3

Programming

The Analytical Engine has no pretensions whatever to originate any thing. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with.

Augusta Ada, Countess of Lovelace,
in *Notes on the Analytical Engine*, 1843

What distinguishes a computer from other tools is its *programmability*. Without a program, a computer is an overpriced and not very effective door stopper. With the right program, though, a computer can be a tool for communicating across the continent, discovering a new molecule that can cure cancer, writing and recording a symphony, or managing the logistics of a retail empire.

Golden Gate Bridge

Programming is the act of writing instructions that make the computer do something useful. It is an intensely creative activity, involving aspects of art, engineering, and science. Good programs are written to be executed efficiently by computers, but also to be read and understood by humans. The best programs are delightful in ways similar to the best architecture, elegant in both form and function.

The ideal programmer would have the vision of Issac Newton, the intellect of Albert Einstein, the memory of Joshua Foer, the courage of Amelia Earhart, the determination of Michael Jordan, the pragmatism of Abraham Lincoln, the creativity of Miles Davis, the aesthetic sense of Maya Lin, the wisdom of Benjamin Franklin, the foresight of Garry Kasparov, the hindsight of Edward Gibbon, the writing talents of William Shakespeare, the oratorical skills of Martin Luther King, the audacity of John Roebling, the humility of Socrates, and the self-confidence of Grace Hopper.

Fortunately, it is not necessary to possess all of those rare qualities to be a good programmer! Indeed, anyone who is able to master the intellectual challenge of learning a language (which, presumably, anyone who has gotten this far has done at least for English) can become a good programmer. Since programming is a new way of thinking, many people find it challenging and even frustrating at first. Because the computer does exactly what

it is told, any small mistake in a program may prevent it from working as intended. With a bit of patience and persistence, however, the tedious parts of programming become easier, and you will be able to focus your energies on the fun and creative problem solving parts.

In the previous chapter, we explored the components of language and mechanisms for defining languages. In this chapter, we explain why natural languages are not a satisfactory way for defining procedures and introduce languages for programming computers and how they are used to define procedures.

3.1 Problems with Natural Languages

Natural languages, such as English, work adequately (most, but certainly not all, of the time) for human-human communication, but are not well-suited for human-computer or computer-computer communication. Why can't we use natural languages to program computers?

Next, we survey several of the reasons for this, focusing on specifics from English, although all natural languages suffer from all of these problems to varying degrees.

Complexity. Although English may seem simple to you now, it took many years of intense effort (most of it subconscious) for you to learn it. Despite using it for most of your waking hours for many years (assuming you are a native English speaker), you only know a small fraction of the entire language. The Oxford English Dictionary contains 615,000 words, of which a typical native English speaker knows about 40,000.

Ambiguity. Not only do natural languages have huge numbers of words, most words have many different meanings. To understand which meaning is intended requires knowing the context, and sometimes pure guesswork.

For example, what does it mean to be paid *biweekly*? According to the American Heritage Dictionary [Her07], *biweekly* has two definitions:

1. *Happening every two weeks.*
2. *Happening twice a week; semiweekly.*

Merriam-Webster's Dictionary takes the opposite approach [Onl08]:

1. *occurring twice a week*
2. *occurring every two weeks : fortnightly*

So, depending on which definition is intended, someone who is paid biweekly could either be paid once or four times every two weeks! One would not want the correct behavior of a payroll management program to depend on how biweekly is interpreted.

Even if we can agree on the definition of every word, the meaning of a sentence is often ambiguous. Here is one of my favorite examples, taken from the instructions with a shipment of ballistic missiles from the British Admiralty:

> *It is necessary for technical reasons that these warheads be stored upside down, that is, with the top at the bottom and the bottom at the top. In order that there be no doubt as to which is the bottom and which is the top, for storage purposes, it will be seen that the bottom of each warhead has been labeled 'TOP'.* [Par88]

Irregularity. Because natural languages evolve over time as different cultures interact and speakers misspeak and listeners mishear, natural languages end up a morass of irregularity. Nearly all grammar rules have exceptions. For example, English has a rule that we can make a word plural by appending an *s*. The new word means "more than one of the original word's meaning". This rule works for most words: *word* $\mapsto$ *words*, *language* $\mapsto$ *languages*, *person* $\mapsto$ *persons*.[1] It does not work for *all* words, however. The plural of *goose* is *geese* (and *gooses* is not an English word), the plural of *deer* is *deer* (and *deers* is not an English word), and the plural of *beer* is controversial (and may depend on whether you speak American English or Canadian English). These irregularities can be charming for a natural language, but they are a constant source of difficulty for non-native speakers attempting to learn a language. There is no sure way to predict when the rule can be applied, and it is necessary to memorize each of the irregular forms.

Uneconomic. It requires a lot of space to express a complex idea in a natural language. Many superfluous words are needed for grammatical correctness, even though they do not contribute to the desired meaning. Since natural languages evolved for everyday communication, they are not well suited to describing the precise steps and decisions needed in a computer program.

I didn't have time to write a short letter, so I wrote a long one instead.
Mark Twain

As an example, consider a procedure for finding the maximum of two numbers. In English, we could describe it like this:

> *To find the maximum of two numbers, compare them. If the first number is greater than the second number, the maximum is the first number. Otherwise, the maximum is the second number.*

[1] Or is it *people*? What is the singular of *people*? What about *peeps*? Can you only have one *peep*?

