, use the list-two function defined in In-Class Problem 14.1.2 to create the desired list of lists.

3. Define split as a wrapper function that simply calls split-acc with appropriate initial values for its accumulator inputs.

---

### In-Class Problem 14.5.6: The merge helper function

Define the merge helper function to satisfy the contract seen earlier. Here are some hints:

1. When either list is empty, the answer is easy.

2. When both lists are non-empty, compare their first elements to see which one comes first.

Define two versions of the merge function: one that is not tail recursive (and perhaps easier to define), and one that is just a wrapper for a tail-recursive helper function called merge-acc. The contract for merge-acc is given below:

```scheme
;; MERGE-ACC
;; -----------------------------
;; INPUTS: SORTED-LEFTY, SORTED-RIGHTY, two lists of
;;         numbers, each sorted into non-decreasing order
;;         ACC, a list-accumulator
;; OUTPUT: When called with ACC=(), the output is a
;;         single list containing all of the elements
;;         of SORTED-LEFTY and SORTED-RIGHTY, sorted
;;         into non-decreasing order.

Hint: When both sorted-lefty and sorted-righty are non-empty, “accumulate” the smaller of (first sorted-lefty) and (first sorted-righty). In the base case, use the built-in reverse function to reverse the accumulated list.
14.5.3 Comparing the Performance of Insertion Sort and Merge Sort

This section shows how we can write Scheme functions to automate a rigorous comparison of the insertion-sort and merge-sort algorithms. Some considerations include:

- We want to test these algorithms on really long lists of randomly generated numbers.
- For each randomly generated list, we want to test both algorithms on the same list.
- We’d like to know how long it takes each algorithm to sort the lists.

We already have the list-of-n-random-numbers function, from In-Class Problem 14.5.2. And since the two sorting algorithms are non-destructive, we can simply store the randomly generated list of numbers in a local variable, and then apply each sorting algorithm to the same list. As for timing their performance, Scheme provides a special form, called time, described below.

The time special form. The purpose of the time special form is to report how long it takes to evaluate a given expression. The syntax and semantics of the time special form are simple.

(Syntax) Any expression of the form (time expr) is a legal instance of the time special form.

(Semantics – Output Value) Any expression of the form (time expr) evaluates to whatever expr evaluates to.

(Semantics – Side Effect) The evaluation of an expression of the form (time expr) causes three pieces of timing information to be displayed in the Interactions Window:

- cpu time how many milliseconds DrScheme spent evaluating expr. (CPU is an acronym for the computer’s central processing unit.)
- real time how many milliseconds elapsed while expr was evaluated.
- gc time how many milliseconds were spent in a memory-management process called garbage collection. (Garbage collection is an extremely interesting and important concept in the management of a computer’s memory, but a discussion of it is beyond the scope of this book.)

The cpu time is typically a bit less than the real time because a computer’s CPU typically does more than one thing during any given time interval; thus, the time the CPU devotes to DrScheme’s evaluation of expr will typically be less than the elapsed time. For our purposes, the cpu time is the most relevant, because it most accurately reflects how much time DrScheme spent evaluating the given expression.

Example 14.5.4: Using the time special form

Here are some examples of the time special form in action:

```
> (time (list-of-n-random-numbers 10000))
cpu time: 4 real time: 5 gc time: 0
(19207 53390 65067 65764 68321 75622 81451 38038 86109 ...)
> (time (insertion-sort (list-of-n-random-numbers 10000)))
cpu time: 7643 real time: 7849 gc time: 62
(10 12 26 30 50 65 70 77 80 83 94 104 108 113 114 150 ...)
> (let ((listy (list-of-n-random-numbers 10000)))
  (time (insertion-sort listy)))
cpu time: 7519 real time: 7674 gc time: 61
(2 9 14 16 26 31 32 37 38 40 84 85 113 114 115 119 171 ...)
```

The first example shows that it doesn’t take DrScheme long to generate a list of 10,000 random numbers. The second example shows how long it takes to generate and sort a list of numbers, using the
insertion-sort function. The last example is the most important: it shows how long the sorting process takes; it ignores the time needed to generate the original list of random numbers.

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

Example 14.5.5: Comparing the performance of the sorting algorithms

The following function can be used to compare the performance of the insertion-sort and merge-sort algorithms.

;;; COMPARE-SORTING-ALGS
;;; -----------------------------------------------------
;;; INPUT: N, a positive integer
;;; OUTPUT: None
;;; SIDE EFFECT: Reports how long it took for the
;;; insertion-sort and merge-sort algorithms to sort
;;; the *same* randomly generated list of N numbers.

(define compare-sorting-algs
  (lambda (n)
    (let ((listy (list-of-n-random-numbers n)))
      (printf "Running insertion-sort ...\n"
              (time (insertion-sort listy))
      (printf "%\n"
              (time (merge-sort listy))
      ;; Return VOID
      (void)))))

Here is an example:

> (compare-sorting-algs 1000)
Running insertion-sort ...
  cpu time: 87 real time: 93 gc time: 0
Running merge-sort ...
  cpu time: 6 real time: 6 gc time: 0

In-Class Problem 14.5.7: A thorough comparison of merge-sort and insertion-sort

Use the compare-sorting-algs function to compare the performance of the two sorting algorithms on lists of the following lengths: 1000, 2000, 4000, 8000, 16000, etc. Which algorithm would you recommend? Try running the faster of the two algorithms on really long lists (e.g., with 100,000 elements, or even a million elements).

Example 14.5.6: The built-in sort function

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

;;; SORT -- built-in function
14.6 The Underlying Structure of Non-Empty Lists

Up to this point, we have seen that non-empty lists can often be effectively processed recursively using only the first and rest accessor functions. The reason for this is that the underlying structure of non-empty lists in Scheme is, in fact, based on decomposing them into their first and rest parts. The rest of this section explores that structure, revealing the central role of a data structure called a cons cell—also known as a pair.

14.6.1 Data Structures

In Computer Science, the term, data structure, refers to any organized (or structured) collection of data. Typically, each data structure has one or more slots for holding data. In some data structures, the slots for holding data are indexed so that any particular slot can be accessed by its corresponding (numerical) index. For example, the slots in vectors—to be discussed in Chapter 15—are indexed in this way. In other data structures, the slots for holding data are named so that any particular slot can be accessed by its name. Named slots are often called fields. For example, a bank-account data structure might have fields called password and balance. The rest of this section restricts attention to a very simple field-based data structure that, for historical reasons, is called a cons cell. Each cons cell has only two fields. For this reason, cons cells are also called pairs. General field-based data structures will be addressed thoroughly in Chapter 16.

14.6.2 Cons Cells (a.k.a. Pairs)

A cons cell is a field-based data structure structure that has only two fields: one named first, and one named rest. (Yes, that’s right! Stay tuned for the relationship between cons cells and non-empty lists.) Scheme provides the following built-in functions for computing with cons cells, one of which we have already seen:

```
cons     For constructing a new cons cell
cons?    Type-checker predicate for cons cells
```
**Example 14.6.1: The cons function revisited**

Here is a more accurate contract for the cons function. Notice that the second input need not be a list.

```scheme
;; CONS -- built-in function
;; ------------------------------------------------------
;; INPUTS: FST, RST, any Scheme data
;; OUTPUT: A cons cell whose FIRST field contains FST,
;; and whose REST field contains RST.
```

The following Interactions Window session demonstrates that the output generated by the cons function is indeed a cons cell, as confirmed by the built-in cons? type-checker predicate:

```scheme
> (cons 1 2)
(1 . 2)
> (cons? (cons 1 2))
#t
> (cons 'x "1232")
(x . "1232")
> (cons? (cons 'x "1232"))
#t
> (cons #t 'abc)
(#t . abc)
> (cons? (cons #t 'abc))
#t
```

Notice that if the output value is a cons cell, DrScheme displays the result using the dotted-pair notation. For example, a cons cell whose first field contains 1 and whose rest field contains 2 is displayed as (1 . 2) by DrScheme.

* DrScheme uses the dotted-pair notation when the rest field of a cons cell is something other than a list.

* The dotted-pair notation is not legal Scheme syntax; so we cannot use it in our Scheme programs or in the Interactions Window.

It must be stressed that:

* Although the dotted-pair notation shown above utilizes parentheses, it does not represent a list!

However:

* When the rest field of a cons cell contains a list, then that cons cell is a non-empty list!

In such cases, the Scheme datum is both a cons cell and a non-empty list. This does not contradict the statement made long ago—in Chapter 2—that a datum can only belong to one data type because:

* The set of non-empty lists is an example of a compound data type. Each non-empty list is, in fact, a cons cell that has special contents, in particular, one whose rest field contains a list.

**Example 14.6.2: Cons cells vs. non-empty lists**

The following interactions demonstrate that a non-empty list is a cons cell whose rest field contains a list, whereas a cons cell whose rest field contains some other kind of data is not a list.

```scheme
> (cons? '(2 3 4))
#t
```
> (list? (rest (cons '(2 3 4))))
#t
> (cons? (rest (cons '(2 3 4))))
#t
> (cons 1 2) ← A dotted pair is not a list
(1 . 2)
> (list? (cons 1 2))
#f

Furthermore, as seen previously, when the rest field of a cons cell contains a list, DrScheme displays that cons cell using the familiar list notation:

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

Fig. 14.1 shows one way of depicting the non-empty list, (3 4 6), as a single cons cell—with very particular contents. In this case, the list is indeed represented as a single cons cell—the biggest one in the picture. The first field in this cons cell contains the datum 3; the rest field of this cons cell contains another cons cell—one that represents the rest of the list (i.e., (4 6)). The first field of that cons cell contains the datum 4; the rest field contains ... yet another cons cell! The first field of the innermost cons cell contains the datum 6; the rest field contains the empty list, which signals that we have reached the end of the list (3 4 6). Notice that the list represented by these three nested cons cells has three elements: 3, 4 and 6. Notice further that the first field of each cons cell contains one of the elements of the list.

* In general, if a list contains n elements, it can be represented by a nested structure of n cons cells.

**Example 14.6.3: The structure of non-empty lists**

The following interactions demonstrate that a list containing n elements can be represented by a nested structure of n cons cells.

> (cons 3 (cons 4 (cons 6 ())))
(3 4 6)
> (cons 1 (cons 2 (cons 3 (cons 4 ()))))
(1 2 3 4)
> (cons 'x (cons 'y (cons 'z ())))
(x y z)
Figure 14.2: An alternative depiction of the non-empty list, \((3 \ 4 \ 6)\), as a chain of cons cells

Although Fig. 14.1 provides an accurate depiction of the nested structure of cons cells that can be used to represent a non-empty list, this kind of picture would get awfully difficult to draw for lists containing more than, say, five or ten elements. For this reason, we prefer to depict non-empty lists as \(\textit{chains}\) of cons cells, using arrows, as illustrated in Fig. 14.2. It is important to realize that the non-empty list depicted by this figure is the same list as that depicted in Fig. 14.1 (i.e., we have two kinds of picture-syntax for one semantic list!). Instead of showing the rest of the list as a cons cell \(\textit{nested inside}\) the \(\textit{rest}\) field, this depiction uses an arrow from the \(\textit{rest}\) field of one cons cell to the next cons cell in the chain. Similarly, the \(\textit{rest}\) field of the second cons cell points to the third cons cell in the chain. Finally, the \(\textit{rest}\) field of the last cons cell, which contains the empty list, is often depicted as a box with an \(X\) in it, signalling the end of the chain.

So... is a non-empty list a single cons cell? Or is it a chain of cons cells? The answer is: it depends on how you look at it! For example, according to the \texttt{cons?} type-checker predicate, a non-empty list is most definitely a single cons cell:

\[
> (\texttt{cons? '}(2 \ 3 \ 4))
\]

\texttt{#t}

On the other hand, if the \(\textit{rest}\) field of a given cons cell \(C_1\) contains a nested cons cell \(C_2\), then the thing that actually gets written into the \(\textit{rest}\) field of \(C_1\) in the computer’s memory is undoubtedly the \textit{address} of \(C_2\) (i.e., the location in the computer’s memory where \(C_2\) can be found). In other words, the \(\textit{rest}\) field of \(C_1\) contains a \textit{pointer} to \(C_2\)—which can be represented by an arrow, as in Fig. 14.2! In short, you can look at it both ways. For our purposes, thinking of non-empty lists as chains of cons cells will be most convenient.

---

### In-Class Problem 14.6.1: Defining our own type-checker predicate for lists

\textit{Define a predicate that satisfies the following contract:}

\[
;; \textit{WELL-FORMED-LIST?}
;; \textbf{-------------------------------------------------------}
;; \textbf{INPUT:} \ \textsc{datum}, anything
;; \textbf{OUTPUT:} \ #t if \textsc{datum} is an empty or non-empty list.
;; \textbf{If non-empty,} \textsc{datum} should be a chain of cons
;; \textbf{cells, each of whose} \texttt{rest} \textbf{slot is filled by}
;; \textbf{a well-formed list.}
\]

\textit{Here are some examples of its use:}

\[
> \ (\texttt{well-formed-list? ()})
\]

\texttt{#t}

\[
> \ (\texttt{well-formed-list? '}(a \ b \ c \ d))
\]

\texttt{#t}

\[
> \ (\texttt{well-formed-list? (cons 1 (cons 2 3)))}
\]

\texttt{#f}

\[
> \ (\texttt{well-formed-list? 'xyz})
\]

\texttt{#f}

\textit{Since this function is a predicate, you should be able to define it using \texttt{and}, \texttt{or} and \texttt{not}, without using \texttt{if} or \texttt{cond}.}
Now that we have explored the underlying structure of non-empty lists in terms of cons cells, you should review all of the examples from earlier in this chapter to make sure that you understand the underlying structures of the lists involved.

Example 14.6.4: The double-all function revisited

Recall the definition of the double-all function seen in Example 14.3.1 which takes a list of numbers as its input, and generates a list of the same length whose elements are obtained by doubling the corresponding elements from the input list.

