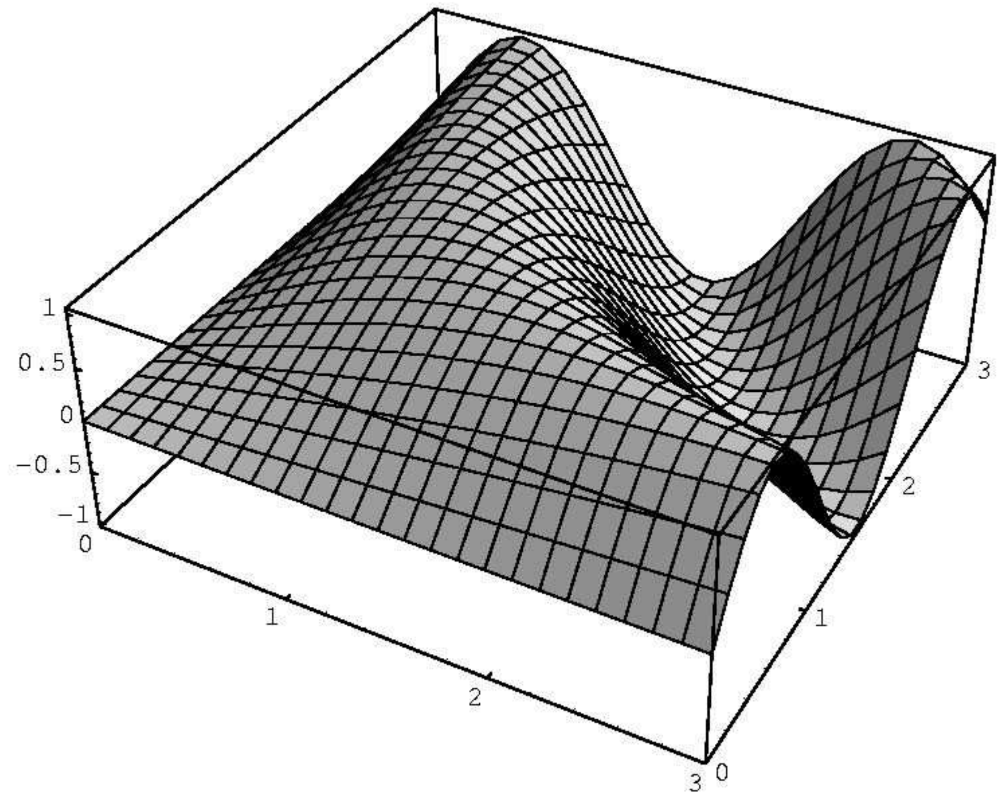

The function $f(x, y) = \sin xy$ plotted by computer

2 Functions

Throughout most of this book we're going to be using a technique called *functional programming*. We can't give a complete definition of this term yet, but in this chapter we introduce the building block of functional programming, the *function*.

Basically we mean by "function" the same thing that your high school algebra teacher meant, except that our functions don't necessarily relate to numbers. But the essential idea is just like the kind of function described by $f(x) = 6x - 2$. In that example, f is the name of a function; that function takes an *argument* called x, which is a number, and *returns* some other number.

In this chapter you are going to use the computer to explore functions, but you are *not* going to use the standard Scheme notation as in the rest of the book. That's because, in this chapter, we want to separate the idea of functions from the complexities of programming language notation. For example, real Scheme notation lets you write expressions that involve more than one function, but in this chapter you can only use one at a time.

To get into this chapter's special computer interface, first start running Scheme as you did in the first chapter, then type

```
(load "functions.scm")
```

to tell Scheme to read the program you'll be using. (If you have trouble loading the program, look in Appendix A for further information about `load`.) Then, to start the program, type

```
(functions)
```

You'll then be able to carry out interactions like the following.* In the text below we've printed what *you* type in **`boldface`** and what the *computer* types in `lightface` printing:

```
Function: +
Argument: 3
Argument: 5

The result is: 8

Function: sqrt
Argument: 144

The result is: 12
```

As you can see, different functions can have different numbers of arguments. In these examples we added two numbers, and we took the square root of one number. However, every function gives exactly one result each time we use it.