Perhaps shorter descriptions are possible, but any much shorter description probably assumes the reader already knows a lot. By contrast, we can express the same steps in the Scheme programming language in very concise way: (**define** (*bigger a b*) (**if** (> *a b*) *a b*)). (Don't worry if this doesn't make sense yet—it should by the end of this chapter.)

Limited means of abstraction. Natural languages provide small, fixed sets of pronouns to use as means of abstraction, and the rules for binding pronouns to meanings are often unclear. As discussed in Section 2.2, the means of abstraction available in English are particularly poor. Since programming often involves using simple names to refer to complex things, we need more powerful means of abstraction than natural languages provide.

3.2 Programming Languages

For programming computers, we want languages that are simple, unambiguous, regular, economical, and that provide more powerful means of abstraction. A *programming language* is a language that is designed to be read and written by humans to create programs that can be executed by computers.

Programming languages come in many flavors. It is difficult to simultaneously satisfy all the goals, in particular simplicity is often at odds with economy and powerful means of abstraction. Every feature that is added to a language to increase its expressiveness incurs a cost in reducing simplicity and regularity.

Another reason there are many different programming languages is that they are at different *levels of abstraction*. Some languages provide programmers with detailed control over machine resources, such as selecting a particular location in memory where a value is stored. Other languages hide most of the details of the machine operation from the programmer, allowing them to focus on higher-level actions.

Ultimately, we want a program the computer can execute. This means at the lowest level we need languages the computer can understand directly. At this level, the program is just a sequence of bits encoding machine instructions. Code at this level is not easy for humans to understand or write, but it is easy for a processor to execute quickly. The machine code encodes instructions that direct the processor to take simple actions like moving data from one place to another, performing simple arithmetic, and jumping around to find the next instruction to execute.

For example, the bit sequence `1110101111111110` encodes an instruction

in the Intel x86 instruction set (used on most PCs) that tells the processor to jump backwards two locations. Since two locations is the amount of space needed to hold this instruction, jumping back two locations actually jumps back to the beginning of this instruction. Hence, it gets stuck running forever without making any progress. The computer's processor is designed to execute very simple instructions like this one. This means each instruction can be executed very quickly. A typical modern processor can execute *billions* of instructions in a single second.[2]

Grace Hopper, 1952

Image courtesy Computer History Museum

Until the early 1950s, all programming was done at the level of simple instructions. The problem with instructions at this level is that they are not easy for humans to write and understand, and you need many simple instructions before you have a useful program.

In the early 1950s, Admiral Grace Hopper developed the first compilers. A *compiler* is a computer program that generates other programs. It can translate an input program written in a high-level language that is easier for humans to create into a program in a machine-level language that is easier for a computer to execute.

An alternative to a compiler is an interpreter. An *interpreter* is a tool that translates between a higher-level language and a lower-level language, but where a compiler translates an entire program at once and produces a machine language program that can be executed directly, an interpreter interprets the program a small piece at a time while it is running. This has the advantage that we do not have to run a separate tool to compile a program before running it; we can simply enter our program into the interpreter and run it right away. This makes it easy to make small changes to a program and try it again, and to observe the state of our program as it is running.

Nobody believed that I had a running compiler and nobody would touch it. They told me computers could only do arithmetic.
Grace Hopper

A disadvantage of using an interpreter instead of a compiler is that because the translation is happening while the program is running, the program may execute much slower than a similar compiled program would. Another advantage of compilers over interpreters is that since the compiler translates the entire program it can also analyze the program for consistency and detect certain types of programming mistakes automatically instead of encountering them when the program is running (or worse, not detecting them at all and producing unintended results). This is especially important when writing large, critical programs such as flight control software — we want to detect as many problems as possible in the flight control software before the plane is flying!

[2] When a computer is marketed as a "2GHz processor" that means the processor executes 2 billion cycles per second. This does not map directly to the number of instructions it can execute in a second, though, since some instructions take several cycles to execute.

3.3 Scheme

For now, we are more concerned with interactive exploration than with performance and detecting errors early, so we use an interpreter instead of a compiler. The programming system we use for the first part of this book is depicted in Figure 3.1.

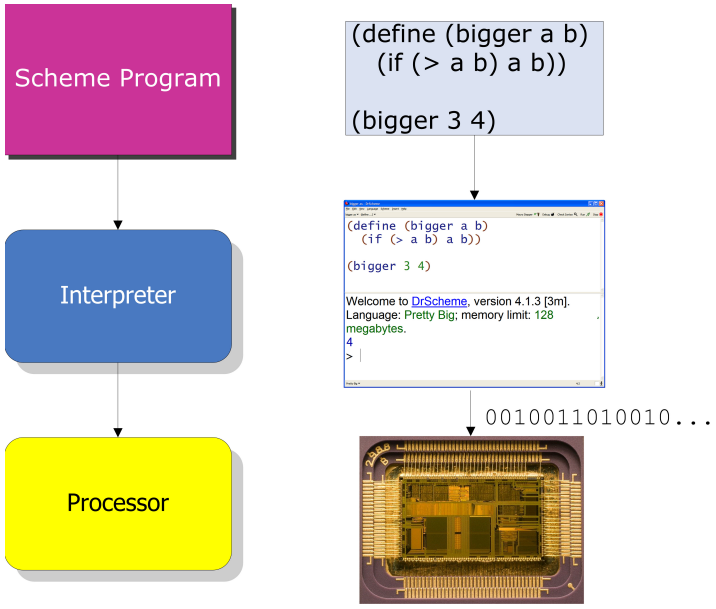


Figure 3.1. Running a Scheme program.