```
(define double-all
  (lambda (listy)
    (cond
      ;; Base Case: LISTS is empty
      ((null? listy)
        (;; The double-all of () is ...
        ()))
      ;; Recursive Case: LISTY is non-empty
      (else
        ;; Double the first element and attach it to the
        ;; double-all of the rest of the list
        (cons (* 2 (first listy))
          (double-all (rest listy)))))))
```

Here's an example of its behavior:

```
> (double-all '(3 1 4 7))
(6 2 8 14)
```

In general, the double-all function returns a list containing the same number of elements as its input. Equivalently, we may say that the double-all function is length preserving. This can be formally proved using the technique of mathematical induction; however, we shall content ourselves with a less formal analysis.

First, note that for any datum \(d\) and any list \(\ell\), the list \((\text{cons } d \ \ell)\) has one more element than \(\ell\). Thus, for example, the list \((3\ 1\ 4\ 7)\), which is equivalent to \((\text{cons } 3\ (1\ 4\ 7))\), has one more element than \((1\ 4\ 7)\). But now consider \((\text{double-all } (3\ \text{cons } 4\ \text{cons } 7))\). By the recursive case, \((\text{double-all } (3\ 1\ 4\ 7))\) effectively evaluates to \((\text{cons } 6\ (\text{double-all } (1\ 4\ 7)))\), which has one more element than \((\text{double-all } (1\ 4\ 7))\). Therefore, if we want to show that 

\((\text{double-all } (3\ 1\ 4\ 7))\) and \((3\ 1\ 4\ 7)\) have the same number of elements, we need only show that 

\((\text{cons } 6\ (\text{double-all } (1\ 4\ 7)))\) and \((\text{cons } 3\ (1\ 4\ 7))\) have the same number of elements. But then, by a similar line of reasoning, this will hold if and only if

\((\text{double-all } (4\ 7))\) and \((4\ 7)\) have the same number of elements. And that will hold if and only if

\((\text{double-all } (7))\) and \((7)\) have the same number of elements. And that holds—since

\((\text{double-all } ()))\) evaluates to ()!

The technique described in the preceding example can be used to show that the built-in \text{map} function is also length preserving. For example, \((1\ 2\ 3\ 4)\) and the list generated by evaluating \((\text{map } \text{facty } (1\ 2\ 3\ 4))\) must have the same length.
In-Class Problem 14.6.2: Picturing the length preserving nature of `double-all` and `map`

Draw the chain of cons cells corresponding to the list `(3 1 4 7)`. Draw a circle around the portion of that chain that corresponds to the rest of the list. Then draw the chain of cons cells corresponding to the list `(6 2 8 14)` generated by evaluating `(double-all '(3 1 4 7))`. Draw a circle around the portion of the chain corresponding to the rest of that list. Notice that the first cons cell in `(3 1 4 7)` is matched by the first cons cell in `(6 2 8 14)`; and that the rest of the cons cells in `(3 1 4 7)` are matched by the rest of the output list `(6 2 8 14)` generated by the recursive function call. In other words, each call to `double-all` effectively consumes one cons cell from the input list and produces one cons cell in the output list. For that reason, the input and output lists must have the same number of cons cells and, hence, the same number of elements.

14.7 Hierarchical/Deep/Nested Lists

The syntax of Scheme expressions allows lists that contain other lists as elements. Indeed, lists may contain lists that contain other lists that contain other lists, and so on, to any desired depth.

* A list that has at least one element that is itself a list is called a hierarchical (or deep or nested) list.

* A list that does not contain any lists as elements is sometimes called a flat list.

For example, the expression `(x (2 (3) 2) #t)` denotes a hierarchical list whose three elements are: the symbol `x`, the subsidiary list `(2 (3) 2)`, and the boolean `#t`. This section demonstrates that recursively processing hierarchical lists is frequently only slightly more complicated than recursively processing flat lists. Indeed, when recursively processing the items in a deep list, it often happens that one need only insert one extra case to handle the possibility that the item currently under consideration is itself a list.

⇒ By convention, functions that recursively process hierarchical lists frequently have names ending in an asterisk (e.g., `sum-all*` instead of `sum-all`).

Example 14.7.1: Summing the items in a hierarchical list

Summing all of the items in a hierarchical list turns out to be only slightly more involved that summing the items in a flat list. (You may wish to review the `sum-all` function defined in Example 14.2.2.) The contract for the hierarchical version, called `sum-all*`, is given below, followed by some examples of its use.

```scheme
;; SUM-ALL*
;; -----------------------------
;; INPUT: HLISTY, a (possibly hierarchical) list of numbers
;; OUTPUT: The sum of all of the numbers appearing anywhere
;;         within HLISTY

> (sum-all* '(1 (2 (3 4) 5) 6))
21
> (sum-all* '(((10 100))))
111
```

You may recall that the `sum-all` function contained a `cond` expression with two cases: a base case and a recursive case. Below, the `sum-all*` function includes an extra recursive case that handles the possibility that the item currently under consideration (i.e., `(first hlisty)`) is itself a list.

```scheme
(define sum-all*
  (lambda (hlisty)
    (cond
```

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

Notice that when (first hlisty) is itself a list, it follows that both (first hlisty) and (rest hlisty) are lists. Therefore, the sum-all* function can be recursively applied to both of these lists, and the results added together to generate the desired sum. For example, if hlisty is the list (((1 2 (3)) 4 (5 1)), then (first hlisty) is the list (1 2 (3)) and (rest hlisty) is the list (4 (5 1)). Recursively applying sum-all* to these two lists yields the results, 6 and 10, respectively. The sum of these two numbers (i.e., 16) is the sum of all of the numbers in hlisty.
⇒ Notice that, as usual, we let the recursive function calls do most of the work!

Note. Using the list? predicate (e.g., in Recursive Case 1, above) to check whether (first hlisty) is a list can be terribly inefficient because, in cases where (first hlisty) happens to be a long list, the list? predicate will walk down its entire length, checking that it is a well formed chain of cons cells. Instead, if we assume that hlisty does not contain any malformed chains of cons cells, we can greatly increase the efficiency of Recursive Case 1 by using the quick-list? predicate, defined below.

;; QUICK-LIST?
;; ----------------------------------------------
;; INPUT: DATUM, anything
;; OUTPUT: #t if DATUM is either () or a cons cell;
;;         #f otherwise.
(define quick-list? (lambda (datum)
                      (or (null? datum) (cons? datum))))

Unlike list?, the quick-list? predicate does not walk down any chains of cons cells; instead, if datum is a cons cell, it simply assumes that it is the first cons cell in a well formed chain (i.e., that it is a non-empty list).

* The rest of the examples in this section assume that all hierarchical lists are well formed (i.e., that they do not contain any malformed chains of cons cells).

Example 14.7.2: Top-level elements vs. leaf items in hierarchical lists

Recall In-Class Problem 14.2.2, whose goal was to define a function to compute the number of elements in a flat list. Here is one solution:

;; LENGTHY
;; ----------------------------------------------
;; INPUT: LISTY, any list
;; OUTPUT: The number of elements of LISTY (i.e., its length)
As demonstrated below, the `lengthy` function does not care whether the individual elements of `listy` are symbols, numbers, booleans, or ... even other lists! Thus, it counts what we sometimes call the top-level elements of `listy`.

```
> (lengthy '(a b c d e))
5
> (lengthy '(x (1 1) (2 (3) 2) y))
4
> (lengthy '((((((3 3 3)))))))
1
```

For contrast, the function, `num-leaf-items*`, counts the number of so-called leaf items in a possibly hierarchical list—that is, the items that appear at any level of the hierarchy.

```
;; NUM-LEAF-ITEMS*
;; ------------------------------------------------------
;; INPUT: HLISTY, a (possibly hierarchical) list
;; OUTPUT: The number of items that appear in HLISTY
;; at any level of the hierarchy.

Here is how `num-leaf-items*` treats the same lists encountered above:

```
> (num-leaf-items* '(a b c d e))
5
> (num-leaf-items* '(x (1 1) (2 (3) 2) y))
7
> (lengthy '((((((3 3 3)))))))
3
```

Notice that for flat lists such as `(a b c d e)`, where each item occurs as a top-level element, `num-leaf-items*` outputs the same answer as `lengthy`. However, `num-leaf-items*` treats hierarchical lists much differently. Note that it does not count subsidiary lists, but only the primitive data that appear within them. Thus, `(num-leaf-items* '(x (1 1) (2 (3) 2) y))` outputs 7, for the seven leaf items: x, 1, 1, 2, 3, 2 and y. Although `num-leaf-items*` descends into the hierarchy of the input list, counting all the leaf items it finds along the way, defining this function is not difficult—as long as we let recursive function calls do most of the work! The following solution demonstrates that `num-leaf-items*` need only include one additional case, to handle the possibility that the element currently under consideration is itself a list:
(define num-leaf-items*
  (lambda (hlisty)
    (cond
      ;; Base Case: HLISTY is empty
      ((null? hlisty) 0)
      ;; Recursive Case 1: (FIRST HLISTY) is itself a list!
      ((quick-list? hlisty)
        ;; Recursive calls on (FIRST HLISTY) and (REST HLISTY)
        ;; compute the numbers of items in each part of HLISTY.
        (+ (num-leaf-items* (first hlisty))
          (num-leaf-items* (rest hlisty))))
      ;; Recursive Case 2: (FIRST HLISTY) is NOT a list
      (else
        ;; Count 1 for (FIRST HLISTY); let the recursive
        ;; function call count the items in (REST HLISTY).
        (+ 1 (num-leaf-items* (rest hlisty)))))))

Notice that the Base Case and Recursive Case 2 are completely analogous to the Base Case and Recursive Case for \texttt{lengthy}. The only difference is the insertion of Recursive Case 1, which handles the possibility that \texttt{(first hlisty)} is itself a list. And that case is easily handled because, in that case, \texttt{(first hlisty)} and \texttt{(rest hlisty)} are both lists. Recursively applying \texttt{num-leaf-items*} to both of those lists, and then summing the results, gives the desired answer.

\section*{In-Class Problem 14.7.1: A hierarchical version of the \texttt{map} function}

Define a function, called \texttt{map*}, that satisfies the following contract:

\begin{verbatim}
;; MAP*
;; -------------------------------------------------------------
;; INPUTS: FUNC, a function that expects one input
;; HLISTY, a (possibly hierarchical) list of
;; suitable inputs for FUNC
;; OUTPUT: A list with the same structure as HLISTY, where
;; each item is obtained by applying FUNC to the
;; corresponding item in HLISTY.
\end{verbatim}

Here are some examples of its behavior:

\begin{verbatim}
> (map* abs '((-1) (2 -3) (-4 (5))))
((-1) (2 3) (4 (5)))
> (map* (lambda (x) (* x x)) '(1 (2 (3 4) 5) 6))
(1 (4 (9 16) 25) 36) 49)
\end{verbatim}

\section*{In-Class Problem 14.7.2: Flattening a hierarchical list}

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

\begin{verbatim}
;; FLATTEN*
;; -------------------------------------------------------------
\end{verbatim}
;; INPUT: HLISTY, a (possibly hierarchical) list
;; OUTPUT: A flat (i.e., non-hierarchical) list that contains
;; all of the items from HLISTY "in the same order".

