```
(defun sentence ()    (append (noun-phrase) (verb-phrase)))
(defun noun-phrase () (append (Article) (Noun)))
(defun verb-phrase () (append (Verb) (noun-phrase)))
(defun Article ()     (one-of '(the a)))
(defun Noun ()        (one-of '(man ball woman table)))
(defun Verb ()        (one-of '(hit took saw liked)))
```

Each of these function definitions has an empty parameter list, `()`. That means the functions take no arguments. This is unusual because, strictly speaking, a function with no arguments would always return the same thing, so we would use a constant instead. However, these functions make use of the `random` function (as we will see shortly), and thus can return different results even with no arguments. Thus, they are not functions in the mathematical sense, but they are still called functions in Lisp, because they return a value.

All that remains now is to define the function `one-of`. It takes a list of possible choices as an argument, chooses one of these at random, and returns a one-element list of the element chosen. This last part is so that all functions in the grammar will return a list of words. That way, we can freely apply append to any category.

```
(defun one-of (set)
  "Pick one element of set, and make a list of it."
  (list (random-elt set)))

(defun random-elt (choices)
  "Choose an element from a list at random."
  (elt choices (random (length choices))))
```

There are two new functions here, `elt` and `random`. `elt` picks an element out of a list. The first argument is the list, and the second is the position in the list. The confusing part is that the positions start at 0, so `(elt choices 0)` is the first element of the list, and `(elt choices 1)` is the second. Think of the position numbers as telling you how far away you are from the front. The expression `(random n)` returns an integer from 0 to n-1, so that `(random 4)` would return either 0,1,2, or 3.

Now we can test the program by generating a few random sentences, along with a noun phrase and a verb phrase:

```
> (sentence) ⇒ (THE WOMAN HIT THE BALL)

> (sentence) ⇒ (THE WOMAN HIT THE MAN)

> (sentence) ⇒ (THE BALL SAW THE WOMAN)

> (sentence) ⇒ (THE BALL SAW THE TABLE)

> (noun-phrase) ⇒ (THE MAN)

> (verb-phrase) ⇒ (LIKED THE WOMAN)
> (trace sentence noun-phrase verb-phrase article noun verb) ⇒
(SENTENCE NOUN-PHRASE VERB-PHRASE ARTICLE NOUN VERB)

> (sentence) ⇒
(1 ENTER SENTENCE)
  (1 ENTER NOUN-PHRASE)
    (1 ENTER ARTICLE)
    (1 EXIT ARTICLE: (THE))
    (1 ENTER NOUN)
    (1 EXIT NOUN: (MAN))
  (1 EXIT NOUN-PHRASE: (THE MAN))
  (1 ENTER VERB-PHRASE)
    (1 ENTER VERB)
    (1 EXIT VERB: (HIT))
    (1 ENTER NOUN-PHRASE)
      (1 ENTER ARTICLE)
      (1 EXIT ARTICLE: (THE))
      (1 ENTER NOUN)
      (1 EXIT NOUN: (BALL))
    (1 EXIT NOUN-PHRASE: (THE BALL))
  (1 EXIT VERB-PHRASE: (HIT THE BALL))
(1 EXIT SENTENCE: (THE MAN HIT THE BALL))
(THE MAN HIT THE BALL)
```

The program works fine, and the trace looks just like the sample derivation above, but the Lisp definitions are a bit harder to read than the original grammar rules. This problem will be compounded as we consider more complex rules. Suppose we wanted to allow noun phrases to be modified by an indefinite number of adjectives and an indefinite number of prepositional phrases. In grammatical notation, we might have the following rules:

*Noun-Phrase* $\Rightarrow$ *Article* + *Adj** + *Noun* + *PP**
*Adj** $\Rightarrow \emptyset$, *Adj* + *Adj**
*PP** $\Rightarrow \emptyset$, *PP* + *PP**
*PP* $\Rightarrow$ *Prep* + *Noun-Phrase*
*Adj* $\Rightarrow$ *big, little, blue, green, . . .*
*Prep* $\Rightarrow$ *to, in, by, with, . . .*