The input to our programming system is a program written in a programming language named *Scheme*.[3] Scheme was developed at MIT in the 1970s by Guy Steele and Gerald Sussman, based on the LISP programming language that was developed by John McCarthy in the 1950s. A Scheme interpreter interprets a Scheme program and executes it on the machine processor.

Although Scheme is not widely used in industry, it is a great language for learning about computing and programming. The primary advantage of using Scheme to learn about computing is its simplicity and elegance. The language is simple enough that you will learn nearly the entire language by the end of this chapter (we defer describing a few aspects until Chapter 11), and by the end of this book you will know enough to implement your own Scheme interpreter. By contrast, some programming languages that are widely used in industrial programming such as C++ and Java re-

[3]Originally, it was named "Schemer", but the machine used to develop it only supported 6-letter file names, so the name was shortened to "Scheme".

quire thousands of pages to describe, and even the world's experts in those languages do not agree on exactly what all programs mean.

Although almost everything we describe should work in all Scheme interpreters, for the examples in this book we assume the DrScheme programming environment which is freely available from *http://www.drscheme.org/*. DrScheme includes interpreters for many different languages, so you must select the desired language using the Language menu. The selected language defines the grammar and evaluation rules that will be used to interpret your program. For all the examples in this book, we use the language named Pretty Big.

3.4 Expressions

Scheme programs are composed of expressions and definitions (Section 3.5). An *expression* is a syntactic element that has a *value*. The act of determining the value associated with an expression is called *evaluation*. A Scheme interpreter, such as the one provided in DrScheme, is a machine for evaluating Scheme expressions. If you enter an expression to a Scheme interpreter, it responds by displaying the value of that expression.

Expressions may be primitives. Scheme also provides means of combination for producing complex expressions from simple expressions. The next subsections describe primitive expressions and application expressions. Section 3.6 describes expressions for making procedures and Section 3.7 describes expressions that can be used to make decisions.

3.4.1 Primitives

An expression can be replaced with a primitive:

$$\textit{Expression} ::\Rightarrow \textit{PrimitiveExpression}$$

As with natural languages, primitives are the smallest units of meaning. Hence, the value of a primitive is its pre-defined meaning.

Scheme provides many different primitives. Three useful types of primitives are described next: numbers, Booleans, and primitive procedures.

Numbers. Numbers represent numerical values. Scheme provides all the kinds of numbers you are familiar with, and they mean almost exactly what

you think they mean.[4]

Example numbers include:

```
150        0     −12
3.14159    3/4   999999999999999999999
```

Numbers evaluate to their value. For example, the value of the primitive expression 150 is 150.

Booleans. Booleans represent truth values. There are two primitives for representing true and false:

$$\textit{PrimitiveExpression} ::\Rightarrow \texttt{true} \mid \texttt{false}$$

Unsurprisingly, the meaning of true is true, and the meaning of false is false.[5]

function

Primitive Procedures. Scheme provides primitive procedures corresponding to many common functions. Mathematically, a *function* is a mapping from inputs to outputs. A function has a *domain*, the set of all inputs that it accepts. For each input in the domain, there is exactly one associated output. For example, + is a procedure that takes zero or more inputs, each of which must be a number. The output it produces is the sum of the values of the inputs. (We cover how to apply a function in the next subsection.)

Table 3.1 describes some of the primitive procedures.

[4]The details of managing numbers on computers are complex, and we do not consider them here.

[5]In the DrScheme interpreter, #t and #f are used as the primitive truth values; they mean the same thing as true and false. So, when you evaluate something that evaluates to true, it will appear as #t in the interactions window.

Symbol	Description	Inputs	Output
+	add	zero or more numbers	sum of the input numbers (0 if there are no inputs)
*	multiply	zero or more numbers	product of the input numbers (1 if there are no inputs)
−	subtract	two numbers	the value of the first number minus the value the second number
/	divide	two numbers	the value of the first number divided by the value of the second number
zero?	is zero?	one number	true if the input value is 0, otherwise false
=	is equal to?	two numbers	true if the input values have the same value, otherwise false
<	is less than?	two numbers	true if the first input value has lesser value than the second input value, otherwise false
>	is greater than?	two numbers	true if the first input value has greater value than the second input value, otherwise false
<=	is less than or equal to?	two numbers	true if the first input value is not greater than the second input value, otherwise false
>=	is greater than or equal to?	two numbers	true if the first input value is not less than the second input value, otherwise false

Table 3.1. Selected Scheme Primitive Procedures.
All of these primitive procedures operate on numbers. The first four are the basic arithmetic operators; the rest are comparison procedures. Some of these procedures are defined for more inputs than just the ones shown here. For example, the subtract procedure also works on one number, producing its negation.

