

Chapter 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. This it is calculated to effect primarily and chiefly of course, through its executive faculties; but it is likely to exert an indirect and reciprocal influence on science itself in another manner. For, in so distributing and combining the truths and the formula of analysis, that they may become most easily and rapidly amenable to the mechanical combinations of the engine, the relations and the nature of many subjects in that science are necessarily thrown into new lights, and more profoundly investigated.

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.

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. The best programs are written to be executed efficiently by computers, but also to be read and understood by humans. The ideal programmer would have the vision of Issac Netwon, the intellect of Albert Einstein, the memory of Joshua Foer, the courage of Amelia Earhart, the determination of Michael Jordan, the political savvy 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 pragmatism of Abraham Lincoln, 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 language¹ 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 become easier, and you will be able to focus your energies on the fun and creative problem solving parts.

3.1 Problems with Natural Languages

Natural languages, such as English, work reasonably well for human-human communication, but are not well-suited for human-computer or computer-computer communication. There are many reasons for this including:

Complexity. Although English may seem simple to you now (at least compared to learning a new language!), it took many years of intense learning for you to learn it. Despite all those years of effort, 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 (Fourth Edition), *biweekly* has two definitions:

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

¹Which, presumably, anyone who has gotten this far has done, at least in English!

So, depending on which definition is intended, someone who is paid biweekly could either be paid once or four times every two weeks!

Even if we can agree on the definition of every word, the meaning of English sentences is often ambiguous. Here is one of my favorite examples, taken from the instructions with a shipment of ballistic missiles from the British Admiralty:²

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'.

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 adding an *s*. The new word means “more than one of the original word’s meaning” (actually, even the standard rule is complicated, since it may also be used to mean zero of them). This rule works for most words: *word* :: \Rightarrow *words*, *language* :: \Rightarrow *languages*, *person* :: \Rightarrow *persons*. It does not work for others, 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 English³). These irregularities may be charming for a natural language, but they are a constant source of difficulty.

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

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.

As an example, consider a process 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.

²Reported in The Hummus Report (<http://www.textfiles.com/magazines/HUMUS/humus.005>).

³See http://crofsblogs.typepad.com/english/2005/06/beer_or_beers.html.

Perhaps shorter descriptions are possible, but any much shorter description probably assumes the reader knows a lot already. By contrast, we can express the same steps in Scheme in very concise way⁴:

```
(define (max a b) (if (> a b) a b))
```

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. 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

Hence, natural languages are not well suited to programming computers. Instead, we need languages that are simple, less ambiguous, more regular, more economic, and that provide more powerful means of abstraction than natural languages. A *programming language* is a language that is designed to be read and written by humans to create programs that can be executed by computers⁵.

Programming languages come in many flavors. One reason for this is that they are at different levels of abstraction. 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 zeros and ones (e.g., 1110101111111110...). 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 sequence of zeros and ones encodes an instruction for the Intel x86 processor (used on most PCs) that tells the processor to jump backwards two locations. In fact, two locations is the amount of space needed to hold this instruction, so jumping back two locations actually jumps back to the beginning of this instruction (hence, it gets stuck running forever without making any progress).

⁴Don't worry if this doesn't make sense yet. It should by the end of this chapter.

⁵We will provide a more precise definition of *programming language* in Chapter ??, after we have a formal model of a computer.

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.⁶

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.

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!

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

⁶When 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 is depicted in Figure 3.3. The input to our programming system is a program written in the Scheme programming language. Scheme⁷ 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.