Here are some examples of its behavior:

> (flatten* '((4 2) 3 (x (y))))
(4 2 3 x y)

> (flatten* '(1 (2 (3) 4) 5))
(1 2 3 4 5)

Hint: In one case, use the built-in append function; in another, use cons.

Problems

Problem 14.25: Computing the depth of a hierarchical list

Define a function, called depth*, that computes the maximum depth of any item in the given (possibly hierarchical) list. Here is its contract, followed by some examples illustrating the desired behavior.

;; DEPTH*
;; --------------------------------------------------------------
;; INPUT: HLISTY, a (possibly hierarchical) list
;; OUTPUT: The maximum depth of any item in HLISTY

> (depth* '(a b c))
1

> (depth* '(1 (2 (3) 2) 1))
3

The first example involves a flat list, each of whose elements is considered to be at depth one. Thus, the maximum depth for that list is one. In the second example, each 1 occurs at depth one, each 2 occurs at depth two, and the 3 occurs at depth three. Thus, the maximum depth for that list is three. (Notice that the 3 is nested within three sets of matching parentheses.) By convention, the depth of the empty list is taken to be zero.

Problem 14.26: Replacing items in a hierarchical list

Define a function that satisfies the following contract.

;; REPLACE*
;; --------------------------------------------------------------
;; INPUT: OLD, anything
;; NEW, anything
;; HLISTY, a (possibly hierarchical) list
;; OUTPUT: A list that is the same as HLISTY, except that
;; each occurrence of OLD in HLISTY (as judged by
;; EQ?) has been replaced by an occurrence of NEW.

Here are some examples:

> (replace* 1 ’one ’(1 2 (1 2 (1 (2)) 1)))
(one 2 (one 2 (one one (2)) one))
> (replace* 'x 'ecks '(a ((x) x) b (x (s) x)))
(a ((ecks) ecks) b (ecks (s) ecks))

---

Problem 14.27: A hierarchical version of is-elt-of

Define a function that satisfies the following contract.

;;; IS-ELT-OF
;;; -----------------------------------------
;;; INPUTS: ITEM, anything
;;;        HLISTY, a (possibly hierarchical) list
;;; OUTPUT: #t if ITEM appears somewhere within HLISTY (as
;;;         judged by EQ?); #f otherwise.

> (is-elt-of* 3 '(1 2 (4 (8 (2 3 9) 6 5)))))
#t
> (is-elt-of* 3 '(1 2 (4 (8 (2 0 9) 6 5)))))
#f

Note: Since this is a predicate, you may wish to define it using some combination of and, or and not, instead of using cond or if.

Problem 14.28

Define a function, called equal?*, that satisfies the following contract:

;;; EQUAL? *
;;; -----------------------------------------
;;; INPUTS: HLISTY, HLISTZ, two possibly hierarchical lists
;;; OUTPUT: #t if HLISTY and HLISTZ contain the same items,
;;;         in the same order, at every level of their hierarchies.

Here are some examples of its use:

> (equal?* '(a b c) '(a b c))
#t
> (equal?* '(a (b (c) d)) '(a (b (c) d)))
#t
> (equal?* '(a (b (c c) e)) '(a (b (c x) d)))
>#f

Note that, unlike the list-equal? function seen in Problem 14.12, the equal?* function does not use the built-in eq? predicate to test the equality of corresponding items, because the eq? predicate is unable to confirm the equality of lists in general. Instead, the equal?* function should use a recursive function call to deal with lists that appear as items anywhere within the hierarchy.

* Be careful! It may be that (first hlisty) is a list, but (first hlistz) is not.

Incidentally, the built-in equal? function is able to correctly determine whether two hierarchical lists have the same contents, at every level of their hierarchies.

*When comparing non-empty lists, the eq? predicate only checks whether they start with the same cons cell; it doesn’t even look at the contents of that cons cell. The differences are illustrated by the following interactions:
> (eq? '(a b) '(a b))
#f
> (let ((listy '(a b))) (eq? listy listy))
#t

**Problem 14.29**

Define a function, called replace-by-depth*, that satisfies the following contract:

```scheme
;; REPLACE-BY-DEPTH*
;; -------------------------------------
;; INPUT: HLISTY, a (possibly hierarchical) list
;; OUTPUT: A list that is just like DLISTY, except that
;; each item in the list has been replaced by a NUMBER
;; that is equal to the depth of that item in the list.
```

Here are some examples:

```scheme
> (replace-by-depth* '(a (b ((c) d) e) (f)))
(1 (2 ((4) 3) 2) (2))
> (replace-by-depth* '(((x))))
(((3)))
```

Hint: Define a helper function that includes an extra input, curr-depth, that keeps track of the current depth. That way, when you come across an item that needs to be replaced, you can just replace it by curr-depth. When should the value of curr-depth be increased?

---

**Special Forms Introduced in this Chapter**

`time` Displays timing information
## Built-in Functions Introduced in this Chapter

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>first, rest</code></td>
<td>Accessor functions for lists</td>
</tr>
<tr>
<td><code>cons</code></td>
<td>Create a new list by attaching a new item to the front of a given list</td>
</tr>
<tr>
<td><code>cons?</code></td>
<td>Type-checker for cons cells</td>
</tr>
<tr>
<td><code>second, third, fourth, etc.</code></td>
<td>Additional accessor functions for lists</td>
</tr>
<tr>
<td><code>list</code></td>
<td>Create a list containing the specified items</td>
</tr>
<tr>
<td><code>member</code></td>
<td>Does an item appear in a list?</td>
</tr>
<tr>
<td><code>map</code></td>
<td>Apply given function to each element of a list, in turn</td>
</tr>
<tr>
<td><code>length</code></td>
<td>Compute the number of elements in a list</td>
</tr>
<tr>
<td><code>list-ref</code></td>
<td>Fetch the $n^{th}$ element of a list—general purpose accessor function</td>
</tr>
<tr>
<td><code>append</code></td>
<td>Concatenate two lists</td>
</tr>
<tr>
<td><code>reverse</code></td>
<td>Reverse the elements of a list</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>Sort a list according to a given comparison function</td>
</tr>
<tr>
<td><code>equal?</code></td>
<td>Test equality of two lists (cf. Problem 14.12)</td>
</tr>
</tbody>
</table>
Part II

Destructive Programming in Scheme
Chapter 15

Vectors

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

Recall that lists can be incrementally extended, one element at a time, using the cons function. As a result, as illustrated in Fig. 15.1, the individual cons cells in which the various elements of a list are stored frequently end up being scattered haphazardly about the computer's memory. For this reason, accessing elements of a list can be relatively slow. For example, to access the one-thousandth element of a list typically requires recursively walking through the first thousand cons cells in the chain. However, as has been demonstrated repeatedly in previous chapters, much recursive processing of lists can be done while only accessing the first and rest of a list. For these sorts of applications, lists are quite handy. Furthermore, many list-based functions can be written non-destructively, which facilitates testing and debugging, since the performance of a non-destructive function depends only on the code within its body.

In sharp contrast to lists, vectors are stored within a single, contiguous block of memory—which has both advantages and disadvantages. The principal advantage is that every item of a vector can be accessed very quickly, whether the first element or the thousandth. To demonstrate why, Fig. 15.2 illustrates the layout of the elements in a typical vector. Notice that each slot has a numerical index, starting at zero. A crucial feature of the layout is that each slot takes up the same amount of space (e.g., four bytes per slot in the figure). Since each slot has the same size, the memory address of any given slot is very easy to compute. For example, if the start of the vector is located at memory address 1000, and each slot is four bytes wide, then the address of the $i$th slot is $1000 + 4 \times i$. (Thankfully, DrScheme takes care of such low-level details. A Scheme programmer need never deal directly with memory addresses.) An important feature of the “Random Access Memory” (RAM) found in modern computers is that the contents of any memory address can be fetched in the same (very small) amount of time. (The term “random” here is used to indicate that the contents of any randomly selected memory address can be fetched in the same, very small amount of time.) Thus, the time required to fetch the first element of a vector (i.e., the element with index 0) is the same as the time required to fetch the thousandth element.

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

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

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

A vector is a datum. Scheme provides a special syntax, the pound syntax, for denoting vectors. Just as the expression (a b c) can be used to represent (or denote) a list containing the three elements a, b and c, the expression #(a b c) can be used to represent (or denote) a vector containing the three elements a, b and c. In general, any expression of the form #(e<sub>1</sub>,e<sub>2</sub>,...,e<sub>n</sub>) can be used to represent a vector containing the n elements denoted by the subsidiary expressions e<sub>1</sub>,e<sub>2</sub>,...,e<sub>n</sub>. Thus, for example, the expression #(x #t () 32) denotes a vector containing four elements: a symbol, a boolean, the empty list, and a number.

One important fact about the #(...) syntax is that the vectors it represents are immutable (i.e., their contents cannot be changed). This limits the usefulness of the #(...) syntax. However, it can be useful when testing functions that don’t need to modify their vector inputs (e.g., functions that print out the contents of a vector).

* Unlike lists, vectors evaluate to themselves!
Example 15.1.1: Demonstrating that vectors evaluate to themselves

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

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

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

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

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

15.2 Constructing Vectors

Scheme provides two built-in functions, called vector and make-vector, that can be used to create new vectors. The vector function is similar to the built-in list function, mentioned briefly in Section 14.1. The make-vector function is typically the most practical. It enables the programmer to create a vector of any specified length.

15.2.1 The Built-in vector Function

The vector function is a built-in function that works very much like the built-in list function, mentioned briefly in Section 14.1. Whereas list constructs a list containing the specified elements, vector constructs a vector containing the specified elements. Unlike the vectors denoted by the #(...) syntax, vectors created by the vector function are mutable (i.e., their contents can be changed—we’ll see how momentarily).

Example 15.2.1: Using the vector function

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

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

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

The make-vector function is a built-in function that can be used to create a vector of any specified length. It is the most common way of creating a vector because it can be used, for example, to easily create a vector with, say, 1000 slots. Like with the vector function, vectors created by make-vector are mutable (i.e., their contents can be changed). By default, make-vector inserts the value 0 into each slot of the vector it creates.

Example 15.2.2: Using the make-vector function

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

```scheme
;; MAKE-VECTOR -- built-in function
;; -----------------------------------------------
;; INPUT: NUM-SLOTS, a non-negative integer
;; OUTPUT: A vector containing NUM-SLOTS slots, each initially 0

> (make-vector 5)
#(0 0 0 0 0)
> (make-vector 15)
#(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

15.3 Accessing Information Stored in a Vector

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

15.3.1 The vector-ref Function

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

Example 15.3.1: Using vector-ref

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

```scheme
;; VECTOR-REF -- built-in function
;; -----------------------------------------------
;; INPUTS: VECKY, a vector
;; INDY, an index
;; OUTPUT: The item of VECKY stored at slot INDY

> (define vecky #(a b c d e))
> (vector-ref vecky 0)
a
> (vector-ref vecky 2)
c
```
Although the `vector-ref` and `list-ref` functions may appear similar syntactically, they operate quite differently. The `vector-ref` function accesses the specified element of a vector nearly instantaneously, while the `list-ref` function walks through each element of the specified list until it finds the one with the desired index. Thus, `vector-ref` is much more efficient than `list-ref`.

### 15.3.2 The `vector-length` Function

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

#### Example 15.3.2: Using `vector-length`

Here is the contract for `vector-length`, followed by some examples of its use.

