The Syntax and Semantics of the Scheme Programming Language

Luke Hunsberger

Fall 2010

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 (i.e., 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 entire 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 helpful to have a 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., semantics), it is as computationally powerful as any programming language. (That has been proven mathematically.)

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!

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 article is itself 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 are legal programs.

Each Scheme program consists of one or more 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. 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 subsection below addresses a different type of primitive data.

A primitive datum is one that is atomic, in the sense that it does not have any parts that we 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.

We won’t bother to explicitly write down the full set of syntax rules for numerical expressions, since Scheme’s syntax rules for numerical expressions are quite similar to those for numerical expressions seen in math classes.

As a reminder of the importance of distinguishing the character sequences from the data they denote, we use the following sort of notation:

| Character Sequence | → | Datum |

⇒ A primitive datum is one that is atomic, in the sense that it does not have any parts that we can access.
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 these character sequences.

### 2.2 Booleans

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

\[
\begin{align*}
\texttt{#t} & \rightarrow \text{ the true truth value} \\
\texttt{#f} & \rightarrow \text{ the false truth value}
\end{align*}
\]

Again, keep in mind the difference between the character sequences and the truth values they denote. The \texttt{boolean} data type consists solely of these two truth values (i.e., pieces of data). As programmers, we type the character sequences; 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, \texttt{()}, is a legal Scheme expression. According to the semantics of Scheme, it denotes the \texttt{null} datum, which is also called the \texttt{empty list}.

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

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

### 2.4 Symbols

Another kind of data in Scheme is a \texttt{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 is a legal symbol expression in Scheme. For example, \texttt{hello}, \texttt{goodBye} and \texttt{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, \texttt{brave-new-world}, \texttt{gas-mileage} and \texttt{xyz-prq-abc} are legal symbol expressions in Scheme. Finally, commonly used one-character expressions, such as \texttt{*}, \texttt{+} and \texttt{-}, also constitute legal symbol expressions in Scheme.

The semantics of Scheme stipulates the symbol denoted by each legal symbol expression. For example, the legal expression, \texttt{hello}, denotes the symbol \texttt{hello}; and the legal expression, \texttt{*}, denotes the asterisk symbol.

\[
\begin{align*}
\texttt{hello} & \rightarrow \text{ the symbol hello} \\
\ast & \rightarrow \text{ the asterisk symbol}
\end{align*}
\]

Again, it is important to keep in mind the difference between the typewritten character sequences (e.g., \texttt{hello} and \texttt{bye-bye}) and the \texttt{symbols} (i.e., the Scheme data) that they denote (e.g., the symbol \texttt{hello} and the symbol \texttt{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.)
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 section 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. In particular, the evaluation function takes one Scheme datum as its input, and generates another Scheme datum as its output, as illustrated below.

```
<table>
<thead>
<tr>
<th>Input Datum</th>
<th>Output Datum</th>
</tr>
</thead>
<tbody>
<tr>
<td>the number two</td>
<td>the number two</td>
</tr>
<tr>
<td>the boolean true</td>
<td>the boolean true</td>
</tr>
</tbody>
</table>
```

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 up to this point.

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.

```
<table>
<thead>
<tr>
<th>Input Datum</th>
<th>Output Datum</th>
</tr>
</thead>
<tbody>
<tr>
<td>the number zero</td>
<td>the number zero</td>
</tr>
<tr>
<td>the boolean true</td>
<td>the boolean true</td>
</tr>
<tr>
<td>the empty list</td>
<td>the empty list</td>
</tr>
</tbody>
</table>
```

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.\footnote{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 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:

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

The Scheme datum associated with a symbol in the global environment can be of any type. Thus, it might be that the boolean true is associated with the symbol $pq$. Similarly, the empty list might be associated with the symbol my-empty-list.

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

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

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

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

4 Introduction to DrScheme

This section introduces the piece of software known as DrScheme.\footnote{The DrScheme software is freely available from drscheme.org.} 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 below, 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, \( C_{\text{in}} \), into the Interactions Window, and then hit the Return (or Enter) key, DrScheme does the following:

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

This process is illustrated below.

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

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

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

### 4.1 Some Sample Interactions

We can use DrScheme to confirm some of the things discussed in previous sections. 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:

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

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

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

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

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

### 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. It is important to realize that each primitive function is a Scheme datum, just like numbers and booleans.

Since each primitive function is a Scheme datum, you might be wondering what character sequences in the Scheme language denote the 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.

For example, the global environment contains entries such that each of the following evaluations holds:

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

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:

\[
\begin{align*}
> + & \rightarrow \#<\text{procedure: +}> \\
> - & \rightarrow \#<\text{procedure: -}> \\
> * & \rightarrow \#<\text{procedure: *}> \\
> / & \rightarrow \#<\text{procedure: /}>
\end{align*}
\]
The behind-the-scenes work involved in these interactions can be summarized as follows:

\[
\begin{align*}
+ & \rightarrow [ \text{the + symbol} \implies \text{the addition function}] \rightarrow \text{#<procedure:+>} \\
- & \rightarrow [ \text{the - symbol} \implies \text{the subtraction function}] \rightarrow \text{#<procedure:->} \\
* & \rightarrow [ \text{the * symbol} \implies \text{the multiplication function}] \rightarrow \text{#<procedure:*>} \\
/ & \rightarrow [ \text{the / symbol} \implies \text{the division function}] \rightarrow \text{#<procedure:/}}
\end{align*}
\]

Notice that the character sequences reported by DrScheme are not 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 \text{#<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 \text{+} 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.

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 there is no way for us to confirm it yet, it is nonetheless true that primitive functions evaluate to themselves. In other words, if you feed a primitive function as input to the evaluation function, you get that same primitive function as the output. For example, the addition function evaluates to the addition function; and the multiplication function evaluates to the multiplication function.

6 Non-Empty Lists (Non-Primitive Scheme Data)

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 can inspect. In contrast, this section presents an example of non-primitive data—that is, data that has parts that we can inspect. In particular, this section 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 \text{+}, the number \text{three}, and the boolean \text{true}. Other examples of non-empty lists are given below:

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

The last example illustrates that a list can contain data 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 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 sample character sequences that the programmer can use to denote the Scheme lists described above:

\[
\begin{align*}
(3 \ 4) & \rightarrow \text{a list containing the number \text{three} and the number \text{four}} \\
(+ \ 3 \ 4) & \rightarrow \text{a list containing the + symbol, the number \text{three} and the number \text{four}} \\
(\text{eval} \ (+ \ 3 \ 4)) & \rightarrow \text{a list containing: (1) the symbol \text{eval}, and (2) a subsidiary list containing the + symbol, the number \text{three} and the number \text{four}}
\end{align*}
\]
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 denotes a list containing \(n\) items—namely, the \(n\) items denoted by \(E_1, E_2, \ldots, E_n\). Thus, if

\[
\begin{align*}
E_1 &\rightarrow D_1 \\
E_2 &\rightarrow D_2 \\
\vdots \\
E_n &\rightarrow D_n
\end{align*}
\]

(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 the picture at the top of page 6.) 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.

