Simply Scheme

Part I
Introduction: Functions

The purpose of these introductory pages before each part of the book is to call attention to a big idea that runs through all the work of several chapters. In the chapters themselves, the big idea may sometimes be hidden from view because of the technical details that we need to make the idea work. If you ever feel lost in the forest, you might want to refer back here.

In these first two chapters, our goal is to introduce the Scheme programming language and the idea of using *functions* as the building blocks of a computation.

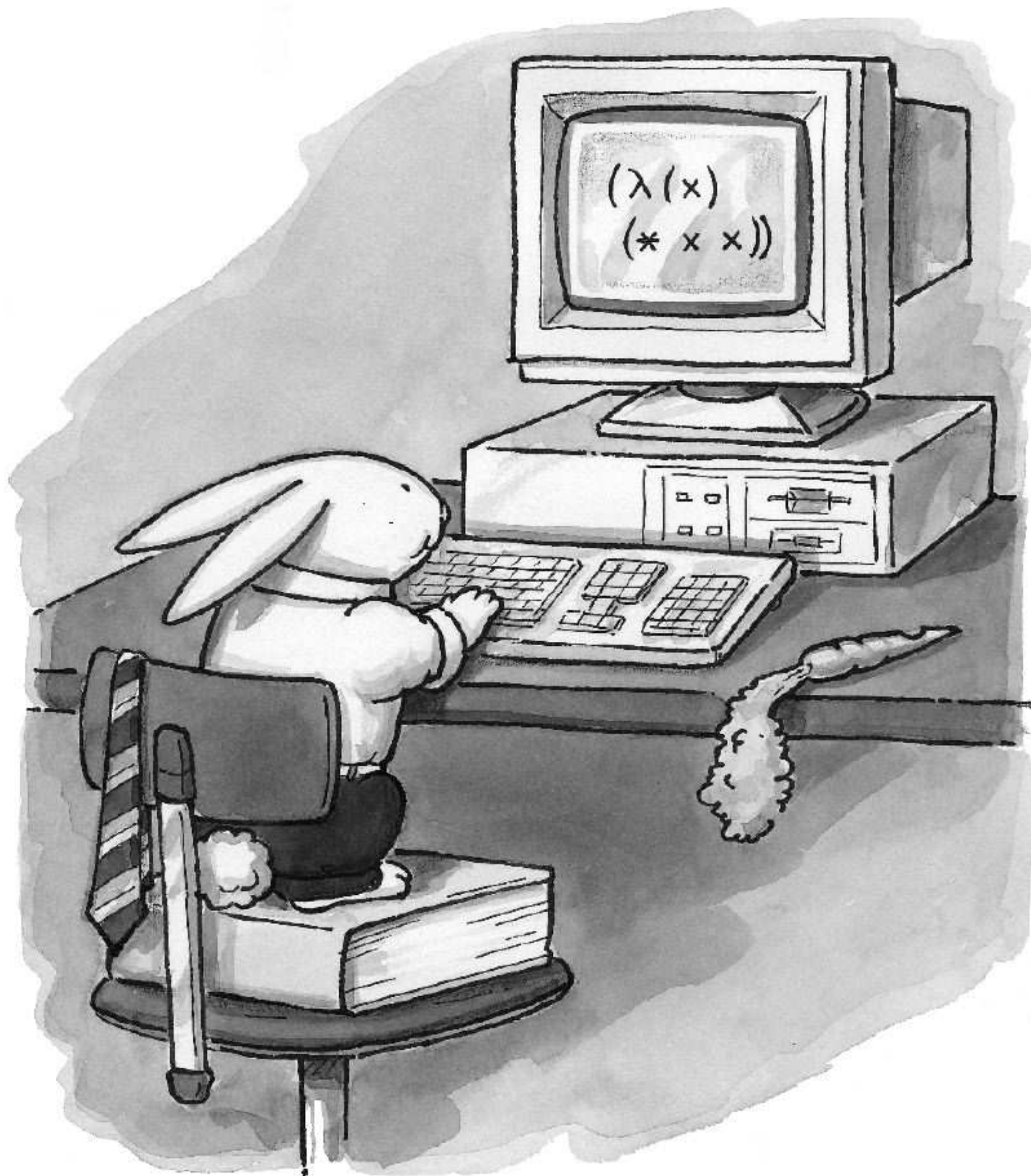
The first chapter is a collection of short Scheme programs, presented to show off what Scheme can do. We'll try to explain enough of the mechanism so that you don't feel completely mystified, but we'll defer the details until later. Our goal is not for you to feel that you could re-create these programs, but rather that you get a sense of what *kinds* of programs we'll be working with.

