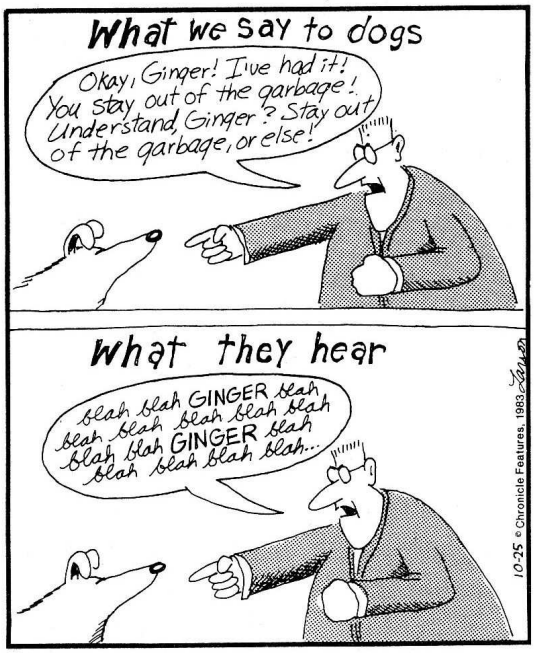
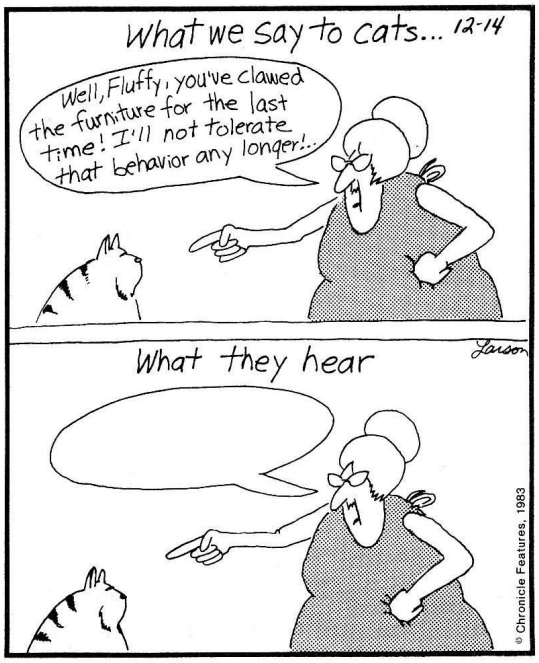
THE FAR SIDE By GARY LARSON



THE FAR SIDE By GARY LARSON



5 Words and Sentences

We started out, in Part I, with examples about acronyms and so on, but since then we've been working with numbery old numbers. That's because the discussions about evaluation and procedure definition were complicated enough without introducing extra ideas at the same time. But now we're ready to get back to symbolic programming.

As we mentioned in Chapter 3, everything that you type into Scheme is evaluated and the resulting value is printed out. Let's say you want to use "square" as a word in your program. For example, you want your program to solve the problem, "Give me an adjective that describes Barry Manilow." If you just type `square` into Scheme, you will find out that `square` is a procedure:

```
> square
#<PROCEDURE>
```

(Different versions of Scheme will have different ways of printing out procedures.)

What you need is a way to say that you want to use the word "`square`" *itself,* rather than the *value* of that word, as an expression. The way to do this is to use `quote`:

```
> (quote square)
SQUARE

> (quote (tomorrow never knows))
(TOMORROW NEVER KNOWS)

> (quote (things we said today))
(THINGS WE SAID TODAY)
```

`Quote` is a special form, since its argument isn't evaluated. Instead, it just returns the argument as is.

Scheme programmers use `quote` a lot, so there is an abbreviation for it:

```
> 'square
SQUARE

> '(old brown shoe)
(old brown shoe)
```

(Since Scheme uses the apostrophe as an abbreviation for `quote`, you can't use one as an ordinary punctuation mark in a sentence. That's why we've been avoiding titles like `(can't buy me love)`. To Scheme this would mean `(can (quote t) buy me love)`!)*

This idea of quoting, although it may seem arbitrary in the context of computer programming, is actually quite familiar from ordinary English. What is a book? It's a bunch of pieces of paper, with printing on them, bound together. What is "a book"? It's a noun phrase, made up of an article and a noun. See? Similarly, what's 2 + 3? It's five. What's "2 + 3"? It's an arithmetic formula. When you see words inside quotation marks, you understand that you're supposed to think about the words themselves; you don't evaluate what they mean. Scheme is the same way.

