Once you see how it works, it's not so mysterious.
In Chapter 2 you used a program called functions to explore some of Scheme’s primitive functions. Now we’re going to go back to that program from the other point of view: instead of using the program to learn about functions, we’re going to look at how the program works as an example of programming with input and output.

The Main Loop

The functions program is an infinite loop similar to Scheme’s read-eval-print loop. It reads in a function name and some arguments, prints the result of applying that function to those arguments, and then does the whole thing over again.

There are some complexities, though. The functions program keeps asking you for arguments until it has enough. This means that the read portion of the loop has to read a function name, figure out how many arguments that procedure takes, and then ask for the right number of arguments. On the other hand, each argument is an implicitly quoted datum rather than an expression to be evaluated; the functions evaluator avoids the recursive complexity of arbitrary subexpressions within expressions. (That’s why we wrote it: to focus attention on one function invocation at a time, rather than on the composition of functions.) Here’s the main loop:

```
(define (functions-loop)
  (let ((fn-name (get-fn)))
    (if (equal? fn-name 'exit)
      "Thanks for using FUNCTIONS!"
      (let ((args (get-args (arg-count fn-name))))
        (if (not (in-domain? args fn-name))
          (show "Argument(s) not in domain."")
          (show-answer (apply (scheme-procedure fn-name) args)))
        (functions-loop))))
```
This invokes a lot of helper procedures. \textit{Arg-count} takes the name of a procedure as its argument and returns the number of arguments that the given procedure takes. \textit{In-domain?} takes a list and the name of a procedure as arguments; it returns \texttt{#t} if the elements of the list are valid arguments to the given procedure. \textit{Scheme-procedure} takes a name as its argument and returns the Scheme procedure with the given name. We’ll get back to these helpers later.

The other helper procedures are the ones that do the input and output. The actual versions are more complicated because of error checking; we’ll show them to you later.

\begin{verbatim}
(define (get-fn) ;; first version
  (display "Function: ")
  (read))

(define (get-args n)
  (if (= n 0)
    '()
    (let ((first (get-arg)))
      (cons first (get-args (- n 1))))))

(define (get-arg) ;; first version
  (display "Argument: ")
  (read))

(define (show-answer answer)
  (newline)
  (display "The result is: ")
  (if (not answer)
    (show "#F")
    (show answer))
  (newline))
\end{verbatim}

(That weird if expression in \textit{show-answer} is needed because in some old versions of Scheme the empty list means the same as \texttt{#f}. We wanted to avoid raising this issue in Chapter 2, so we just made sure that false values always printed as \texttt{#F}.)

\textbf{The Difference between a Procedure and Its Name}

You may be wondering why we didn’t just say

\begin{verbatim}
(show-answer (apply fn-name args))
\end{verbatim}
in the definition of functions-loop. Remember that the value of the variable fn-name comes from get-fn, which invokes read. Suppose you said

\begin{verbatim}
(define x (read))
\end{verbatim}

and then typed

\begin{verbatim}
(+ 2 3)
\end{verbatim}

The value of x would be the three element list (+ 2 3), not the number five.

Similarly, if you type “butfirst,” then read will return the word butfirst, not the procedure of that name. So we need a way to turn the name of a function into the procedure itself.

---

**The Association List of Functions**

We accomplish this by creating a huge association list that contains all of the functions the program knows about. Given a word, such as butfirst, we need to know three things:

- The Scheme procedure with that name (in this case, the butfirst procedure).
- The number of arguments required by the given procedure (one).*
- The types of arguments required by the given procedure (one word or sentence, which must not be empty).

We need to know the number of arguments the procedure requires because the program prompts the user individually for each argument; it has to know how many to ask for. Also, it needs to know the domain of each function so it can complain if the arguments you give it are not in the domain.**

This means that each entry in the association list is a list of four elements:

---

* Some Scheme procedures can accept any number of arguments, but for the purposes of the functions program we restrict these procedures to their most usual case, such as two arguments for +.