3.4.2 Application Expressions

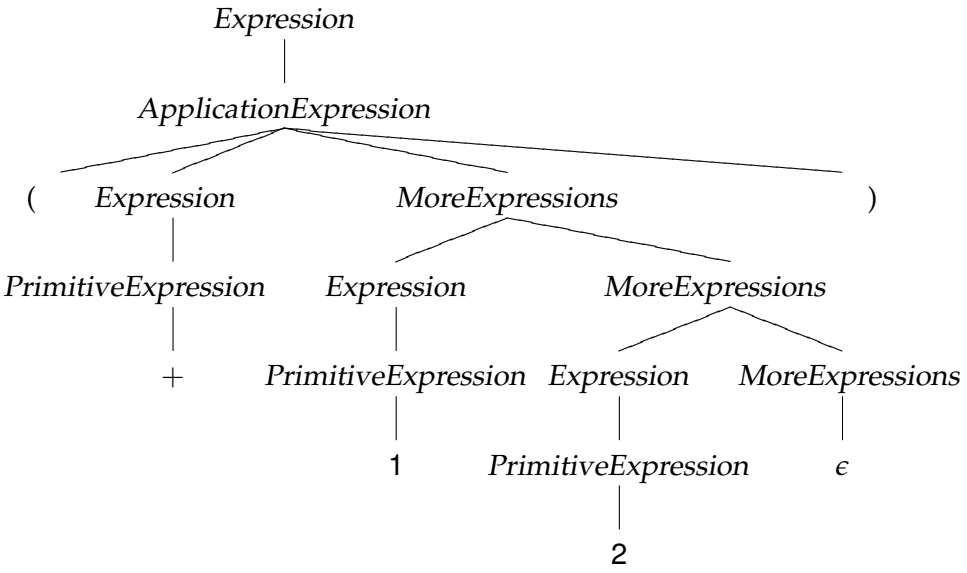
Most of the actual work done by a Scheme program is done by application expressions. The grammar rule for application is:

$$
\begin{array}{lll}
\textit{Expression} & ::\Rightarrow & \textit{ApplicationExpression} \\
\textit{ApplicationExpression} & ::\Rightarrow & (\textit{Expression MoreExpressions}) \\
\textit{MoreExpressions} & ::\Rightarrow & \epsilon \mid \textit{Expression MoreExpressions}
\end{array}
$$

This rule generates a list of one or more expressions surrounded by parentheses. The value of the first expression should be a procedure. All of the primitive procedures are procedures; in Section 3.6, we will see how to create new procedures. The remaining expressions are the inputs to the procedure.

For example, the expression (+ 1 2) is an *ApplicationExpression*, consisting of three subexpressions. Although this example is probably simple enough that you can probably guess that it evaluates to 3, we will demonstrate in detail how it is evaluated by breaking down into its subexpressions using the grammar rules. The same process will allow us to understand how *any* expression is evaluated.

Here is a parse tree for the expression (+ 1 2):

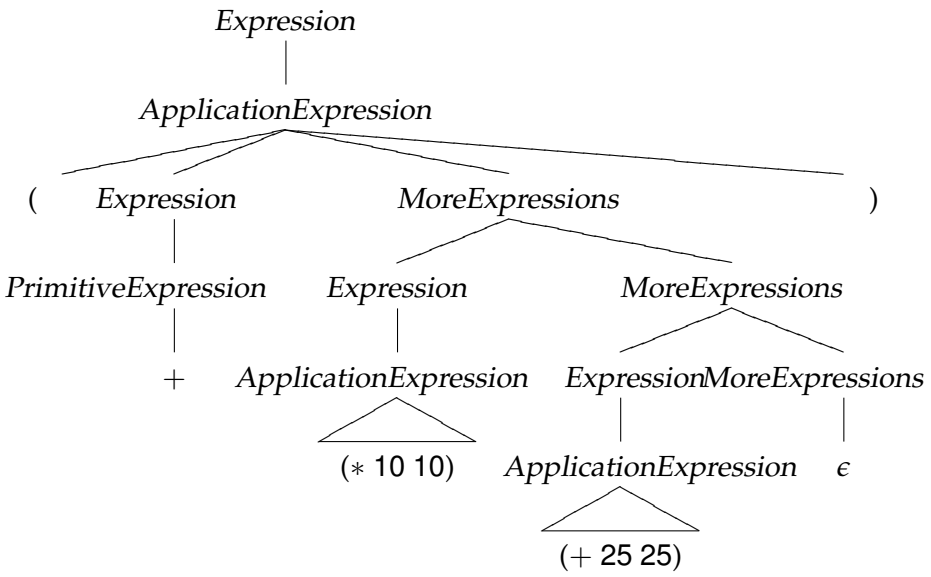


Following the grammar rules, we replace *Expression* with *ApplicationExpression* at the top of the parse tree. Then, we replace *ApplicationExpres-*

sion with (*Expression MoreExpressions*). The *Expression* term is replaced *PrimitiveExpression*, and finally, the primitive addition procedure $+$. This is the first subexpression of the application, so it is the procedure to be applied. The *MoreExpressions* term produces the two operand expressions: 1 and 2, both of which are primitives that evaluate to their own values. The application expression is evaluated by applying the value of the first expression (the primitive procedure $+$) to the inputs given by the values of the other expressions. Following the meaning of the primitive procedure, $(+\ 1\ 2)$ evaluates to 3 as expected.