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

```scheme
> (+ 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.

**An Example.** Consider the first case. The expression is: \((+ 2 3)\). This expression denotes a list containing three items: the \(+\) symbol, the number \(two\), and the number \(three\). The first step in evaluating this list is to **evaluate each item in the list**. Now, the \(+\) symbol evaluates to the built-in addition function because the global environment is guaranteed to contain an entry associating the \(+\) symbol with the addition function. The remaining items in the list are numbers; thus, they trivially evaluate to themselves. This first step is summarized below:

\[
\begin{align*}
+ &\rightarrow \text{the }\ +\text{ symbol }\implies \text{the addition function} \\
2 &\rightarrow \text{the number }two\implies \text{the number }two
\end{align*}
\]
Okay, so after evaluating all of the items in the list we have the addition function and two numbers. The second step in the default rule involves applying that function to the remaining items (i.e., feeding the remaining items as input into that function), as illustrated below:

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:

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

where the evaluation step is explained by:

First Step of Default Rule:

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

Second Step of Default Rule:

addition function applied to two and three yields output of five

Pictorially, the evaluation of this list looks like this:

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

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

32

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

Okay, so far so good: we have seen that our input expression denotes a particular list. This list shall be the Input Datum for the evaluation function.

To evaluate this list, we need to use the default rule. The first step of the default rule requires us to evaluate each item in the list:
the + symbol $\Rightarrow$ the addition function
the number two $\Rightarrow$ the number two
the subsidiary list $\Rightarrow$ oops!

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

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

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

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

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

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

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

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

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

These Scheme data evaluate to the following:

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

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

\[
> (+ 1 2 3)
\]

\[
6
\]

Notice that because the addition function is a primitive function, its operation is invisible to us. We observe the inputs going in and the output coming out, but we do not get to see how the output is generated.

Summary. 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 to the rest of the new items (i.e., feeding the rest of the items as input to the function that is the first item). The result of this function application is taken to be the result of evaluating the original list.

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

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

Surprise! The evaluation function that is so important to the computational model of Scheme is itself provided as a built-in function. Since it is a primitive function, we don’t get to see how it does its thing; however, as described in this handout, we already understand what it does—at least, for many kinds of Scheme data.

The global environment contains an entry that associates the eval symbol with the built-in evaluation function, as confirmed by the following interaction:

\[
> \text{eval}
\]

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

We can even use the default rule to explicitly apply the evaluation function to some inputs, as demonstrated below:

\[
> \text{(eval +)}
\]

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

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

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

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

The second step of the default rule requires us to apply the first item (i.e., the evaluation function) to the second item (i.e., the addition function). Since primitive 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.
7 Special Forms

In DrScheme, there is a special class of symbol expressions called *keywords*. Examples of keywords include: `and`, `cond`, `define`, `do`, `else`, `if`, `lambda`, `let`, `or` and `quote`. Each of these keywords is a legal Scheme expression that denotes a symbol. For example, `else` denotes the *else* symbol, and `lambda` denotes the *lambda* symbol. For expository convenience, we may refer to expressions such as *else* and *lambda* as keyword expressions, and the corresponding symbols (i.e., the *else* 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 special forms have their own special mode of evaluation.

For example, each of the following character sequences denotes a special form:

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

The important thing about special forms is that they are *not* evaluated according to the default rule seen earlier. 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 weeks, you will be introduced to about a dozen different kinds of special form. For each kind of special form, you will have to learn the corresponding evaluation rule. However, you will use these special forms so often that their special modes of evaluation will become second nature after a short time. (Once you get the hang of it, learning the evaluation rule for another 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 indicated by the `define` keyword. As a character sequence, it has the form

```
(define C1 C2)
```

where `C1` is an expression denoting some Scheme symbol `s`, and `C2` can be any expression denoting any Scheme datum, `e`, as illustrated below.

```
C1 → s and C2 → e
```

What happens when a `define` special form is evaluated is illustrated below:

Evaluating the special form: `(define C1 C2)`

```
C1    C2
  ↓    ↓
s    e  ⇒  E
```

Global Environment Entry: `s E`
Notice that \( C_1 \) and \( C_2 \) respectively denote \( s \) and \( e \). But only \( e \) is evaluated, the result being some datum \( E \). (The symbol \( s \) is not evaluated.) Next, an entry is created in the global environment in which the datum \( E \) is associated with the symbol \( s \). Since the purpose of a \texttt{define} special form is to create an entry in the global environment, a \texttt{define} special form does not result in any \textit{output}; thus, DrScheme does not generate any output on the computer screen, as illustrated below:

\[
> \text{(define } x \text{ 6)}
\]

An Example. Typing the character sequence \texttt{(define x (+ 1 2 3))} into the Interactions Window and hitting return would result in the number \textit{six} being associated with the symbol \textit{x} in the global environment, as illustrated below.

Evaluating the special form: \( \text{(define x (+ 1 2 3))} \)

\[
\begin{align*}
\text{x} & \quad \text{(+ 1 2 3)} \\
\downarrow & \quad \downarrow \\
\text{the symbol x} & \quad \text{[a list containing the + symbol and three numbers]} \\
\text{Global Environment Entry:} & \quad \text{the symbol x: the number six}
\end{align*}
\]

However, DrScheme does not report any output value:

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

However, subsequent attempts to evaluate the symbol \( x \) would result in the value \( 6 \), as illustrated below:

\[
\begin{align*}
> x \\
& \quad \text{. . reference to undefined identifier: x} \\
> \text{(define x (+ 1 2 3))} \\
> x \\
& \quad 6 \\
> (* x 100) \\
& \quad 600 \\
> x \\
& \quad 6 \\
> (* x 1000) \\
& \quad 6000 \\
> (* x x) \\
& \quad 36 \\
> x \\
& \quad 6
\end{align*}
\]

Notice that the first attempt to evaluate the symbol \( x \) resulted in an error; however, after the \texttt{define} special form, attempts to evaluate \( x \) result in the value \textit{six}. The subsequent expressions can be evaluated using what we have learned in previous sections. 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.

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 the \texttt{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 the figure at the top of page 6.) 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.

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

\[(quote \ C)\]

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

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

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

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

**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. 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.

Here are some examples:

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

In the first example, \((quote \ x)\) denotes a list containing the *quote* symbol and the symbol \(x\). That list is the Input Datum. The result of evaluating that list is the symbol \(x\)—that is the Output Datum. Notice that the list is evaluated, but its second element is not. We can abbreviate this evaluation as follows:

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

\[\text{In fact, the keyword symbol is never evaluated in a special form of any kind. The purpose of the keyword symbol is simply to indicate that the given list is a special form, thereby requiring a special mode of evaluation.}\]
This is quite different from the default rule for evaluating non-empty lists. Well, that’s to be expected since the default rule was not used!

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

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

\[
\text{\texttt{(quote \( C \))} \longrightarrow \left\{ \begin{array}{l}
\text{list containing \texttt{quote} symbol and \( D \) } \rightarrow \ D \\
\end{array} \right.} \longrightarrow \ C'
\]

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

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

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

\[
> \texttt{6}
> \texttt{(quote x)}
> \texttt{x}
\]

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

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

\textbf{Alternate Syntax for quote Special Forms.} Since \texttt{quote} special forms are used so frequently, there is an alternate syntax for them. In particular, if \( C \) is any Scheme expression denoting some datum \( D \), then the expressions, \( \texttt{(quote} \ C\texttt{)} \) and \( \texttt{'}C\texttt{',} \) denote the same two-element list—namely, a list containing the \texttt{quote} symbol and the datum \( D \):

\[
\texttt{(quote } \ C\texttt{)} \longrightarrow \text{list containing \texttt{quote} symbol and } \ D
\]

\[
\texttt{'}C\texttt{'} \longrightarrow \text{list containing \texttt{quote} symbol and } \ D
\]

The two character expressions are quite different, but both represent the same list! (Syntax vs. Semantics!) For example, \( \texttt{'}num\texttt{'} \) and \( \texttt{(quote num)} \) both represent a list containing the \texttt{quote} symbol and the \texttt{num} symbol, as illustrated below:

\[
> (\texttt{quote num})
\]

\[
\texttt{num}
\]

\[
> \texttt{'}num\texttt{'}
\]

\[
\texttt{num}
\]

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

The following examples demonstrate the equivalence between the two kinds of syntax for the \texttt{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 \texttt{quote} symbol and the \texttt{x} symbol.

\[
> (\texttt{quote (quote x)})
> \texttt{'}x
> \texttt{'}x
\]

\[
> \texttt{'}\texttt{'}x
\]

\[
\texttt{'}x
\]
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 section 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, as illustrated by the following Interactions Window session.

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

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?.

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, as illustrated in the following Interactions Window session:

> (number? 3)
#t
> (number? #t)
#f
> (boolean? #f)
#t
> (boolean? 'x)
#f
> (symbol? +)
#f
> (symbol? '+)
#t
> (null? ())
#t
> (null? '(+ 1 2))
#f

4This 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.
> (procedure? +)
#t
> (procedure? '+)
#f
> (list? '(+ 1 2))
#t
> (list? ())
#t
> (list? +)
#f

Each of these expressions denotes a list that is evaluated according to the default rule for evaluating non-empty lists. 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.

8.2 Arithmetic Predicates

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

\begin{verbatim}
> (greater-than 3 4) #f
> (greater-than 4 3) #t
> (less-than 4 3) #t
> (less-than-or-equal 4 3) #t
> (less-than-or-equal 3 4) #f
> (less-than-or-equal 3 3) #t
\end{verbatim}

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 a lot 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 section describes how to do this in the Scheme programming language.

9.1 Defining Functions vs. Applying Them to Inputs

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

\[ f(x) = x \times x \]

\footnote{In other contexts, these predicates are commonly called relational operators.}
\footnote{These predicates can also be applied to more than two inputs; however, we shall postpone discussion of such things until the section on recursion.}
In this case, the name of the function is $f$, and we might casually describe it as the \textit{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 definition of the function, $f$, 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 \textit{input parameter}, $x$, which is used to refer to potential input values. In addition, the expression, $x \times x$, 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 \textit{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, $x \times x$, yielding $3 \times 3$. \textit{Evaluating} the expression, $3 \times 3$, 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, $x \times x$, yielding $4 \times 4$, which evaluates to 16.

**Another Example.** 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 $h$:

$$g(w, h) = w \times h$$

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, $w \times h$, yielding $3 \times 7$. Evaluating this expression results in the desired output value, 21.

**Summary.** The 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 \textit{input parameters} and a \textit{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.

**Yet Another Example.** 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, $\frac{1}{3} \pi \times 3^2 \times 2$. 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 \textit{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 \textit{body} of the function being defined. If you’re wondering where the \textit{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 \textit{except its name}.

⇒ The character sequence that denotes a \texttt{lambda} special form is called a \textit{lambda expression}. Thus, a \textit{lambda expression} is a piece of syntax, whereas a \texttt{lambda} special form is a Scheme list datum.
Example: The Squaring Function in Scheme. Recall the mathematical definition of the squaring function:

\[ f(x) = x \times x \]

This mathematical definition does three things:

- It specifies a single input parameter, \( x \), for the function being defined;
- It specifies a body, \( x \times x \), 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. Once a function is defined, we can then use the \texttt{define} special form to give it a name. In particular, the following \texttt{lambda} expression can be used to define a squaring function in Scheme:

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

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:

\[ \Rightarrow \text{The evaluation of a } \lambda \text{ special form always results in a function.} \]

Thus, if the expression, \( \texttt{(lambda (x) (* x x))} \), is typed into the Interactions Window, DrScheme will report that a function has been created, as illustrated below:

\[
\texttt{> (lambda (x) (* x x))}
\]

\#	exttt{<procedure>}

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

\[ \Rightarrow \text{At this point, it is important to stress that the function has been created; however, it has not yet been applied to any inputs!} \]

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:

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

\[
\texttt{> square}
\]

\#	exttt{<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:

\[
\texttt{> (square 3)}
\]

\[
9
\]

\[
\texttt{> (square 4)}
\]

\[
16
\]

\[
\texttt{> (square -8)}
\]

\[
64
\]

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

Incidentally, it is possible to define and apply a function without ever having given it a name, as the following Interactions Window session demonstrates:
> ((lambda (x) (* x x)) 4)
16

The default rule for evaluating non-empty lists is used to evaluate the above expression. In the process, each element of the list is evaluated. The first element of the list is the lambda special form, which evaluates to the 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 more convenient to use functions without bothering to name them.

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

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

In the cone function, 3.14159 is used as an approximation of \( \pi \). In addition, 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.

**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.

The following are examples of well-formed lambda expressions:

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

For the last expression, \( (x y z) \) specifies the parameter list and \( (* x (- y z)) \) specifies the body. In contrast, the following are examples of malformed lambda expressions:

- \( (\text{lambda } (x y) (* x y)) \)
- \( (\text{lambda } (x 10) (* x 10)) \)
- \( (\text{lambda } x) \)
The Semantics of a Lambda Expression. The semantics of a lambda expression stipulates what Scheme datum 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

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

denotes a Scheme list whose elements are as follows:

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

This list is referred to as a lambda special form.

Note. By now, you should be getting used to the fact that a piece of syntax, such as \( \text{\texttt{(lambda (}}x\text{\texttt{)}} (* x x)) \), 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 when writing a handout such as this. Therefore, for the sake of expository convenience, the rest of this handout 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 \( \text{\texttt{(lambda (}}x\text{\texttt{)}} (* x x)) \) evaluates to a function, when we really mean that the list denoted by the expression \( \text{\texttt{(lambda (}}x\text{\texttt{)}} (* x x)) \) 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.

9.3.1 Applying a lambda Function to Input Values

Example: Applying the Squaring Function. Consider the lambda expression, \( \text{\texttt{(lambda (}}x\text{\texttt{)}} (* x x)) \). 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 according to the local environment, not the global environment. The evaluation of \( (* x x) \) therefore yields the result \( 16 \).
- 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. 1.
Another Example: Computing the Volume of a Sphere.  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 symbol \( r \) evaluates to 2, and the symbol \( \pi \) evaluates to 3.14159.  The resulting value is: 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, \( r \)'s value came from the local environment, whereas \( \pi \)'s value 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. 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
```
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, (sphere-volume 1), above.