To leave the `functions` program, type `exit` when it asks for a function.

Arithmetic

Experiment with these arithmetic functions: `+`, `-`, `*`, `/`, `sqrt`, `quotient`, `remainder`, `random`, `round`, `max`, and `expt`. Try different kinds of numbers, including integers and numbers with decimal fractions. What if you try to divide by zero? Throughout this chapter we are going to let you experiment with functions rather than just give you a long, boring list of how each one works. (The boring list is available for reference on page 553.)

Try these:

```
Function: /
Argument: 1
Argument: 987654321987654321

Function: remainder
Argument: 12
Argument: -5
```

* If you get no response at all after you type `(functions)`, just press the Return or Enter key again. Tell your instructor to read Appendix A to see how to fix this.

```
Function: round
Argument: 17.5
```

These are just a few suggestions. Be creative; don't just type in our examples.

Words

Not all Scheme functions deal with numbers. A broader category of argument is the *word,* including numbers but also including English words like `spaghetti` or `xylophone`. Even a meaningless sequence of letters and digits such as `glo87rp` is considered a word.* Try these functions that accept words as arguments: `first`, `butfirst`, `last`, `butlast`, `word`, and `count`. What happens if you use a number as the argument to one of these?

```
Function: butfirst
Argument: a

Function: count
Argument: 765432
```

So far most of our functions fall into one of two categories: the arithmetic functions, which require numbers as arguments and return a number as the result; and the word functions, which accept words as arguments and return a word as the result. The one exception we've seen is `count`. What kind of argument does `count` accept? What kind of value does it return? The technical term for "a kind of data" is a *type.*

In principle you could think of almost anything as a type, such as "numbers that contain the digit 7." Such *ad hoc* types are legitimate and sometimes useful, but there are also official types that Scheme knows about. Types can overlap; for example, numbers are also considered words.

```
Function: word
Argument: 3.14
Argument: 1592654

Function: +
Argument: 6
Argument: seven
```

* Certain punctuation characters can also be used in words, but let's defer the details until you've gotten to know the word functions with simpler examples.

Domain and Range

The technical term for “the things that a function accepts as an argument” is the *domain* of the function. The name for “the things that a function returns” is its *range*. So the domain of `count` is words, and the range of `count` is numbers (in fact, nonnegative integers). This example shows that the range may not be exactly one of our standard data types; there is no “nonnegative integer” type in Scheme.

How do you talk about the domain and range of a function? You could say, for example, “The `cos` function has numbers as its domain and numbers between −1 and 1 as its range.” Or, informally, you may also say “`Cos` takes a number as its argument and returns a number between −1 and 1.”*

For functions of two or more arguments, the language is a little less straightforward. The informal version still works: “`Remainder` takes two integers as arguments and returns an integer.” But you can’t say “The domain of `remainder` is two integers,” because the domain of a function is the *set* of all possible arguments, not just a statement about the characteristics of legal arguments.**

(By the way, we’re making certain simplifications in this chapter. For example, Scheme’s `+` function can actually accept any number of arguments, not just two. But we don’t want to go into all the bells and whistles at once, so we’ll start with adding two numbers at a time.)

Here are examples that illustrate the domains of some functions:

```
Function: expt
Argument: -3
Argument: .5

Function: expt
Argument: -3
Argument: -3

Function: remainder
Argument: 5
Argument: 0
```

* Unless your version of Scheme has complex numbers.

** Real mathematicians say, “The domain of `remainder` is the Cartesian cross product of the integers and the integers.” In order to avoid that mouthful, we’ll just use the informal wording.

