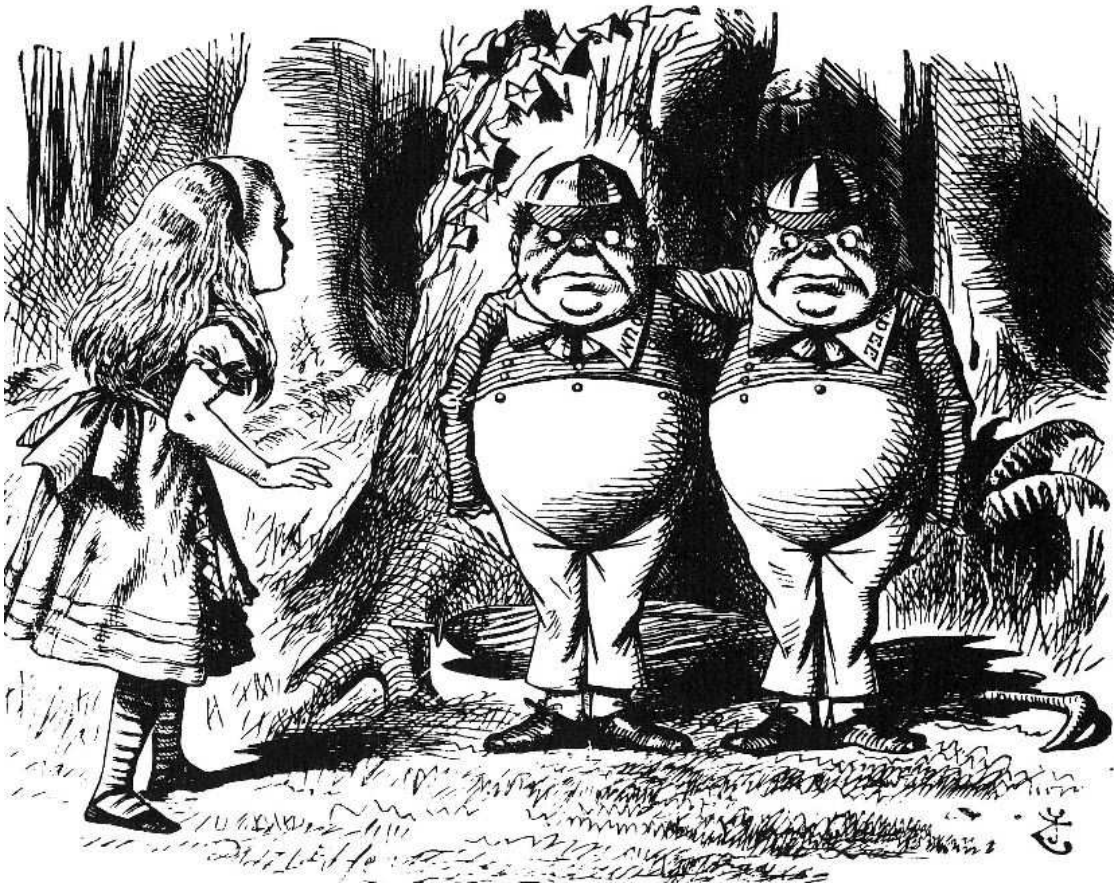

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

6 True and False

We still need one more thing before we can write more interesting programs: the ability to make decisions. Scheme has a way to say "if this is true, then do this thing, otherwise do something else."

Here's a procedure that greets a person:

```
(define (greet name)
  (if (equal? (first name) 'professor)
      (se '(i hope i am not bothering you) 'professor (last name))
      (se '(good to see you) (first name))))

> (greet '(matt wright))
(GOOD TO SEE YOU MATT)

> (greet '(professor harold abelson))
(I HOPE I AM NOT BOTHERING YOU PROFESSOR ABELSON)
```

The program greets a person by checking to see if that person is a professor. If so, it says, "I hope I am not bothering you" and then the professor's name. But if it's a regular person, the program just says, "Good to see you," and then the person's first name.

`If` takes three arguments. The first has to be either true or false. (We'll talk in a moment about exactly what true and false look like to Scheme.) In the above example, the first word of the person's name might or might not be equal to the word "Professor." The second and third arguments are expressions; one or the other of them is evaluated depending on the first argument. The value of the entire `if` expression is the value of either the second or the third argument.