```
;; VECTOR-LENGTH -- built-in function
;; ____________________________________________________________
;; INPUT: VECKY, a vector
;; OUTPUT: The number of elements/slots in that vector

> (define vecky #(a b c d e))
> (vector-length vecky)
 5
> (vector-length (make-vector 25))
25
```

#### In-Class Problem 15.3.1: Fetching a random element from a vector

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

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

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

*Hint: Use the built-in `vector-length`, random and `vector-ref` functions.*
15.4 Recursively Processing Vectors

Since a vector is an ordered sequence of elements, we could define recursive functions to walk through vectors in much the same way that we defined recursive functions to walk through lists. However, we would quickly find ourselves writing the same sort of recursive code over and over again. To avoid this kind of repetition, which would quickly grow tiresome, we will instead use the `dotimes` special form which will enable us to automate the recursion needed to walk through any vector. Before introducing the `dotimes` special form, we first demonstrate how to manually write a recursive function to walk through some or all of a vector.

---

**Example 15.4.1: Manually walking through a vector**

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

```scheme
;; PRINT-VECTOR-FROM
;; ----------------------------------------------------
;; INPUTS: VECKY, a vector
;; I, a non-negative integer, no greater than
;; the length of VECKY
;; OUTPUT: None
;; SIDE EFFECT: Prints out the elements of VECKY
;; from index I onward.
(define print-vector-from
  (lambda (vecky i)
    (cond
      ;; Base Case: I is too big
      ;; Note: The last legal index of VECKY is LENGTH-1
      ((>= i (vector-length vecky)) (void))
      ;; Recursive Case: I is a legal index
      (else
       ;; Print one element
       (printf "Element \~A of VECKY is: \~A\%" i (vector-ref vecky i))
       ;; Let recursion print the rest of the elements
       (print-vector-from vecky (+ i 1))))))

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

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

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

```scheme
;; PRINT-VECTOR-WR
;; ----------------------------------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: None
```

---
;; SIDE EFFECT: Prints out the elements of VECKY

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

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

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

However, the dotimes special form, introduced below, simplifies the task for the programmer by automating this kind of recursion.

15.4.1 The dotimes Special Form

The dotimes special form has the following syntax:

\[
\text{(dotimes (var numExpr)}
  \text{expr}_1
  \text{expr}_2
  \ldots
  \text{expr}_k
\text{)}
\]

where:

* \( \text{var} \) is a symbol that will be the name of a counter variable in a local environment created by the dotimes;

* \( \text{numExpr} \) is any expression that evaluates to a non-negative integer, say, \( n \), that will specify the number of iterations to be performed by the dotimes; and

* The expressions, \( \text{expr}_1, \text{expr}_2, \ldots, \text{expr}_k \), are any \( k \) expressions that together constitute the body of the dotimes.

The semantics of the dotimes special form is as follows. First, a local environment is created containing a variable \( \text{var} \) whose value is initially set to zero. Next, the following steps are performed \( n \) times:

* The expressions, \( \text{expr}_1, \text{expr}_2, \ldots, \text{expr}_k \), are evaluated with respect to that new local environment.

* The value of \( \text{var} \) in the local environment is incremented by one.

Thus, the expressions in the body of the dotimes are evaluated \( n \) times, once for each value of \( \text{var} \) in the range, \( \{0, 1, 2, \ldots, n - 1\} \). Note that the expressions in the body may refer to the variable \( \text{var} \). When they are evaluated, the current value of \( \text{var} \) will be taken from the local environment.

Finally, the dotimes special form evaluates to \#<void>. Therefore, the usefulness of dotimes comes not from any output value, but from the side effects that occur by evaluating the expressions in its body with respect to the new local environment.
Example 15.4.2: Illustrating the `dotimes` special form

The following examples illustrate how the `dotimes` special form can be used.

```scheme
> (dotimes (i 5)
   ;; The body:
   (printf "i: ~A~%" i))
i: 0
i: 1
i: 2
i: 3
i: 4
> (dotimes (i (+ 1 2))
   ;; The body:
   (printf "~A + ~A = ~A~%" i i (+ i i)))
0 + 0 = 0
1 + 1 = 2
2 + 2 = 4
```

In these examples, the body consists of a single expression; however, that need not be the case in general. Notice that in the first example, the numerical expression 5 ensures that the expression in the body will be evaluated five times, once for each value of i in the range \{0, 1, 2, 3, 4\}. In the second example, the body of the `dotimes` is evaluated three times, since (+ 1 2) evaluates to 3.

Because the counter variable in a `dotimes` takes on the successive values, 0, 1, 2, ..., n - 1, the `dotimes` special form is tailor-made for walking through the elements of any given vector, as demonstrated below.

Example 15.4.3: Printing the contents of a vector using `dotimes`

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

```scheme
;;; PRINT-VECTOR
;;; --------------------------------------------------------------------------------------------------
;;; INPUT:   VECKY, a vector
;;; OUTPUT:  None
;;; SIDE EFFECT: Displays the contents of VECKY in the Interactions Window, one element per row

(define print-vector
  (lambda (vecky)
    ;; I takes on the values: 0, 1, 2, ..., LENGTH-1
    (dotimes (i (vector-length vecky))
      ;; Print out the Ith element of VECKY
      (printf "~A: ~A~%" i (vector-ref vecky i))))))

> (define vecky #(a b c d e))
> (print-vector vecky)
a
b
c```
Example 15.4.4: Printing the contents of a vector in reverse order

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

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

(define print-in-reverse
  (lambda (vecky)
    ;; LEN = number of elements in VECKY
    (let ((len (vector-length vecky)))
      ;; I : takes on the values from 0 to LEN-1
      ;; REV-I: takes on the values from LEN-1 to 0
      (dotimes (i len)
        ;; The BODY:
        (let ((rev-i (- len i 1)))
          (printf "vecky[\%d] = \%s\n" rev-i (vector-ref vecky rev-i)))))
    (void))))

> (print-in-reverse #(a b c d))
vecky[3] = d
vecky[2] = c
vecky[1] = b
vecky[0] = a
```

Example 15.4.5: Implementing our own version of $\text{dotimes}$

Since $\text{dotimes}$ is merely automating a version of recursion that we already know how to do, we can define a function, called $\text{my-dotimes}$, that does essentially the same thing as $\text{dotimes}$. The only difference is that our version is a little clunkier because it is a function, not a special form. First, we define a helper function that includes the input parameter $i$.

```scheme
;;; MY-DOTIMES-HELPER
;;; --------------------------------------------------------------
;;; INPUTS: I, a non-negative integer
;;; N, a non-negative integer
;;; FUNK, a function that can be applied to I
;;; OUTPUT: none
;;; SIDE EFFECT: Successively applies FUNK to the values
```

```scheme
(define my-dotimes-helper
  (lambda (i n funk)
    ;; The BODY:
    (let ((rev-i (- n i 1)))
      (funk (vector-ref vecky rev-i)))
    (void))))
```

```scheme
(my-dotimes-helper 0 3 my-dotimes-helper)
```

```scheme
vecky[3] = d
vecky[2] = c
vecky[1] = b
vecky[0] = a
```
 ;; I, I+1, I+2, ..., N-1

(define my-dotimes-helper
  (lambda (i n funk)
    (cond
    ;; Base Case: I >= N
    (>= i n)
    ;; Time to stop!
    (void))
    ;; Recursive Case: I < N
    (else
    ;; Apply FUNK to the current value of I
    (funk i)
    ;; Let recursion handle applying FUNK to all of
    ;; the rest of the values of I.
    (my-dotimes-helper (+ i 1) n funk))))

Next, we define my-dotimes as a wrapper function for my-dotimes-helper:

 ;; MY-DOTIMES
 ;; -----------------------------------------------
 ;; INPUTS: N, a non-negative integer
 ;; FUNK, a function that can be applied to any
 ;; non-negative integer from 0 to N-1
 ;; OUTPUT: none
 ;; SIDE EFFECT: Successively applies FUNK to the
 ;; non-negative integers from 0 to N-1.
 (define my-dotimes
  (lambda (n funk)
    ;; Call the helper function with I=0
    (my-dotimes-helper 0 n funk)))

Here are some parallel examples of dotimes and my-dotimes in action:

> (dotimes (i 3)
  (printf "i = ")
  (printf "%A\%" i))
i = 0
i = 1
i = 2
> (my-dotimes 3 (lambda (i)
  (printf "i = ")
  (printf "%A\%" i)))
i = 0
i = 1
i = 2
> (define vecky #(a b c))
> (dotimes (j (+ 1 2))
  (printf "vecky[\A] = " j)
  (printf "%A\%" (vector-ref vecky j)))
vecky[0] = a
vecky[1] = b
vecky[2] = c
> (my-dotimes-helper (+ 1 2)
  (lambda (j)
    (printf "vecky[\^A] = " j)
    (printf "\^A\%^" (vector-ref vecky j))))
vecky[0] = a
vecky[1] = b
vecky[2] = c

Notice that in these examples, a lambda function is used to do two things. First, its input parameter, which can be any symbol, identifies the counter variable for my-dotimes. Second, the body of the lambda function, which can contain any number of expressions, contains the expressions you would normally find in the body of a dotimes. In the first example, the counter variable is called i; in the second example, the counter variable is called j. In each example, for illustrative purposes only, the bodies contain two printf expressions. By now, it should be clear that my-dotimes effectively simulates the dotimes special form; however, in practice, it is much more convenient to use dotimes than my-dotimes!

In-Class Problem 15.4.1: Using dotimes to print out every other element of a vector

Define a function, called print-every-other-one-veck, that takes a vector as its only input. It should not return any output value. Instead, it should print out every other element of the given vector.

Here is the contract, followed by some examples:

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

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

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

In-Class Problem 15.4.2: Testing whether two vectors are “equal”

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

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

\[
\begin{align*}
> & \text{(vector-equal? #(a b c) #(a b c))} \\
& \text{#t} \\
> & \text{(vector-equal? (make-vector 3) (vector 0 0 0))} \\
& \text{#t} \\
> & \text{(vector-equal? #(a b c) #(a b c d))} \\
& \text{#f}
\end{align*}
\]

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

\[
\begin{align*}
;; & \text{VECTOR-EQUAL?-HELPER} \\
;; & \text{--------------------------------------------------------------} \\
;; & \text{INPUTS: VECK-ONE, VECK-TWO, two vectors of the same length} \\
;; & \text{I, an index} \\
;; & \text{OUTPUT: #t if the corresponding elements of VECK-ONE and} \\
;; & \text{VECK-TWO are the same from index I onward.} \\
;; & \text{#f otherwise.}
\end{align*}
\]

Notice that the helper function is only ever called on vectors having the same length.

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

\[
\begin{align*}
> & \text{(equal? #(1 2 3) #(1 2 3))} \\
& \text{#t} \\
> & \text{(equal? #(a b c) (vector ’a ’b ’c))} \\
& \text{#t} \\
> & \text{(equal? #(0 0 0) (make-vector 4))} \\
& \text{#t}
\end{align*}
\]

15.5 Destructively Modifying a Vector

The vector-set! function is provided to enable a programmer to destructively modify the contents of a specified slot in a vector.

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

Example 15.5.1: The vector-set! function

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

\[
\begin{align*}
;; & \text{VECTOR-SET! -- Built-in Function} \\
;; & \text{--------------------------------------------------------------} \\
;; & \text{INPUTS: VECKY, a vector} \\
;; & \text{INDY, a numerical index} \\
;; & \text{NEW-VAL, anything}
\end{align*}
\]
;; OUTPUT: *void*
;; SIDE EFFECT: Destructively changes the contents of VECKY
;; at slot INDY to become NEW-VAL

> (define vecko (vector 0 10 20 30))
> vecko
#(0 10 20 30)
> (vector-set! vecko 2 'x)
> vecko
#(0 10 x 30)

In-Class Problem 15.5.1: Initializing a vector

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

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

Here is an example of its use:

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

Hint: Use dotimes and vector-set!.

In-Class Problem 15.5.2: Swapping elements of a vector

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

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

Here are some examples of its use:

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

Problems

Problem 15.1

Define a (destructive) function that satisfies the following contract.