As with any nonterminal, the *Expression* nonterminals in the application expression can be replaced with anything that appears on the right side of an expression rule, including the application expression rule. Hence, we can build up complex expressions like $(+\ (*\ 10\ 10)\ (+\ 25\ 25))$.

The parse tree is:



This tree is similar to the previous tree, except instead of the subexpressions of the first application expression being simple primitive expressions, they are now application expressions. (Instead of showing the complete parse tree for the nested application expressions, we use triangles.)

To evaluate the output application, we need to evaluate all the subexpressions. The first subexpression, $+$, evaluates to the primitive procedure. The second subexpression, $(*\ 10\ 10)$, evaluates to 100, and the third expression, $(+\ 25\ 25)$, evaluates to 50. Now, we can evaluate the original expression using the values for its three component subexpressions: $(+\ 100\ 50)$ evaluates to 150.

Exercise 3.1. Draw a parse tree for the Scheme expression

$$(+\ 100\ (*\ 5\ (+\ 5\ 5)))$$

and show how it would be evaluated.

Exercise 3.2. Predict how each of the following Scheme expressions is evaluated. After making your prediction, try evaluating the expression in DrScheme. If the result is different from your prediction, explain why the Scheme interpreter evaluates the expression as it does.

a. 150

b. $(+\ 150)$

c. $(+\ (+\ 100\ 50)\ (*\ 2\ 0))$

d. $(=\ (+\ 100\ 50)\ (*\ 15\ (+\ 5\ 5)))$

e. $(zero?\ (-\ 150\ (+\ 50\ 50\ (+\ 25\ 25))))$

f. $+$

g. [⋆] $(+\ +\ <)$

Exercise 3.3. For each problem, construct a Scheme expression that calculates the result and try evaluating it in DrScheme.

a. How many seconds are there in a year?

b. For how many seconds have you been alive?

c. For what fraction of your life have you been in school?

Exercise 3.4. Construct a Scheme expression to calculate the distance in inches that light travels during the time it takes the processor in your computer to execute one cycle.

A meter is defined as the distance light travels in $1/299792458^{th}$ of a second in a vacuum. One meter is 100 centimeters, and one inch is defined as 2.54 centimeters. Your processor speed is probably given in *gigahertz* (GHz), which are 1,000,000,000 hertz. One hertz means once per second, so 1GHz means the processor executes 1,000,000,000 cycles per second. On a Windows machine, you can find the speed of your processor by opening the Control Panel (select it from the Start menu) and selecting System. Note that Scheme performs calculations exactly, so the result will be displayed as a fraction. To see a more useful answer, use (*exact->inexact Expression*) to convert the value of the expression to a decimal representation.

3.5 Definitions

Scheme provides a simple, yet powerful, mechanism for abstraction. We can introduce a new name using a definition:

$$\textit{Definition} ::\Rightarrow \text{ (}\textbf{define}\ \textit{Name}\ \textit{Expression}\text{)}$$

After a definition, the name in the definition is now associated with the value of the expression in the definition.[6] A definition is not an expression since it does not evaluate to a value.

A name can be any sequence of letters, digits, and special characters (such as $-$, $>$, *?*, and *!*) that starts with a letter or special character. Examples of valid names include *a*, *Ada*, *Augusta-Ada*, *gold49*, *!yuck*, and *yikes!*\%@\#. We don't recommend using some of these names in your programs, however! A good programmer will pick names that are easy to read, pronounce, and remember, and that are not easily confused with other names.

After a name has been bound to a value by a definition, that name may be used in an expression:

$$\begin{array}{ll} \textit{Expression} & ::\Rightarrow \textit{NameExpression} \\ \textit{NameExpression} & ::\Rightarrow \textit{Name} \end{array}$$

The value of a *NameExpression* is the value associated with the name.

For example, below we define *speed-of-light* to be the speed of light in meters per second, define *seconds-per-hour* to be the number of seconds in an hour, and use them to calculate the speed of light in kilometers per hour:

```
> (define speed-of-light 299792458)
> speed-of-light
299792458
> (define seconds-per-hour (* 60 60))
> (/ (* speed-of-light seconds-per-hour) 1000)
1079252848 4/5
```

[6] Alert readers should be worried that we need a more precise definition of the meaning of definitions to know what it means for a value to be associated with a name. This one will serve us well for now, but we will provide a more precise explanation of the meaning of a definition in Chapter 11.

3.6 Procedures

In Chapter 1 we defined a procedure as a description of a process. Scheme provides a way to define procedures that take inputs, carry out a sequence of actions, and produce an output. In Section 3.4.1, we saw that Scheme provides some primitive procedures. To construct complex programs, however, we need to be able to create our own procedures.

Procedures are similar to mathematical functions in that they provide a mapping between inputs and outputs, but they are different from mathematical functions in two key ways:

- State — in addition to producing an output, a procedure may access and modify state. This means that even when the same procedure is applied to the same inputs, the output produced may vary. Because mathematical functions do not have external state, when the same function is applied to the same inputs it always produces the same result. State makes procedures much harder to reason about. In particular, it breaks the substitution model of evaluation we introduce in the next section. We will ignore this issue until Chapter 11, and focus until then only on procedures that do not involve any state.