** Scheme would complain all by itself, of course, but would then stop running the functions program. We want to catch the error before Scheme does, so that after seeing the error message you’re still in functions. As we mentioned in Chapter 19, a program meant for beginners, such as the readers of Chapter 2, should be especially robust.
(define *the-functions* ;; partial listing
  (list (list '* * 2 (lambda (x y) (and (number? x) (number? y)))))
  (list '+ + 2 (lambda (x y) (and (number? x) (number? y)))))
  (list 'and (lambda (x y) (and x y)) 2
    (lambda (x y) (and (boolean? x) (boolean? y))))
  (list 'equal? equal? 2 (lambda (x y) #t))
  (list 'even? even? 1 integer?)
  (list 'word word 2 (lambda (x y) (and (word? x) (word? y)))))))

The real list is much longer, of course, but you get the idea.* It’s a convention in Scheme programming that names of global variables used throughout a program are surrounded by asterisks* to distinguish them from parameters of procedures.

Here are the selector procedures for looking up information in this a-list:

(define (scheme-procedure fn-name)
  (cadr (assoc fn-name *the-functions*)))

(define (arg-count fn-name)
  (caddr (assoc fn-name *the-functions*)))

(define (type-predicate fn-name)
  (cadddr (assoc fn-name *the-functions*)))

---

Domain Checking

Note that we represent the domain of a procedure by another procedure.** Each

---

* Since and is a special form, we can’t just say

(list 'and and 2 (lambda (x y) (and (boolean? x) (boolean? y))))

That’s because special forms can’t be elements of lists. Instead, we have to create a normal procedure that can be put in a list but computes the same function as and:

(l lambda (x y) (and x y))

We can get away with this because in the functions program we don’t care about argument evaluation, so it doesn’t matter that and is a special form. We do the same thing for if and or.

** The domain of a procedure is a set, and sets are generally represented in programs as lists. You might think that we’d have to store, for example, a list of all the legal arguments to butfirst. But that would be impossible, since that list would have to be infinitely large. Instead, we can take advantage of the fact that the only use we make of this set is membership testing, that is, finding out whether a particular argument is in a function’s domain.
domain-checking procedure, or *type predicate*, takes the same arguments as the procedure whose domain it checks. For example, the type predicate for + is

\[
\text{(lambda} \ (x \ y) \ \text{(and} \ (\text{number?} \ x) \ (\text{number?} \ y)))
\]

The type predicate returns \#t if its arguments are valid and \#f otherwise. So in the case of +, any two numbers are valid inputs, but any other types of arguments aren’t.

Here’s the in-domain? predicate:

\[
\text{(define} \ \text{(in-domain?} \ \text{args} \ \text{fn-name})
\]

\[
\text{(apply} \ \text{(type-predicate} \ \text{fn-name}) \ \text{args}))
\]

Of course, certain type predicates are applicable to more than one procedure. It would be silly to type

\[
\text{(lambda} \ (x \ y) \ \text{(and} \ (\text{number?} \ x) \ (\text{number?} \ y)))
\]

for +, -, =, and so on. Instead, we give this function a name:

\[
\text{(define} \ \text{(two-numbers?} \ x \ y)
\]

\[
\text{(and} \ (\text{number?} \ x) \ (\text{number?} \ y)))
\]

We then refer to the type predicate by name in the a-list:

\[
\text{(define} \ \text{*the-functions*} \ \text{;; partial listing, revised}
\]

\[
\text{(list} \ \text{(list} \ \text{’*} \ * \ 2 \ \text{two-numbers?})
\]

\[
\text{(list} \ \text{’+} \ + \ 2 \ \text{two-numbers?})
\]

\[
\text{(list} \ \text{’and} \ \text{(lambda} \ (x \ y) \ \text{(and} \ x \ y)) \ 2
\]

\[
\text{(lambda} \ (x \ y) \ \text{(and} \ \text{(boolean?} \ x) \ \text{(boolean?} \ y)))
\]

\[
\text{(list} \ \text{’equal?} \ \text{equal?} \ 2 \ \text{(lambda} \ (x \ y) \ \text{#t})
\]

\[
\text{(list} \ \text{’even?} \ \text{even?} \ 1 \ \text{integer?})
\]

\[
\text{(list} \ \text{’word} \ \text{word} \ 2 \ \text{(lambda} \ (x \ y) \ \text{(and} \ \text{(word?} \ x) \ \text{(word?} \ y))))
\]