More Types: Sentences and Booleans

We're going to introduce more data types, and more functions that include those types in their domain or range. The next type is the *sentence:* a bunch of words enclosed in parentheses, such as

```
(all you need is love)
```

(Don't include any punctuation characters within the sentence.) Many of the functions that accept words in their domain will also accept sentences. There is also a function `sentence` that accepts words and sentences. Try examples like `butfirst` of a sentence.

```
Function: sentence
Argument: (when i get)
Argument: home

Function: butfirst
Argument: (yer blues)

Function: butlast
Argument: ()
```

Other important functions are used to ask yes-or-no questions. That is, the range of these functions contains only two values, one meaning "true" and the other meaning "false." Try the numeric comparisons `=`, `<`, `>`, `<=`, and `>=`, and the functions `equal?` and `member?` that work on words and sentences. (The question mark is part of the name of the function.) There are also functions `and`, `or`, and `not` whose domain and range are both true-false values. The two values "true" and "false" are called *Booleans,* named after George Boole (1815–1864), who developed the formal tools used for true-false values in mathematics.

What good are these true-false values? Often a program must choose between two options: If the number is positive, do this; if negative, do that. Scheme has functions to make such choices based on true-false values. For now, you can experiment with the `if` function. Its first argument must be true or false; the others can be anything.

Our Favorite Type: Functions

So far our data types include numbers, words, sentences, and Booleans. Scheme has several more data types, but for now we'll just consider one more. A *function* can be used as data. Here's an example:

```
Function: number-of-arguments
Argument: equal?

The result is: 2
```

The range of `number-of-arguments` is nonnegative integers. But its domain is *functions*. For example, try using it as an argument to itself!

If you've used other computer programming languages, it may seem strange to use a function—that is, a part of a computer program—as data. Most languages make a sharp distinction between program and data. We'll soon see that the ability to treat functions as data helps make Scheme programming very powerful and convenient.

Try these examples:

```
Function: every
Argument: first
Argument: (the long and winding road)

Function: keep
Argument: vowel?
Argument: constantinople
```

Think carefully about these. You aren't applying the function `first` to the sentence `(the long and winding road)`; you're applying the function `every` to a function and a sentence.

Other functions that can be used with `keep` include `even?` and `odd?`, whose domains are the integers, and `number?`, whose domain is everything.

Play with It

If you've been reading the book but not trying things out on the computer as you go along, get to work! Spend some time getting used to these ideas and thinking about them. When you're done, read ahead.

Thinking about What You've Done

The idea of *function* is at the heart of both mathematics and computer science. For example, when mathematicians want to think very formally about the system of numbers, they use functions to create the integers. They say, let's suppose we have one number, called zero; then let's suppose we have the *function* given by $f(x) = x + 1$. By applying that function repeatedly, we can create $1 = f(0)$, then $2 = f(1)$, and so on.

Functions are important in computer science because they give us a way to think about *process*—in simple English, a way to think about something happening, something changing. A function embodies a *transformation* of information, taking in something we know and returning something we didn't know. That's what computers do: They transform information to produce new results.

A lot of the mathematics taught in school is about numbers, but we've seen that functions don't have to be about numbers. We've used functions of words and sentences, such as `first`, and even functions of functions, such as `keep`. You can imagine functions that transform information of any kind at all, such as the function French(window)=fenêtre or the function capital(California)=Sacramento.

You've done a lot of thinking about the *domain* and *range* of functions. You can add two numbers, but it doesn't make sense to add two words that aren't numbers. Some two-argument functions have complicated domains because the acceptable values for one argument depend on the specific value used for the other one. (The function `expt` is an example; make sure you've tried both positive and negative numbers, and fractional as well as whole-number powers.)