**Example: 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 the expressions, (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.
Here's an example of a function that takes more than one input argument (i.e., parameter).

```scheme
> (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 $f(x) = x \times x$. 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 $x$ by various input 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:

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

The corresponding mathematical equations/expressions would be:

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

**A Lambda Expression with a Bigger Body!** The following example illustrates that a `lambda` expression can have more than one expression in its body.

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

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

\[\Rightarrow\] 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.

\[\Rightarrow\] 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.
10 Strings, printf, load, the Run Button, and Comments

This section 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 Program 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/function 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.

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

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

> (string? "abc")
#t
> (string? ("a" "b" "c"))
#f
> (string? #t)
#f

10.2 The printf Function

Scheme includes a built-in printf function that operates on format strings as in many other languages. The following examples demonstrate the use of the printf function. The printf function interprets the character sequences, \n, %- and %-A, in a special way that is discussed below. Such character sequences are commonly called escape sequences.

> (printf "Hi there!\n")
Hi there!
> (printf "Oh, I get it!-%")
Oh, I get it!
> (printf "First thing: %-A, second thing: %-A-%" (+ 2 3) (* 6 7))
First thing: 5, second thing: 42

The printf function causes the format string (i.e., its first argument) to be displayed in the Interactions Window, except that:

- the quotation marks are omitted;
• each instance of the escape sequences, \n or \%, is interpreted as a new-line 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 ~A, then there must be n input expressions following the format string, as follows:

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

Here are a few more examples:

```scheme
> (printf "Line One!\n Line Two!!\n Line Three!!!\n")
Line One!
Line Two!!
Line Three!!!
```

```scheme
> (printf "First ===> ~A, Second ===> ~A, Third ===> ~A\n"
(+ 4 2) (- 9 6.3) (* 4 100))
First ===> 6, Second ===> 2.7, Third ===> 400
```

```scheme
> (printf "A symbol: ~A, a string: ~A, a boolean: ~A\n"
'I-am-a-symbol
'I am a String!' (>
A symbol: I-am-a-symbol, a string: I am a String!, a boolean: #t
```

Notice that expressions involving an application of the printf function evaluate to...nothing! That’s because the whole point of a printf function is its side effect. In reality, Scheme provides a special datum that is interpreted as “no value”. This “no value” datum belongs to the void data type. In fact, the “no value” datum is the only datum belonging to the void data type. Like any other data type, there is a corresponding type-checker predicate for the void data type. It is called void?. Its use is demonstrated below.

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

In this example, the Default Rule for evaluating non-empty lists is used to evaluate the expression, (void? (printf "hi\n")). In the process, the void? symbol evaluates to the built-in void? function and (printf "hi\n") evaluates to the special “no value” datum belonging to the void data type. The “no value” datum is fed into the void? function, resulting in the output value #t, as reported by drScheme. The character sequence, hi, was printed out as a side-effect of the evaluation of the expression, (printf "hi\n"). DrScheme uses different colors to distinguish side-effect printing (e.g., hi) from output values (e.g., #t). Of course, in a black-and-white handout such as this, it’s sort of hard to see the different colors!

**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. The value of the last expression determines the output value for the function call.

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\n" a)
        (printf "The value of the second input is: ~A\n" b)
        (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.

**Defining a useful tester function.** The following example defines a tester function that can be used to demonstrate the evaluation of Scheme expressions. The tester function takes a Scheme datum as its input, prints out a character sequence representing that datum, and then evaluates that datum. The output value generated by the tester function is the result of evaluating the input datum.

```
> (define tester
  (lambda (datum)
    (printf "~A ===> ~A~%" datum (eval datum)))
> (tester '+)
+ ===> #<primitive: +>
> (tester +)
#<primitive: +> ===> #<primitive: +>
> (tester '(+ 1 2))
(+ 1 2) ===> 3
> (tester (+ 1 2))
3 ===> 3
```

Notice how the `quote` special form is used to shield the input expression from evaluation. Thus, the unevaluated Scheme datum is fed as input to the tester function. Then, when desired, inside the body of the tester function, the `eval` function is used to explicitly evaluate the test expression.

**Question...** 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. For example, suppose the file "test.scm" contains the following expressions:

```
(printf "Loading test.scm!!")