Some of the type predicates are more complicated. For example, here’s the one for the member? and appearances functions:

\[
\text{(define} \ \text{(member-types-ok?} \ \text{small} \ \text{big})
\]

\[
\text{(and} \ \text{(word?} \ \text{small})
\]

\[
\text{(or} \ \text{(sentence?} \ \text{big}) \ \text{(and} \ \text{(word?} \ \text{big}) \ \text{(#=} \ \text{(count} \ \text{small}) \ 1))
\]
Item also has a complicated domain:

```lisp
(lambda (n stuff)
  (and (integer? n) (> n 0)
       (word-or-sent? stuff) (<= n (count stuff))))
```

This invokes `word-or-sent?`, which is itself the type predicate for the `count` procedure:

```lisp
(define (word-or-sent? x)
  (or (word? x) (sentence? x)))
```

On the other hand, some are less complicated. `Equal?` will accept any two arguments, so its type predicate is just

```lisp
(lambda (x y) #t)
```

The complete listing at the end of the chapter shows the details of all these procedures. Note that the `functions` program has a more restricted idea of domain than Scheme does. For example, in Scheme

```lisp
(and 6 #t)
```

returns `#t` and does not generate an error. But in the `functions` program the argument 6 is considered out of the domain.*

If you don’t like math, just ignore the domain predicates for the mathematical primitives; they involve facts about the domains of math functions that we don’t expect you to know.**

---

* Why did we choose to restrict the domain? We were trying to make the point that invoking a procedure makes sense only with appropriate arguments; that point is obscured by the complicating fact that Scheme interprets any non-`#f` value as true. In the `functions` program, where composition of functions is not allowed, there’s no benefit to Scheme’s more permissive rule.

** A reason that we restricted the domains of some mathematical functions is to protect ourselves from the fact that some version of Scheme support complex numbers while others do not. We wanted to write one version of `functions` that would work in either case; sometimes the easiest way to avoid possible problems was to restrict some function’s domain.
Intentionally Confusing a Function with Its Name

Earlier we made a big deal about the difference between a procedure and its name, to make sure you wouldn’t think you can apply the word butfirst to arguments. But the functions program completely hides this distinction from the user:

Function: count
Argument: butlast

The result is: 7

Function: every
Argument: butlast
Argument: (helter skelter)

The result is: (HELTE SKELTE)

When we give butlast as an argument to count, it’s as if we’d said

(count 'butlast)

In other words, it’s taken as a word. But when we give butlast as an argument to every, it’s as if we’d said

(every butlast '(helter skelter))

How can we treat some arguments as quoted and others not? The way this works is that everything is considered a word or a sentence by the functions program. The higher-order functions every and keep are actually represented in the functions implementation by Scheme procedures that take the name of a function as an argument, instead of a procedure itself as the ordinary versions do:

(define (named-every fn-name list)
  (every (scheme-procedure fn-name) list))

(define (named-keep fn-name list)
  (keep (scheme-procedure fn-name) list))

> (every first '(another girl))
(A G)
> (named-every 'first '(another girl))
(A G)
> (every 'first '(another girl))
ERROR: ATTEMPT TO APPLY NON-PROCEDURE FIRST

Chapter 21  Example: The Functions Program
This illustration hides a subtle point. When we invoked named-every at a Scheme prompt, we had to quote the word first that we used as its argument. But when you run the functions program, you don’t quote anything. The point is that functions provides an evaluator that uses a different notation from Scheme’s notation. It may be clearer if we show an interaction with an imaginary version of functions that does use Scheme notation:

Function: first
Non-Automatically-Quoted-Argument: ’datum

The result is: D

Function: first
Non-Automatically-Quoted-Argument: datum

ERROR: THE VARIABLE DATUM IS UNBOUND.

We didn’t want to raise the issue of quoting at that early point in the book, so we wrote functions so that every argument is automatically quoted. Well, if that’s the case, it’s true even when we’re invoking every. If you say

Function: every
Argument: first
...

then by the rules of the functions program, that argument is the quoted word first. So named-every, the procedure that pretends to be every in the functions world, has to “un-quote” that argument by looking up the corresponding procedure.