(It's no accident that kids who make jokes like

> Matt: "Say your name."
>
> Brian: "Your name."

grow up to be computer programmers. The difference between a thing and its name is one of the important ideas that programmers need to understand.)

---

* Actually, it *is* possible to put punctuation inside words as long as the entire word is enclosed in double-quote marks, like this:

```
> '("can't" buy me love)
("can't" BUY ME LOVE)
```

Words like that are called *strings*. We're not going to use them in any examples until almost the end of the book. Stay away from punctuation and you won't get in trouble. However, question marks and exclamation points are okay. (Ordinary words, the ones that are neither strings nor numbers, are officially called *symbols*.)

Selectors

So far all we've done with words and sentences is quote them. To do more interesting work, we need tools for two kinds of operations: We have to be able to take them apart, and we have to be able to put them together.* We'll start with the take-apart tools; the technical term for them is *selectors*.

```
> (first 'something)
S

> (first '(eight days a week))
EIGHT

> (first 910)
9

> (last 'something)
G

> (last '(eight days a week))
WEEK

> (last 910)
0

> (butfirst 'something)
OMETHING

> (butfirst '(eight days a week))
(DAYS A WEEK)

> (butfirst 910)
10

> (butlast 'something)
SOMETHIN
```

* The procedures we're about to show you are not part of standard, official Scheme. Scheme does provide ways to do these things, but the regular ways are somewhat more complicated and error-prone for beginners. We've provided a simpler way to do symbolic computing, using ideas developed as part of the Logo programming language.

```
> (butlast '(eight days a week))
(EIGHT DAYS A)

> (butlast 910)
91
```

Notice that the `first` of a sentence is a word, while the `first` of a word is a letter. (But there's no separate data type called "letter"; a letter is the same as a one-letter word.) The `butfirst` of a sentence is a sentence, and the `butfirst` of a word is a word. The corresponding rules hold for `last` and `butlast`.

The names `butfirst` and `butlast` aren't meant to describe ways to sled; they abbreviate "all `but` the `first`" and "all `but` the `last`."

You may be wondering why we're given ways to find the first and last elements but not the 42nd element. It turns out that the ones we have are enough, since we can use these primitive selectors to define others:

```
(define (second thing)
  (first (butfirst thing)))

> (second '(like dreamers do))
DREAMERS

> (second 'michelle)
I
```

There is, however, a primitive selector `item` that takes two arguments, a number *n* and a word or sentence, and returns the *n*th element of the second argument.

```
> (item 4 '(being for the benefit of mister kite!))
BENEFIT

> (item 4 'benefit)
E
```

Don't forget that a sentence containing exactly one word is different from the word itself, and selectors operate on the two differently:

```
> (first 'because)
B

> (first '(because))
BECAUSE
```

```
> (butfirst 'because)
ECAUSE

> (butfirst '(because))
()
```

The value of that last expression is the *empty sentence.* You can tell it's a sentence because of the parentheses, and you can tell it's empty because there's nothing between them.

```
> (butfirst 'a)
""

> (butfirst 1024)
"024"
```

As these examples show, sometimes `butfirst` returns a word that has to have double-quote marks around it. The first example shows the *empty word,* while the second shows a number that's not in its ordinary form. (Its numeric value is 24, but you don't usually see a zero in front.)

```
> 024
24

> "024"
"024"
```

We're going to try to avoid printing these funny words. But don't be surprised if you see one as the return value from one of the selectors for words. (Notice that you don't have to put a single quote in front of the double quotes. Strings are self-evaluating, just as numbers are.)

Since `butfirst` and `butlast` are so hard to type, there are abbreviations `bf` and `bl`. You can figure out which is which.

Constructors

Functions for putting things together are called *constructors.* For now, we just have two of them: `word` and `sentence`. `Word` takes any number of words as arguments and joins them all together into one humongous word:

```
> (word 'ses 'qui 'pe 'da 'lian 'ism)
SESQUIPEDALIANISM
```

```
> (word 'now 'here)
NOWHERE

> (word 35 893)
35893
```

`Sentence` is similar, but slightly different, since it can take both words and sentences as arguments:

```
>  (sentence 'carry 'that 'weight)
(CARRY THAT WEIGHT)

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

`Sentence` is also too hard to type, so there's the abbreviation `se`.