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

(define x 34)
```

Then the following Interactions Window session could ensue:

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

Thus, 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, Indentation, and Choosing Names for Variables, etc.

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

;;; TESTER
;;; ==============
;;; Takes an arbitrary Scheme datum as its only input.
;;; Displays that datum in the Interactions Window.
;;; Then returns as its output, the result of evaluating
;;; that datum.
;;; ---------------------
;;; NOTE: In typical usage, the desired input expression
;;; must be quoted to shield it from evaluation.

(define tester
  (lambda (datum)
    (printf "~A ==> " datum) ;; Notice, no newline character!
    (eval datum) ;; A rare example of explicitly calling
                ;; the EVAL function!
  ))

;;; Examples
;;; ================

(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 comments make the code easier for people to understand. DrScheme ignores the comments completely.
11 Conditional Expressions

This section introduces conditional expressions. A conditional expression is one whose evaluation depends on the value of a boolean expression. (The boolean expression is the condition.) In Scheme, a conditional expression is typically composed using the if special form.

The boolean condition in a conditional expression can be simple or complex. A complex boolean condition can be composed using the and and or special forms, as well as the built-in not function.

Frequently, it is useful to nest one conditional expression within another. In such cases, the resulting Scheme expression can become quite complex. Thus, Scheme provides another special form, the cond special form, to simplify this common kind of nested conditional expression.

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

Syntactically, the strictest version of an if special form looks like this:

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

where:
- \text{boolExpr} is an expression that evaluates to a boolean (i.e., either \#t or \#f); and
- \text{thenExpr} and \text{elseExpr} are any Scheme expressions.

Thus, the following expressions are examples of the if special form:

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

The semantics of Scheme stipulates that an if special form is evaluated thusly:

- First, the boolean expression, \text{boolExpr}, is evaluated.
- If \text{boolExpr} evaluates to \#t, then \text{thenExpr} is evaluated—and the value of the if special form is whatever \text{thenExpr} evaluates to.
- On the other hand, if \text{boolExpr} evaluates to \#f, then \text{elseExpr} is evaluated—and the value of the if special form is whatever \text{elseExpr} evaluates to.

Notice that the boolean expression is always evaluated; however, only one of the remaining expressions, \text{thenExpr} or \text{elseExpr}, is evaluated.

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

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

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

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

In the third expression, the condition, #f, evaluates to #f. Thus, the else expression, "else", is evaluated. Its value, "else", is the value of the entire if expression.
Using an if expression in the body of a function. Below, a function, howBig, is defined. If given a number less than 10, its output is the symbol, small; otherwise, its output is the symbol, big.

> (define howBig
   (lambda (num)
     (if (< num 10)
       'small
       'big))

> (howBig 5)
small
> (howBig 102)
big

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. Thus, 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).

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

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

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

11.2 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 value as input, the not function returns the opposite boolean value, as demonstrated below:

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

However, the not function also accepts any other kind of Scheme datum as input. It, too, observes the rule that anything other than #f counts as boolean true, as demonstrated below:
> (not 'symbol)  
#f  
> (not (+ 2 3))  
#f  
> (not ())  
#f  
> (not "string")  
#f

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

11.2.2 The and Special Form

In the simplest case, the \texttt{and} special form takes two boolean inputs:

\[
\text{and boolOne boolTwo}
\]

If both boolean expressions, \texttt{boolOne} and \texttt{boolTwo}, evaluate to \#t, then the \texttt{and} expression itself evaluates to \#t. If either or both evaluate to \#f, then the \texttt{and} expression itself evaluates to \#f. The following Interactions Window session demonstrates this behavior:

\[
\begin{align*}
> & (\text{and \#t \#t})  
& \#t  
> & (\text{and (> 3 2) (< 5 9)})  
& \#t  
> & (\text{and \#t \#f})  
& \#f  
> & (\text{and (> 3 2) (= 5 9)})  
& \#f  
> & (\text{and \#f \#t})  
& \#f  
> & (\text{and (> 2 5) \#t})  
& \#f  
> & (\text{and \#f \#f})  
& \#f  
> & (\text{and (> 2 5) (= 9 91)})  
& \#f
\end{align*}
\]

Although the \texttt{and} operator could have been provided as a built-in function, Scheme provides it as a special form. To see why, suppose \texttt{myBigBadFunc} is a function that takes a really long time to compute its output value. Now consider the expression, \(\text{and (= 9 21) (myBigBadFunc 32)}\). Since the value of the first boolean expression, \(= 9 21\), is \#f, the value of the entire \texttt{and} expression must be \#f. Thus, there is no reason to waste time computing the value of \(\text{myBigBadFunc 32}\). If \texttt{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 \texttt{and} as a special form. The evaluation rule for the \texttt{and} special form 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 \texttt{and} expression.

The non-strict version of the \texttt{and} special form. The \texttt{and} special form accepts non-boolean input expressions. Like the \texttt{not} function, it treats any non-boolean expression as though it were boolean \textit{true} (i.e., anything other than \#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. The following Interactions Window session demonstrates this behavior:

\[
\begin{align*}
> & (\text{and 3 4}) \;; \text{ the output value is 4, which counts as boolean true}  
& 4  
> & (\text{and (* 3 4) (* 8 8)}) \;; \text{ the output value is 64, which counts as true}  
& 64
\end{align*}
\]
> (and (* 3 4) (= 9 7)); the output value is boolean false
#f

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

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

> (and #t #t #t #t)
#t
> (and #t #t #f #t #t)
#f
> (and (> 3 2) (= 9 9) (<= 5 20))
#t
> (and 1 2 3 4 5);; 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-strict, then the and expression might evaluate to a non-strict 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:

> (or #f #f #f #f)
#f
> (or #f #f #t #f);; at least one input evaluates to #t, so output is #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

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.

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.

Consider the following letterGrade function:

> (define letterGrade
  (lambda (num)

> (define letterGrade
  (lambda (num)
(if (>= num 90)
   'A
   (if (>= num 80)
      'B
      (if (>= num 70)
         'C
         'D))))

> (letterGrade 86)
  B
> (letterGrade 95)
  A
> (letterGrade 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, (letterGrade 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, (letterGrade 86).

The following Interactions Window session defines a similar function, letGrd, that uses a cond expression instead of the nested if expressions seen above. This cond expression serves the same purpose as the nested if expressions.

> (define letGrd
   (lambda (num)
      (cond
         ((>= num 90)
          'A)
         ((>= num 80)
          'B)
         ((>= num 70)
          'C)
         (#t
          'D))))

Consider the evaluation of the expression, (letGrd 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, (letGrd 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.

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

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

where:

- each cond$_i$ is a (strict or non-strict) condition;
- the last condition, cond$_n$, is #t; and
- each expr$_i$ is some Scheme expression.

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

Each condition, cond$_i$, is evaluated in turn until one is found that evaluates to #t (or something that counts as true).

The value of the cond expression is the value of the corresponding expression, expr$_i$.

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

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

In a cond expression, each condition, cond$_i$, can be followed by multiple subsidiary expressions. Typically, having multiple expressions for a single condition only makes sense if the expressions before the last one have side effects. As with the body of a lambda expression, it is the value of the last subsidiary expression that serves as the value of the cond expression. The following Interactions Window session defines a function, condEffects, that includes a cond expression in which each condition has multiple subsidiary expressions associated with it. Notice how comments are used to make the code easier on the eyes.

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

> (condEffects 94)
Oh my gosh! You did great!!!
A
> (condEffects 86)
Well, you know, a B is pretty good!!
Nothing to be ashamed of at all!!
B
> (condEffects 75)
According to Vassar regulations, a C is considered average!
Thus, your grade, 75, is average!
C
> (condEffects 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, `(condEffects 86)`, was evaluated as follows. First, the condition, `(>= num 90)`, was evaluated. Since it evaluated to #f, the second condition, `(>= num 80)`, was evaluated. This one evaluated to #t. Thus, the associated subsidiary expressions were evaluated in turn. The value of the last subsidiary expression was B. Thus, B was returned as the output value for the entire cond expression. Notice that only the subsidiary expressions associated with the condition, `(>= num 80)`, were evaluated. The subsidiary expressions associated with the other conditions were ignored. The remaining conditions (i.e., `(>= num 70)` and #t) were also ignored.

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

This section introduces recursive functions. Defining recursive functions in Scheme requires no new computational constructs (i.e., no new special forms); 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:

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

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

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 boolean 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 on some stopping point.

Defining Recursive Functions in Scheme. In Scheme, the typical characteristics of the definition of a recursive function, \( f \), are:

- a \text{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 input(s).

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

Example 12.1 The factorial function. The factorial function, \( f(n) = n! \), is frequently defined as follows:

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

This kind of definition is somewhat casual, as evidenced by the “dot-dot-dot”. What does the “dot-dot-dot” mean exactly?

We can give a more precise, recursive definition of the factorial function, as follows:

- Base Case \((n = 1)\): \(1! = 1\).
- Recursive Case \((n > 1)\): \(n! = n \cdot (n - 1)!\)

According to this definition, the following equalities hold:

- \(4! = 4 \cdot 3!\)
- \(3! = 3 \cdot 2!\)

\(^7\)Of course, if the first element of the list evaluates to something other than a function, then no function application can happen.
• $2! = 2 \cdot 1$
• $1! = 1$

Putting all of this information together yields:

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

The following Scheme expression defines a recursive function, \texttt{facty-v1}, whose definition is based on the above insights.\(^8\) The main job of \texttt{facty-v1} is to use recursion to compute the factorial of its input, \(n\).

\[
\text{(define facty-v1}
\text{(lambda (n)
  (if (= n 1)
    1
    (* n (facty-v1 (- n 1)))))
\text{)}
\]

Notice that the \texttt{define} special form effectively gives the name, \texttt{facty-v1}, to the function defined by the \texttt{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 \texttt{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 \texttt{define} special form. So, the symbol, \texttt{facty-v1}, is not evaluated. Only the third element of the \texttt{define} special form—namely the \texttt{lambda} expression—is evaluated. Like any \texttt{lambda} expression, the one above evaluates to a function. However:

\(⇒\) It is important to remember that evaluating the above \texttt{lambda} expression only creates a function. It does not call the function! Thus, the expressions in the body of the \texttt{lambda} expression are not evaluated—yet!

The reason this is important is that the global environment does not yet associate any value with the symbol, \texttt{facty-v1}.\(^9\) Thus, any attempt to evaluate that symbol, at this time, would cause an error.

However, after the \texttt{lambda} expression has been evaluated (to a function), the evaluation of the \texttt{define} special form can continue: in particular, by entering a value (i.e., the newly created function) for the symbol, \texttt{facty-v1}, in the global environment.

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

\[
\begin{align*}
> & (\text{facty-v1 1}) \\
& 1 \\
> & (\text{facty-v1 2}) \\
& 2 \\
> & (\text{facty-v1 3}) \\
& 6 \\
> & (\text{facty-v1 4}) \\
& 24
\end{align*}
\]

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

\[
\text{(define facty-v2}
\text{(lambda (n)
  (cond
    ;; Base Case: n = 1
    ((= n 1)
     1)
    ;; Recursive Case: n > 1
    (#t
     (* n (facty-v2 (- n 1))))))
\text{)}
\]

\(^8\)The function is called, \texttt{facty-v1}, because it is the first version of the factorial function we will look at.

\(^9\)The order of events is: (1) an entry for \texttt{facty-v1} in the global environment is created with a temporary value: \texttt{void}; (2) the \texttt{lambda} expression is evaluated, which yields a function; and (3) that function is entered as the value associated with \texttt{facty-v1}. 
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:

```scheme
> (facty-v2 1)
1
> (facty-v2 2)
2
> (facty-v2 3)
6
> (facty-v2 4)
24
```

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

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

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

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

The application of the `facty` function to the input 3 is depicted at the top of Fig. 3. First, a local environment is created with an entry associating the input parameter `n` with the value 3. Next, the expression in the body of the `facty` function, shown below, is evaluated with respect to that local environment.¹⁰

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

Since the value of `n` is 3 in the local environment, the condition, `( = n 1)`, evaluates to `#f`. Thus, we skip to the second condition, `#t`, which of course evaluates to `#t`. Thus, the expressions associated with the recursive case are evaluated in turn. The first expression causes the line, `Recursive Case (n = 3)`, to be displayed in the Interactions Window. Then, the second expression, `( * n (facty (- n 1)))`, must be evaluated—according to the Default Rule. The `*` symbol evaluates to the `multiplication` function, `n` evaluates to 3, and `(facty (- n 1))` evaluates to ... Gosh, we need a new paragraph!

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

¹⁰To decrease clutter, only a portion of the body is shown in each function-call box in the figure.
Figure 3: DrScheme's evaluation of (facty 3)
⇒ Notice that the evaluation of the expression, \((\ast (\text{facty} (- \text{n} 1)))\), in the top function-call box cannot continue until the subsidiary expression, \((\text{facty} (- \text{n} 1))\), is evaluated. However, this value cannot be known until the output value for the second function-call box has been generated! In other words, the evaluation of the expression in the top box must be suspended, pending the outcome of the second box.

The application of the \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, \text{n}, with the value 2.

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

Thus, the evaluation of the body of the function in the second box proceeds in the environment where \text{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 \text{printf} expression causes the line, \text{Recursive Case (n = 2)}, to be displayed in the Interactions Window; and the evaluation of the expression, \((\ast \text{n} (\text{facty} (- \text{n} 1)))\), 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. 3.

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

The application of the \text{facty} function to the value 1 begins by creating a local environment entry that associates the input parameter \text{n} with the value 1. (Again, this is a new input parameter, distinct from the other \text{n}'s!) Next, the \text{cond} expression in the body of the function is evaluated. This time, however, the condition \((= \text{n} 1)\) evaluates to \#t; thus, the base case expressions are evaluated. Evaluating the \text{printf} expression causes the line, \text{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, \((\text{facty} (- \text{n} 1))\), that was being evaluated in the middle function-call box (where \text{n} has the value 2). Now that that the value of \((\text{facty} (- \text{n} 1))\) is in hand, the evaluation of the expression, \((\ast \text{n} (\text{facty} (- \text{n} 1)))\), in the middle box can continue. To wit, the \text{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, \((\text{facty} (- \text{n} 1))\), that was being evaluated in the top function-call box (where \text{n} has the value 3). Now that the value of \((\text{facty} (- \text{n} 1))\) is in hand, the evaluation of the expression, \((\ast \text{n} (\text{facty} (- \text{n} 1)))\), in the top box can continue. To wit, the \text{multiplication} function is applied to 3 and 2, yielding the output value 6 for the top function-call box.

Phew!

Here is what it looks like when \((\text{facty} 3)\) is evaluated in the Interactions Window:

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

Example 12.1 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, 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.

Example 12.2 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. We can define \( g \) recursively, as follows:

- **Base Case** \( (n = 1) \): \( g(1) = 1 \)
- **Recursive Case** \( (n > 1) \): \( g(n) = n^2 + g(n - 1) \)

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:

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

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

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

---

11There are exceptions to this observation that involve destructive functions. However, this course focuses primarily on non-destructive functions.
12.1 Tail Recursion

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

If this kind of memory-usage problem were characteristic of all recursive functions, it might severely limit the usefulness of recursive functions. However, if the body of the recursive function is defined in a certain way—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 12.3 Printing Dashes. Recall the \text{print-n-dashes} function seen in class:

\begin{verbatim}
;;; PRINT-N-DASHES
;;; -----------------------------------------------
;;; INPUT: N, number (non-negative integer)
;;; OUTPUT: none.
;;; SIDE EFFECT: prints a row of 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 function call print the rest of the dashes
       (print-n-dashes (- n 1)))))
\end{verbatim}

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

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 \texttt{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, \((\texttt{printf } \"-\")\), 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 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 corresponding to the expressions, \((\text{print-n-dashes} 5), (\text{print-n-dashes} 4), (\text{print-n-dashes} 3), (\text{print-n-dashes} 2), (\text{print-n-dashes} 1)\) and \((\text{print-n-dashes} 0)\), 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?

The defining characteristic of a tail-recursive function is that the value of the recursive function-call expression, *without any subsequent computation*, is the output value for the current function-call box.

Suppose, for example, that the body of a tail-recursive function, \(\text{tail-rec-func}\), consists of a single \(\text{cond}\) expression. Suppose further that this \(\text{cond}\) has only two cases: a base case and a recursive case. Then the recursive function-call expression, \((\text{tail-rec-func} \ldots)\), must be, by itself, the *last* expression in the recursive case; it must not be a subsidiary expression within some larger expression. Thus, the definition of \(\text{tail-rec-func}\) must have the following form:

\[
\begin{align*}
(\text{define}\ \text{tail-rec-func} & \\
(\lambda & (\ldots) \\
(\text{cond} & \\
& (\ldots) \\
& (\ldots) \\
& \ldots \\
& (\text{tail-rec-func} \ldots))
\end{align*}
\]

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

\[
\begin{align*}
(\text{define}\ \text{facty} & \\
(\lambda & (n) \\
(\text{cond} & \\
& (\ldots))
\end{align*}
\]

\(\text{This is actually the facty-v2 function, but the same points apply to all versions of the facty function seen earlier.}\)
((= n 1) 1) ;; Recursive Case: n > 1 (#t (* n (facty (- n 1)))))

Notice that the last expression in the recursive case of the cond is (* n (facty (- n 1))). This expression includes the recursive function-call expression, (facty (- n 1)), as a subsidiary expression. This means that the value of the recursive function-call expression is not simply returned as the output value of the parent function-call box. Instead, when the value of the recursive function-call expression is known, some additional computation—in this case, multiplying by n—has to be performed in order to generate the desired output value. For this reason, DrScheme must keep track of the contents of the original function call-box while it processes the recursive function call. Thus, DrScheme must create a separate function call-box for the recursive function call. Thus, DrScheme cannot use the memory-saving trick described for tail-recursive functions. The problem? The function, facty, is not tail recursive.

Tail-recursive functions like print-n-dashes do not generate interesting output values; instead, their primary purpose is to display information in the Interactions Window as a side effect. Functions that generate interesting output values can also be tail recursive; however, they typically require one or more additional input parameters called accumulators, which are discussed in the next section. All of the examples in the next section involve accumulator-based tail-recursive functions.

12.2 Accumulators

In the factorial example, seen earlier, each recursive function call generated an output value that represented a solution to a simpler problem. For example, the evaluation of (facty 4) (i.e., 4!) resulted in the recursive function calls, (facty 3), (facty 2) and (facty 1), whose values were 3!, 2! and 1!, respectively. This section explores a slightly different way of organizing recursive computations using accumulators.

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

As each recursive function call is made, the value of the accumulator gets closer and closer to the desired output value, until finally, when the base case is reached, the accumulator holds the desired answer.

Accumulator-based recursive functions are typically tail recursive. This section explores the use of accumulators in tail-recursive functions.

Example 12.4 Computing sums of the form, 0 + 1 + 2 + ... + n.

We begin with a non-tail-recursive function, sum-to-n, that computes sums of the form 0 + 1 + 2 + ... + n.

;;;; SUM-TO-N
;;;; ------------------------------------------------
;;;; INPUT: N, number (non-negative integer)
;;;; OUTPUT: The value of the sum 0 + 1 + 2 + ... + n
;;;; NOTE: This function is NOT tail recursive
;;;; and does NOT have any accumulators!

(define sum-to-n (lambda (n)  
(cond  
;;;; Base Case: n = 0  
((= n 0)  
(printf "Base Case (n=0)"~%)
 0)  
;;;; Recursive Case: n > 0  
(#t  
(printf "Recursive Case (n=~A) ..."~% n)
 (+ n (sum-to-n (- n 1)))))

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

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

### Using an Accumulator.

Below, we define a function, \texttt{sum-to-n-acc}, that solves the same problem using an extra input parameter, called an \textit{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 \texttt{printf} expressions serve only to display useful information; they do not affect the output value.

\[
\text{;; SUM-TO-N-ACC}
\text{;; --------------------------------------------------}
\text{;; INPUTS: N, number}
\text{;; ACC, number (an accumulator)}
\text{;; OUTPUT: A sum of the form: 0 + 1 + 2 + \ldots + n}
\text{;; --------------------------------------------------}
\text{;; NOTE: To achieve the desired results, the initial value}
\text{;; of the accumulator must be 0.}
\]

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

Since the function, \texttt{sum-to-n-acc}, includes an extra input parameter, we need to supply the values for both \texttt{n} and \texttt{acc} when calling this function. Thus, to compute the sum, 0 + 1 + 2 + 3, using this function, we would evaluate the expression, \((\text{sum-to-n-acc} \; 3 \; 0)\). Notice that the initial accumulator has a value of 0, which is akin to our basket being initially empty. Here’s what the evaluation of \((\text{sum-to-n-acc} \; 3 \; 0)\) looks like in the Interactions Window:

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

First off, notice that we see a similar sequence of function calls, where the value of \texttt{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 \texttt{n}. In other words, our basket has accumulated \texttt{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. 4 illustrates the sequence of recursive function calls generated by DrScheme’s evaluation of \((\text{sum-to-n-acc } 3 \text{ 0})\). Notice that each function-call box has its own input parameters, called \(n\) and \(\text{acc}\), 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 12.5 Factorial Revisited. Here is a tail-recursive version of the factorial function, called \(\text{facty-acc}\):