In this notation, $\emptyset$ indicates a choice of nothing at all, a comma indicates a choice of several alternatives, and the asterisk is nothing special—as in Lisp, it's just part of the name of a symbol. However, the convention used here is that names ending in an asterisk denote zero or more repetitions of the underlying name. That is, *PP** denotes zero or more repetitions of *PP*. This is known as "Kleene star" notation (pronounced "clean-E") after the mathematician Stephen Cole Kleene.[1]

The problem is that the rules for *Adj** and *PP** contain choices that we would have to represent as some kind of conditional in Lisp. For example:

```
(defun Adj* ()
  (if (= (random 2) 0)
      nil
      (append (Adj) (Adj*))))

(defun PP* ()
  (if (random-elt '(t nil))
      (append (PP) (PP*))
      nil))

(defun noun-phrase () (append [illegible] (Adj*) (Noun) (PP*)))
(defun PP () (append (Prep) (noun-phrase)))
(defun Adj () (one-of '(big little blue green adiabatic)))
(defun Prep () (one-of '(to in by with on)))
```

I've chosen two different implementations for `Adj*` and `PP*`; either approach would work in either function. We have to be careful, though; here are two approaches that would not work:

```
(defun Adj* ()
  "Warning - incorrect definition of Adjectives."
  (one-of '(nil (append (Adj) (Adj*)))))

(defun Adj* ()
  "Warning - incorrect definition of Adjectives."
  (one-of (list nil (append (Adj) (Adj*)))))
```

The first definition is wrong because it could return the literal expression `((append (Adj) (Adj*)))` rather than a list of words as expected. The second definition would cause infinite recursion, because computing the value of `(Adj*)` always involves a recursive call to `(Adj*)`. The point is that what started out as simple functions are now becoming quite complex. To understand them, we need to know many Lisp conventions—`defun`, `()`, `case`, `if`, `quote`, and the rules for order of evaluation—when ideally the implementation of a grammar rule should use only *linguistic* conventions. If we wanted to develop a larger grammar, the problem could get worse, because the rule-writer might have to depend more and more on Lisp.

[1]We will soon see "Kleene plus" notation, wherein *PP+* denotes one or more repetition of *PP*.

## 2.3 A Rule-Based Solution

An alternative implementation of this program would concentrate on making it easy to write grammar rules and would worry later about how they will be processed. Let's look again at the original grammar rules:

*Sentence* ⇒ *Noun-Phrase* + *Verb-Phrase*
*Noun-Phrase* ⇒ *Article* + *Noun*
*Verb-Phrase* ⇒ *Verb* + *Noun-Phrase*
*Article* ⇒ *the, a, . . .*
*Noun* ⇒ *man, ball, woman, table . . .*
*Verb* ⇒ *hit, took, saw, liked . . .*

Each rule consists of an arrow with a symbol on the left-hand side and something on the right-hand side. The complication is that there can be two kinds of right-hand sides: a concatenated list of symbols, as in "*Noun-Phrase* ⇒ *Article* + *Noun*," or a list of alternate words, as in "*Noun* ⇒ *man, ball, . . .*" We can account for these possibilities by deciding that every rule will have a list of possibilities on the right-hand side, and that a concatenated list, *for example* "*Article* + *Noun*," will be represented as a Lisp list, *for example* "(Article Noun)". The list of rules can then be represented as follows:

```
(defparameter *simple-grammar*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Noun))
    (verb-phrase -> (Verb noun-phrase))
    (Article -> the a)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked))
  "A grammar for a trivial subset of English.")

(defvar *grammar* *simple-grammar*
  "The grammar used by generate.  Initially, this is
  *simple-grammar*, but we can switch to other grammers.")
```

Note that the Lisp version of the rules closely mimics the original version. In particular, I include the symbol "->", even though it serves no real purpose; it is purely decorative.