;; DOUBLE-ALL!
;; -------------------------------
;; INPUT: VECKY, a vector of numbers
;; OUTPUT: The same vector, modified as described below
;; SIDE EFFECT: Doubles the contents of each slot (destructively)

> (define vecky (vector 10 20 30 40 50))
> vecky
#(10 20 30 40 50)
> (double-all! vecky)
#(20 40 60 80 100)
> vecky
#(20 40 60 80 100)

Problem 15.2

Define a (destructive) function, called roll-em!, that satisfies the following contract:

;; ROLL-EM!
;; -------------------------------
;; INPUT: DICE, a vector of numbers
;; OUTPUT: The same vector, but with contents destructively modified, as follows
;; SIDE EFFECT: Replaces each slot with a random toss of a six-sided die

> (define dice (make-vector 5))
> (roll-em! dice)
#(3 2 5 3 4)
> dice
#(3 2 5 3 4)
> (roll-em! dice)
#(1 6 1 5 5)
> dice
#(1 6 1 5 5)

Hint: Use the toss-die function from the previous chapter.
Problem 15.3

Define a (destructive) function called roll-some! that satisfies the following contract:

;; ROLL-SOME!
;; --------------------------------
;; INPUTS: DICE-VECK, a vector of dice values
;; ROLLER, a vector of the same length as DICE-VECK,
;; but consisting solely of 1s and 0s
;; OUTPUT: DICE-VECK, modified as described below
;; SIDE EFFECT: Walks through the two input vectors in parallel.
;; For each index I, if the Ith element of ROLLER is a 1, then
;; the Ith element of DICE-VECK is replaced by a random toss
;; of a 6-sided die; otherwise, it is unchanged.

Here are some examples:

> my-dice
#(1 2 1 6 6 3 2 3)
> (roll-some! my-dice #(1 1 1 0 0 1 1 1))
#(5 4 2 6 6 3 4)
> (roll-some! my-dice #(1 1 1 0 0 0 1 1))
#(1 6 3 6 6 6 2 2)
> (roll-some! my-dice #(1 0 1 0 0 0 1 1))
#(4 6 5 6 6 1 6)
> my-dice
#(4 6 5 6 6 1 6)

Problem 15.4

Define a non-destructive function, called vector-reverse, that satisfies the following contract. Because it is non-destructive, it must create a new vector, instead of modifying the given vector.

;; VECTOR-REVERSE
;; --------------------------------
;; INPUT: VECK, a vector
;; OUTPUT: A *new* vector that is just like VECK, except
;; that its elements are in the reverse order.
;; SIDE EFFECTS: none

Here are some examples:

> (vector-reverse #(a b c d))
#(d c b a)
> (vector-reverse #(1 2 3))
#(3 2 1)

Notice that the inputs in the above examples are immutable! So, if the function had tried to modify them, it would have caused an error!

Hints: Create a new vector of the appropriate length. Then use dotimes to walk thru the vector, setting its elements to appropriate values. Recall the print-in-reverse function from Example 15.4.4 for ideas. Don’t forget to return the new vector as output.
**Problem 15.5**

Define a (destructive) function that satisfies the following contract:

```scheme
;; VECTOR-REVERSE!
;; ---------------------------------------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: The same vector, modified as described below
;; SIDE EFFECT: Destructively reverses the order of the
;;     elements in VECKY.

> (vector-reverse! (vector 1 2 3 4 5))
#(5 4 3 2 1)
> (define vecky (vector 'a 'b 'c 'd))
> vecky
#(a b c d)
> (vector-reverse! vecky)
#(d c b a)
> vecky
#(d c b a)
```

**Problem 15.6**

Define a non-destructive function that satisfies the following contract:

```scheme
;; VECTOR-MAP
;; -----------------------------------------------------------
;; INPUTS: FUNK, a function that expects one input
;;         VEKK, a vector of suitable inputs for FUNK
;; OUTPUT: A *new* vector of the same length as VEQUE
;;         each of whose elements is obtained by applying FUNK to
;;         the corresponding element of VEKK.
;; SIDE EFFECT: *NONE*

Here is an example that assumes that the facty function has already been defined.

```
> (define vek #(1 2 3 4 5)) ←─ #(1 2 3 4 5) is an immutable vector
> (vector-map facty vek)
#(1 2 6 24 120) ←─ New vector created by vector-map
> vek
#(1 2 3 4 5) ←─ vek hasn’t changed
```

Notice that vek has been defined to be an immutable vector (i.e., its contents can’t be changed) and, thus, even if vector-map wanted to change its contents, it could not. (Attempting to do so would cause an error.) After vector-map is finished, vek remains the same (i.e., vector-map is non-destructive).

Hints: Use make-vector to create a new vector of the same length as vek. Then use dotimes to walk thru the new vector, setting each element to the result obtained by applying funk to the corresponding element of vek.
### Problem 15.7

**Define a destructive function that satisfies the following contract:**

```
;; VECTOR-MAP!
;; -------------------------------
;; INPUTS:  FUNK, a function that expects one input
;; VEQUE, a vector of suitable inputs for FUNK
;; OUTPUT: VEQUE, destructively modified...
;; SIDE EFFECT: Destructively modifies VEQUE by replacing
;; each of its elements by the result of applying FUNK to
;; the corresponding element of VEQUE.
```

**Here is an example:**

```
> (define vec (vector 1 2 3 4 5))  ← VEC is a mutable vector
> (vector-map! facty vec)
#(1 2 6 24 120)  ← VECTOR-MAP! does its thing
> vec
#(1 2 6 24 120)  ← VEC has been changed!
```

Recall that the built-in `vector` function creates a mutable vector (i.e., one whose contents can be changed). `vector-map!` walks through the input vector, destructively modifying its contents. Afterward, `vec` is shown to be modified. Thus, `vector-map!` is destructive!

*Hints: No need to create a new vector. Just use `do` to walk through the given vector, destructively modifying its contents as you go.*

### Problem 15.8

**Define a destructive function that satisfies the following contract:**

```
;; VECK-REPLACE!
;; -------------------------------
;; INPUTS:  OLD, anything
;; NEW, anything
;; VECKY, a vector
;; OUTPUT: VECKY, destructively modified as follows.
;; SIDE EFFECT: Replaces each occurrence of OLD in VECKY
;; by NEW, where equality is as judged by EQ?.
```

**Here is an example:**

```
> (define vequi (vector 1 2 1 3 1 4))  ← VEQUI is mutable
> vequi
#(1 2 1 3 1 4)
> (veck-replace! 1 111 vequi)
#(#(111 2 111 3 111 4)
> vequi
#(#(111 2 111 3 111 4)  ← VEQUI has changed!
```
Problem 15.9

Define a non-destructive function, called `veck-index-of`, that satisfies the following contract:

```
;; VECK-INDEX-OF
;; --------------------------------------------------------------
;; INPUTS: ITEM, anything
;;    VECKY, a vector
;; OUTPUT: The index of the first slot of VECKY that contains
;; an first occurrence of ITEM; or #f if ITEM does
;; not appear in VECKY.
```

Here are some examples of its use:

```
> (veck-index-of 'x #(a b c x y z x x))
3
> (veck-index-of 'z #(a b c x y z x x))
5
> (veck-index-of 'w #(a b c x y z x x))
#f
```

Hint: Define a helper function, called `veck-index-of-helper`, that includes an extra input I, that
serves as an index into the given vector. Make recursive function calls, incrementing I, until you find an
occurrence of ITEM, or I gets too big. Here’s the contract for the helper function:

```
;; INDEX-OF-HELPER
;; -------------------------------
;; INPUTS: ITEM, anything
;;    VECKY, a vector
;;    I, a numerical index
;; OUTPUT: The index of the slot of VECKY that contains the
;; first occurrence of ITEM at or after index I;
;; or #f if ITEM does not appear in VECKY.
```

Problem 15.10: Converting a vector into a list

The goal of this problem is to define a function, called `veck-to-list`, that takes a vector as its only
input, and returns as its output a list containing the same elements, in the same order, as illustrated below.

```
> (veck-to-list #(a b c d))
(a b c d)
> (veck-to-list (make-vector 5))
(0 0 0 0 0)
> (veck-to-list (vector 1 #t ()))
(1 #t ())
```

Here is the contract for `veck-to-list`:

```
;; VECK-TO-LIST
;; -----------------------------------------------
;; INPUT: VECK, a vector
;; OUTPUT: A list having the same elements as VECK,
;; and in the same order.
```
Given the tools that we have seen so far, for this problem, it is probably easiest to define a recursive helper function, called veck-to-list-helper, that includes an extra input I that identifies the current element of the vector. Here is the contract:

;; VECK-TO-LIST-HELPER
;; ---------------------
;; INPUTS: VECK, a vector
;; I, an index
;; OUTPUT: A list containing the elements of VECK from
;; the index I onward

Here are some examples:

> (veck-to-list-helper #(a b c d e) 2)
(c d e)
> (veck-to-list-helper #(a b c d e) 3)
(d e)

Hints:

⋆ When the index I reaches the length of the vector, then it is no longer a valid index for that vector, indicating that you have reached the base case of the recursion.
⋆ Remember, you are returning a list in the base case, and a list in the recursive case!
⋆ For this problem, it is probably easier not to use an accumulator.

Incidentally, after you have implemented this function, you can compare it to the built-in function, vector->list, that does the same thing!

Problem 15.11: Converting a list into a vector

The goal of this problem is to define a function, called list-to-veck, that takes a list as its only input, and returns as its output a vector containing the same elements, in the same order, as illustrated below.

> (list-to-veck '(a b c d))
#(a b c d)
> (list-to-veck (list 1 #t ()))
#(1 #t ())

Here is the contract for list-to-veck:

;; LIST-TO-VECK
;; ------------------
;; INPUT: LISTY, a list
;; OUTPUT: A vector containing the same elements as LISTY, and in the same order
;; SIDE EFFECTS: none

Hints:

⋆ You can call the built-in length function once to find out how long the list is—but don’t call it more than once! Use it to create a vector of the same length. Then, you might think about using dotimes to walk thru the vector, but that would be inefficient if you are planning to use
the `fetch-nth-element` or `list-ref` functions (cf. In-class problem 14.2.3) to access the successive elements of the list. Instead, define a recursive helper function that satisfies the following contract. It will use list-based recursion to walk thru the list. On each recursive function call, the first element of the list will be the one you want to give to the `vector-set!` function. This approach will be much more efficient.

;;; LIST-TO-VECK-HELPER
;;; -----------------------------------------------------------
;;; INPUTS: LISTY, a list
;;; VECK, a vector
;;; I, an index (non-negative integer)
;;; OUTPUT: The vector, VECK, modified as described below
;;; SIDE EFFECTS: When called with I=0, copies the contents
;;; of LISTY into VECK. When called with I>0, copies the
;;; contents of LISTY into the slots of VECK, starting at
;;; index I.

Here are some examples:

> (list-to-veck-helper '(a b c d e) (make-vector 5) 0)
#(a b c d e)
> (list-to-veck-helper '(c d e) (make-vector 5) 2)
#(0 0 c d e)

Incidentally, after you have finished defining these functions, you may want to compare them to the built-in `list->vector` function that does the same thing!

Problem 15.12

Define a non-destructive function, called `every-other-one-vector`, that satisfies the following contract.

;;; EVERY-OTHER-ONE-VECTOR
;;; -----------------------------------------------------------
;;; INPUT: VECKY, a vector
;;; OUTPUT: A vector containing every other element of VECKY.
;;; Note: The output vector should contain roughly half the
;;; elements of VECKY.

Here are some examples of its behavior:

> (every-other-one #(0 1 2 3 4 5 6))
#(0 2 4 6)
> (every-other-one #(1 2 3 4 5 6))
#(1 3 5)

Hints:

* Create a new vector whose length is roughly half that of the input vector. Then use `dotimes` to walk through that new vector, copying relevant elements from the input vector to the new vector.

* You may find the built-in `even?`, `odd?`, or `quotient` functions helpful. (`even?` and `odd?` were introduced in Problem 12.3; `quotient` was introduced in Section 5.3.)
You should not use lists for this function; use vectors!

**Problem 15.13: Computing a histogram**

Define a function, called `compute-histogram`, that satisfies the following contract:

```
;;  COMPUTE-HISTOGRAM
;;  -------------------------------
;;  INPUT:  VECK-O-DICE, a vector of dice values (each 1 thru 6)
;;  OUTPUT: A vector of length 7, where the ith slot contains
;;          the number of occurrences of i in VECK-O-DICE.
;;          (The 0th slot of the output vector is ignored.)
```

Here's an example:

```
> (define my-dice #(1 2 1 2 6 6 6 5 2))
> (compute-histogram my-dice)
#(0 2 3 0 0 1 3)
```

In this case, `my-dice` contains:

- two 1s
- three 2s
- no 3s
- no 4s
- one 5
- three 6s

These counts are reflected in the histogram computed by this function. Note that the histogram is a vector with seven slots, numbered 0 thru 6. The zeroeth slot is ignored. We only care about slots 1 thru 6. For each \( i > 0 \), the slot of the output vector at index \( i \) contains the number of occurrences of \( i \) in `my-dice` (or `veck-o-dice`). Here's another example:

```
> (define my-dice #(3 3 3 3 3 1 1 1))
> (compute-histogram my-dice)
#(0 4 0 5 0 0 0)
```

**Hint:** Create a vector of length seven called `histy`, then use `doitimes` to walk through the `veck-o-dice` (not `histy`). For each index \( i \), look at the \( i \)th slot of `veck-o-dice` and use its value to figure out which slot of `histy` to increment. In effect, `histy` is a vector with seven accumulators, one of which we are ignoring.

**Problem 15.14**

You may wish to review Problem 14.6 before starting this problem.

Define a function, called `veck-has-satisfier?`, that satisfies the following contract:

```
;;  VECK-HAS-SATISIFIER?
;;  -----------------------------------------------
;;  INPUTS:  FUNK, a predicate that expects one input
;;           VECK, a vector of suitable inputs for FUNK
```
Here are some examples:

> (veck-has-satisfier? number? #(a #t 3 x))
#t
> (veck-has-satisfier? symbol? #(1 2 3 4))
#f

Note that it would be inefficient to implement this function using `dotimes` because `dotimes` invariably walks through the entire vector. This function should stop as soon as it finds an element of `veck` that satisfies `funk`. Therefore, you should define a recursive helper function that takes an extra input, `i`, that serves as an index into the vector `veck`.

**Problem 15.15**

Define a function, called `has-three-of-a-kind?`, that satisfies the following contract:

```
;; HAS-THREE-OF-A-KIND?
;; -----------------------------------------------
;; INPUT: VECK-O-DICE
;; OUTPUT: #t if the vector of dice contains *at least* three of one kind; #f otherwise.
```

Here are some examples:

> (has-three-of-a-kind? #(1 2 1 2 1)) ← has three ones
#t
> (has-three-of-a-kind? #(4 2 4 4 4)) ← has four fours
#t
> (has-three-of-a-kind? #(6 6 6 6 6)) ← has five sixes
#t
> (has-three-of-a-kind? #(6 5 6 5 2)) ← does not have three of a kind
#f

Notice that having four or five of a kind also counts as having three of a kind.

Hint: One way to solve this problem: Compute a histogram vector, then check whether it has an element that is 3 or bigger. Can you think of a way to use `veck-has-satisfier?` from Problem 15.14 to check whether the histogram vector contains an element that is 3 or bigger?

**Problem 15.16**

Define a function, called `has-large-straight?`, that satisfies the following contract:

```
;; HAS-LARGE-STRAIGHT?
;; -----------------------------------------------
;; INPUT: VECK-O-DICE, a vector of five dice values
;; OUTPUT: #t if the vector of dice contains all of the numbers in {1,2,3,4,5} or {2,3,4,5,6}, in any order;
```
Here are some examples:

```scheme
> (has-large-straight? #(2 4 5 3 1))
#t
> (has-large-straight? #(6 5 4 2 3))
#t
> (has-large-straight? #(6 4 1 2 3))
#f
```

Hints: You may assume that the input vector has exactly five slots. Compute a histogram and go from there! What must the histogram look like for a large straight? (Use the built-in `equal?` predicate (cf. In-class problem 15.4.2) to make your life easier!) Alternatively, convert the vector of dice into a list, then use the built-in `sort` function (cf. Example 14.5.6) to sort its contents into non-decreasing order. What must it look like at that point?

Special Forms Introduced in this Chapter

- `dotimes` Automates a certain kind of numerical recursion

Built-In Functions Introduced in this Chapter

- `make-vector` Construct a vector of a specified length
- `vector` Construct a vector containing the specified items
- `vector-length` Fetches the length of a given vector
- `vector-ref` Fetches a specified element of a given vector
- `vector->list` Convert vector into a list (cf. Problem 15.10)
- `list->vector` Convert list into a vector (cf. Problem 15.11)
Chapter 16

Data Structures

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

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

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

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

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

This chapter focuses on field-based data structures.

Example 16.0.1: Motivating field-based data structures, I

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

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

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

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

> (define my-birthday (vector ’i-am-a-date! 1958 10 22))

¹ Although a vector is primarily an index-based data structure, it also contains a field, called length, whose value is accessed by the vector-length function.
Notice that to access a particular item requires knowing which index to use (e.g., 2 for month, and 1 for year).

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

---

**Example 16.0.2: Motivating field-based data structures, II**

*First, we define some useful constants.*

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

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

;; *MONTHS* -- A vector of names for the months
;; -----------------------------------------------------
;; To enable using the standard numbers for months,
;; we use a vector of length 13, ignoring slot 0.
;; And, since the names of the months won’t change,
;; we can use an *immutable* vector.

(define *months* 
  #(0 Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec))

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

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

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

*Although it may not be used all the much, here’s a type-checker predicate for a date-as-vector.*

;; DATE? -- type-checker predicate for dates
;; INPUT: DATUM, anything
;; OUTPUT: #t, if DATUM is a "date-as-vector"

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

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

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

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

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

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

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

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

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

;; SET-DATE-YEAR! -- Mutator Function
;; -----------------------------------------------
;; INPUTS: DATEY, a "date-as-vector"
;;         NEW-VAL
;; OUTPUT: Don’t care
;; SIDE EFFECT: Destructively modifies the contents of
;;              the YEAR slot to be NEW-VAL.
The following interactions demonstrate the use of dates-as-vectors to store date information:

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

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

A define-struct special form has the following syntax:

```
(define-struct structName (fname1 fname2 ... fnamek))
```

where:
- `structName` is a symbol that will be the name of the data structure; and
- `fname1`, `fname2`, …, `fnamek` are `k` symbols that will be the names of the fields of the new data structure.

The semantics of the define-struct special form stipulates that its evaluation generates no output, but has the side effect of defining all of the following functions:
- a constructor function, named `make-structName`;
- a type-checker predicated, named `structName?`;
- `k` accessor functions, named `structName-fname1`, …, `structName-fnamek`; and
- `k` mutator functions, named `set-structName-fname1!`, …, `set-structName-fnamek!`.

Example 16.1.1: Using define-struct to define a date data structure

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

```
(define-struct date (year month day))
```

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

The following interactions demonstrate their use:

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

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

Here are some examples of its use:

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

**Example 16.1.2: Implementing a deck of cards**

Below, we define a deck data structure that has two fields: cards and num-left. The cards field will be a vector that is partitioned into two sections, as illustrated in Fig. 16.1. The first section will contain all of the cards that have not yet been dealt; the second section will contain those that have already been
dealt. Thus, the second section of the vector is effectively a “discard pile”. The \texttt{num-left} field serves two purposes. First, it keeps track of the number of cards that have not yet been dealt (i.e., that remain available for dealing). Second, it corresponds to the index at the top of the discard pile. For example, as shown in the figure, suppose that there are ten cards in the deck and \texttt{num-left} is 6. This means that there are six cards that have not yet been dealt. Those cards are located in the first part of the \texttt{cards} vector, at positions 0, 1, \ldots, 5. The “discard pile” begins at index 6. The four cards in the discard pile are at positions 6, 7, 8 and 9. In general, the cards available for dealing have indices between 0 and \((\texttt{num-left} - 1)\), while the discard pile starts at index \texttt{num-left}.

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

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

(define-struct deck (cards num-left))
\end{verbatim}

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

\begin{verbatim}
;; PRINT-DECK
;; -----------------------------------------------
;; INPUT: DECKY, a DECK data structure
;; OUTPUT: None
;; SIDE EFFECT: Displays contents of DECKY

(define print-deck
  (lambda (decky)
    (printf "Deck of cards: °A, num-left: °A°\n"
        (deck-cards decky) (deck-num-left decky))))
\end{verbatim}

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

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

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

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

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

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

> (define decky
  (make-deck (vector 'a 'b 'c 'd 'e 'f 'g 'h 'i 'j) 10))
> (print-deck decky)
Deck of cards: #(a b c d e f g h i j), num-left: 10
> (deal! decky)
d
> (print-deck decky)
Deck of cards: #(a b c j e f g h i d), num-left: 9
> (deal! decky)
e
We conclude this chapter by introducing two built-in functions. The first function, called \texttt{format}, works just like \texttt{printf}, except that instead of printing stuff out as a side effect, it generates a Scheme string as its output.

\begin{example}
We run the same test as in Example 16.1.2.
\begin{verbatim}
> (format "\~A + \~A = \~A\~%" 2 3 (+ 2 3))
"2 + 3 = 5"
> (printf (format "\~A + \~A = \~A\~%" 2 3 (+ 2 3))) 2 + 3 = 5
> (printf "\~A + \~A = \~A\~%" 2 3 (+ 2 3)) 2 + 3 = 5
> (format "\~A\~%  \~A\~%  \~A\~%" 1 2 3)
"1\n 2\n 3"
> (printf (format "\~A\~%  \~A\~%  \~A\~%" 1 2 3)) 1
 2
 3
> (printf "\~A\~%  \~A\~%  \~A\~%" 1 2 3) 1
 2
 3
\end{verbatim}
\end{example}

The next function is called \texttt{string-append}. It is analogous to the built-in \texttt{append} function, except that it appends strings instead of lists.
Example 16.1.4: The built-in `string-append` function

*The contract for the `string-append` function is given below, followed by some examples of its use.*

```scheme
;; STRING-APPEND -- built-in function
;; ------------------------------------------------------
;; INPUT: Any number of strings
;; OUTPUT: A single string formed by concatenating all
;; of the input strings.