The second chapter explores functions in some detail. Traditionally, computer programs are built out of *actions:* First do this, then do that, and finally print the results. Each step in the program *does* something. Functional programming is different, in that we are less concerned with actions and more concerned with values.

For example, if you have a pocket calculator with a square root button, you could enter the number 3, push the button, and you'll see something like 1.732050808 in the display. How does the calculator know? There are several possible processes that the calculator could carry out. One process, for example, is to make a guess, square it, see if the result is too big or too small, and use that information to make a closer guess. That's a sequence of actions. But ordinarily you don't care what actions the calculator takes;

what interests you is that you want to apply the square root *function* to the *argument* 3, and get back a *value.* We're going to focus on this business of functions, arguments, and result values.

Don't think that functions have to involve numbers. We'll be working with functions like "first name," "plural," and "acronym." These functions have words and sentences as their arguments and values.



Scheme-Brained Hare

1 Showing Off Scheme

We are going to use the programming language Scheme to teach you some big ideas in computer science. The ideas are mostly about *control of complexity*—that is, about how to develop a large computer program without being swamped in details.

For example, once you've solved part of the large problem, you can give that partial solution a *name* and then you can use the named subprogram as if it were an indivisible operation, just like the ones that are built into the computer. Thereafter, you can forget about the details of that subprogram. This is the beginning of the idea of *abstraction,* which we'll discuss in more depth throughout the book.

The big ideas are what this book is about, but first we're going to introduce you to Scheme. (Scheme is a dialect of Lisp, a family of computer programming languages invented for computing with words, sentences, and ideas instead of just numbers.)

Talking to Scheme

The incantations to get Scheme running will be different for each model of computer. Appendix A talks about these details; you can look up the particular version of Scheme that you're using. That appendix will also tell you how to load the file `simply.scm`, which you need to make the examples in this book work.

When Scheme has started up and is ready for you to interact with it, you'll see a message on the screen, something like this:

```
Welcome to XYZ Brand Scheme.
>
```

The `>` is a *prompt,* Scheme's way of telling you that it's ready for you to type something. Scheme is an *interactive* programming language. In other words, you type a request to Scheme, then Scheme prints the answer, and then you get another prompt. Try it out:

```
> 6
6
```

We just asked Scheme, "What is 6?" and Scheme told us that 6 is 6. Most of the time we ask harder questions:

```
> (+ 4 7)
11
> (- 23 5)
18
> (+ 5 6 7 8)
26
```

Whenever something you type to Scheme is enclosed in parentheses, it indicates a request to carry out a *procedure.* (We'll define "procedure" more formally later, but for now it means something that Scheme knows how to do. A procedure tells Scheme how to compute a particular function.) The first thing inside the parentheses indicates what procedure to use; the others are *arguments,* i.e., values that are used as data by the procedure.

Scheme has non-numeric procedures, too:

```
> (word 'comp 'uter)
COMPUTER
```

(If this last example gives an error message saying that Scheme doesn't understand the name `word`, it means that you didn't load the file `simply.scm`. Consult Appendix A.)

In these first examples, we've shown what you type in **`boldface`** and what the computer responds in `lightface`. Hereafter, we will rely on the prompt characters to help you figure out who's talking on which line.

For the most part, Scheme doesn't care about whether you type in UPPER CASE or lower case. For the examples in this book, we'll assume that you always type in lower case and that the computer prints in upper case. Your Scheme might print in lower case; it doesn't matter.

Recovering from Typing Errors

Don't worry if you make a mistake typing these examples in; you can just try again. One of the great things about interactive programming languages is that you can experiment in them.

The parentheses and single quote marks are important; don't leave them out. If Scheme seems to be ignoring you, try typing a bunch of right parentheses, `))))))`, and hitting the `return` or `enter` key. (That's because Scheme doesn't do anything until you've closed all the parentheses you've opened, so if you have an extra left parenthesis, you can keep typing forever with no response.)