- Resources — unlike an ideal mathematical function, which provides an instantaneous and free mapping between inputs and outputs, a procedure requires resources to execute before the output is produced. The most important resources are *space* (memory) and *time*. A procedure may need space to keep track of intermediate results while it is executing. Each step of a procedure requires some time to execute. Predicting how long a procedure will take to execute, and finding the fastest procedure possible for solving some problem, are core problems in computer science. We will consider this throughout this book, and in particular in Chapter 9. Even knowing if a procedure will finish (that is, ever produce an output) is a challenging problem. In Chapter 13 we will see that it is impossible to solve in general.

For the rest of this chapter, however, we will view procedures as idealized mathematical functions: we will consider only procedures that involve no state, and we will not worry about the resources our procedures require.

3.6.1 Making Procedures

Scheme provides a general mechanism for making a procedure:

Expression	::⇒	*ProcedureExpression*
ProcedureExpression	::⇒	(**lambda** (*Parameters*) *Expression*)
Parameters	::⇒	ϵ | ***Name*** *Parameters*

Evaluating a *ProcedureExpression* produces a procedure that takes as inputs the *Parameters* following the **lambda**.[7] You can think of **lambda** as meaning "make a procedure". The body of the procedure is the *Expression*, which is not evaluated until the procedure is applied.

Note that a *ProcedureExpression* can replace an *Expression*. This means anywhere an *Expression* is used we can create a new procedure. This is very powerful since it means we can use procedures as inputs to other procedures and create procedures that return new procedures as their output!

Here are some example procedures:

- (**lambda** (*x*) (∗ *x x*)) — a procedure that takes one input, and produces the square of the input value as its output.
- (**lambda** (*a b*) (+ *a b*)) — a procedure that takes two inputs, and produces the sum of the input values as its output.
- (**lambda** () 0) — a procedure that takes no inputs, and produces 0 as its output.
- (**lambda** (*a*) (**lambda** (*b*) (+ *a b*))) — a procedure that takes one input (*a*), and produces as its output a procedure that takes one input and produces the sum of that input at *a* as its output. We can think of this procedure as a procedure that makes an adding procedure.

3.6.2 Substitution Model of Evaluation

For a procedure to be useful, we need to apply it. In Section 3.4.2, we saw the syntax and evaluation rule for an *ApplicationExpression* when the procedure to be applied is a primitive procedure. The syntax for applying a constructed procedure is identical to the syntax for applying a primitive procedure:

Expression	::⇒	*ApplicationExpression*
ApplicationExpression	::⇒	(*Expression MoreExpressions*)
MoreExpressions	::⇒	ϵ | *Expression MoreExpressions*

[7] Scheme uses **lambda** to make a procedure because it is based on LISP which is based on Lambda Calculus (see Chapter 17).

To understand how constructed procedures are evaluated, we need a new evaluation rule. In this case, the first *Expression* evaluates to a procedure that was created using a *ProcedureExpression*, so we can think of the *ApplicationExpression* as:

ApplicationExpression ::⇒
((**lambda** (*Parameters*)*Expression*) *MoreExpressions*)

(The underlined part is the replacement for the *ProcedureExpression*.)

To evaluate the application, we evaluate the *MoreExpressions* in the application expression. These expressions are known as the *operands* of the application. The resulting values are the input to the procedure. There must be exactly one expression in the *MoreExpressions* corresponding to each name in the parameters list. Next, evaluate the expression that is the body of the procedure. Whenever any parameter name is used inside the body expression, the name evaluates to the value of the corresponding input. This is similar to the way binding worked in Post production systems (Section 2.3). When a value is matched with a procedure parameter, that parameter is bound to the value. When the parameter name is evaluated, the result is the bound value.

Example 3.1: Square. Consider evaluating the following expression, which apples the squaring procedure to 2:

((**lambda** (*x*) (∗ *x x*)) 2)

It is an *ApplicationExpression* where the first sub-expression is the *ProcedureExpression*, (**lambda** (*x*) (∗ *x x*)). To evaluate the application, we evaluate all the subeexpressions and apply the value of the first subexpression to the values of the remaining subexpressions. The first subexpression evaluates to a procedure that takes one parameter named *x* and has the expression body (∗ *x x*). There is one operand expression, the primitive 2, that evaluates to 2.

To evaluate the application we bind the first parameter, *x*, to the value of the first operand, 2, and evaluate the procedure body, (∗ *x x*). After substituting the parameter values, we have (∗ 2 2). This is an application of the primitive multiplication procedure. Evaluating the application results in the value 4.

The procedure in our example, (**lambda** (*x*) (∗ *x x*)), is a procedure that takes a number as input and as output produces the square of that number. We

can use the definition mechanism (from Section 3.5) to give this procedure a name so we can reuse it:

(**define** *square* (**lambda** (*x*) (∗ *x x*)))

This defines the name *square* as the procedure. After this, we can use *square* to produce the square of any number:

```
> (square 2)
4
> (square 1/4)
1/16
> (square (square 2))
16
```

Example 3.2: Make adder. For the make an adding procedure example,

((**lambda** (*a*) (**lambda** (*b*) (+ *a b*))) 3)

produces a procedure that adds 3 to its input. Applying that procedure,

(((**lambda** (*a*) (**lambda** (*b*) (+ *a b*))) 3) 4)

evaluates to 7. By using **define**, we can give these procedures sensible names:

```
(define make-adder
        (lambda (a)
                (lambda (b) (+ a b))))
```

Then,

(**define** *add-three* (*make-adder* 3))

defines *add-three* as a procedure that takes one parameter and outputs the value of that parameter plus 3.