More on Higher-Order Functions

One of the higher-order functions that you can use in the functions program is called number-of-arguments. It takes a procedure (actually the name of a procedure, as we’ve just been saying) as argument and returns the number of arguments that that procedure accepts. This example is unusual because there’s no such function in Scheme. (It would be complicated to define, for one thing, because some Scheme procedures can accept a variable number of arguments. What should number-of-arguments return for such a procedure?)

The implementation of number-of-arguments makes use of the same a-list of functions that the functions evaluator itself uses. Since the functions program
needs to know the number of arguments for every procedure anyway, it's hardly any extra effort to make that information available to the user. We just add an entry to the a-list:

(list 'number-of-arguments arg-count 1 valid-fn-name?)

The type predicate merely has to check that the argument is found in the a-list of functions:

(define (valid-fn-name? name)
 (assoc name *the-functions*))

The type checking for the arguments to every and keep is unusually complicated because what's allowed as the second argument (the collection of data) depends on which function is used as the first argument. For example, it's illegal to compute

(every square '“(think for yourself)"

even though either of those two arguments would be allowable if we changed the other one:

> (every square '(3 4 5))
(9 16 25)

> (every first '“(think for yourself)"
(T F Y)

The type-checking procedures for every and keep use a common subprocedure. The one for every is

(lambda (fn stuff)
 (hof-types-ok? fn stuff word-or-sent?))

and the one for keep is

(lambda (fn stuff)
 (hof-types-ok? fn stuff boolean?))

The third argument specifies what types of results fn must return when applied to the elements of stuff.
(define (hof-types-ok? fn-name stuff range-predicate)
  (and (valid-fn-name? fn-name)
       (= 1 (arg-count fn-name))
       (word-or-sent? stuff)
       (empty? (keep (lambda (element)
                       (not ((type-predicate fn-name) element)))
                   stuff))
       (null? (filter (lambda (element)
                       (not (range-predicate element)))
                     (map (scheme-procedure fn-name)
                          (every (lambda (x) x) stuff))))))

This says that the function being used as the first argument must be a one-argument function (so you can’t say, for example, every of word and something); also, each element of the second argument must be an acceptable argument to that function. (If you keep the unacceptable arguments, you get nothing.) Finally, each invocation of the given function on an element of stuff must return an object of the appropriate type: words or sentences for every, true or false for keep.*

More Robustness

The program we’ve shown you so far works fine, as long as the user never makes a mistake. Because this program was written for absolute novices, we wanted to bend over backward to catch any kind of strange input they might give us.

Using read to accept user input has a number of disadvantages:

• If the user enters an empty line, read continues waiting silently for input.
• If the user types an unmatched open parenthesis, read continues reading forever.
• If the user types two expressions on a line, the second one will be taken as a response to the question the functions program hasn’t asked yet.

* That last argument to and is complicated. The reason we use map instead of every is that the results of the invocations of fn might not be words or sentences, so every wouldn’t accept them. But map has its own limitation: It won’t accept a word as the stuff argument. So we use every to turn stuff into a sentence—which, as you know, is really a list—and that’s guaranteed to be acceptable to map. (This is an example of a situation in which respecting a data abstraction would be too horrible to contemplate.)

376 Part VI Sequential Programming
We solve all these problems by using `read-line` to read exactly one line, even if it's empty or ill-formed, and then checking explicitly for possible errors.

`read-line` treats parentheses no differently from any other character. That's an advantage if the user enters mismatched or inappropriately nested parentheses. However, if the user correctly enters a sentence as an argument to some function, `read-line` will include the initial open parenthesis as the first character of the first word, and the final close parenthesis as the last character of the last word. `Get-arg` must correct for these extra characters.