Another problem you might encounter is seeing a long message that you don't understand and then finding yourself with something other than a Scheme prompt. This happens when Scheme considers what you typed as an error. Here's an example; for now, never mind exactly why this is an error. We just want to talk about the result:

```
> (+ 2 a)

Unbound variable a
;Package: (user)

2 Error->
```

The exact form of the message you get will depend on the version of Scheme that you're using. For now, the important point is that some versions deal with errors by leaving you talking to a *debugger* instead of to Scheme itself. The debugger may have a completely different language. It's meant to help you figure out what's wrong in a large program you've written. For a beginner, though, it's more likely to get in the way. Read the documentation for your particular Scheme dialect to learn how to escape from the debugger. (In some versions you don't get trapped in a debugger when you make an error, so this problem may not arise.)

Exiting Scheme

Although there's no official standard way to exit Scheme, most versions use the notation

```
> (exit)
```

for this purpose. If you type in some of the examples that follow and then exit from Scheme, what you type won't be remembered the next time you use Scheme. (Appendix A talks about how to use a text editor along with Scheme to make a permanent record of your work.)

More Examples

We're about to show you a few examples of (we hope) interesting programs in Scheme. Play with them! Type them into your computer and try invoking them with different data. Again, don't worry too much if something doesn't work—it probably just means that you left out a parenthesis, or some such thing.

While you're going through these examples, see how much you can figure out for yourself about how they work. In particular, try guessing what the names of procedures, such as `first` and `keep`, mean. Some of them will probably be obvious, some of them harder. The point isn't to see how smart you are, but to get you thinking about the *kinds* of things you want to be able to do in a computer program. Later on we'll go through these examples in excruciating detail and tell you the official meanings of all the pieces.

Besides learning the *vocabulary* of Scheme, another point of this activity is to give you a feeling for the ways in which we put these names together in a program. Every programming language has its own flavor. For example, if you've programmed before in other languages, you may be surprised not to find anything that says `print` in these examples.

On the other hand, some of these examples are programs that we won't expect you to understand fully until most of the way through this book. So don't worry if something doesn't make sense; just try to get some of the flavor of Scheme programming.

Example: Acronyms

Here's our first new program. So far we have just been using procedures built into Scheme: `+`, `-`, and `word`. When you first start up Scheme, it knows 100–200 procedures. These are called *primitive* procedures. Programming in Scheme means defining new procedures, called *compound* procedures. Right now we're going to invent one that finds the acronym for a title:

```
(define (acronym phrase)
  (accumulate word (every first phrase)))
```

```
> (acronym '(american civil liberties union))
ACLU

> (acronym '(reduced instruction set computer))
RISC

> (acronym '(quod erat demonstrandum))
QED
```

Did you have trouble figuring out what all the pieces do in the `acronym` procedure? Try these examples:

```
> (first 'american)
A

> (every first '(american civil liberties union))
(A C L U)

> (accumulate word '(a c l u))
ACLU
```

Notice that this simple `acronym` program doesn't always do exactly what you might expect:

```
> (acronym '(united states of america))
USOA
```

We can rewrite the program to leave out certain words:

```
(define (acronym phrase)
  (accumulate word (every first (keep real-word? phrase))))

(define (real-word? wd)
  (not (member? wd '(a the an in of and for to with))))

> (acronym '(united states of america))
USA

> (acronym '(structure and interpretation of computer programs))
SICP

> (acronym '(association for computing machinery))
ACM

> (real-word? 'structure)
#T
```

```
> (real-word? 'of)
#F*

> (keep real-word? '(united network command for law and enforcement))
(UNITED NETWORK COMMAND LAW ENFORCEMENT)
```

Example: Pig Latin

Our next example translates a word into Pig Latin.**

```
(define (pigl wd)
  (if (member? (first wd) 'aeiou)
      (word wd 'ay)
      (pigl (word (butfirst wd) (first wd)))))

> (pigl 'spaghetti)
AGHETTISPAY

> (pigl 'ok)
OKAY
```

(By the way, if you've used other programming languages before, don't fall into the trap of thinking that each line of the `pigl` definition is a "statement" and that the

* In some versions of Scheme you might see `()` instead of `#F`.