Abbreviated Procedure Definitions. Since we commonly need to define new procedures, Scheme provides a condensed notation for defining a procedure[8]:

[8]The condensed notation also includes a begin expression, which is a special form. We will not need the begin expression until we start dealing with procedures that have side-effects. We describe the **begin** special form in Chapter 11.

$$\textit{Definition} ::\Rightarrow \ (\textbf{define}\ (\textit{Name}\ \textit{Parameters})\ \textit{Expression})$$

This incorporates the **lambda** invisibly into the definition, but means exactly the same thing. For example,

(**define** *square* (**lambda** (*x*) (∗ *x x*)))

can be written equivalently as:

(**define** (*square x*) (∗ *x x*))

The two definitions mean exactly the same thing.

Exercise 3.5. Define a procedure, *cube*, that takes one number as input and produces as output the cube of that number.

Exercise 3.6. Define a procedure, *compute-cost*, that takes as input two numbers, the first represents that price of an item, and the second represents the sales tax rate. The output should be the total cost, which is computed as the price of the item plus the sales tax on the item, which is its price times the sales tax rate. For example, (*compute-cost* 13 0.05) should evaluate to 13.65.

3.7 Decisions

We would like to be able to make procedures where the actions taken depend on the input values. For example, we may want a procedure that takes two numbers as inputs and evaluates to the maximum value of the two inputs. To define such a procedure we need a way of making a decision. A *predicate* is a test expression that is used to determine which actions to take next. Scheme provides the **if** expression for determining actions based on a predicate.

The *IfExpression* replacement has three *Expression* terms. For clarity, we give each of them names as denoted by the subscripts:

$$\begin{aligned} \textit{Expression} &::\Rightarrow \textit{IfExpression} \\ \textit{IfExpression} &::\Rightarrow (\textbf{if}\ \textit{Expression}_{\text{Predicate}} \\ &\qquad\qquad \textit{Expression}_{\text{Consequent}} \\ &\qquad\qquad \textit{Expression}_{\text{Alternate}}) \end{aligned}$$

The evaluation rule for an *IfExpression* is to first evaluate $\textit{Expression}_{\text{Predicate}}$, the predicate expression. If it evaluates to any non-false value, the value of

the *IfExpression* is the value of $Expression_{\text{Consequent}}$, the consequent expression, and the alternate expression is not evaluated at all. If the predicate expression evaluates to false, the value of the *IfExpression* is the value of $Expression_{\text{Alternate}}$, the alternate expression, and the consequent expression is not evaluated at all. The predicate expression determines which of the two following expressions is evaluated to produce the value of the *IfExpression*.

Note that if the value of the predicate is *anything* other than false, the consequent expression is used. For example, if the predicate evaluates to true, to a number, or to a procedure the consequent expression is evaluated.

The if-expression is a *special form*. This means that although it looks syntactically identical to an application (that is, it could be an application of a procedure named **if**), it is not evaluated as a normal application would be. Instead, we have a special evaluation rule for if-expressions. The reason a special rule is needed is because we do not want all the subexpressions to be evaluated. With the normal application rule, all the subexpressions are evaluated, and then the procedure resulting from the first subexpression is applied to the values resulting from the others. With the if special form evaluation rule, the predicate expression is always evaluated, but only one of the following subexpressions is evaluated depending on the result of evaluating the predicate expression.

This means an if-expression can evaluate to a value even if evaluating one of its subexpressions would produce an error. For example,

(**if** (> 3 4) (∗ + +) 7)

evaluates to 7 even though evaluating the subexpression (∗ + +) would produce an error. Because of the special evaluation rule for if-expressions, the consequent expression is never evaluated.

Example 3.3: Bigger. Now that we have procedures, decisions, and definitions, we can understand the *bigger* procedure from the beginning of the chapter. The definition,

(**define** (*bigger a b*) (**if** (> *a b*) *a b*))

is a condensed procedure definition. It is equivalent to:

(**define** *bigger* (**lambda** (*a b*) (**if** (> *a b*) *a b*)))

This defines the name *bigger* as the value of evaluating the procedure expression (**lambda** (*a b*) (**if** (> *a b*) *a b*)). This is a procedure that takes two

inputs, named *a* and *b*. Its body is an if-expression with predicate expression (> *a b*). The predicate expression compares the value that is bound to the first parameter, *a*, with the value that is bound to the second parameter, *b*, and evaluates to true if the value of the first parameter is greater, and false otherwise. According to the evaluation rule for an if-expression, if the predicate evaluates to any non-false value (in this case, true), the value of the if-expression is the value of the consequent expression, *a*. If the predicate evaluates to false, the value of the if-expression is the value of the alternate expression, *b*. Hence, our *bigger* procedure takes two numbers as inputs and produces as output the greater of the two inputs.

Exercise 3.7. Follow the evaluation and application rules to evaluate the following Scheme expression:

(*bigger* 3 4)

where *bigger* is the maximum procedure defined as,

(**define** *bigger* (**lambda** (*a b*) (**if** (> *a b*) *a b*)))

It is very tedious to follow all of the steps (that's why we normally rely on computers to do it!), but worth doing once to make sure you understand the evaluation rules.

Exercise 3.8. Define a procedure, *xor*, that implements the logical exclusive-or operation. The *xor* function takes two inputs, and outputs true if exactly one of those outputs has a true value. Otherwise, it outputs false. For example, (*xor* true true) should evaluate to false and (*xor* (< 3 5) (= 8 8)) should evaluate to true.

Exercise 3.9. Define a procedure, *abs*, that takes a number as input and produces the absolute value of that number as its output. For example, (*abs* 3) should evaluate to 3, (*abs* −150) should evaluate to 150, and (*abs* 0) should evaluate to 0.

Exercise 3.10. [⋆] Define a procedure, *bigger-magnitude*, that takes two inputs, and produces as output the value of the input with the maximum magnitude (that is, absolute distance from zero). For example,

(*bigger-magnitude* 5 −7)

should evaluate to −7, and

(*bigger-magnitude* 9 −3)

should evaluate to 9.

Exercise 3.11. [⋆] Define a procedure, *biggest*, that takes three inputs, and produces as output the maximum value of the three inputs. For example,

(*biggest* 5 7 3)

should evaluate to 7. Try to find at least two different ways to define *biggest*, one using *bigger*, and one without using it.

3.8 Summary

At this point, we have covered enough of Scheme to write useful programs (even if the programs we have seen so far seem rather dull). In fact (as we will see in Chapter 17), we have covered enough to express *every* possible computation! We just need to combine the constructs we know in more complex ways to perform more interesting computations. The next chapter, and much of the rest of this book, focuses on ways to combine the constructs for making procedures, making decisions, and applying procedures in more powerful ways.

Here we summarize the grammar rules and evaluation rules. Each grammar rule has an associated evaluation rule. This means that any Scheme fragment that can be described by the grammar also has an associated meaning that can be produced by combining the evaluation rules corresponding to the grammar rules.

Program	::⇒	ϵ | *ProgramElement Program*
ProgramElement	::⇒	*Expression* | *Definition*

A program is a sequence of expressions and definitions.

Definition	::⇒	(**define** ***Name*** *Expression*)

A definition evaluates the expression, and associates the value of the expression with the name.

Definition	::⇒	(**define** (***Name*** *Parameters*) *Expression*)

Abbreviation for (**define** ***Name*** (**lambda** *Parameters*) *Expression*)

Expression ::⇒ *PrimitiveExpression* | *NameExpression* | *ApplicationExpression* | *ProcedureExpression* | *IfExpression*

The value of the expression is the value of the replacement expression.

PrimitiveExpression ::⇒ ***Number*** | true | false | *primitive procedure*

Evaluation Rule 1: Primitives. A primitive expression evaluates to its pre-defined value.

NameExpression ::⇒ ***Name***

Evaluation Rule 2: Names. A name evaluates to the value associated with that name.

ApplicationExpression ::⇒ (*Expression MoreExpressions*)

Evaluation Rule 3: Application. To evaluate an application expression:

a. **Evaluate** all the subexpressions;

b. Then, **apply** the value of the first subexpression to the values of the remaining subexpressions.

MoreExpressions ::⇒ ϵ | *Expression MoreExpressions*
ProcedureExpression ::⇒ (**lambda** (*Parameters*) *Expression*)

Evaluation Rule 4: Lambda. Lambda expressions evaluate to a procedure that takes the given parameters and has the expression as its body.

Parameters ::⇒ ϵ | ***Name*** *Parameters*

IfExpression ::⇒ (**if** $Expression_{\text{Predicate}}$ $Expression_{\text{Consequent}}$ $Expression_{\text{Alternate}}$)

Evaluation Rule 5: If. To evaluate an if-expression, (a) evaluate the predicate expression; then, (b) if the value of the predicate expression is a false value then the value of the if-expression is the value of the alternate expression; otherwise, the value of the if-expression is the value of the consequent expression.

The evaluation rule for an application (Rule 3b) uses **apply** to perform the application. We define **apply** using the two application rules:

- **Application Rule 1: Primitives.** If the procedure to apply is a primitive procedure, just do it.
- **Application Rule 2: Constructed Procedures.** If the procedure to apply is a constructed procedure, **evaluate** the body of the procedure with each parameter name bound to the corresponding input expression value.

Note that **evaluate** in the Application Rule 2 means use the evaluation rules above to evaluate the expression. Thus, the evaluation rules are defined using the application rules, which are defined using the evaluation rules! This appears to be a circular definition, but as with the grammar examples, it has a base case. There are some expressions we can evaluate without using the application rules (e.g., primitive expressions, name expressions), and some applications we can evaluate without using the evaluation rules (when the procedure to apply is a primitive). Hence, the process of evaluating an expression will sometimes finish and when it does we end with the value of the expression.[9]

[9]This does not guarantee it will *always* finish, however! We will see in some examples in the next chapter where evaluation never finishes.