The special forms defvar and defparameter both introduce special variables and assign a value to them; the difference is that a *variable*, like *grammar*, is routinely changed during the course of running the program. A *parameter*, like *simple-grammar*, on the other hand, will normally stay constant. A change to a parameter is considered a change *to* the program, not a change *by* the program.

Once the list of rules has been defined, it can be used to find the possible rewrites of a given category symbol. The function assoc is designed for just this sort of task.

It takes two arguments, a "key" and a list of lists, and returns the first element of the list of lists that starts with the key. If there is none, it returns `nil`. Here is an example:

```
> (assoc 'noun *grammar*) ⇒ (NOUN -> MAN BALL WOMAN TABLE)
```

Although rules are quite simply implemented as lists, it is a good idea to impose a layer of abstraction by defining functions to operate on the rules. We will need three functions: one to get the right-hand side of a rule, one for the left-hand side, and one to look up all the possible rewrites (right-hand sides) for a category.

```
(defun rule-lhs (rule)
  "The left-hand side of a rule."
  (first rule))

(defun rule-rhs (rule)
  "The right-hand side of a rule."
  (rest (rest rule)))

(defun rewrites (category)
  "Return a list of the possible rewrites for this category."
  (rule-rhs (assoc category *grammar*)))
```

Defining these functions will make it easier to read the programs that use them, and it also makes changing the representation of rules easier, should we ever decide to do so.

We are now ready to address the main problem: defining a function that will generate sentences (or noun phrases, or any other category). We will call this function `generate`. It will have to contend with three cases: (1) In the simplest case, `generate` is passed a symbol that has a set of rewrite rules associated with it. We choose one of those at random, and then generate from that. (2) If the symbol has no possible rewrite rules, it must be a terminal symbol—a word, rather than a grammatical category—and we want to leave it alone. Actually, we return the list of the input word, because, as in the previous program, we want all results to be lists of words. (3) In some cases, when the symbol has rewrites, we will pick one that is a list of symbols, and try to generate from that. Thus, `generate` must also accept a list as input, in which case it should generate each element of the list, and then append them all together. In the following, the first clause in `generate` handles this case, while the second clause handles (1) and the third handles (2). Note that we used the `mappend` function from section 1.7 (page 18).

```
(defun generate (phrase)
  "Generate a random sentence or phrase"
  (cond ((listp phrase)
         (mappend #'generate phrase))
```

```
    ((rewrites phrase)
     (generate (random-elt (rewrites phrase))))
    (t (list phrase))))
```

Like many of the programs in this book, this function is short, but dense with information: the craft of programming includes knowing what *not* to write, as well as what to write.

This style of programming is called *data-driven* programming, because the data (the list of rewrites associated with a category) drives what the program does next. It is a natural and easy-to-use style in Lisp, leading to concise and extensible programs, because it is always possible to add a new piece of data with a new association without having to modify the original program.

Here are some examples of generate in use:

```
> (generate 'sentence) ⇒ (THE TABLE SAW THE BALL)

> (generate 'sentence) ⇒ (THE WOMAN HIT A TABLE)

> (generate 'noun-phrase) ⇒ (THE MAN)

> (generate 'verb-phrase) ⇒ (TOOK A TABLE)
```

There are many possible ways to write generate. The following version uses if instead of cond:

```
(defun generate (phrase)
  "Generate a random sentence or phrase"
  (if (listp phrase)
      (mappend #'generate phrase)
      (let ((choices (rewrites phrase)))
        (if (null choices)
            (list phrase)
            (generate (random-elt choices))))))
```

This version uses the special form let, which introduces a new variable (in this case, choices) and also binds the variable to a value. In this case, introducing the variable saves us from calling the function rewrites twice, as was done in the cond version of generate. The general form of a let form is:

```
(let ((var value)...)
  body-containing-vars)
```

let is the most common way of introducing variables that are not parameters of functions. One must resist the temptation to use a variable without introducing it: