Part II
Composition of Functions

The big idea in this part of the book is deceptively simple. It's that we can take the value returned by one function and use it as an argument to another function. By "hooking up" two functions in this way, we invent a new, third function. For example, let's say we have a function that adds the letter `s` to the end of a word:

add-s("`run`") = "`runs`"

and another function that puts two words together into a sentence:

sentence("`day`", "`tripper`") = "`day tripper`"

We can combine these to create a new function that represents the third person singular form of a verb:

third-person(verb) = *sentence*("`she`", *add-s*(verb))

That general formula looks like this when applied to a particular verb:

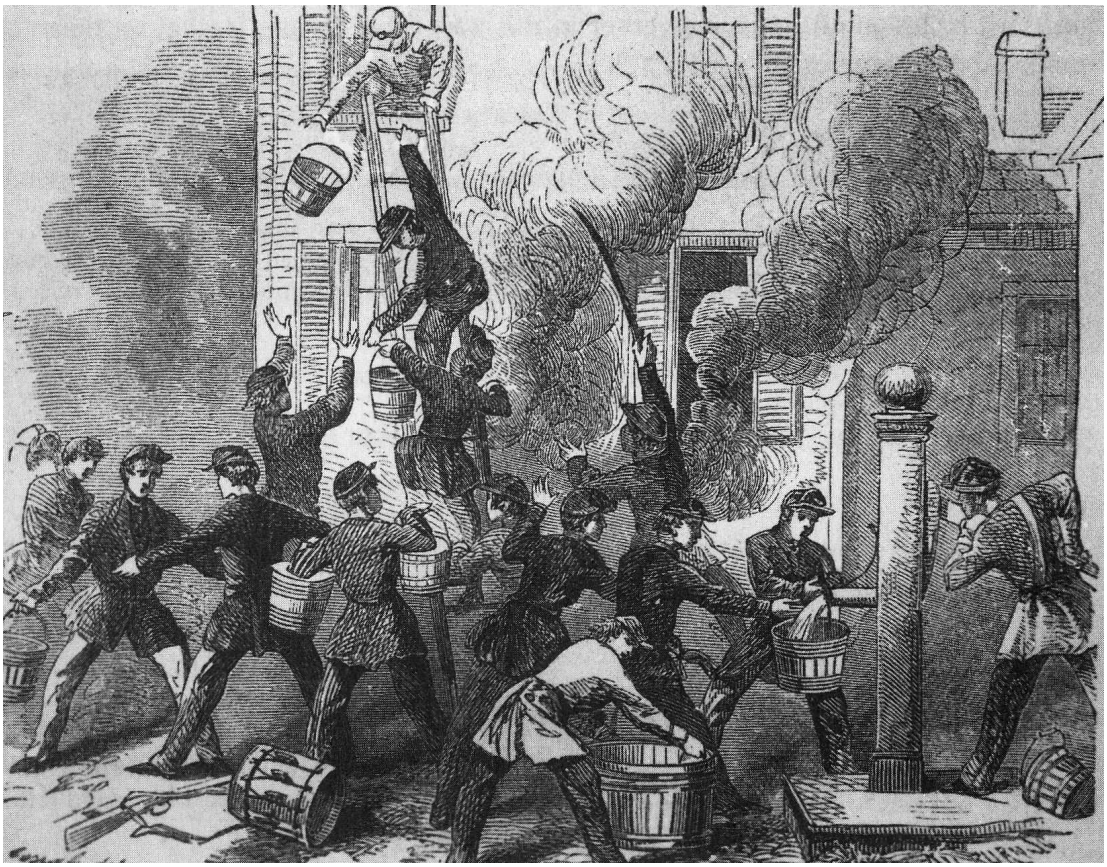
third-person("`sing`") = "`she sings`"

The way we say it in Scheme is

```
(define (third-person verb)
  (sentence 'she (add-s verb)))
```

(When we give an example like this at the beginning of a part, don't worry about the fact that you don't recognize the notation. The example is meant as a preview of what you'll learn in the coming chapters.)

We know that this idea probably doesn't look like much of a big deal to you. It seems obvious. Nevertheless, it will turn out that we can express a wide variety of computational algorithms by linking functions together in this way. This linking is what we mean by "functional programming."



In a bucket brigade, each person hands a result to the next.

3 Expressions

evaluates

expression.

`26` `(+ 14 7)`

atom *atomic expression*

compound expression,

`+ 14 7`

subexpressions.

call *invoke*

apply

* In other programming languages, the name for what you type might be a “command” or an “instruction.” The name “expression” is meant to emphasize that we are talking about the notation in which you ask the question, as distinct from the idea in your head, just as in English you express an idea in words.