```scheme
;; FACTY-ACC
;; -----------------------------------------------
;; INPUTS: \(N\), number (positive integer)
;; \(\text{ACC}\), number (accumulator, initially \(1\))
;; OUTPUT: \(N!\) (i.e., the factorial of \(N\))
;; -----------------------------------------------
;; NOTE: An expression of the form \((\text{facty-acc } N\ 1)\) evaluates to \(N!\)!
;; (define facty-acc
  (lambda (n acc)
    (cond
      ;; Base Case: \(n = 1\)
      ((= n 1)
       (printf "Base Case (n=1, acc=\"A\")\n" acc)
       ;; Return the accumulator!
       acc)
      ;; Recursive Case: \(n > 1\)
      (#t
       (printf "Recursive Case (n=\"A\", acc=\"A\")\n" n acc)
       ;; Recursive function call (tail-recursive)
       (facty-acc (- n 1) (* n acc))))))
```

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 \(1\) 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.\(^{13}\)

For \(\text{facty-acc}\), the current accumulator, \(\text{acc}\), 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:

\(^{13}\text{In contrast, the non-tail-recursive function, \text{facy}, seen earlier, included the recursive function-call expression, \((\text{facy } (- n\ 1))\), within the larger expression, \((\text{\star n (facy } (- n\ 1)))\).}
Figure 4: DrScheme's evaluation of \((\text{sum-to-n-acc} \, 3 \, 0)\)
> (facty-acc 4 1)
  Recursive Case (n=4, acc=1)
  Recursive Case (n=3, acc=4)
  Recursive Case (n=2, acc=12)
  Base Case (n=1, acc=24)
  24

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

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

⇒ The output value for (facty-acc n acc) is equal to the factorial of n multiplied by acc.

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

Example 12.6 Summing squares: $1^2 + 2^2 + \ldots + n^2$ Here’s a tail-recursive function for computing the sums of squares from 1 to n:

;;; SUM-SQUARES-ACC
;;; ----------------------------------------------------
;;; INPUTS: N, a non-negative integer
;;; ACC, a number (accumulator)
;;; OUTPUT: If the accumulator is 0, then the output
;;; is equal to the sum 0*0 + 1*1 + 2*2 + \ldots + N*N
(define sum-squares-acc
  (lambda (n acc)
    (cond
      ;; Base Case: n <= 0
      ((<= n 0)
        (printf "Base Case: n=~A, acc=~A~%" n acc)
        ;; Return the accumulator!
        acc)
      ;; Recursive Case: n > 0
      (#t
        (printf "Recursive Case: n=~A, acc=~A~%" n acc)
        (sum-squares-acc (- n 1) (+ acc (* n n)))))))

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

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

> (sum-squares-acc 3 0) ;; compute 0*0 + 1*1 + 2*2 + 3*3 = 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 \(0^2 + 1^2 + \ldots + n^2 + acc\).

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

**Example 12.7 Approximating \(e\)**

Mathematicians tell us that the number \(e\) is well approximated by sums of the form

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

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

- the input parameter \(\text{curr-exp}\) (i.e., the current exponent) will take on the values 0, 1, 2, \ldots, \(n\);
- the input parameter \(\text{max-exp}\) (i.e., the maximum exponent) will stay the same across all recursive function calls (i.e., it will stay fixed at \(n\));
- the input parameter \(\text{curr-denom}\) (i.e., current denominator) will accumulate the factorials that comprise the various denominators in the terms in the sum (i.e., it will take on the values 1, 1, 2, 6, 24, \ldots); and
- the input parameter \(\text{acc}\) will accumulate the desired sum (i.e., it will take on the values 1, 2, 2.5, 2.66666666666, \ldots).

```
;; EXP-ACC
;; ---------------------------------------------------
;; INPUTS: MAX-EXP, number (non-negative integer, maximum exponent)
;; CURR-EXP, number (non-negative integer, current exponent)
;; CURR-DENOM, number (accumulator for current denominator)
;; ACC, number (accumulator for desired answer)
;; OUTPUT: Number equal to: 1 + 1/(1!) + 1/(2!) + 1/(3!) + ... + 1/(N!)
;; ---------------------------------------------------
;; NOTE: To achieve the desired result, use the following
;; initial values: CURR-EXP = 0, CURR-DENOM = 1, ACC = 0

(define exp-acc
  (lambda (max-exp curr-exp curr-denom acc)
    ;; Print out the values of the input parameters first...
    (printf "max-exp=~A, curr-exp=~A, curr-denom=~A, acc=~A~%\n" max-exp curr-exp curr-denom acc)
    (cond
      ;; Base Case: curr-exp > max-exp
      (> curr-exp max-exp)
      ;; Return the accumulator!
      acc)
    ;; Recursive Case: curr-exp <= max-exp
    (#t
      ;; Make recursive function call with updated inputs
      (exp-acc max-exp (+ 1 curr-exp) (* (+ 1 curr-exp) curr-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 \(\text{curr-exp}\), \(\text{curr-denom}\) and \(\text{acc}\) must be 0, 1 and 0, respectively. Thus, to compute the sum

\[
1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!}
\]

would involve evaluating \((\text{exp-acc } 4 \ 0 \ 1 \ 0)\), as illustrated below:
Notice that the max-exp parameter stays fixed at 4 across all the recursive function calls. This parameter is used only to distinguish the base case from the recursive case. The parameter curr-exp represents the exponent of the term currently being worked on; thus, it starts at 0 and increments by 1 each time. The parameter curr-denom represents the denominator of the term currently being worked on; thus, it takes on the values of successive factorials: 0!, 1!, 2!, 3!, .... The parameter acc accumulates the desired sum. By the time the base case is reached (i.e., when curr-exp > max-exp in the last line), the accumulator holds the desired answer. Thus, the accumulator is simply returned as the output value for this function.