** Pig Latin is a not-very-secret secret language that many little kids learn. Each word is translated by moving all the initial consonants to the end of the word, and adding "ay" at the end. It's usually spoken rather than written, but that's a little harder to do on a computer.

statements are executed one after the other. That's not how it works in Scheme. The entire thing is a single expression, and what counts is the grouping with parentheses. Starting a new line is no different from a space between words as far as Scheme is concerned. We could have defined `pigl` on one humongous line and it would mean the same thing. Also, Scheme doesn't care about how we've indented the lines so that subexpressions line up under each other. We do that only to make the program more readable for human beings.)

The procedure follows one of two possible paths, depending on whether the first letter of the given word is a vowel. If so, `pigl` just adds the letters `ay` at the end:

```
> (pigl 'elephant)
ELEPHANTAY
```

The following examples might make it a little more clear how the starting-consonant case works:

```
> (first 'spaghetti)
S

> (butfirst 'spaghetti)
PAGHETTI

> (word 'paghetti 's)
PAGHETTIS

> (define (rotate wd)
    (word (butfirst wd) (first wd)))

> (rotate 'spaghetti)
PAGHETTIS

> (rotate 'paghettis)
AGHETTISP

> (pigl 'aghettisp)
AGHETTISPAY
```

You've seen `every` before, in the `acronym` example, but we haven't told you what it does. Try to guess what Scheme will respond when you type this:

```
(every pigl '(the ballad of john and yoko))
```

Example: Ice Cream Choices

Here's a somewhat more complicated program, but still pretty short considering what it accomplishes:

```
(define (choices menu)
  (if (null? menu)
      '(())
      (let ((smaller (choices (cdr menu))))
        (reduce append
                (map (lambda (item) (prepend-every item smaller))
                     (car menu))))))

(define (prepend-every item lst)
  (map (lambda (choice) (se item choice)) lst))

> (choices '((small medium large)
             (vanilla (ultra chocolate) (rum raisin) ginger)
             (cone cup)))
((SMALL VANILLA CONE)
 (SMALL VANILLA CUP)
 (SMALL ULTRA CHOCOLATE CONE)
 (SMALL ULTRA CHOCOLATE CUP)
 (SMALL RUM RAISIN CONE)
 (SMALL RUM RAISIN CUP)
 (SMALL GINGER CONE)
 (SMALL GINGER CUP)
 (MEDIUM VANILLA CONE)
 (MEDIUM VANILLA CUP)
 (MEDIUM ULTRA CHOCOLATE CONE)
 (MEDIUM ULTRA CHOCOLATE CUP)
 (MEDIUM RUM RAISIN CONE)
 (MEDIUM RUM RAISIN CUP)
 (MEDIUM GINGER CONE)
 (MEDIUM GINGER CUP)
 (LARGE VANILLA CONE)
 (LARGE VANILLA CUP)
 (LARGE ULTRA CHOCOLATE CONE)
 (LARGE ULTRA CHOCOLATE CUP)
 (LARGE RUM RAISIN CONE)
 (LARGE RUM RAISIN CUP)
 (LARGE GINGER CONE)
 (LARGE GINGER CUP))
```

Notice that in writing the program we didn't have to say how many menu categories there are, or how many choices in each category. This one program will work with any menu—try it out yourself.

Example: Combinations from a Set

Here's a more mathematical example. We want to know all the possible combinations of, let's say, three things from a list of five possibilities. For example, we want to know all the teams of three people that can be chosen from a group of five people. "Dozy, Beaky, and Tich" counts as the same team as "Beaky, Tich, and Dozy"; the order within a team doesn't matter.

Although this will be a pretty short program, it's more complicated than it looks. We don't expect you to be able to figure out the algorithm yet.* Instead, we just want you to marvel at Scheme's ability to express difficult techniques succinctly.

```
(define (combinations size set)
  (cond ((= size 0) '(()))
        ((empty? set) '())
        (else (append (prepend-every (first set)
                                     (combinations (- size 1)
                                                   (butfirst set)))
                      (combinations size (butfirst set))))))

> (combinations 3 '(a b c d e))
((A B C) (A B D) (A B E) (A C D) (A C E)
 (A D E) (B C D) (B C E) (B D E) (C D E))

> (combinations 2 '(john paul george ringo))
((JOHN PAUL) (JOHN GEORGE) (JOHN RINGO)
 (PAUL GEORGE) (PAUL RINGO) (GEORGE RINGO))
```

(If you're trying to figure out the algorithm despite our warning, here's a hint: All the combinations of three letters shown above can be divided into two groups. The first group consists of the ones that start with the letter `A` and contain two more letters; the second group has three letters not including `A`. The procedure finds these two groups separately and combines them into one. If you want to try to understand all the pieces, try playing with them separately, as we encouraged you to do with the `pigl` and `acronym` procedures.)