Notice that in the example above we asked + to add *three* numbers. In the `functions` program of Chapter 2 we pretended that every Scheme function accepts a fixed number of arguments, but actually, some functions can accept any number. These include `+`, `*`, `word`, and `sentence`.

Result Replacement

Since a little person can't do his or her job until all of the necessary subexpressions have been evaluated by other little people, we can "fast forward" this process by skipping the parts about "Alice waits for Bernie and Cordelia" and starting with the completion of the smaller tasks by the lesser little people.

To keep track of which result goes into which larger computation, you can write down a complicated expression and then *rewrite* it repeatedly, each time replacing some small expression with a simpler expression that has the same value.

```
(+ (* (- 10 7) (+ 4 1)) (- 15 (/ 12 3)) 17)
(+ (* 3        (+ 4 1)) (- 15 (/ 12 3)) 17)
(+ (* 3        5      ) (- 15 (/ 12 3)) 17)
(+ 15                    (- 15 (/ 12 3)) 17)
(+ 15                    (- 15 4       ) 17)
(+ 15                    11              17)
43
```

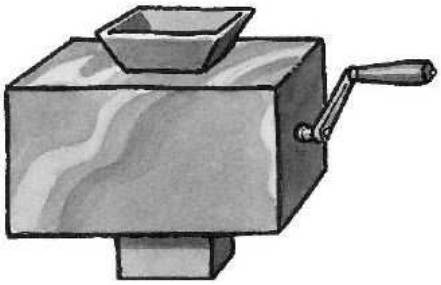
In each line of the diagram, the boxed expression is the one that will be replaced with its value on the following line.

If you like, you can save some steps by evaluating *several* small expressions from one line to the next:

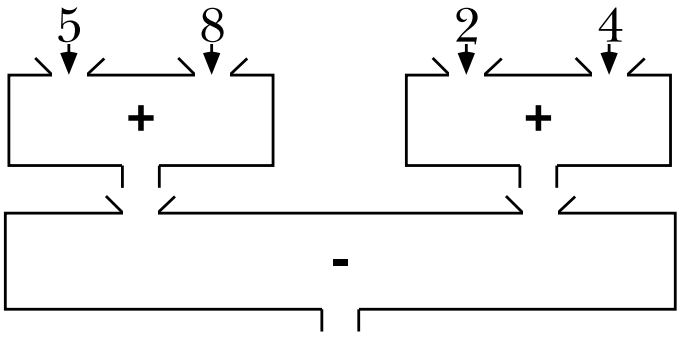
```
(+ (* (- 10 7) (+ 4 1)) (- 15 (/ 12 3)) 17)
(+ (* 3        5      ) (- 15 4       ) 17)
(+ 15                    11              17)
43
```

Plumbing Diagrams

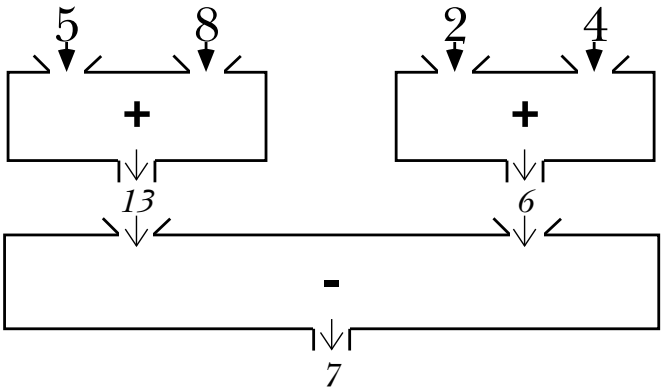
Some people find it helpful to look at a pictorial form of the connections among subexpressions. You can think of each procedure as a machine, like the ones they drew on the chalkboard in junior high school.



Each machine has some number of input hoppers on the top and one chute at the bottom. You put something in each hopper, turn the crank, and something else comes out the bottom. For a complicated expression, you hook up the output chute of one machine to the input hopper of another. These combinations are called "plumbing diagrams." Let's look at the plumbing diagram for `(- (+ 5 8) (+ 2 4))`:



You can annotate the diagram by indicating the actual information that flows through each pipe. Here's how that would look for this expression:



Pitfalls

⇒ One of the biggest problems that beginning Lisp programmers have comes from trying to read a program from left to right, rather than thinking about it in terms of expressions and subexpressions. For example,

```
(square (cos 3))
```

doesn't mean "square three, then take the cosine of the answer you get." Instead, as you know, it means that the argument to `square` is the return value from `(cos 3)`.