```
> (se '(one plus one) 'makes 2)
(ONE PLUS ONE MAKES 2)
```

By the way, why did we have to quote `makes` in the last example, but not `2`? It's because numbers are self-evaluating, as we said in Chapter 3. We have to quote `makes` because otherwise Scheme would look for something named `makes` instead of using the word itself. But numbers can't be the names of things; they represent themselves. (In fact, you could quote the `2` and it wouldn't make any difference—do you see why?)

First-Class Words and Sentences

If Scheme isn't your first programming language, you're probably accustomed to dealing with English text on a computer quite differently. Many other languages treat a sentence, for example, as simply a collection (a "string") of *characters* such as letters, spaces, and punctuation. Those languages don't help you maintain the two-level nature of English text, in which a sentence is composed of words, and a word is composed of letters.

Historically, computers just dealt with numbers. You could add two numbers, move a number from one place in the computer's memory to another place, and so on. Since each instruction in the computer's native *machine language* couldn't process anything larger than a number, programmers developed the attitude that a single number is a "real thing" while anything more complicated has to be considered as a collection of things, rather than as a single thing in itself.

The computer represents a text character as a single number. In many programming languages, therefore, a character is a "real thing," but a word or sentence is understood only as a collection of these character-code numbers.

But this isn't the way in which human beings normally think about their own language. To you, a word isn't primarily a string of characters (although it may temporarily seem like one if you're competing in a spelling bee). It's more like a single unit of meaning. Similarly, a sentence is a linguistic structure whose parts are words, not letters and spaces.

A programming language should let you express your ideas in terms that match *your* way of thinking, not the computer's way. Technically, we say that words and sentences should be *first-class data* in our language. This means that a sentence, for example, can be an argument to a procedure; it can be the value returned by a procedure; we can give it a name; and we can build aggregates whose elements are sentences. So far we've seen how to do the first two of these. We'll finish the job in Chapter 7 (on *variables*) and Chapter 17 (on *lists*).

Pitfalls

⇒ We've been avoiding apostrophes in our words and sentences because they're abbreviations for the `quote` special form. You must also avoid periods, commas, semicolons, quotation marks, vertical bars, and, of course, parentheses, since all of these have special meanings in Scheme. You may, however, use question marks and exclamation points.

⇒ Although we've already mentioned the need to avoid names of primitives when choosing formal parameters, we want to remind you specifically about the names `word` and `sentence`. These are often very tempting formal parameters, because many procedures have words or sentences as their domains. Unfortunately, if you choose these names for parameters, you won't be able to use the corresponding procedures within your definition.

```
(define (plural word)                          ;; wrong!
  (word word 's))

> (plural 'george)
ERROR: GEORGE isn't a procedure
```

The result of substitution was not, as you might think,

```
(word 'george 's)
```

but rather

```
('george 'george 's)
```

We've been using `wd` and `sent` as formal parameters instead of `word` and `sentence`, and we recommend that practice.

⇒ There's a difference between a word and a single-word sentence. For example, people often fall into the trap of thinking that the `butfirst` of a two-word sentence such as `(sexy sadie)` is the second word, but it's not. It's a one-word-long sentence. For example, its `count` is one, not five.*

```
> (bf '(sexy sadie))
(SADIE)

> (first (bf '(sexy sadie)))
SADIE
```

⇒ We mentioned earlier that sometimes Scheme has to put double-quote marks around words. Just ignore them; don't get upset if your procedure returns `"6-of-hearts"` instead of just `6-of-hearts`.

⇒ `Quote` doesn't mean "print." Some people look at interactions like this:

```
> '(good night)
(GOOD NIGHT)
```

and think that the quotation mark was an instruction telling Scheme to print what comes after it. Actually, Scheme *always* prints the value of each expression you type, as part of the read-eval-print loop. In this case, the value of the entire expression is the subexpression that's being quoted, namely, the sentence `(good night)`. That value wouldn't be printed if the quotation were part of some larger expression:

```
> (bf '(good night))
(NIGHT)
```

---

* You met `count` in Chapter 2. It takes a word or sentence as its argument, returning either the number of letters in the word or the number of words in the sentence.

⇒ If you see an error message like

```
> (+ 3 (bf 1075))
ERROR: INVALID ARGUMENT TO +: "075"
```

try entering the expression