Part of the definition of a function is that you always get the same answer whenever you call a function with the same argument(s). The value returned by the function, in other words, shouldn't change regardless of anything else you may have computed meanwhile. One of the "functions" you've explored in this chapter isn't a real function according to this rule; which one? The rule may seem too restrictive, and indeed it's often convenient to use the name "function" loosely for processes that can give different results in different circumstances. But we'll see that sometimes it's important to stick with the strict definition and refrain from using processes that aren't truly functions.

We've hinted at two different ways of thinking about functions. The first is called *function as process.* Here, a function is a rule that tells us how to transform some information into some other information. The function is just a rule, not a thing in its own right. The actual "things" are the words or numbers or whatever the function manipulates. The second way of thinking is called *function as object.* In this view, a function is a perfectly good "thing" in itself. We can use a function as an argument to another function, for example. Research with college math students shows that this second idea is hard for most people, but it's worth the effort because you'll see that *higher-order functions* (functions of functions) like `keep` and `every` can make programs much easier to write.

As a homey analogy, think about a carrot peeler. If we focus our attention on the carrots—which are, after all, what we want to eat—then the peeler just represents a process. We are peeling carrots. We are applying the function `peel` to carrots. It's the carrot that counts. But we can also think about the peeler as a thing in its own right, when we clean it, or worry about whether its blade is sharp enough.

The big idea that we *haven't* explored in this chapter (although we used it a lot in Chapter 1) is the *composition* of functions: using the result from one function as an argument to another function. It's a crucial idea; we write large programs by defining a bunch of small functions and then composing them with each other to produce the desired result. We'll start doing that in the next chapter, where we return to real Scheme notation.

Exercises

Use the `functions` *program for all these exercises.*

2.1 In each line of the following table we've left out one piece of information. Fill in the missing details.

function	arg 1	arg 2	result
`word`	`now`	`here`	
`sentence`	`now`	`here`	
`first`	`blackbird`	none	
`first`	`(blackbird)`	none	
	`3`	`4`	`7`
`every`		`(thank you girl)`	`(hank ou irl)`
`member?`	`e`	`aardvark`	
`member?`	`the`		`#t`
`keep`	`vowel?`	`(i will)`	
`keep`	`vowel?`		`eieio`*
`last`	`()`	none	
	`last`	`(honey pie)`	`(y e)`
		`taxman`	`aa`

2.2 What is the domain of the `vowel?` function?

* Yes, there is an English word. It has to do with astronomy.

2.3 One of the functions you can use is called `appearances`. Experiment with it, and then describe fully its domain and range, and what it does. (Make sure to try lots of cases. Hint: Think about its name.)

2.4 One of the functions you can use is called `item`. Experiment with it, and then describe fully its domain and range, and what it does.

The following exercises ask for functions that meet certain criteria. For your convenience, here are the functions in this chapter: `+`, `-`, `/`, `<=`, `<`, `=`, `>=`, `>`, `and`, `appearances`, `butfirst`, `butlast`, `cos`, `count`, `equal?`, `every`, `even?`, `expt`, `first`, `if`, `item`, `keep`, `last`, `max`, `member?`, `not`, `number?`, `number-of-arguments`, `odd?`, `or`, `quotient`, `random`, `remainder`, `round`, `sentence`, `sqrt`, `vowel?`, and `word`.

2.5 List the one-argument functions in this chapter for which the type of the return value is always different from the type of the argument.

2.6 List the one-argument functions in this chapter for which the type of the return value is sometimes different from the type of the argument.

2.7 Mathematicians sometimes use the term "operator" to mean a function of two arguments, both of the same type, that returns a result of the same type. Which of the functions you've seen in this chapter satisfy that definition?

2.8 An operator f is *commutative* if $f(a,b) = f(b,a)$ for all possible arguments a and b. For example, `+` is commutative, but `word` isn't. Which of the operators from Exercise 2.7 are commutative?

2.9 An operator f is *associative* if $f(f(a,b),c) = f(a,f(b,c))$ for all possible arguments a, b, and c. For example, `*` is associative, but not `/`. Which of the operators from Exercise 2.7 are associative?