⇒ Another big problem that people have is thinking that Scheme cares about the spaces, tabs, line breaks, and other "white space" in their Scheme programs. We've been indenting our expressions to illustrate the way that subexpressions line up underneath each other. But to Scheme,

```
(+ (* 2 (/ 14 7) 3) (/ (* (- (* 3 5) 3) (+ 1
1)) (- (* 4 3) (* 3 2))) (- 15 18))
```

means the same thing as

```
(+ (* 2 (/ 14 7) 3)
   (/ (* (- (* 3 5) 3) (+ 1 1))
      (- (* 4 3) (* 3 2)))
   (- 15 18))
```

So in this expression:

```
(+ (* 3 (sqrt 49)                          ;; weirdly formatted
   (/ 12 4)))
```

there aren't two arguments to `+`, even though it looks that way if you think about the indenting. What Scheme does is look at the parentheses, and if you examine these carefully, you'll see that there are three arguments to `*`: the atom `3`, the compound expression `(sqrt 49)`, and the compound expression `(/ 12 4)`. (And there's only one argument to `+`.)

⇒ A consequence of Scheme's not caring about white space is that when you hit the return key, Scheme might not do anything. If you're in the middle of an expression, Scheme waits until you're done typing the entire thing before it evaluates what you've typed. This is fine if your program is correct, but if you type this in:

```
(+ (* 3 4)
   (/ 8 2)                                ; note missing right paren
```

then *nothing* will happen. Even if you type forever, until you close the open parenthesis next to the + sign, Scheme will still be reading an expression. So if Scheme seems to be ignoring you, try typing a zillion close parentheses. (You'll probably get an error message about too many parentheses, but after that, Scheme should start paying attention again.)

⇒ You might get into the same sort of trouble if you have a double-quote mark (") in your program. Everything inside a pair of quotation marks is treated as one single *string*. We'll explain more about strings later. For now, if your program has a stray quotation mark, like this:

```
(+ (* 3 " 4)                              ; note extra quote mark
   (/ 8 2))
```

then you can get into the same predicament of typing and having Scheme ignore you. (Once you type the second quotation mark, you may still need some close parentheses, since the ones you type inside a string don't count.)

⇒ One other way that Scheme might seem to be ignoring you comes from the fact that you don't get a new Scheme prompt until you type in an expression and it's evaluated. So if you just hit the `return` or `enter` key without typing anything, most versions of Scheme won't print a new prompt.

Boring Exercises

3.1 Translate the arithmetic expressions $(3+4)\times5$ and $3+(4\times5)$ into Scheme expressions, and into plumbing diagrams.

3.2 How many little people does Alonzo hire in evaluating each of the following expressions:

```
(+ 3 (* 4 5) (- 10 4))
```

```
(+ (* (- (/ 8 2) 1) 5) 2)
```

```
(* (+ (- 3 (/ 4 2))
      (sin (* 3 2))
      (- 8 (sqrt 5)))
   (- (/ 2 3)
      4))
```

3.3 Each of the expressions in the previous exercise is compound. How many subexpressions (not including subexpressions of subexpressions) does each one have?

For example,

```
(* (- 1 (+ 3 4)) 8)
```

has three subexpressions; you wouldn't count `(+ 3 4)`.

3.4 Five little people are hired in evaluating the following expression:

```
(+ (* 3 (- 4 7))
   (- 8 (- 3 5)))
```

Give each little person a name and list her specialty, the argument values she receives, her return value, and the name of the little person to whom she tells her result.

3.5 Evaluate each of the following expressions using the result replacement technique:

```
(sqrt (+ 6 (* 5 2)))
```

```
(+ (+ (+ 1 2) 3) 4)
```

3.6 Draw a plumbing diagram for each of the following expressions:

```
(+ 3 4 5 6 7)
```

```
(+ (+ 3 4) (+ 5 6 7))
```

```
(+ (+ 3 (+ 4 5) 6) 7)
```

3.7 What value is returned by `(/ 1 3)` in your version of Scheme? (Some Schemes return a decimal fraction like `0.33333`, while others have exact fractional values like `1/3` built in.)

3.8 Which of the functions that you explored in Chapter 2 will accept variable numbers of arguments?

Real Exercises

3.9 The expression `(+ 8 2)` has the value `10`. It is a compound expression made up of three atoms. For this problem, write five other Scheme expressions whose values are also the number ten:

- An atom
- Another compound expression made up of three atoms
- A compound expression made up of four atoms
- A compound expression made up of an atom and two compound subexpressions
- Any other kind of expression