```
> (strings-are-numbers #t)
OKAY
```

and try again. (The extension to Scheme that allows arithmetic operations to work on nonstandard numbers like `"075"` makes ordinary arithmetic slower than usual. So we've provided a way to turn the extension on and off. Invoking `strings-are-numbers` with the argument `#f` turns off the extension.)*

Boring Exercises

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

```
(sentence 'I '(me mine))

(sentence '() '(is empty))

(word '23 '45)

(se '23 '45)

(bf 'a)

(bf '(aye))

(count (first '(maggie mae)))

(se "" '() "" '())

(count (se "" '() "" '()))
```

* See Appendix A for a fuller explanation.

5.2 For each of the following examples, write a procedure of two arguments that, when applied to the sample arguments, returns the sample result. Your procedures may not include any quoted data.

```
> (f1 '(a b c) '(d e f))
(B C D E)

> (f2 '(a b c) '(d e f))
(B C D E AF)

> (f3 '(a b c) '(d e f))
(A B C A B C)

> (f4 '(a b c) '(d e f))
BE
```

5.3 Explain the difference in meaning between `(first 'mezzanine)` and `(first '(mezzanine))`.

5.4 Explain the difference between the two expressions `(first (square 7))` and `(first '(square 7))`.

5.5 Explain the difference between `(word 'a 'b 'c)` and `(se 'a 'b 'c)`.

5.6 Explain the difference between `(bf 'zabadak)` and `(butfirst 'zabadak)`.

5.7 Explain the difference between `(bf 'x)` and `(butfirst '(x))`.

5.8 Which of the following are legal Scheme sentences?

```
(here, there and everywhere)
(help!)
(all i've got to do)
(you know my name (look up the number))
```

5.9 Figure out what values each of the following will return *before* you try them on the computer:

```
(se (word (bl (bl (first '(make a))))
          (bf (bf (last '(baseball mitt)))))
    (word (first 'with) (bl (bl (bl (bl 'rigidly))))
          (first 'held) (first (bf 'stitches))))

(se (word (bl (bl 'bring)) 'a (last 'clean))
    (word (bl (last '(baseball hat))) (last 'for) (bl (bl 'very))
          (last (first '(sunny days)))))
```

5.10 What kinds of argument can you give `butfirst` so that it returns a word? A sentence?

5.11 What kinds of argument can you give `last` so that it returns a word? A sentence?

5.12 Which of the functions `first`, `last`, `butfirst`, and `butlast` can return an empty word? For what arguments? What about returning an empty sentence?

Real Exercises

5.13 What does `''banana` stand for?

What is `(first ''banana)` and why?

5.14 Write a procedure `third` that selects the third letter of a word (or the third word of a sentence).

5.15 Write a procedure `first-two` that takes a word as its argument, returning a two-letter word containing the first two letters of the argument.

```
> (first-two 'ambulatory)
AM
```

5.16 Write a procedure `two-first` that takes two words as arguments, returning a two-letter word containing the first letters of the two arguments.

```
> (two-first 'brian 'epstein)
BE
```

Now write a procedure `two-first-sent` that takes a two-word sentence as argument, returning a two-letter word containing the first letters of the two words.

```
> (two-first-sent '(brian epstein))
BE
```

5.17 Write a procedure `knight` that takes a person's name as its argument and returns the name with "Sir" in front of it.

```
> (knight '(david wessel))
(SIR DAVID WESSEL)
```

5.18 Try the following and explain the result:

```
(define (ends word)
  (word (first word) (last word)))

> (ends 'john)
```

5.19 Write a procedure `insert-and` that takes a sentence of items and returns a new sentence with an "and" in the right place:

```
> (insert-and '(john bill wayne fred joey))
(JOHN BILL WAYNE FRED AND JOEY)
```

5.20 Define a procedure to find somebody's middle names:

```
> (middle-names '(james paul mccartney))
(PAUL)

> (middle-names '(john ronald raoul tolkien))
(RONALD RAOUL)

> (middle-names '(bugs bunny))
()

> (middle-names '(peter blair denis bernard noone))
(BLAIR DENIS BERNARD)
```

5.21 Write a procedure `query` that turns a statement into a question by swapping the first two words and adding a question mark to the last word:

```
> (query '(you are experienced))
(ARE YOU EXPERIENCED?)

> (query '(i should have known better))
(SHOULD I HAVE KNOWN BETTER?)
```