> (string-append "String one!" "String two!"")
"String one!String two!"
> (printf (string-append "String one!\n" "String two!\n"))
String one!
String two!
```

In later chapters, when we implement games, it will be useful to separate the tasks of (1) generating the output string as a Scheme datum, and (2) printing it out as a side effect. Although for a text-based interface it may suffice to print out the string in the Interactions Window, for a graphical interface, we may want to display the string in a graphical window, which `printf` cannot do.

**Special Forms Introduced in this Chapter**

- `define-struct` For specifying new data structures

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

- `format` Like `printf`, but generates a string as output
- `string-append` For concatenating strings together
Chapter 17

The Model-View-Controller Paradigm

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

- the Model consists of the data structures that contain all of the information needed to keep track of the game;
- the Controller consists of the functions that are used to play the game (e.g., to select moves or take action); and
- the View consists of functions for displaying the current state of the game to the player(s).

The relative independence comes from the following:

- the Model typically contains no functions, or just some basic wrapper functions that facilitate the use of the data structures it defines;
- the Controller functions can access or modify the contents of the data structures defined in the Model; and
- the View functions only access the contents of the data structures in the Model; they do not modify them.

An important responsibility of the model is to keep track of the current state of the game (e.g., whose turn it is, how many rolls of the dice the current player has, which slots of the scoresheet are already filled in, and so on). Although the controller functions can access and modify the contents of the data structures in the model, they do not get involved with displaying any information to the player(s). Similarly, the view functions only show the player whatever information is relevant for the current state of the game; they do not get involved in taking action (e.g., rolling dice or choosing cards).

17.1 Implementing a Simple Game using the MVC Paradigm

This section presents a lengthy example that implements a simple dice-throwing game, called dice-dice, as an illustration of the use of the Model-View-Controller paradigm.

<table>
<thead>
<tr>
<th>Example 17.1.1: Implementing the dice-dice game using the MVC paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>The rules of the dice-dice game are as follows. It is a one-player game. The player gets five turns. On each turn, the player seeks to accumulate points by repeatedly throwing a pair of dice. The player can throw the dice as many times as he or she wants; however, if a toss ever comes up doubles (e.g., a pair of threes), the turn is immediately ended, with the player receiving a score of zero for that turn. The overall score is the sum of the points accumulated across all five turns. The goal is to achieve a high overall score.</em></td>
</tr>
</tbody>
</table>
The flow of the game can be represented in many ways. We will use the diagram in Fig. 17.1 as a guide. In the figure, the boxes with rounded corners represent the different states that the game could be in. The three states are:

- **ready-to-play**: It is time for the player to either roll the dice or score the current total.
- **waiting**: Waiting for the player to begin the next turn.
- **game-over**: The game is over!

The game starts in the **ready-to-play** state, which is indicated by the > symbol pointing at that rounded box.

The arrows in the figure represent transitions from one state to another. A transition is typically triggered by some action of the player, but the particular transition may also depend on the outcome of the action (e.g., whether the player happened to roll doubles). Actions are implemented as Scheme functions (e.g., roll! to roll the dice, save! to save the total for the current turn, continue! to start the next turn, and reset! to start a new game). For example, if the game is in the **ready-to-play** state, then the player can choose either to roll the dice or save the current score. If the player chooses to roll the dice and anything other than doubles turn up, then the total of that pair of dice is accumulated for the current turn, and the state remains **ready-to-play**. However, if doubles turn up, then the current turn is ended with a score of zero, and the state changes to **waiting**. When the state is **ready-to-play**, if the player decides to save the points accumulated so far for the current turn, then the current turn will end, and a new turn will begin, with the state remaining **ready-to-play**. From the **waiting** state, the player’s only option is to do the **continue!** action. If there are no more turns left, then the state changes to **game-over**; otherwise, it changes to **ready-to-play**. From the **game-over** state, the only option is to start a new game.

Now that the flow of the game has been described, it is time to talk about the model, the view, and the controller:

- **The model consists primarily of a dice-dice data structure** that contains the information needed to keep track of the game; however, it is also convenient to define some global constants and to provide a wrapper function for make-dice-dice that facilitates creating a new game.

- **The view consists of non-destructive** functions that can be used to display the current contents of the dice-dice data structure in a way that is useful for the player.
The controller consists of destructive functions that implement the different actions available to the player (e.g., roll!, save!, continue! and reset!).

We begin with the model. Notice that the definition of the dice-dice data structure is introduced by a block of comments explaining what each of the fields in the data structure is for.

;;; DICE-DICE struct
;;; ----------------------------------
;;; STATE: a symbol: READY-TO-PLAY, WAITING, or GAME-OVER
;;; CURR-TURN: an integer from 0 to 4 (the "current turn")
;;; LAST-TOSS: a pair (T1 T2) showing last toss of the dice;
;;; or #f if at the start of a new turn
;;; CURR-TOTAL: the points accumulated so far for the current turn
;;; SCORES: a vector of saved totals (5 entries, one per turn)
;;; GRAND TOTAL: the sum of all totals saved so far

(define-struct dice-dice
  (state curr-turn last-toss curr-total scores grand-total))

;;; Global constants for the different states of the game

(define *ready-to-play* 'ready-to-play)
(define *waiting* 'waiting)
(define *game-over* 'game-over)

;;; NEW-GAME -- wrapper for MAKE-DICE-DICE
;;; ----------------------------------------
;;; INPUT: none
;;; OUTPUT: A new DICE-DICE struct, ready for a new game

(define new-game
  (lambda ()
    (make-dice-dice *ready-to-play* ;; state
                    0 ;; curr-turn
                    #f ;; last toss
                    0 ;; curr total
                    (make-vector 5) ;; scores
                    0 ;; grand total )))

The view functions are next. To facilitate later conversion to a graphical interface, the task of generating the output string is handled by dd->string, while the task of printing out that string to the Interactions Window is handled by a wrapper function called show-game. Notice that the information contained in the output string depends on which state the game is in. (See the cond cases in dd->string.) Not only does the output string report on information about the current state of the game, but it also displays the actions that are now available to the player. The string-append function is used only for convenience, to avoid having a single format expression that is too long to fit on one line.

;;; DD->STRING (i.e., DICE-DICE->STRING)
;;; --------------------------------------
;;; INPUT: DD, a DICE-DICE struct
;;; OUTPUT: A string containing info about the state of the game
;;; SIDE EFFECT: none
(define dd->string
  (lambda (dd)
    ;; Some local variables, just for convenience
    (let ((st (dice-dice-state dd))
         (last-toss (dice-dice-last-toss dd))
         (turn (dice-dice-curr-turn dd))
         (scores (dice-dice-scores dd))
         (grand-total (dice-dice-grand-total dd))
         (curr-total (dice-dice-curr-total dd)))
      (cond
       ;; Case 1: Starting new turn
       ;; ---------------------------------------------
       ;; Haven’t tossed any dice yet
       ;; Generate a string that says we are starting a new
       ;; turn, and that displays the scores saved so far.
       ;; Generate a string that says we are starting a new
       ;; turn, and that displays the scores saved so far.
       ;; ---------------------------------------------
       (and (eq? st *ready-to-play*)
            (not last-toss)
            ;; Generate a string that says we are starting a new
            ;; turn, and that displays the scores saved so far.
            (string-append
             (format "Start turn ~A. Scores: ~A. Grand total: ~A\n" turn scores grand-total)
             (format "==> ROLL! or SAVE!\n")))
       ;; ---------------------------------------------
       ;; Case 2: Turn already in progress
       ;; ---------------------------------------------
       (eq? st *ready-to-play*)
       ;; Generate a string that says we are continuing the
       ;; current turn, shows the most recent toss and
       ;; the current total so far accumulated.
       (string-append
        (format "Turn ~A. Last Toss: ~A, Curr Total: ~A\n" turn last-toss curr-total)
        (format "==> ROLL! or SAVE!\n")))
       ;; ---------------------------------------------
       ;; Case 3: Whoops! We must’ve zeroed out!
       ;; ---------------------------------------------
       (eq? st *waiting*)
       ;; Generate a string that explains the (sad) situation.
       (string-append
        (format "Sorry! You zeroed out! Last toss: ~A.\n" last-toss)
        (format "==> CONTINUE!\n")))
       ;; ---------------------------------------------
       ;; Case 4: Game over
       (else
        (format "Game over! Scores: ~A. Grand total: ~A\n" scores grand-total)))))

;; SHOW-GAME -- view function for text-based interface
;; ---------------------------------------------
;; INPUT: DD, a DICE-DICE struct
Finally, the controller functions are listed below.

;; RESET!
;; ---------------------------------------------
;; INPUT: DD, a DICE-DICE struct
;; OUTPUT: DD, destructively modified
;; SIDE EFFECT: Resets contents of DD to a fresh game

(define reset!
  (lambda (dd)
    ;; Reset contents of DD in preparation for a new game
    (set-dice-dice-state! dd *ready-to-play*)
    (set-dice-dice-curr-turn! dd 0)
    (set-dice-dice-last-toss! dd #f)
    (set-dice-dice-curr-total! dd 0)
    (set-dice-dice-scores! dd (make-vector 5))
    (set-dice-dice-grand-total! dd 0)
    ;; Return the modified struct as output
    dd))

;; ROLL!
;; ---------------------------------------------
;; INPUT: DD, a DICE-DICE struct
;; OUTPUT: none
;; SIDE EFFECT: If appropriate, rolls the dice and updates game

(define roll!
  (lambda (dd)
    (let* ((st (dice-dice-state dd)))
      (cond
        ;; Good Case: It’s time to roll
        ;; ---------------------------------------------
        ;; Case 1: Doubles!
        ;; ---------------------------------------------
        ((eq? toss-one toss-two)
          ;; CURR-TOTAL goes to zero
          (set-dice-dice-last-toss! dd (list toss-one toss-two))
          (cond
            ;; Case 1: Doubles!
            ;; ---------------------------------------------
            ((eq? toss-one toss-two)
              ;; CURR-TOTAL goes to zero
              )))
      )))
(set-dice-dice-curr-total! dd 0)
;; Set state to WAITING so player can see results
(set-dice-dice-state! dd *waiting*)
;; ------------------------------
;; Case 2: Not doubles!
;; ------------------------------
(else
 ;; Update CURR-TOTAL
 (set-dice-dice-curr-total! dd (+ (dice-dice-curr-total dd)
 toss-one
 toss-two))
 ;; No change to state
)))))
;; ------------------------------
;; Bad Cases
;; ------------------------------
(else
 ;; Print out an error message
 (printf "Sorry! Can’t roll now!"
)))))
;; SAVE!
;; -----------------------------------------------
;; INPUT: DD, a DICE-DICE struct
;; OUTPUT: None
;; SIDE EFFECT: If appropriate, saves curr-total to score vector
;; and prepares for the next turn (or the end of the game)
(define save!
 (lambda (dd)
 ;; For convenience, define some local variables
 (let ((st (dice-dice-state dd))
 (vecky (dice-dice-scores dd))
 (indy (dice-dice-curr-turn dd))
 (cond
 ;; Good case: Saving is allowed
 ;; -----------------------------------------------
 ((eq? st *ready-to-play*)
 ;; Store the score in the vector
 (vector-set! vecky indy (dice-dice-curr-total dd))
 ;; Update the grand total
 (set-dice-dice-grand-total! dd
 (+ (dice-dice-curr-total dd)
 (dice-dice-grand-total dd))))
 (cond
 ;; -----------------------------------------------
 ;; Case 1: That was the last turn
 ;; -----------------------------------------------
 ((= indy 4)
 ;; The game is OVER!
 (set-dice-dice-state! dd *game-over*)))
;; CONTINUE!
;; INPUT: DD, a DICE-DICE struct
;; OUTPUT: None
;; SIDE EFFECT: When appropriate, prepares for the
;; next turn (or game over)
(define continue!
  (lambda (dd)
    ;; For convenience, create some local variables
    (let ((st (dice-dice-state dd))
          (indy (dice-dice-curr-turn dd))
          (vecky (dice-dice-scores dd)))
      (cond
        ;; Good Case: Waiting to go to next turn (or game over)
        ;; ------------------------------------------------------
        ((eq? st *waiting*)
         (cond
          ;; Case 1: That was the last turn
          ;; ---------------------------------------
          ((= indy 4)
           (set-dice-dice-state! dd *game-over*)
           (set-dice-dice-curr-total! dd 0))
          ;; Case 2: That was not the last turn
          ;; ---------------------------------------
          (else
           (set-dice-dice-state! dd *ready-to-play*)
           (set-dice-dice-curr-turn! dd (+ indy 1))
           (set-dice-dice-last-toss! dd #f)
           (set-dice-dice-curr-total! dd 0)))))
    ;; No need to update grand total :
    ;; )))))))))

Now that the model, view and controller modules have been defined, we can provide a few wrapper functions that are tailored to the text-based version of the game. Each simply calls the corresponding controller
function, followed by the show-game view function to display the results.

;; ROLL-WR!, SAVE-WR!, CONTINUE-WR!, RESET-WR!
;; ---------------------------------------------------------------------
;; INPUT: DD, a DICE-DICE struct
;; OUTPUT: None
;; SIDE EFFECT: Carries out the corresponding controller
;; function, which destructively modifies DD, and then
;; calls SHOW-GAME to see the results.

(define roll-wr!
  (lambda (dd)
    (roll! dd)
    (show-game dd)))

(define save-wr!
  (lambda (dd)
    (save! dd)
    (show-game dd)))

(define continue-wr!
  (lambda (dd)
    (continue! dd)
    (show-game dd)))

(define reset-wr!
  (lambda (dd)
    (reset! dd)
    (show-game dd)))

The following interactions demonstrate the first few turns of a new game of dice-dice.

> (define dd (new-game))
> (show-game dd)
Start turn 0. Scores: #(0 0 0 0 0). Grand total: 0
  ==> ROLL! or SAVE!
> (roll-wr! dd)
Turn 0. Last Toss: (4 1), Curr Total: 5
  ==> ROLL! or SAVE!
> (roll-wr! dd)
Turn 0. Last Toss: (4 5), Curr Total: 14
  ==> ROLL! or SAVE!
> (save-wr! dd)
Start turn 1. Scores: #(14 0 0 0 0). Grand total: 14
  ==> ROLL! or SAVE!
> (roll-wr! dd)
Turn 1. Last Toss: (2 1), Curr Total: 3
  ==> ROLL! or SAVE!
> (roll-wr! dd)
Turn 1. Last Toss: (4 1), Curr Total: 8
  ==> ROLL! or SAVE!
> (roll-wr! dd)
Turn 1. Last Toss: (1 6), Curr Total: 15
===> ROLL! or SAVE!
> (roll-wr! dd)
Turn 1. Last Toss: (3 4), Curr Total: 22
===> ROLL! or SAVE!
> (roll-wr! dd)
Sorry! You zeroed out! Last toss: (5 5).
===> CONTINUE!
> (continue-wr! dd)
Start turn 2. Scores: #(14 0 0 0 0). Grand total: 14
===> ROLL! or SAVE!
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 system, we may refer to these commands as Linux commands.) The rest of this chapter describes the file system, how to explore the file system using the commands issued from a Terminal window, and how to format, submit and print out your assignment files.

A.1 The File System

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

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

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

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

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

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

The Linux operating system provides numerous commands that enable you to navigate through the file system. These commands are processed by a program called *Terminal.* When you start the *Terminal* program, it opens up a *Terminal window.* When a command is typed into the *Terminal* window, and the **Enter** key is tapped, the *Terminal* program will attempt to execute the command.
When using Linux commands in a Terminal window to navigate the file system, the Terminal program keeps track of your current location within the directory tree. That current location is called your working directory. The 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:

```
;; ===========================================
;; 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:

;;; -----------------------------------
(p 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:
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.