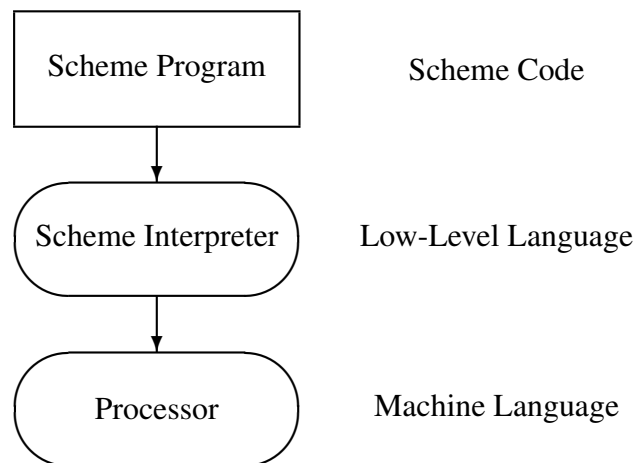


Figure 3.1: **Levels of Abstraction.**

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 ??), and by the end of this book you will know enough to implement your own Scheme interpreter. By contrast, programming languages such as C++ and Java require thousands of pages to describe, and even the world’s experts in those languages do not agree on exactly what all programs mean.

⁷Originally, it was named “Schemer”, but the machine used to develop it only supported 6-letter file names, so the name was shortened to “Scheme”

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 printing 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 ?? 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:

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

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—numbers, Booleans, and primitive procedures—are described next.

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 mean⁸.

Example numbers include:

150 0 -12

⁸The details of managing numbers on computers are complex, and we do not consider them here.

3.14159 3/4 999999999999999999999999

Numbers evaluate to their value. For example, the value of the primitive expression `150` is `150`.⁹

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

PrimitiveExpression $::\Rightarrow$ `#t` | `#f`

The meaning of `#t` is true, and the meaning of `#f` is false.

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.

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:

Expression $::\Rightarrow$ *ApplicationExpression*
ApplicationExpression $::\Rightarrow$ (*Expression* *MoreExpressions*)
MoreExpressions $::\Rightarrow$ ϵ | *Expression* *MoreExpressions*

⁹By convention, we use the `typewriter` font to show program text, and the *slanted* font to show values. In the DrScheme interactions window, values are shown in blue.

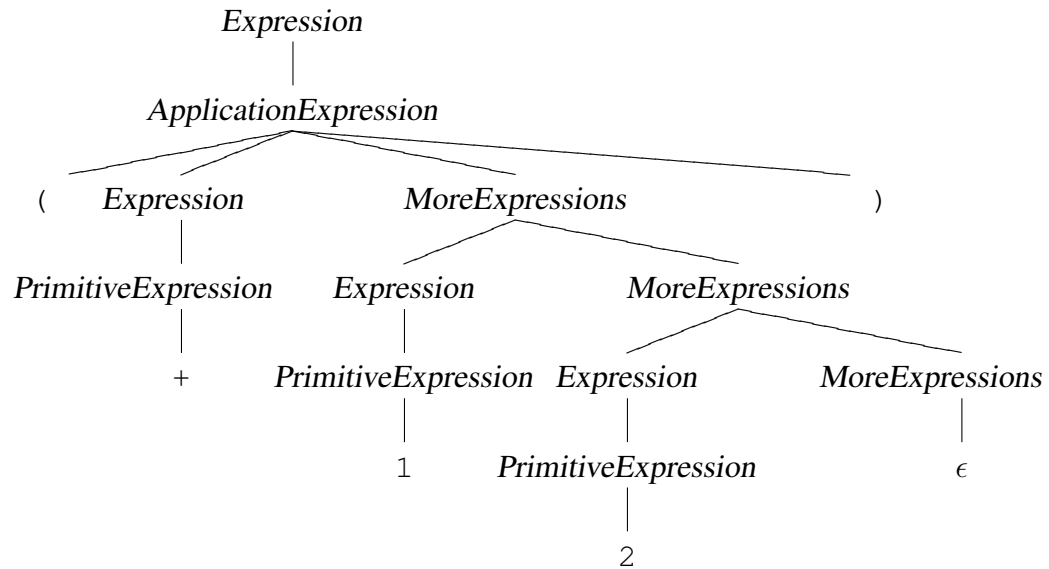
Symbol	Description	Inputs	Output
Arithmetic			
+	add	zero or more numbers	sum of the input numbers (<i>0</i> if there are no inputs)
*	multiply	zero or more numbers	product of the input numbers (<i>1</i> if there are no inputs)
-	subtract	two numbers	the value of the first number minus the value the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

Table 3.1: **Selected Scheme Primitive Procedures.** Some of these are defined for more inputs than just the ones shown here. For example, the `-` procedure works on one number and on more than two numbers.

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 application expression, 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 complex expression is evaluated.

Here is a parse tree for the expression:



The first subexpression, $+$, is a primitive expression. It evaluates to the primitive addition procedure. The next subexpression, 1 , evaluates to the number 1 . The final subexpression, 2 , evaluates to the number 2 . 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. In this case, $(+ 1 2)$ evaluates to 3 .

The subexpressions in an expression can also be complex expressions. As with any nonterminal, the *Expression* nonterminals in the application expression can be

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) (< 100 50))
- e. (not (= (>= 100 100) (> 100 (+ 51 51))))
- f. +
- g. (◇) (+ + <)

3.5 Definitions

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

Definition $::\Rightarrow$ (define *Name Expression*)

A definition is not an expression since it does not evaluate to a value. Instead of producing a value, a definition changes the state of the interpreter. After a definition, the name in the definition is now associated with the value of the expression in the definition.¹⁰

¹⁰Alert 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 ??.

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`, and `yikes!%#@#`.¹¹

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

```
Expression           ::=⇒ NameExpression
NameExpression      ::=⇒ Name
```

For example,

```
                                % meters per second12
> (define speed-of-light 299792458)
> speed-of-light
299792458
> (define seconds-per-hour (* 60 60))
> (* speed-of-light seconds-per-hour)
1079252848800
```

3.6 Procedures

A *procedure* is a sequence of actions that take inputs 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:

¹¹We don't recommend using most 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.

¹²Text after a `%` is a *comment*. The Scheme interpreter will ignore any text after a `%` until the end of the line.

- **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. 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 ??, 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 discuss these in Chapter ?. In fact, even knowing if a procedure will finish (that is, ever produce an output) is a challenging problem. In Chapter ?? 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. The syntax is:

<i>Expression</i>	$::\Rightarrow$	<i>ProcedureExpression</i>
<i>ProcedureExpression</i>	$::\Rightarrow$	(<code>lambda</code> (<i>Parameters</i>) <i>Expression</i>)
<i>Parameters</i>	$::\Rightarrow$	ϵ <i>Name</i> <i>Parameters</i>

Evaluating a *ProcedureExpression* produces a procedure that takes as inputs the *Parameters* following the `lambda`.¹³ The body of the procedure is the *Expression*

¹³Scheme uses `lambda` to make a procedure because it is based on LISP which is based on Lambda Calculus (which we will cover in Chapter ??).

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!

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:

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

We need a new rule, though, for evaluating the application. In this case, the first *Expression* evaluates to a procedure that was created using a *ProcedureExpression*, so we can think of the *ApplicationExpression* as (the underlined part is the replacement for the *ProcedureExpression*):

$$\textit{ApplicationExpression} ::= (\underline{(\textit{lambda} (\textit{Parameters}) \textit{Expression})} \textit{MoreExpressions})$$

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 one exactly expression in the *MoreExpressions* corresponding to each name in the parameters list. Then, evaluate the evaluate that is the body of the procedure. Except, instead of evaluating it normally, whenever any name that matches a name in the parameters list appears that name evaluates to the value of the corresponding input. This is similar to the way binding worked in Post Production Systems (Section ??). 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: Squaring. Consider evaluating the following expression:

```
((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 subexpressions 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
```

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

Definition $::\Rightarrow$ `(define (Name Parameters) 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 3.2.