---

* What's an *algorithm*? It's a method for solving a problem. The usual analogy is to a recipe in cooking, although you'll see throughout this book that we want to get away from the aspect of that analogy that emphasizes the *sequential* nature of a recipe—first do this, then do that, etc. There can be more than one algorithm to solve the same problem.

If you've taken a probability course, you know that there is a formula for the *number* of possible combinations. The most traditional use of computers is to work through such formulas and compute numbers. However, not all problems are numeric. Lisp, the programming language family of which Scheme is a member, is unusual in its emphasis on *symbolic* computing. In this example, listing the actual combinations instead of just counting them is part of the flavor of symbolic computing, along with our earlier examples about manipulating words and phrases. We'll try to avoid numeric problems when possible, because symbolic computing is more fun for most people.

Example: Factorial

Scheme can handle numbers, too. The factorial of n (usually written in mathematical notation as $n!$) is the product of all the numbers from 1 to n:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))

> (factorial 4)
24

> (factorial 1000)
4023872600770937735437024339230039857193748642107146325437999104299385
1239862902059204420848696940480047998861019719605863166687299480855890
1323829669944590997424504087073759918823627727188732519779505950995276
1208749754624970436014182780946464962910563938874378864873371191810458
2578364784997701247663288983595573543251318532395846307555740911426241
7474349347553428646576611667797396668820291207379143853719588249808126
8678383745597317461360853795345242215865932019280908782973084313928444
0328123155861103697680135730421616874760967587134831202547858932076716
9132448426236131412508780208000261683151027341827977704784635868170164
3650241536913982812648102130927612448963599287051149649754199093422215
6683257208082133318611681155361583654698404670897560290095053761647584
7728421889679646244945160765353408198901385442487984959953319101723355
5566021394503997362807501378376153071277619268490343526252000158885351
4733161170210396817592151090778801939317811419454525722386554146106289
2187960223838971476088506276862967146674697562911234082439208160153780
8898939645182632436716167621791689097799119037540312746222899880051954
4441428201218736174599264295658174662830295557029902432415318161721046
5832036786906117260158783520751516284225540265170483304226143974286933
0616908979684825901254583271682264580665267699586526822728070757813918
```

```
5817888965220816434834482599326604336766017699961283186078838615027946
5955131156552036093988180612138558600301435694527224206344631797460594
6825731037900840244324384656572450144028218852524709351906209290231364
9327349756551395872055965422874977401141334696271542284586237738753823
0483865688976461927383814900140767310446640259899490222221765904339901
8860185665264850617997023561938970178600408118897299183110211712298459
0164192106888438712185564612496079872290851929681937238864261483965738
2291123125024186649353143970137428531926649875337218940694281434118520
1580141233448280150513996942901534830776445690990731524332782882698646
0278986432113908350621709500259738986355427719674282224875758676575234
4220207573630569498825087968928162753848863396909959826280956121450994
8717012445164612603790293091208890869420285106401821543994571568059418
7274899809425474217358240106367740459574178516082923013535808184009699
6372524230560855903700624271243416909004153690105933983835777939410970
0277534720000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000
```

If this doesn't work because your computer is too small, try a more reasonably sized example, such as the factorial of 200.

Play with the Procedures

This chapter has introduced a lot of new ideas at once, leaving out all the details. Our hope has been to convey the *flavor* of Scheme programming, before we get into Chapter 2, which is full of those missing details. But you can't absorb the flavor just by reading; take some time out to play with these examples before you go on.

Exercises

1.1 Do 20 push-ups.

1.2 Calculate 1000 factorial by hand and see if the computer got the right answer.

1.3 Create a file called `acronym.scm` containing our acronym program, using the text editor provided for use with your version of Scheme. Load the file into Scheme and run the program. Produce a transcript file called `acronym.log`, showing your interaction with Scheme as you test the program several times, and print it.