Similarly, `read-line` treats number signs (`#`) like any other character, so it doesn’t recognize `#t` and `#f` as special values. Instead it reads them as the strings "#t" and "#f". `Get-arg` calls `booleanize` to convert those strings into Boolean values.

```scheme
(define (get-arg)
  (display "Argument: ")
  (let ((line (read-line)))
    (cond ((empty? line)
            (show "Please type an argument!")
            (get-arg))
          ((and (equal? "(" (first (first line)))
                (equal? ")" (last (last line))))
            (let ((sent (remove-first-paren (remove-last-paren line))))
                (if (any-parens? sent)
                    (begin (show "Sentences can’t have parentheses inside.")
                            (get-arg))
                    (map booleanize sent))))
          ((any-parens? line)
            (show "Bad parentheses")
            (get-arg))
          ((empty? (bf line)) (booleanize (first line)))
          (else (show "You typed more than one argument! Try again.")
                 (get-arg)))))

Get-arg invokes `any-parens?`, `remove-first-paren`, `remove-last-paren`, and `booleanize`, whose meanings should be obvious from their names. You can look up their definitions in the complete listing at the end of this chapter.

Get-fn is simpler than `get-arg`, because there’s no issue about parentheses, but it’s still much more complicated than the original version, because of error checking.
(define (get-fn)
  (display "Function: ")
  (let ((line (read-line)))
    (cond ((empty? line)
              (show "Please type a function!"))
         ((not (= (count line) 1))
              (show "You typed more than one thing! Try again.")
              (get-fn))
         ((not (valid-fn-name? (first line)))
              (show "Sorry, that’s not a function.")
              (get-fn))
         (else (first line))))

This version of get-fn uses valid-fn-name? (which you’ve already seen) to ensure that what the user types is the name of a function we know about.

There’s a problem with using read-line. As we mentioned in a pitfall in Chapter 20, reading some input with read and then reading the next input with read-line results in read-line returning an empty line left over by read. Although the functions program doesn’t use read, Scheme itself used read to read the (functions) expression that started the program. Therefore, get-fn’s first attempt to read a function name will see an empty line. To fix this problem, the functions procedure has an extra invocation of read-line:

(define (functions)
  (read-line)
  (show "Welcome to the FUNCTIONS program.")
  (functions-loop))

---

**Complete Program Listing**

;;; The functions program

(define (functions)
  ;; (read-line)
  (show "Welcome to the FUNCTIONS program.")
  (functions-loop))
(define (functions-loop)
  (let ((fn-name (get-fn)))
    (if (equal? fn-name 'exit)
      "Thanks for using FUNCTIONS!"
      (let ((args (get-args (arg-count fn-name))))
        (if (not (in-domain? args fn-name))
            (show "Argument(s) not in domain.")
            (show-answer (apply (scheme-function fn-name) args)))
        (functions-loop)))))

(define (get-fn)
  (display "Function: ")
  (let ((line (read-line)))
    (cond ((empty? line) (show "Please type a function!") (get-fn))
      ((not (= (count line) 1)) (show "You typed more than one thing! Try again.") (get-fn))
      ((not (valid-fn-name? (first line))) (show "Sorry, that’s not a function.") (get-fn))
      (else (first line))))))

(define (get-arg)
  (display "Argument: ")
  (let ((line (read-line)))
    (cond ((empty? line) (show "Please type an argument!") (get-arg))
      ((and (equal? "(" (first (first line)))
        (equal? ")" (last (last line))))
        (let ((sent (remove-first-paren (remove-last-paren line))))
          (if (any-parens? sent)
            (begin
              (show "Sentences can’t have parentheses inside.")
              (get-arg))
            (map booleanize sent))))
      ((any-parens? line) (show "Bad parentheses") (get-arg))
      ((empty? (bf line)) (booleanize (first line))
        (else (show "You typed more than one argument! Try again.") (get-arg)))))))

Chapter 21  Example: The Functions Program
(define (get-args n)
  (if (= n 0)
      '()
      (let ((first (get-arg)))
        (cons first (get-args (- n 1))))))

(define (any-parens? line)
  (let ((letters (accumulate word line)))
    (or (member? "(" letters)
        (member? ")" letters))))

(define (remove-first-paren line)
  (if (equal? (first line) "(
      (bf line)
    (se (bf (first line)) (bf line)))))

(define (remove-last-paren line)
  (if (equal? (last line) ")")
    (bl line)
    (se (bl line) (bl (last line)))))

(define (booleanize x)
  (cond ((equal? x "#t") #t)
        ((equal? x "#f") #f)
        (else x)))

(define (show-answer answer)
  (newline)
  (display "The result is: ")
  (if (not answer)
      (show "#F")
    (show answer))
  (newline))

(define (scheme-function fn-name)
  (cadr (assoc fn-name *the-functions*)))

(define (arg-count fn-name)
  (caddr (assoc fn-name *the-functions*)))

(define (type-predicate fn-name)
  (caddr (assoc fn-name *the-functions*)))

(define (in-domain? args fn-name)
  (apply (type-predicate fn-name) args))
;; Type predicates

(define (word-or-sent? x)  
  (or (word? x) (sentence? x)))

(define (not-empty? x)  
  (and (word-or-sent? x)  
       (not (empty? x))))

(define (two-numbers? x y)  
  (and (number? x) (number? y)))

(define (two-reals? x y)  
  (and (real? x) (real? y)))

(define (two-integers? x y)  
  (and (integer? x) (integer? y)))

(define (can-divide? x y)  
  (and (number? x) (number? y) (not (= y 0))))

(define (dividable-integers? x y)  
  (and (two-integers? x y) (not (= y 0))))

(define (trig-range? x)  
  (and (number? x)  
       (<= (abs x) 1)))

(define (hof-types-ok? fn-name stuff range-predicate)  
  (and (valid-fn-name? fn-name)  
       (= 1 (arg-count fn-name))  
       (word-or-sent? stuff)  
       (empty? (keep (lambda (element)  
                       (not ((type-predicate fn-name) element)))  
                   stuff))  
       (null? (filter (lambda (element)  
                        (not (range-predicate element)))  
                    (map (scheme-function fn-name)  
                         (every (lambda (x) x) stuff))))))

(define (member-types-ok? small big)  
  (and (word? small)  
       (or (sentence? big) (and (word? big) (= (count small) 1)))))

Chapter 21  Example: The Functions Program 381
;; Names of functions as functions

(define (named-every fn-name list)
  (every (scheme-function fn-name) list))

(define (named-keep fn-name list)
  (keep (scheme-function fn-name) list))

(define (valid-fn-name? name)
  (assoc name *the-functions*))

;; The list itself

(define *the-functions*
  (list (list '* * 2 two-numbers?)
        (list '+ + 2 two-numbers?)
        (list '− − 2 two-numbers?)
        (list '/ / 2 can-divide?)
        (list '< < 2 two-reals?)
        (list '≤ <= 2 two-reals?)
        (list '=' = 2 two-numbers?)
        (list '> > 2 two-reals?)
        (list '≥ >= 2 two-reals?)
        (list 'abs abs 1 real?)
        (list 'acos acos 1 trig-range?)
        (list 'and (lambda (x y) (and x y)) 2
                 (lambda (x y) (and (boolean? x) (boolean? y))))
        (list 'appearances appearances 2 member-types-ok?)
        (list 'asin asin 1 trig-range?)
        (list 'atan atan 1 number?)
        (list 'bf bf 1 not-empty?)
        (list 'bl bl 1 not-empty?)
        (list 'butfirst butfirst 1 not-empty?)
        (list 'butlast butlast 1 not-empty?)
        (list 'ceiling ceiling 1 real?)
        (list 'cos cos 1 number?)
        (list 'count count 1 word-or-sent?)
        (list 'equal? equal? 2 (lambda (x y) #t))
        (list 'even? even? 1 integer?)
        (list 'every named-every 2
                 (lambda (fn stuff)
                     (hof-types-ok? fn stuff word-or-sent?)))
        (list 'exit '() 0 '())
   ; in case user applies number-of-arguments to exit
        (list 'exp exp 1 number?)

382 Part VI Sequential Programming
Chapter 21  Example: The Functions Program

(list 'expt expt 2
  (lambda (x y)
    (and (number? x) (number? y)
      (or (not (real? x)) (>= x 0) (integer? y))))
)
(list 'first first 1 not-empty?)
(list 'floor floor 1 real?)
(list 'gcd gcd 2 two-integers?)
(list 'if (lambda (pred yes no) (if pred yes no)) 3
  (lambda (pred yes no) (boolean? pred)))
(list 'item item 2
  (lambda (n stuff)
    (and (integer? n) (> n 0)
      (word-or-sent? stuff) (<= n (count stuff))))))
(list 'keep named-keep 2
  (lambda (fn stuff)
    (hof-types-ok? fn stuff boolean?)))
(list 'last last 1 not-empty?)
(list 'lcm lcm 2 two-integers?)
(list 'log log 1 (lambda (x) (and (number? x) (not (= x 0)))))
(list 'max max 2 two-reals?)
(list 'member? member? 2 member-types-ok?)
(list 'min min 2 two-reals?)
(list 'modulo modulo 2 dividable-integers?)
(list 'not not 1 boolean?)
(list 'number-of-arguments arg-count 1 valid-fn-name?)
(list 'odd? odd? 1 integer?)
(list 'or (lambda (x y) (or x y)) 2
  (lambda (x y) (and (boolean? x) (boolean? y))))
(list 'quotient quotient 2 dividable-integers?)
(list 'random random 1 (lambda (x) (and (integer? x) (> x 0))))
(list 'remainder remainder 2 dividable-integers?)
(list 'round round 1 real?)
(list 'se se 2
  (lambda (x y) (and (word-or-sent? x) (word-or-sent? y))))
(list 'sentence sentence 2
  (lambda (x y) (and (word-or-sent? x) (word-or-sent? y))))
(list 'sentence? sentence? 1 (lambda (x) #t))
(list 'sin sin 1 number?)
(list 'sqrt sqrt 1 (lambda (x) (and (real? x) (>= x 0))))
(list 'tan tan 1 number?)
(list 'truncate truncate 1 real?)
(list 'vowel? (lambda (x) (member? x '(a e i o u))) 1
  (lambda (x) #t))
(list 'word word 2 (lambda (x y) (and (word? x) (word? y))))
(list 'word? word? 1 (lambda (x) #t)))
Exercises

21.1 The `get-args` procedure has a `let` that creates the variable `first`, and then that variable is used only once inside the body of the `let`. Why doesn’t it just say the following?

```scheme
(define (get-args n)
  (if (= n 0)
      '()
      (cons (get-arg) (get-args (- n 1)))))
```

21.2 The domain-checking function for `equal?` is

```scheme
(lambda (x y) #t)
```

This seems silly; it’s a function of two arguments that ignores both arguments and always returns `#t`. Since we know ahead of time that the answer is `#t`, why won’t it work to have `equal?`’s entry in the a-list be

```scheme
(list 'equal? equal? 2 #t)
```

21.3 Every time we want to know something about a function that the user typed in, such as its number of arguments or its domain-checking predicate, we have to do an `assoc` in `*the-functions*`. That’s inefficient. Instead, rewrite the program so that `get-fn` returns a function’s entry from the a-list, instead of just its name. Then rename the variable `fn-name` to `fn-entry` in the `functions-loop` procedure, and rewrite the selectors `scheme-procedure`, `arg-count`, and so on, so that they don’t invoke `assoc`.

21.4 Currently, the program always gives the message “argument(s) not in domain” when you try to apply a function to bad arguments. Modify the program so that each record in `*the-functions*` also contains a specific out-of-domain message like “both arguments must be numbers,” then modify `functions` to look up and print this error message along with “argument(s) not in domain.”
21.5 Modify the program so that it prompts for the arguments this way:

Function: if
First Argument: #t
Second Argument: paperback
Third Argument: writer

The result is: PAPERBACK

but if there’s only one argument, the program shouldn’t say First:

Function: sqrt
Argument: 36

The result is 6

21.6 The assoc procedure might return #f instead of an a-list record. How come it’s okay for arg-count to take the caddr of assoc’s return value if (caddr #f) is an error?

21.7 Why is the domain-checking predicate for the word? function

(lambda (x) #t)

instead of the following procedure?

(lambda (x) (word? x))

21.8 What is the value of the following Scheme expression?

(functions)

21.9 We said in the recursion chapters that every recursive procedure has to have a base case and a recursive case, and that the recursive case has to somehow reduce the size of the problem, getting closer to the base case. How does the recursive call in get-fn reduce the size of the problem?