You learned in Chapter 2 that Scheme includes a special data type called *Booleans* to represent true or false values. There are just two of them: `#t` for "true" and `#f` for

"false."*

We said that the first argument to `if` has to be true or false. Of course, it would be silly to say

```
> (if #t (+ 4 5) (* 2 7))
9
```

because what's the point of using `if` if we already know which branch will be followed? Instead, as in the `greet` example, we call some procedure whose return value will be either true or false, depending on the particular arguments we give it.

Predicates

A function that returns either `#t` or `#f` is called a *predicate*.** You've already seen the `equal?` predicate. It takes two arguments, which can be of any type, and returns `#t` if the two arguments are the same value, or `#f` if they're different. It's a convention in Scheme that the names of predicates end with a question mark, but that's just a convention. Here are some other useful predicates:

```
> (member? 'mick '(dave dee dozy beaky mick and tich))
#T
> (member? 'mick '(john paul george ringo))
#F
> (member? 'e 'truly)
#F
```

* In *some* versions of Scheme, the empty sentence is considered false. That is, `()` and `#f` may be the same thing. The reason that we can't be definite about this point is that older versions of Scheme follow the traditional Lisp usage, in which the empty sentence is false, but since then a standardization committee has come along and insisted that the two values should be different. In this book we'll consider them as different, but we'll try to avoid examples in which it matters. The main point is that you shouldn't be surprised if you see something like this:

```
> (= 3 4)
()
```

in the particular implementation of Scheme that you're using.

** Why is it called that? Think about an English sentence, such as "Ringo is a drummer." As you may remember from elementary school, "Ringo" is the *subject* of that sentence, and "is a drummer" is the *predicate*. That predicate could be truthfully attached to some subjects but not others. For example, it's true of "Neil Peart" but not of "George Harrison." So the predicate "is a drummer" can be thought of as a function whose value is true or false.

```
> (member? 'y 'truly)
#T
> (= 3 4)
#F
> (= 67 67)
#T
> (> 98 97)
#T
> (before? 'zorn 'coleman)
#F
> (before? 'pete 'ringo)
#T
> (empty? '(abbey road))
#F
> (empty? '())
#T
> (empty? 'hi)
#F
> (empty? (bf (bf 'hi)))
#T
> (empty? "")
#T
```

`Member?` takes two arguments; it checks to see if the first one is a member of the second. The `=`, `>`, `<`, `>=`, and `<=` functions take two numbers as arguments and do the obvious comparisons. (By the way, these are exceptions to the convention about question marks.) `Before?` is like `<`, but it compares two words alphabetically. `Empty?` checks to see if its argument is either the empty word or the empty sentence.

Why do we have both `equal?` and `=` in Scheme? The first of these works on any kind of Scheme data, while the second is defined only for numbers. You could get away with always using `equal?`, but the more specific form makes your program more self-explanatory; people reading the program know right away that you're comparing numbers.

There are also several predicates that can be used to test the type of their argument:

```
> (number? 'three)
#F
> (number? 74)
#T
> (boolean? #f)
#T
> (boolean? '(the beatles))
#F
```

```
> (boolean? 234)
#F
> (boolean? #t)
#T
> (word? 'flying)
#T
> (word? '(dig it))
#F
> (word? 87)
#T
> (sentence? 'wait)
#F
> (sentence? '(what goes on))
#T
```

Of course, we can also define new predicates:

```
(define (vowel? letter)
  (member? letter 'aeiou))

(define (positive? number)
  (> number 0))
```

Using Predicates

Here's a procedure that returns the absolute value of a number:

```
(define (abs num)
  (if (< num 0)
      (- num)
      num))
```

(If you call `-` with just one argument, it returns the negative of that argument.) Scheme actually provides `abs` as a primitive procedure, but we can redefine it.

Do you remember how to play buzz? You're all sitting around the campfire and you go around the circle counting up from one. Each person says a number. If your number is divisible by seven or if one of its digits is a seven, then instead of calling out your number, you say "buzz."

```
(define (buzz num)
  (if (or (divisible? num 7) (member? 7 num))
      'buzz
      num))
```

```
(define (divisible? big little)
  (= (remainder big little) 0))
```

`Or` can take any number of arguments, each of which must be true or false. It returns true if any of its arguments are true, that is, if the first argument is true *or* the second argument is true *or*... (`Remainder`, as you know, takes two integers and tells you what the remainder is when you divide the first by the second. If the remainder is zero, the first number is divisible by the second.)

`Or` is one of three functions in Scheme that combine true or false values to produce another true or false value. `And` returns true if all of its arguments are true, that is, the first *and* second *and*... Finally, there's a function `not` that takes exactly one argument, returning true if that argument is false and vice versa.

In the last example, the procedure we really wanted to write was `buzz`, but we found it useful to define `divisible?` also. It's quite common that the easiest way to solve some problem is to write a *helper procedure* to do part of the work. In this case the helper procedure computes a function that's meaningful in itself, but sometimes you'll want to write procedures with names like `buzz-helper` that are useful only in the context of one particular problem.

Let's write a program that takes a word as its argument and returns the plural of that word. Our first version will just put an "s" on the end:

```
(define (plural wd)
  (word wd 's))

> (plural 'beatle)
BEATLES

> (plural 'computer)
COMPUTERS

> (plural 'fly)
FLYS
```

This works for most words, but not those that end in "y." Here's version two:

```
(define (plural wd)
  (if (equal? (last wd) 'y)
      (word (bl wd) 'ies)
      (word wd 's)))
```

This isn't exactly right either; it thinks that the plural of "boy" is "boies." We'll ask you to add some more rules in Exercise 6.12.

`If` Is a Special Form

There are a few subtleties that we haven't told you about yet. First of all, `if` is a special form. Remember that we're going to need the value of only one of its last two arguments. It would be wasteful for Scheme to evaluate the other one. So if you say

```
(if (= 3 3)
    'sure
    (factorial 1000))
```

`if` won't compute the factorial of 1000 before returning `sure`.

The rule is that `if` always evaluates its first argument. If the value of that argument is true, then `if` evaluates its second argument and returns its value. If the value of the first argument is false, then `if` evaluates its third argument and returns that value.

So Are `And` and `Or`

`And` and `or` are also special forms. They evaluate their arguments in order from left to right* and stop as soon as they can. For `or`, this means returning true as soon as any of the arguments is true. `And` returns false as soon as any argument is false. This turns out to be useful in cases like the following:

```
(define (divisible? big small)
  (= (remainder big small) 0))

(define (num-divisible-by-4? x)
  (and (number? x) (divisible? x 4)))

> (num-divisible-by-4? 16)
#T
```

* Since you can start a new line in the middle of an expression, in some cases the arguments will be "top to bottom" rather than "left to right," but don't forget that Scheme doesn't care about line breaks. That's why Lisp programmers always talk as if their programs were written on one enormously long line.

```
> (num-divisible-by-4? 6)
#F

> (num-divisible-by-4? 'aardvark)
#F

> (divisible? 'aardvark 4)
ERROR: AARDVARK IS NOT A NUMBER
```

We want to see if `x` is a number, and, if so, if it's divisible by `4`. It would be an error to apply `divisible?` to a nonnumber. If `and` were an ordinary procedure, the two tests (`number?` and `divisible?`) would both be evaluated before we would have a chance to pay attention to the result of the first one. Instead, if `x` turns out not to be a number, our procedure will return `#f` without trying to divide it by `4`.

Everything That Isn't False Is True

`#T` isn't the only true value. In fact, *every* value is considered true except for `#f`.

```
> (if (+ 3 4) 'yes 'no)
YES
```

This allows us to have *semipredicates* that give slightly more information than just true or false. For example, we can write an integer quotient procedure. That is to say, our procedure will divide its first argument by the second, but only if the first is evenly divisible by the second. If not, our procedure will return `#f`.

```
(define (integer-quotient big little)
  (if (divisible? big little)
      (/ big little)
      #f))

> (integer-quotient 27 3)
9

> (integer-quotient 12 5)
#F
```

`And` and `or` are also semipredicates. We've already explained that `or` returns a true result as soon as it evaluates a true argument. The particular true value that `or` returns is the value of that first true argument:

```
> (or #f 3 #f 4)
3
```

`And` returns a true value only if all of its arguments are true. In that case, it returns the value of the last argument:

```
> (and 1 2 3 4 5)
5
```

As an example in which this behavior is useful, we can rewrite `integer-quotient` more tersely:

```
(define (integer-quotient big little)        ;; alternate version
  (and (divisible? big little)
       (/ big little)))
```

Decisions, Decisions, Decisions

`If` is great for an either-or choice. But sometimes there are several possibilities to consider:

```
(define (roman-value letter)
  (if (equal? letter 'i)
      1
      (if (equal? letter 'v)
          5
          (if (equal? letter 'x)
              10
              (if (equal? letter 'l)
                  50
                  (if (equal? letter 'c)
                      100
                      (if (equal? letter 'd)
                          500
                          (if (equal? letter 'm)
                              1000
                              'huh?)))))))))
```

That's pretty hideous. Scheme provides a shorthand notation for situations like this in which you have to choose from among several possibilities: the special form `cond`.

```
(define (roman-value letter)
  (cond ((equal? letter 'i) 1)
        ((equal? letter 'v) 5)
        ((equal? letter 'x) 10)
        ((equal? letter 'l) 50)
        ((equal? letter 'c) 100)
        ((equal? letter 'd) 500)
        ((equal? letter 'm) 1000)
        (else 'huh?)))
```

The tricky thing about `cond` is that it doesn't use parentheses in quite the same way as the rest of Scheme. Ordinarily, parentheses mean procedure invocation. In `cond`, *most* of the parentheses still mean that, but *some* of them are used to group pairs of tests and results. We've reproduced the same `cond` expression below, indicating the funny ones in boldface.

```
(define (roman-value letter)
  (cond (((equal? letter 'i) 1))
        (((equal? letter 'v) 5))
        (((equal? letter 'x) 10))
        (((equal? letter 'l) 50))
        (((equal? letter 'c) 100))
        (((equal? letter 'd) 500))
        (((equal? letter 'm) 1000))
        ((else 'huh?)) ))
```

`Cond` takes any number of arguments, each of which is *two expressions* inside a pair of parentheses. Each argument is called a *cond clause*. In the example above, one typical clause is

```
((equal? letter 'l) 50)
```

The outermost parentheses on that line enclose two expressions. The first of the two expressions (the *condition*) is taken as true or false, just like the first argument to `if`. The second expression of each pair (the *consequent*) is a candidate for the return value of the entire `cond` invocation.

`Cond` examines its arguments from left to right. Remember that since `cond` is a special form, its arguments are not evaluated ahead of time. For each argument, `cond` evaluates the first of the two expressions within the argument. If that value turns out to be true, then `cond` evaluates the second expression in the same argument, and returns

that value without examining any further arguments. But if the value is false, then `cond` does *not* evaluate the second expression; instead, it goes on to the next argument.

By convention, the last argument always starts with the word `else` instead of an expression. You can think of this as representing a true value, but `else` doesn't mean true in any other context; you're only allowed to use it as the condition of the last clause of a `cond`.*

Don't get into bad habits of thinking about the appearance of `cond` clauses in terms of "two parentheses in a row." That's often the case, but not always. For example, here is a procedure that translates Scheme true or false values (`#t` and `#f`) into more human-readable words `true` and `false`.

```
(define (truefalse value)
  (cond (value 'true)
        (else 'false)))

> (truefalse (= 2 (+ 1 1)))
TRUE

> (truefalse (= 5 (+ 2 2)))
FALSE
```

When a `cond` tests several possible conditions, they might not be mutually exclusive.** This can be either a source of error or an advantage in writing efficient programs. The trick is to make the *most restrictive* test first. For example, it would be an error to say

```
(cond ((number? (first sent)) ...)           ;; wrong
      ((empty? sent) ...)
      ...)
```

because the first test only makes sense once you've already established that there *is* a first word of the sentence. On the other hand, you don't have to say

```
(cond ((empty? sent) ...)
      ((and (not (empty? sent)) (number? (first sent))) ...)
      ...)
```

---

* What if you don't use an `else` clause at all? If none of the clauses has a true condition, then the return value is unspecified. In other words, always use `else`.

** Conditions are mutually exclusive if only one of them can be true at a time.

because you've already established that the sentence is nonempty if you get as far as the second clause.

`If` Is Composable

Suppose we want to write a `greet` procedure that works like this:

```
> (greet '(brian epstein))
(PLEASED TO MEET YOU BRIAN -- HOW ARE YOU?)

> (greet '(professor donald knuth))
(PLEASED TO MEET YOU PROFESSOR KNUTH -- HOW ARE YOU?)
```

The response of the program in these two cases is almost the same; the only difference is in the form of the person's name.

This procedure could be written in two ways:

```
(define (greet name)
  (if (equal? (first name) 'professor)
      (se '(pleased to meet you)
          'professor
          (last name)
          '(-- how are you?))
      (se '(pleased to meet you)
          (first name)
          '(-- how are you?))))

(define (greet name)
  (se '(pleased to meet you)
      (if (equal? (first name) 'professor)
          (se 'professor (last name))
          (first name))
      '(-- how are you?)))
```

The second version avoids repeating the common parts of the response by using `if` within a larger expression.

Some people find it counterintuitive to use `if` as we did in the second version. Perhaps the reason is that in some other programming languages, `if` is a "command" instead of a function like any other. A mechanism that selects one part of a program to run, and leaves out another part, may seem too important to be a mere argument

subexpression. But in Scheme, the value returned by *every* function can be used as part of a larger expression.*

We aren't saying anything new here. We've already explained the idea of composition of functions, and we're just making the same point again about `if`. But we've learned that many students expect `if` to be an exception, so we're taking the opportunity to emphasize the point: There are no exceptions to this rule.

Pitfalls

⇒ The biggest pitfall in this chapter is the unusual notation of `cond`. Keeping track of the parentheses that mean function invocation, as usual, and the parentheses that just group the parts of a `cond` clause is tricky until you get accustomed to it.

⇒ Many people also have trouble with the asymmetry of the `member?` predicate. The first argument is something small; the second is something big. (The order of arguments is the same as the order of a typical English sentence about membership: "Is Mick a member of the Beatles?") It seems pretty obvious when you look at an example in which both arguments are quoted constant values, but you can get in trouble when you define a procedure and use its parameters as the arguments to `member?`. Compare writing a procedure that says, "does the letter E appear in this word?" with one that says, "is this letter a vowel?"

⇒ Many people try to use `and` and `or` with the full flexibility of the corresponding English words. Alas, Scheme is not English. For example, suppose you want to know whether the argument to a procedure is either the word `yes` or the word `no`. You can't say

```
(equal? argument (or 'yes 'no))              ; wrong!
```

This sounds promising: "Is the `argument` `equal` to the word `yes` `or` the word `no`?" But the arguments to `or` must be true-or-false values, not things you want to check for

* Strictly speaking, since the argument expressions to a special form aren't evaluated, `if` is a function whose domain is expressions, not their values. But many special forms, including `if`, `and`, and `or`, are designed to act as if they were ordinary functions, the kind whose arguments Scheme evaluates in advance. The only difference is that it is sometimes possible for Scheme to figure out the correct return value after evaluating only some of the arguments. Most of the time we'll just talk about the domains and ranges of these special forms as if they were ordinary functions.

equality with something else. You have to make two separate equality tests:

```
(or (equal? argument 'yes) (equal? argument 'no))
```

In this particular case, you could also solve the problem by saying

```
(member? argument '(yes no))
```

but the question of trying to use `or` as if it were English comes up in other cases for which `member?` won't help.

⇒ This isn't exactly a pitfall, because it won't stop your program from working, but programs like

```
(define (odd? n)
  (if (not (even? n)) #t #f))
```

are redundant. Instead, you could just say

```
(define (odd? n)
  (not (even? n)))
```

since the value of `(not (even? n))` is already `#t` or `#f`.

Boring Exercises

6.1 What values are printed when you type these expressions to Scheme? (Figure it out in your head before you try it on the computer.)

```
(cond ((= 3 4) '(this boy))
      ((< 2 5) '(nowhere man))
      (else '(two of us)))

(cond (empty? 3)
      (square 7)
      (else 9))

(define (third-person-singular verb)
  (cond ((equal? verb 'be) 'is)
        ((equal? (last verb) 'o) (word verb 'es))
        (else (word verb 's))))

(third-person-singular 'go)
```

6.2 What values are printed when you type these expressions to Scheme? (Figure it out in your head before you try it on the computer.)

```
(or #f #f #f #t)

(and #f #f #f #t)

(or (= 2 3) (= 4 3))

(not #f)

(or (not (= 2 3)) (= 4 3))

(or (and (= 2 3) (= 3 3)) (and (< 2 3) (< 3 4)))
```

6.3 Rewrite the following procedure using a `cond` instead of the `if`s:

```
(define (sign number)
  (if (< number 0)
      'negative
      (if (= number 0)
          'zero
          'positive)))
```

6.4 Rewrite the following procedure using an `if` instead of the `cond`:

```
(define (utensil meal)
  (cond ((equal? meal 'chinese) 'chopsticks)
        (else 'fork)))
```

Real Exercises

Note: Writing helper procedures may be useful in solving some of these problems.

6.5 Write a procedure `european-time` to convert a time from American AM/PM notation into European 24-hour notation. Also write `american-time`, which does the opposite:

```
> (european-time '(8 am))
8
```

```
> (european-time '(4 pm))
16

> (american-time 21)
(9 PM)

> (american-time 12)
(12 PM)

> (european-time '(12 am))
24
```

Getting noon and midnight right is tricky.

6.6 Write a predicate `teen?` that returns true if its argument is between 13 and 19.

6.7 Write a procedure `type-of` that takes anything as its argument and returns one of the words `word`, `sentence`, `number`, or `boolean`:

```
> (type-of '(getting better))
SENTENCE

> (type-of 'revolution)
WORD

> (type-of (= 3 3))
BOOLEAN
```

(Even though numbers are words, your procedure should return `number` if its argument is a number.)

Feel free to check for more specific types, such as "positive integer," if you are so inclined.

6.8 Write a procedure `indef-article` that works like this:

```
> (indef-article 'beatle)
(A BEATLE)

> (indef-article 'album)
(AN ALBUM)
```

Don't worry about silent initial consonants like the `h` in `hour`.

6.9 Sometimes you must choose the singular or the plural of a word: `1 book` but `2 books`. Write a procedure `thismany` that takes two arguments, a number and a singular noun, and combines them appropriately:

```
> (thismany 1 'partridge)
(1 PARTRIDGE)

> (thismany 3 'french-hen)
(3 FRENCH-HENS)
```

6.10 Write a procedure `sort2` that takes as its argument a sentence containing two numbers. It should return a sentence containing the same two numbers, but in ascending order:

```
> (sort2 '(5 7))
(5 7)

> (sort2 '(7 5))
(5 7)
```

6.11 Write a predicate `valid-date?` that takes three numbers as arguments, representing a month, a day of the month, and a year. Your procedure should return `#t` if the numbers represent a valid date (e.g., it isn't the 31st of September). February has 29 days if the year is divisible by 4, except that if the year is divisible by 100 it must also be divisible by 400.

```
> (valid-date? 10 4 1949)
#T

> (valid-date? 20 4 1776)
#F

> (valid-date? 5 0 1992)
#F

> (valid-date? 2 29 1900)
#F

> (valid-date? 2 29 2000)
#T
```

6.12 Make `plural` handle correctly words that end in `y` but have a vowel before the `y`, such as `boy`. Then teach it about words that end in `x` (box). What other special cases can you find?

6.13 Write a better `greet` procedure that understands as many different kinds of names as you can think of:

```
> (greet '(john lennon))
(HELLO JOHN)

> (greet '(dr marie curie))
(HELLO DR CURIE)

> (greet '(dr martin luther king jr))
(HELLO DR KING)

> (greet '(queen elizabeth))
(HELLO YOUR MAJESTY)

> (greet '(david livingstone))
(DR LIVINGSTONE I PRESUME?)
```

6.14 Write a procedure `describe-time` that takes a number of seconds as its argument and returns a more useful description of that amount of time:

```
> (describe-time 45)
(45 SECONDS)

> (describe-time 930)
(15.5 MINUTES)

> (describe-time 30000000000)
(9.506426344208686 CENTURIES)
```