<table>
<thead>
<tr>
<th>Example A.3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppose that &quot;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:</td>
</tr>
<tr>
<td><code>enscript -P Asprey &quot;hunsberg/Desktop/my101/labs/lab2/hun-lab2.txt</code></td>
</tr>
<tr>
<td>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:</td>
</tr>
<tr>
<td><code>enscript -P Asprey hun-lab2.txt</code></td>
</tr>
<tr>
<td>(See Section A.2 if you need a refresher on cd-ing into a desired directory.)</td>
</tr>
</tbody>
</table>

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><strong>pwd</strong></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/yourAcctName.)</td>
</tr>
<tr>
<td><strong>cd ~</strong></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: cd stands for “change directory”; and ~ is a convenient shorthand for your home directory.</td>
</tr>
<tr>
<td><strong>ls</strong></td>
<td>List the contents of your working directory. It should contain a subdirectory called Desktop.</td>
</tr>
<tr>
<td><strong>cd Desktop</strong></td>
<td>Change the working directory to be your Desktop directory.</td>
</tr>
<tr>
<td><strong>ls</strong></td>
<td>List the contents of the working directory—which should be the Desktop directory at this point. Note that most of the contents of the Desktop directory have a corresponding icon on the Desktop!</td>
</tr>
<tr>
<td><strong>mkdir my101</strong></td>
<td>Create a new directory called my101. Note that because my101 is not a full pathname, the new directory will created within the working directory—in this case, the Desktop directory. And since the new directory is located within the Desktop directory, an icon will automatically appear on the Desktop!</td>
</tr>
<tr>
<td><strong>ls</strong></td>
<td>This should show you that the Desktop directory now contains a subdirectory called my101.</td>
</tr>
<tr>
<td><strong>cd my101</strong></td>
<td>Travel into the my101 directory.</td>
</tr>
<tr>
<td><strong>pwd</strong></td>
<td>Show that you are now in the my101 sub-directory. It will probably be displayed before the prompt as ~/Desktop/my101 or /home/yourAcctName/Desktop/my101.</td>
</tr>
<tr>
<td><strong>ls</strong></td>
<td>Show the (non-existent) contents of your new my101 directory.</td>
</tr>
<tr>
<td><strong>mkdir labs</strong></td>
<td>Create a new directory called labs inside the my101 directory. (It is created within the my101 directory because that is the working directory—i.e., the directory where you are right now.)</td>
</tr>
<tr>
<td><strong>ls</strong></td>
<td>Show that you indeed have created labs.</td>
</tr>
<tr>
<td><strong>cd labs</strong></td>
<td>Move into the labs directory.</td>
</tr>
<tr>
<td><strong>pwd</strong></td>
<td>Show that you are indeed there.</td>
</tr>
<tr>
<td><strong>ls</strong></td>
<td>Show that the labs directory is currently empty.</td>
</tr>
<tr>
<td><strong>mkdir lab1</strong></td>
<td>Create a sub-directory called lab1.</td>
</tr>
<tr>
<td><strong>ls</strong></td>
<td>Show that lab1 is there.</td>
</tr>
<tr>
<td><strong>cd lab1</strong></td>
<td>Move into the lab1 directory.</td>
</tr>
<tr>
<td><strong>pwd</strong></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:

   ```
   ```

   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:

   ```
cp ~cs101/public_html/hun-spr-2017/labs/lab1/* ./
   ```

   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 web site, 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/are 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 numbers 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
append, 175
boolean?: type checker, 45
cons: create a cons cell, 153, 155
eq?, 47
equal?, 170, 208, 224
eval: evaluation function, 27
even?: is its input even?, 103, 126, 167, 172
first: access first element of a list, 153
format, 245
fourth, 155
integer?, 26, 123
length, 160
list, 157
list-ref, 161
list?: type checker, 45
load: for loading files, 68
log: natural logarithm, 129
make-vector, 216
member, 162
not: flip boolean value, 77
null?: type checker, 45
number?: type checker, 45
odd?: is its input odd?, 103, 126, 167
printf: formatted printing, 64
procedure?: type checker, 45
quotient: integer division, 26, 34, 127, 128
random: pseudo-random number gen., 134
remainder: integer division, 26, 34, 127, 128
remove, 174
rest: access the rest of a list, 153
reverse, 184
second, 155
sin: trig function, 112
sort, 197
string-append, 245
string<=?, 168
symbol?: type checker, 45
third, 155
vector, 215
vector->list, 231
vector-length, 217
vector-ref, 216
vector-set!, 224
void, 27, 164
void?: type checker, 45, 65
character sequences, 13, 14
comments, 69
computational model, 8, 21, 34
cons cell, 198, see also pair
contracts, 26
  as comments in a Scheme program, 69
data structures, 198, 237
  field-based, 237
  fields, 198
  implemented using vectors, 237, 238
  index-based, 237
  recursively defined, 237
data type, 13
  booleans, 14
  functions, 25
  non-empty lists, 29
  null, 15
  numbers, 13
  strings, 63
  symbols, 15
  void, 27, 164
debug, 8
deep lists, see also hierarchical lists, nested lists
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

273
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
hierarchical lists, see also deep lists, nested lists
identity function, 17
insertion-sort algorithm, 188

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, 131
local variable, 131
local vs. global environments, 57, 133

nested lists, see also hierarchical lists, deep lists
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

pair, 198, see also cons cell
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, 121

sample data structures
  date, 241
  dice-dice, 249

sample functions
  add-squares, 160
  all-numbers, 165
  alt-sum, 103
  approx-e-acc, 120
  approx-e-wr, 122
  approx-log-acc, 128
  approx-pi-acc, 118
  approx-pi-wr, 122
  bad-reverse, 184
  banana-msg, 76
  bookstore-open?, 93
  cafe-open?, 85
  cafe-open?-alt, 87
  civil-to-mil-hours, 81
  compare-sorting-algs, 197
  compute-histogram, 233
  compute-tax, 93
  conc, 174
  cond-effects, 84
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>print-rightside-up-triangle</td>
<td>110</td>
</tr>
<tr>
<td>print-square</td>
<td>109</td>
</tr>
<tr>
<td>print-thing-n-times</td>
<td>108</td>
</tr>
<tr>
<td>print-upside-down-triangle</td>
<td>109</td>
</tr>
<tr>
<td>print-vector</td>
<td>220</td>
</tr>
<tr>
<td>print-vector-from</td>
<td>218</td>
</tr>
<tr>
<td>print-vector-wr</td>
<td>218</td>
</tr>
<tr>
<td>quadrant</td>
<td>90</td>
</tr>
<tr>
<td>quick-list?</td>
<td>204</td>
</tr>
<tr>
<td>random-tosses</td>
<td>179</td>
</tr>
<tr>
<td>rect-area</td>
<td>54</td>
</tr>
<tr>
<td>rem-dupes</td>
<td>187</td>
</tr>
<tr>
<td>remove-if</td>
<td>176</td>
</tr>
<tr>
<td>remover</td>
<td>174</td>
</tr>
<tr>
<td>replace</td>
<td>177</td>
</tr>
<tr>
<td>replace*</td>
<td>207</td>
</tr>
<tr>
<td>replace-by-depth*</td>
<td>209</td>
</tr>
<tr>
<td>reset!</td>
<td>251</td>
</tr>
<tr>
<td>reset-wr!</td>
<td>254</td>
</tr>
<tr>
<td>reversey</td>
<td>183</td>
</tr>
<tr>
<td>roll!</td>
<td>251</td>
</tr>
<tr>
<td>roll-some</td>
<td>187</td>
</tr>
<tr>
<td>roll-some!</td>
<td>227</td>
</tr>
<tr>
<td>roll-wr!</td>
<td>254</td>
</tr>
<tr>
<td>save!</td>
<td>251</td>
</tr>
<tr>
<td>save-wr!</td>
<td>254</td>
</tr>
<tr>
<td>sekund</td>
<td>155</td>
</tr>
<tr>
<td>set-date-year!</td>
<td>239</td>
</tr>
<tr>
<td>set-date-month!</td>
<td>239</td>
</tr>
<tr>
<td>set-date-year!</td>
<td>239</td>
</tr>
<tr>
<td>show-game</td>
<td>249</td>
</tr>
<tr>
<td>sorted?</td>
<td>168</td>
</tr>
<tr>
<td>sphere-volume</td>
<td>57</td>
</tr>
<tr>
<td>split</td>
<td>193</td>
</tr>
<tr>
<td>split-acc</td>
<td>195</td>
</tr>
<tr>
<td>square</td>
<td>53, 58, 60</td>
</tr>
<tr>
<td>stutter</td>
<td>178</td>
</tr>
<tr>
<td>sum-all*</td>
<td>203</td>
</tr>
<tr>
<td>sum-cubes</td>
<td>150</td>
</tr>
<tr>
<td>sum-recips</td>
<td>102</td>
</tr>
<tr>
<td>sum-squares</td>
<td>100</td>
</tr>
<tr>
<td>sum-squares-acc</td>
<td>117</td>
</tr>
<tr>
<td>sum-the-even-tosses</td>
<td>139</td>
</tr>
<tr>
<td>sum-to-n</td>
<td>100, 113</td>
</tr>
<tr>
<td>sum-to-n-acc</td>
<td>114</td>
</tr>
<tr>
<td>tester</td>
<td>66</td>
</tr>
<tr>
<td>and quote</td>
<td>67</td>
</tr>
<tr>
<td>tester-alt</td>
<td>107</td>
</tr>
<tr>
<td>thurd</td>
<td>155</td>
</tr>
<tr>
<td>time-of-day</td>
<td>92</td>
</tr>
<tr>
<td>toss-die</td>
<td>135</td>
</tr>
<tr>
<td>toss-three-dice-until-beat-target</td>
<td>143</td>
</tr>
<tr>
<td>toss-until-doubles</td>
<td>142</td>
</tr>
<tr>
<td>toss-until-total-beats-target</td>
<td>144</td>
</tr>
<tr>
<td>transfer-all</td>
<td>183, 191</td>
</tr>
<tr>
<td>veck-has-satisfier?</td>
<td>233</td>
</tr>
<tr>
<td>veck-index-of</td>
<td>230</td>
</tr>
<tr>
<td>veck-index-of-helper</td>
<td>230</td>
</tr>
<tr>
<td>vector-replace!</td>
<td>229</td>
</tr>
<tr>
<td>veck-to-list</td>
<td>230</td>
</tr>
<tr>
<td>veck-to-list-helper</td>
<td>231</td>
</tr>
<tr>
<td>vector-equal?</td>
<td>223</td>
</tr>
<tr>
<td>vector-map</td>
<td>228</td>
</tr>
<tr>
<td>vector-map!</td>
<td>229</td>
</tr>
<tr>
<td>vector-reverse!</td>
<td>228</td>
</tr>
<tr>
<td>vector-swap!</td>
<td>225</td>
</tr>
<tr>
<td>verbose-func</td>
<td>66</td>
</tr>
<tr>
<td>well-formed-list?</td>
<td>201</td>
</tr>
</tbody>
</table>

**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

define-struct, 241

semantics of, 241

syntax of, 241
dotimes, 219

if, 73

lambda, 52

semantics of, 55

syntax of, 54

let, 131

in terms of lambda, 136

semantics of, 132

syntax of, 131

let*, 131, 144
equivalence to nested lets, 146

semantics of, 145

syntax of, 144

letrec, 131, 147

semantics of, 148

syntax of, 148

or, 80

quote, 41

alternate syntax (’), 42

semantics of, 41

syntax of, 41
time, 196
syntax, 7

truth values, 13

#<undefined>, 148

variables, 18
vectors
  immutable, 214
  mutable, 215, 216
  the pound syntax, 214
  the #(...) syntax, 214
void datum, 27, 38, 64, 97