- a. Define a procedure, `cube`, that takes one number as input and produces as output the cube of that number.
- b. 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 time 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:

$$\begin{array}{l}
 \textit{Expression} \\
 \textit{IfExpression}
 \end{array}
 \begin{array}{l}
 ::= \Rightarrow \\
 ::= \Rightarrow
 \end{array}
 \begin{array}{l}
 \textit{IfExpression} \\
 (\textit{if } \textit{Expression}_{\textit{Predicate}} \\
 \textit{Expression}_{\textit{Consequent}} \\
 \textit{Expression}_{\textit{Alternate}})
 \end{array}$$

The evaluation rule for an *IfExpression* is to first evaluate the predicate expression. If it evaluates to any non-false value, the value of the *IfExpression* is the value of 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 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 ($\#f$), the consequent expression is used. For example, if the predicate evaluates to $\#t$, to a number, or to a procedure the consequence 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 consequence expression is never evaluated.

Example 3.2: Maximum.

Now that we have procedures, decisions, and definitions, we can understand the `max` procedure from the beginning of the chapter. The definition,

```
(define (max a b) (if (> a b) a b))
```

is a condensed procedure definition. It is equivalent to:

```
(define max (lambda (a b) (if (> a b) a b)))
```

This defines the name `max` 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 `#t` if the value of the first parameter is greater, and `#f` otherwise. According to the evaluation rule for an if expression, if the predicate evaluates to any non-false value (in this case, `#t`), the value of the if expression is the value of the consequent expression, `a`. If the predicate evaluates to `#f`, the value of the if expression is the value of the alternate expression, `b`. Hence, our `max` procedure takes two numbers as inputs and produces as output the greater of the two inputs.

Exercise 3.3. (\diamond) Define a procedure, `max3`, that takes three inputs, and produces as output the maximum value of the three inputs. For example,

```
(max3 5 7 3)
```

should evaluate to 7.

3.8 Summary

At this point, we have covered enough of Scheme to write useful programs. In fact (as we will see in Chapter ??), we have covered enough to express *every* possible computation!

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 $::\Rightarrow \epsilon \mid \textit{Expression Program} \mid \textit{Definition Program}$

A program is a sequence of expressions and definitions.

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

A definition associates the value of the expression with the name.

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

Abbreviation for $(\textit{define Name (lambda Parameters)} \\ \textit{Expression})$

Expression $::\Rightarrow \textit{PrimitiveExpression} \mid \textit{NameExpression} \mid \\ \textit{ApplicationExpression} \mid \\ \textit{ProcedureExpression} \mid \textit{IfExpression}$

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

PrimitiveExpression $::\Rightarrow \textit{Number} \mid \#t \mid \#f \mid \textit{primitive procedure}$

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

NameExpression $::\Rightarrow \textit{Name}$

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

$$\text{ApplicationExpression} \quad ::= \Rightarrow \quad (\text{Expression MoreExpressions})$$

Evaluation Rule 3: Application. To evaluate an expression:

- a. **Evaluate** all the subexpressions;
- b. Then, **apply** the value of the first subexpression to the values of the remaining subexpressions.

$$\text{MoreExpressions} \quad ::= \Rightarrow \quad \epsilon \mid \text{Expression MoreExpressions}$$

$$\text{ProcedureExpression} \quad ::= \Rightarrow \quad (\text{lambda } (\text{Parameters}) \text{ Expression})$$

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

$$\text{Parameters} \quad ::= \Rightarrow \quad \epsilon \mid \text{Name Parameters}$$

$$\text{IfExpression} \quad ::= \Rightarrow \quad (\text{if Expression}_{\text{Predicate}} \text{Expression}_{\text{Consequent}} \text{Expression}_{\text{Alternate}})$$

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.¹⁴

Exercise 3.4. Follow the evaluation and application rules to evaluate the following Scheme expression:

```
(max 3 4)
```

where `max` is the maximum procedure defined as,

```
(define max (lambda (a b) (if (> a b) a b)))
```

It is fairly 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 3.5. 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.

¹⁴This doesn't guarantee it will *always* finish, however! We will see in some examples in the next chapter where evaluation never finishes.