If the 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:

```
> (exp-acc 20 0 1 0)
2.7182818284590455
```

## 12.3 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 12.8 A wrapper for facty-acc** The following defines a wrapper function, facty-wr, for the accumulator-based function, facty-acc, defined earlier. Notice that the wrapper function simply calls facty-acc with the accumulator appropriate initialized to 1.

```
(define facty-wr
  (lambda (n)
    (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 12.9 A wrapper for exp-acc** The function, exp-acc, seen earlier, involved several accumulators. The following wrapper function, exp-wr, shields the user from having to know the appropriate initial values for these accumulators:

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

Here’s what it looks like in the Interactions Window:
> (exp-wr 4)
2.708333333333333
> (exp-wr 5)
2.7166666666666663
> (exp-wr 6)
2.7180555555555554
> (exp-wr 100)
2.7182818284590455

Notice that the user of exp-wr may not even be aware that accumulators are being used!
13 Local Variables and Local Environments: let, let* and letrec

This section introduces the let special form and its somewhat more general variants, let* and letrec. The purpose of a let special form is to set up a local environment, much like those that exist inside a function-call box, and then to evaluate one or more expressions with respect to that local environment. The value of a let expression is the value of the last expression in its body. Once a let expression has been evaluated, its local environment vanishes. 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

When introducing any special form, it is important to specify both the syntax and the semantics.

The syntax of a let expression. The syntax of a let expression is as follows:

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

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; and
- \(\text{expr}_1, \ldots, \text{expr}_k\) are \(k\) Scheme expressions, where \(k \geq 1\).

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

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

Example 13.1 Some legal let expressions

The following expressions are all legal let expressions:

- (let () #t)
- (let ((x (+ 2 3))) (* x x))
- (let ((x (+ 2 3))
  (y 3)
  (z (* 2 2)))
  (printf "x: \"A\", y: \"A\", z: \"A\" x y z")
  (+ x y z))

The first let expression includes no var/val pairs, as indicated by the empty list. Its body consists of the single expression, #t. The second let expression includes a single var/val pair: (x (+ 2 3)). Its body consists of the single expression, (* x x). The third let expression includes three var/val pairs: (x (+ 2 3)), (y 3) and (z (* 2 2)). Its body consists of two expressions: a printf expression and (+ x y z).
The semantics of a let expression. As usual, we specify the semantics of a let special form by the special way in which it is evaluated. A let expression of the form

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

is evaluated as follows.

- First, the expressions, \(\text{val}_1, \ldots, \text{val}_n\), are evaluated.
- Second, a local environment is created containing \(n\) entries—one for each of the \(\text{var}/\text{val}\) pairs in the 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, \(\text{expr}_1, \ldots, \text{expr}_k\), 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 of the symbols \(\text{var}_i\) 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, \(\text{expr}_k\), is the value of the entire let expression.

Example 13.2 Evaluating let expressions

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

\[
\begin{align*}
&> \text{(let ()} \\
&\quad \#t) \\
&\#t \\
&> \text{(let ((x (+ 2 3)))} \\
&\quad (* x x)) \\
&25 \\
&> \text{(let ((x (+ 2 3)))} \\
&\quad (y 3) \\
&\quad (z (* 2 2))) \\
&\quad (\text{printf } "x: ~A, y: ~A, z: ~A~\%" \ x \ y \ z) \\
&\quad (+ x y z)) \\
&x: 5, y: 3, z: 4 \\
&12
\end{align*}
\]

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 \(\text{var}/\text{val}\) 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 13.3 Local vs. Global

The following Interactions Window session demonstrates that the local environment supersedes the global environment when evaluating expressions in the body of a let.
> (define x 1000)
> (define y 100)
> (define z 10)
> (+ x y z)
1110
> (let ((x 3)
       (y 4))
    (+ x y z))
17

The first three expressions use the `define` special form to create three global variables, named `x`, `y`, and `z`. The last expression uses a `let` to create a local environment containing two local variables, named `x` and `y`. When the single expression in the body of the `let` is evaluated, the values for `x` and `y` are drawn from the local environment, whereas the values for `+` and `z` are drawn from the global environment.

**Example 13.4** Typically, a `let` expression is used to store a value in a local variable so that it can be referred to in subsequent expressions. The following Interactions Window session demonstrates how the result of a call to the `random` function can be stored in a local variable.

> (let ((rnd (random 10)))
  (printf "The random number is: ~A~%" rnd)
  (printf "When squared, it is: ~A~%" (* rnd rnd))
  (printf "We’ll return its cube: ~%"
          (* rnd rnd rnd))
The random number is: 6
When squared, it is: 36
We’ll return its cube: 216
> rnd
ERROR: reference to undefined identifier: rnd

However, notice that after the `let` expression has been evaluated, its local environment ceases to exist. Thus, the subsequent attempt to evaluate `rnd` causes DrScheme to report an error. (This example assumes that there is no entry for `rnd` in the global environment.)

**Deriving the `let` special form from the `lambda` special form.** If you’re thinking that the evaluation of a `let` expression seems awfully close to the evaluation of a function call, you’re right. In fact, the `let` special form 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 `let` expressions with certain expressions involving the application of a `lambda` function.

**Example 13.5** The following Interactions Window session shows the evaluation of a `let` expression, followed by the evaluation of an equivalent expression involving the application of a `lambda` function to some inputs.

> (let ((x (+ 2 3))
      (y (* 3 4)))
  (printf "x: ~A, y: ~A~%" x y)
  (+ x y))
x: 5, y: 12
17
> ((lambda (x y)
    (printf "x: ~A, y: ~A~%" x y)
    (+ x y))
  (+ 2 3)
  (* 3 4))
x: 5, y: 12
17

⇒ The semantics for the evaluation of the first expression is identical to the semantics for the evaluation of the second expression!
In particular, for the `let` expression, a local environment is set up in which the symbol \( x \) is associated with the value 5 and the symbol \( y \) is associated with the value 12. After that, the two expressions in the body of the `let` are evaluated with respect to that local environment yielding some side-effect printing and an output value of 17.

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

**Example 13.6** The following Interactions Window session first creates a global variable, \( z \). It then evaluates a `let` expression and an equivalent expression involving the application of a `lambda` function.

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

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

In general, a `let` expression of the form,

\[
\begin{align*}
& (\text{let } \left( (\var_1 \ \text{val}_1) \\
& \quad (\var_2 \ \text{val}_2) \\
& \quad \ldots \\
& \quad (\var_n \ \text{val}_n) \\
& \quad \text{expr}_1 \\
& \quad \text{expr}_2 \\
& \quad \ldots \\
& \quad \text{expr}_n) \\
& \end{align*}
\]

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

\[
\begin{align*}
& ((\text{lambda } (\var_1 \ldots \var_n) \\
& \quad \text{expr}_1 \\
& \quad \text{expr}_2 \\
& \quad \ldots \\
& \quad \text{expr}_n) \\
& \quad \text{val}_1 \ldots \text{val}_n) \\
& \end{align*}
\]

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.

### 13.2 The `let*` Special Form

The syntax of a `let*` special form is nearly identical to that of a `let` special form. However, the semantics is substantially different. In particular, the construction of the local environment happens in a different way. This difference allows a certain kind of incremental computation that turns out to be quite useful.
The syntax of a \texttt{let*} expression. A \texttt{let*} expression has the following form:

\begin{verbatim}
(\texttt{let*} ((\textit{var}_1 \textit{val}_1)
 (\textit{var}_2 \textit{val}_2)
 ... 
 (\textit{var}_n \textit{val}_n))
 \textit{expr}_1
 \textit{expr}_2
 ... 
 \textit{expr}_k)
\end{verbatim}

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

The semantics of a \texttt{let*} expression. A \texttt{let*} expression of the form given above is evaluated as follows:

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

\begin{itemize}
  \item Crucially, the \(i\)th entry in the local environment is created \textit{before} the \((i + 1)\)st value is computed. Thus, the expression, \textit{val}_{i+1}, can refer to \textit{any} of the preceding symbols, \textit{var}_1, \ldots, \textit{var}_i.
  \item Then the expressions in the body of the \texttt{let*} are evaluated, in turn.
  \item The value of the last expression in the body of the \texttt{let*} serves as the value of the entire \texttt{let*} expression.
\end{itemize}

Example 13.7 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 \texttt{let}:

\begin{verbatim}
> (let* ((x 4)
 (y (+ x 2))
 (z (* x y))
 (w (+ x y z)))
 (printf "x: ~A, y: ~A, z: ~A, w: ~A\n" x y z w)

x: 4, y: 6, z: 24, w: 34
68
\end{verbatim}

Notice that the expression, (+ x 2), used to compute the value for \textit{y} refers to the local variable \textit{x}. Similarly, the expression, (* x y), used to compute the value for \textit{z} refers to both \textit{x} and \textit{y}. Finally, the expression, (+ x y z), used to compute the value for \textit{w} refers to \textit{x}, \textit{y} and \textit{z}. Trying to do this with a \texttt{let} expression causes DrScheme to complain.

\begin{verbatim}
> (let ((x 4)
 (y (+ x 2))
 (z (* x y))
 (w (+ x y z)))
 (printf "x: ~A, y: ~A, z: ~A, w: ~A\n" x y z w)
(+ x y z w))
... reference to undefined identifier: x
\end{verbatim}

The reason is due to the difference in the way \texttt{let} and \texttt{let*} expressions are evaluated (i.e., their semantics). In a \texttt{let} expression, \textit{all} of the value expressions are evaluated \textit{first}, \textit{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 \textit{precede} it in the \texttt{let*} expression.

In general, a \texttt{let*} expression of the form,
(let* ((var₁ val₁)
       (var₂ val₂)
       ...
       (varₙ valₙ))
  expr₁
  expr₂
  ...
  exprₙ)

is equivalent to \( n \) nested let expressions:

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

The following example should help to verify this equivalence.

Example 13.8 The following Interactions Window session evaluates a let* expression and the equivalent nested let expression:

```
> (let* ((x 4)
        (y (+ x 2))
        (z (* x y))
        (w (+ x y z)))
  (printf "x: ~A, y: ~A, z: ~A, w: ~A~%
          x y z w)
(+ x y z w))
x: 4, y: 6, z: 24, w: 34
68
```

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

In general, let* provides a simpler syntax than the corresponding set of nested let expressions. Thus, if you ever need to do incremental computations where the value of each local variable depends of the values of the preceding local variables, then you probably will want to use let*.

13.3 The letrec Special Form

To be continued!
14 Lists and List-Based Recursion

We have already seen that an empty list is a primitive datum in Scheme, denoted by (). This section introduces non-empty lists as chains of pairs. In this context, a pair is a data structure that contains two parts, commonly called first and rest. The first part of a pair is used to hold an element of a list; the second part of a pair is used to hold the rest of the list. Thus, the first part of a pair can be any kind of Scheme entity; in contrast, the rest of a pair must be a list (either empty or non-empty). If a pair has its rest equal to the empty list, then that pair is the last link in the chain.

Although it may seem strange to represent a non-empty list in terms of its first part (i.e., the first element of the list) and its rest part (i.e., the rest of the elements in the list), this kind of representation is extremely advantageous because it allows us to define recursive functions on lists. List-based recursion is quite similar to numerical recursion. There is a base case: the empty list (analogous to \( n = 0 \)); and there is a recursive case: a non-empty list (analogous to \( n > 0 \)). Frequently, even quite simple recursive functions can process lists of any length.

To support the use of lists and list-based recursion, Scheme provides four built-in functions:

- **cons**: a function that creates an instance of a pair (i.e., a link in the chain for a non-empty list);
- **cons?**: a type-checker predicate for pairs;
- **first**: a function that provides access to the first part of a pair; and
- **rest**: a function that provides access to the rest of a pair.

Surprisingly, these functions are all that are required to support list-based recursion. Most of the time, we will only require cons, first and rest. To recognize the base case, we will use the null? type-checker predicate, which we have seen before.

14.1 I Thought We Were Already Using Lists...

It is true that the Scheme programming languages uses lists all over the place. We define functions using the define and lambda special forms, which are examples of lists. All of the other special forms are also lists. We also use lists to apply functions to inputs, courtesy of the Default Rule for evaluating non-empty lists. So, why do we need anything else?

These uses of lists have so far enabled us to define functions and apply them to inputs, but they haven’t enabled us to process lists as containers of data. When viewing lists as containers of data, we typically don’t want them to be evaluated. So far, the only way we have seen of shielding a list from evaluation has been by way of the quote special form. However, the quote special form is too limiting because it shields everything in the list from evaluation. For example, in the expression, ‘(a b c), the quote shields the symbols a, b and c from evaluation. However, oftentimes, when creating lists as containers of data, we want to create, for example, lists containing the values of a, b and c. In such cases, we want these symbols to be evaluated. In addition, we have not yet seen any way of accessing the elements of a given list, which is essential to doing any meaningful computation on lists-as-data.

14.2 Non-Empty Lists as Chains of Pairs

The **cons?** Type-Checker Predicate.

⇒ Historically, the pairs that serve as links in the chains of non-empty lists have been called cons cells.

The cons? type-checker predicate has the following contract:

```scheme
;;; CONS?
;;; ------------------------------------------
;;; INPUT: Any Scheme entity
;;; OUTPUT: #t if the input is a cons cell (i.e., a pair);
;;; #f otherwise.
```

The following Interactions Window session uses the cons? type-checker predicate to demonstrate that non-empty lists, which we can easily create using the quote special form, are indeed chains of cons cells, whereas the empty list is something else.
> (cons? '(a b c))
#t
> (cons? '(lambda () "hi"))
#t
> (cons? '(if (> 2 3) 'oh_yeah 'nope))
#t
> (cons? ())
#f

The first and rest Accessor Functions. The first and rest functions are called accessor functions because they enable us to access the parts of a list. The contracts for these functions are given below.

;;; FIRST
;;; ---------------------------------------------------
;;; INPUT: A cons cell (i.e., a non-empty list)
;;; OUTPUT: The first part of the input cons cell.

;;; REST
;;; ---------------------------------------------------
;;; INPUT: A cons cell (i.e., a non-empty list)
;;; OUTPUT: The rest of the input cons cell (i.e., the rest of the list).

The following Interactions Window session uses the first and rest accessor functions to demonstrate that we can access the parts of any given non-empty list.

> (first '(a b c d e))
a
> (rest '(a b c d e))
(b c d e)
> (first '(3 2 1))
3
> (rest '(3 2 1))
(2 1)
> (first '(64))
64
> (rest '(64))()

Notice that the last example, (rest '(64)), demonstrates that the empty list serves to terminate a list.

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

> (first (rest '(a b c d e))) ;; access SECOND ELEMENT
b
> (first (rest (rest '(a b c d e)))) ;; access THIRD element
c
> (first (rest (rest (rest '(a b c d e)))))) ;; access FOURTH element
d

Although we could continue in this fashion, these kinds of examples get ugly pretty quickly. We shall find that when writing recursive functions on lists, we rarely have to make use of such wild combinations of first and rest.

Using cons to Create Pairs (i.e., Cons Cells). The cons function has the following contract:

;;; CONS
;;; ---------------------------------------------------
;;; INPUTS: FST, any Scheme entity
;;; RST, a list (either empty or non-empty)
;;; OUTPUT: A cons cell whose first element is FST and whose
;;; rest element is RST.
Figure 5: The non-empty list, (3 4 6), as a chain of cons cells

Figure 6: An alternative depiction of the non-empty list, (3 4 6)

The following Interactions Window session demonstrates that the cons cells we create using the cons function are treated as non-empty lists by DrScheme.

⇒ The only requirement when using the cons function is that the second input must be a list!

> (cons 8 '(a b c))
(8 a b c)
> (cons 2 '(3 4 a b c))
(2 3 4 a b c)
> (cons 64 ())
(64)

Notice that the list represented by the newly created cons cell contains one more element than the second input. For example, the cons cell (i.e., the non-empty list) created by (cons 8 '(a b c)) contains four elements, one more than (a b c) contains. Similarly, the cons cell (i.e., the non-empty list) created by (cons 64 ()) contains one element, which is one more than the empty list contains.

14.3 Pictures of Non-Empty Lists as Chains of Cons Cells

Fig. 5 shows one way of depicting the non-empty list, (3 4 6), as a chain of cons cells. Notice that the list is indeed represented as a cons cell (the biggest one in the picture). The first element of this cons cell is 3, the rest of this cons cell is itself a cons cell (i.e., non-empty list)—namely, the cons cell whose first element is 4 and whose rest is . . . another cons cell! This innermost cons cell has as its first part, 6, and its rest, () (i.e., the empty list). Since the rest of this cons cell is the empty list, that is the end of the chain. Notice that the list represented by this chain of cons cells has three elements: 3, 4 and 6. Notice, further, that it also has three cons cells!

⇒ A list containing n elements is represented by a chain of n cons cells—one cons cell per element in the list.

Although Fig. 5 is an accurate depiction of a chain of cons cells for 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 chains of cons cells using arrows, as illustrated in Fig. 6. It is important to realize that the non-empty list depicted by this figure is the same list as that depicted in Fig. 5 (i.e., we have two kinds of picture-syntax for one semantic list!).