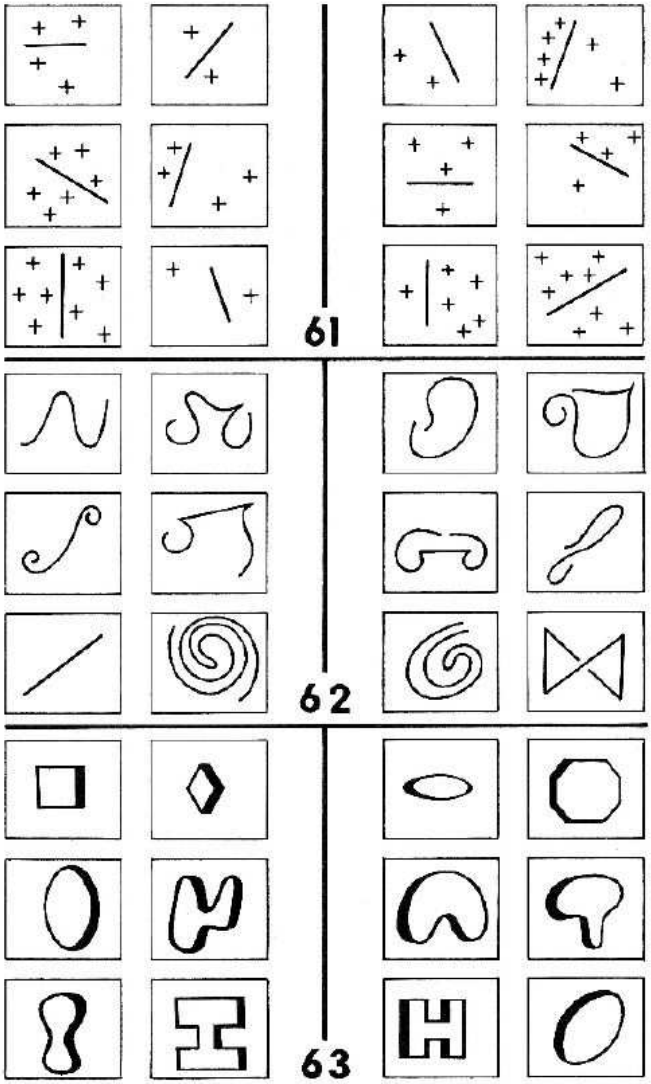

In each set, how do the ones on the left differ from the ones on the right?

16 Example: Pattern Matcher

It's time for another extended example in which we use the Scheme tools we've been learning to accomplish something practical. We'll start by describing how the program will work before we talk about how to implement it.

You can load our program into Scheme by typing

```
(load "match.scm")
```

Problem Description

A *pattern matcher* is a commonly used procedure whose job is to compare a sentence to a range of possibilities. An example may make this clear:

```
> (match '(* me *) '(love me do))
#T

> (match '(* me *) '(please please me))
#T

> (match '(* me *) '(in my life))
#F
```

The first argument, `(* me *)`, is a *pattern*. In the pattern, each asterisk (`*`) means "any number of words, including no words at all." So the entire pattern matches any sentence that contains the word "me" anywhere within it. You can think of `match` as a more general form of `equal?` in the sense that it compares two sentences and tells us whether they're the same, but with a broader meaning of "the same."

Our pattern matcher will accept patterns more complicated than this first example. There are four *special characters* that indicate unspecified parts of a pattern, depending on the number of words that should be allowed:

- `?` At most one word.
- `!` Exactly one word.
- `&` At least one word.
- `*` Any number of words.

These characters are meant to be somewhat mnemonic. The question mark means "maybe there's a word." The exclamation point means "precisely one word!" (And it's vertical, just like the digit 1, sort of.) The ampersand, which ordinarily means "and," indicates that we're matching a word and maybe more. The asterisk doesn't have any mnemonic value, but it's what everyone uses for a general matching indicator anyway.

We can give a *name* to the collection of words that match an unspecified part of a pattern by including in the pattern a word that starts with one of the four special characters and continues with the name. If the match succeeds, `match` will return a sentence containing these names and the corresponding values from the sentence:

```
> (match '(*start me *end) '(love me do))
(START LOVE ! END DO !)

> (match '(*start me *end) '(please please me))
(START PLEASE PLEASE ! END !)

> (match '(mean mr mustard) '(mean mr mustard))
()

> (match '(*start me *end) '(in my life))
FAILED
```

In these examples, you see that `match` doesn't really return `#t` or `#f`; the earlier set of examples showed a simplified picture. In the first of the new examples, the special pattern word `*start` is allowed to match any number of words, as indicated by the asterisk. In this case it turned out to match the single word "love." `Match` returns a result that tells us which words of the sentence match the named special words in the pattern. (We'll call one of these special pattern words a *placeholder*.) The exclamation points in the returned value are needed to separate one match from another. (In the second example, the name `end` was matched by an empty set of words.) In the third

example, the match was successful, but since there were no placeholders the returned sentence was empty. If the match is unsuccessful, `match` returns the word `failed`.*

If the same placeholder name appears more than once in the pattern, then it must be matched by the same word(s) in the sentence each time:

```
> (match '(!twice !other !twice) '(cry baby cry))
(TWICE CRY ! OTHER BABY !)

> (match '(!twice !other !twice) '(please please me))
FAILED
```

Some patterns might be matchable in more than one way. For example, the invocation

```
> (match '(*front *back) '(your mother should know))
```

might return any of five different correct answers:

```
(FRONT YOUR MOTHER SHOULD KNOW ! BACK !)
(FRONT YOUR MOTHER SHOULD ! BACK KNOW !)
(FRONT YOUR MOTHER ! BACK SHOULD KNOW !)
(FRONT YOUR ! BACK MOTHER SHOULD KNOW !)
(FRONT ! BACK YOUR MOTHER SHOULD KNOW !)
```

We arbitrarily decide that in such cases the first placeholder should match as many words as possible, so in this case `match` will actually return the first of these answers.

Before continuing, you might want to look at the first batch of exercises at the end of this chapter, which are about using the pattern matcher. (The rest of the exercises are about the implementation, which we'll discuss next.)

Implementation: When Are Two Sentences Equal?

Our approach to implementation will be to start with something we already know how

* Why not return the sentence if successful or `#f` otherwise? That would be fine in most versions of Scheme, but as we mentioned earlier, the empty sentence `()` is the same as the false value `#f` in some dialects. In those Schemes, a successfully matched pattern with no named placeholders, for which the program should return an empty sentence, would be indistinguishable from an unmatched pattern.

to write: a predicate that tests whether two sentences are exactly equal. We will add capabilities one at a time until we reach our goal.

Suppose that Scheme's primitive `equal?` function worked only for words and not for sentences. We could write an equality tester for sentences, like this:

```
(define (sent-equal? sent1 sent2)
  (cond ((empty? sent1)
         (empty? sent2))
        ((empty? sent2) #f)
        ((equal? (first sent1) (first sent2))
         (sent-equal? (bf sent1) (bf sent2)))
        (else #f)))
```

Two sentences are equal if each word in the first sentence is equal to the corresponding word in the second. They're unequal if one sentence runs out of words before the other.

Why are we choosing to accept Scheme's primitive word comparison but rewrite the sentence comparison? In our pattern matcher, a placeholder in the pattern corresponds to a group of words in the sentence. There is no kind of placeholder that matches only part of a word. (It would be possible to implement such placeholders, but we've chosen not to.) Therefore, we will never need to ask whether a word is "almost equal" to another word.

When Are Two Sentences Nearly Equal?

Pattern matching is just a more general form of this `sent-equal?` procedure. Let's write a very simple pattern matcher that knows only about the "!" special character and doesn't let us name the words that match the exclamation points in the pattern. We'll call this one `match?` with a question mark because it returns just true or false.

```
(define (match? pattern sent)                ;; first version: ! only
  (cond ((empty? pattern)
         (empty? sent))
        ((empty? sent) #f)
        ((equal? (first pattern) '!)
         (match? (bf pattern) (bf sent)))
        ((equal? (first pattern) (first sent))
         (match? (bf pattern) (bf sent)))
        (else #f)))
```

This program is exactly the same as `sent-equal?`, except for the highlighted `cond` clause. We are still comparing each word of the pattern with the corresponding word of the sentence, but now an exclamation mark in the pattern matches *any* word in the sentence. (If `first` of `pattern` is an exclamation mark, we don't even look at `first` of `sent`.)

Our strategy in the next several sections will be to expand the pattern matcher by implementing the remaining special characters, then finally adding the ability to name the placeholders. For now, when we say something like "the `*` placeholder," we mean the placeholder consisting of the asterisk alone. Later, after we add named placeholders, the same procedures will implement any placeholder that begins with an asterisk.

Matching with Alternatives

The `!` matching is not much harder than `sent-equal?`, because it's still the case that one word of the pattern must match one word of the sentence. When we introduce the `?` option, the structure of the program must be more complicated, because a question mark in the pattern might or might not be paired up with a word in the sentence. In other words, the pattern and the sentence might match without being the same length.

```
(define (match? pattern sent)                ;; second version: ! and ?
  (cond ((empty? pattern)
         (empty? sent))
        ((equal? (first pattern) '?)
         (if (empty? sent)
             (match? (bf pattern) '())
             (or (match? (bf pattern) (bf sent))
                 (match? (bf pattern) sent))))
        ((empty? sent) #f)
        ((equal? (first pattern) '!)
         (match? (bf pattern) (bf sent)))
        ((equal? (first pattern) (first sent))
         (match? (bf pattern) (bf sent)))
        (else #f)))
```

Note that the new `cond` clause comes *before* the check to see if `sent` is empty. That's because `sent` might be empty and a pattern of `(?)` would still match it. But if the sentence is empty, we know that the question mark doesn't match a word, so we just have to make sure that the `butfirst` of the pattern contains nothing but question marks. (We don't have a predicate named `all-question-marks?`; instead, we use `match?` recursively to make this test.)

In general, a question mark in the pattern has to match either one word or zero words in the sentence. How do we decide? Our rule is that each placeholder should match as many words as possible, so we prefer to match one word if we can. But allowing the question mark to match a word might prevent the rest of the pattern from matching the rest of the sentence.

Compare these two examples:

```
> (match? '(? please me) '(please please me))
#T

> (match? '(? please me) '(please me))
#T
```

In the first case, the first thing in the pattern is a question mark and the first thing in the sentence is "please," and they match. That leaves "please me" in the pattern to match "please me" in the sentence.

In the second case, we again have a question mark as the first thing in the pattern and "please" as the first thing in the sentence. But this time, we had better not use up the "please" in the sentence, because that will only leave "me" to match "please me." In this case the question mark has to match no words.

To you, these examples probably look obvious. That's because you're a human being, and you can take in the entire pattern and the entire sentence all at once. Scheme isn't as smart as you are; it has to compare words one pair at a time. To Scheme, the processing of both examples begins with question mark as the first word of the pattern and "please" as the first word of the sentence. The pattern matcher has to consider both cases.

How does the procedure consider both cases? Look at the invocation of `or` by the `match?` procedure. There are two alternatives; if either turns out true, the match succeeds. One is that we try to match the question mark with the first word of the sentence just as we matched `!` in our earlier example—by making a recursive call on the `butfirst`s of the pattern and sentence. If that returns true, then the question mark matches the first word.

The second alternative that can make the match succeed is a recursive call to `match?` on the `butfirst` of the pattern and the *entire* sentence; this corresponds to matching

the ? against nothing.*

Let's trace match? so that you can see how these two cases are handled differently by the program.

```
> (trace match?)
> (match? '(? please me) '(please please me))
(match? (? please me) (please please me))
| (match? (please me) (please me))                Try matching ? with please.
|  |  (match? (me) (me))
|  |  |  (match? () ())
|  |  |  #t                                       It works!
|  |  #t
|  #t
#t
#T


> (match? '(? please me) '(please me))
(match? (? please me) (please me))
| (match? (please me) (me))                       Try matching ? with please.
|  #f                                             It doesn't work.
| (match? (please me) (please me))                This time, match ? with nothing.
|  |  (match? (me) (me))
|  |  |  (match? () ())
|  |  |  #t
|  |  #t
|  #t
#t
#T
```

Backtracking

The program structure that allows for two alternative routes to success has more profound implications than you may think at first.

When match? sees a question mark in the pattern, it has to decide whether or not to "use up" a word of the sentence by matching it with the question mark. You might wonder, "How does the question mark decide whether to take a word?" The answer is that the decision isn't made "by the question mark"; there's nothing about the particular

* Actually, since or is a special form, Scheme avoids the need to try the second alternative if the first one succeeds.

word that the question mark might match that helps with the decision! Instead, the decision depends on matching what comes to the right of the question mark.

Compare this situation with the `keep` recursive pattern. There, too, the procedure makes a decision about the first word of a sentence, and each alternative leads to a recursive call for the `butfirst`:

```
(cond ((empty? sent) '())
      ((some-test? (first sent))
       (se (first sent) (recursive-call (bf sent))))
      (else (recursive-call (bf sent))))
```

The difference is that in the `keep` pattern the choice between alternatives *can* be made just by looking at the immediate situation—the single word that might or might not be chosen; the decision doesn't depend on anything in the rest of the problem. As a result, the choice has already been made before any recursive call happens. Therefore, only one of the recursive calls is actually made, to make choices about the remaining words in the sentence.

In `match?`, by contrast, any particular invocation can't make its choice until it knows the result of a recursive invocation. The result from the recursive call determines the choice made by the caller.

Here's a model that might help you think about this kind of recursion. `Match?` sees a question mark in the pattern. It makes a *tentative* decision that this question mark should match the first word of the sentence, and it uses a recursive invocation to see whether that decision allows the rest of the problem to be solved. If so, the tentative choice was correct. If not, `match?` tries an alternative decision that the question mark doesn't match a word. This alternative is still tentative; another recursive call is needed to see if the rest of the pattern can succeed. If not, the overall match fails.

This structure is called *backtracking*.

What if there are two question marks in the pattern? Then there are *four* ways to match the overall pattern. Both question marks can match a word, or only the first question mark, or only the second, or neither. A pattern with several placeholders leads to even more alternatives. A pattern with three question marks will have eight alternatives. (All three match words, the first two do but the third doesn't, and so on.) A pattern with 10 question marks will have 1024 alternatives. How can `match?` try all these alternatives? The procedure seems to make only one two-way choice; how can it accomplish a four-way or many-way decision?

The secret is the same as the usual secret of recursion: Most of the work is done in recursive calls. We take a leap of faith that recursive invocations will take care of the decisions concerning question marks later in the pattern. Think about it using the backtracking model. Let's suppose there are 10 question marks in the pattern. When `match?` encounters the leftmost question mark, it makes a tentative decision to match the question mark with a word of the sentence. To test whether this choice can work, `match?` invokes itself recursively on a pattern with nine question marks. By the leap of faith, the recursive invocation will examine 512 ways to match question marks with words—half of the total number. If one of these 512 works, we're finished. If not, the original `match?` invocation changes its tentative choice, deciding instead *not* to match its question mark (the leftmost one) with a word of the sentence. Another recursive call is made based on that decision, and that recursive call checks out the remaining 512 possibilities.

By the way, the program doesn't always have to try all of the different combinations of question marks matching or not matching words separately. For example, if the problem is

```
(match? '(a b ? ? ? ?) '(x y z w p q))
```

then the very first comparison discovers that `a` is different from `x`, so none of the 16 possible arrangements about question marks matching or not matching words will make a difference.

Here are some traced examples involving patterns with two question marks, to show how the result of backtracking depends on the individual problem.

```
> (match? '(? ? foo) '(bar foo))
(match? (? ? foo) (bar foo))
|  (match? (? foo) (foo))
|  |  (match? (foo) ())
|  |  #f
|  |  (match? (foo) (foo))
|  |  |  (match? () ())
|  |  |  #t
|  |  #t
|  #t
#t
#T
```

In this first example, the first question mark tries to match the word `bar`, but it can't tell whether or not that match will succeed until the recursive call returns. In the recursive call, the second question mark tries to match the word `foo`, and fails. Then the second

question mark tries again, this time matching nothing, and succeeds. Therefore, the first question mark can report success; it never has to try a recursive call in which it doesn't match a word.

In our second example, each question mark will have to try both alternatives, matching and then not matching a word, before the overall match succeeds.

```
> (match? '(? ? foo bar) '(foo bar))
(match? (? ? foo bar) (foo bar))
|  (match? (? foo bar) (bar))
|  |  (match? (foo bar) ())
|  |  #f
|  |  (match? (foo bar) (bar))
|  |  #f
|  #f
|  (match? (? foo bar) (foo bar))
|  |  (match? (foo bar) (bar))
|  |  #f
|  |  (match? (foo bar) (foo bar))
|  |  |  (match? (bar) (bar))
|  |  |  |  (match? () ())
|  |  |  |  #t
|  |  |  #t
|  |  #t
|  #t
#t
#T
```

The first question mark tries to match the word `foo` in the sentence, leaving the pattern `(? foo bar)` to match `(bar)`. The second question mark will try both matching and not matching a word, but neither succeeds. Therefore, the first question mark tries again, this time not matching a word. The second question mark first tries matching `foo`, and when that fails, tries not matching anything. This last attempt is successful.

In the previous example, every question mark's first attempt failed. The following example illustrates the opposite case, in which every question mark's first attempt succeeds.

```
> (match? '(? ? baz) '(foo bar baz))
(match? (? ? baz) (foo bar baz))
|  (match? (? baz) (bar baz))
|  |  (match? (baz) (baz))
|  |  |  (match? () ())
|  |  |  #t
|  |  #t
|  #t
#t
#t
```

The first question mark matches `foo`; the second matches `bar`.

If the sentence is shorter than the pattern, we may end up trying to match a pattern against an empty sentence. This is much easier than the general problem, because there aren't two alternatives; a question mark has no word in the sentence to match.

```
> (match? '(? ? foo) '())
(match? (? ? foo) ())
|  (match? (? foo) ())
|  |  (match? (foo) ())
|  |  #f
|  #f
#f
#f
```

Each question mark knows right away that it had better not try to match a word, so we never have to backtrack.

Matching Several Words

The next placeholder we'll implement is `*`. The order in which we're implementing these placeholders was chosen so that each new version increases the variability in the number of words a placeholder can match. The `!` placeholder was very easy because it always matches exactly one word; it's hardly different at all from a non-placeholder in the pattern. Implementing `?` was more complicated because there were two alternatives to consider. But for `*`, we might match any number of words, up to the entire rest of the sentence.

Our strategy will be a generalization of the `?` strategy: Start with a "greedy" match, and then, if a recursive call tells us that the remaining part of the sentence can't match the rest of the pattern, try a less greedy match.

The difference between `?` and `*` is that `?` allows only two possible match lengths, zero and one. Therefore, these two cases can be checked with two explicit subexpressions of an `or` expression. In the more general case of `*`, any length is possible, so we can't check every possibility separately. Instead, as in any problem of unknown size, we use recursion. First we try the longest possible match; if that fails because the rest of the pattern can't be matched, a recursive call tries the next-longest match. If we get all the way down to an empty match for the `*` and still can't match the rest of the pattern, then we return `#f`.

```
(define (match? pattern sent)              ;; third version: !, ?, and *
  (cond ((empty? pattern)
         (empty? sent))
        ((equal? (first pattern) '?)
         (if (empty? sent)
             (match? (bf pattern) '())
             (or (match? (bf pattern) (bf sent))
                 (match? (bf pattern) sent))))
        ((equal? (first pattern) '*)
         (*-longest-match (bf pattern) sent))
        ((empty? sent) #f)
        ((equal? (first pattern) '!)
         (match? (bf pattern) (bf sent)))
        ((equal? (first pattern) (first sent))
         (match? (bf pattern) (bf sent)))
        (else #f)))

(define (*-longest-match pattern-rest sent)
  (*-lm-helper pattern-rest sent '()))

(define (*-lm-helper pattern-rest sent-matched sent-unmatched)
  (cond ((match? pattern-rest sent-unmatched) #t)
        ((empty? sent-matched) #f)
        (else (*-lm-helper pattern-rest
                           (bl sent-matched)
                           (se (last sent-matched) sent-unmatched)))))
```

If an asterisk is found in the pattern, `match?` invokes `*-longest-match`, which carries out this backtracking approach.

The real work is done by `*-lm-helper`, which has three arguments. The first argument is the still-to-be-matched part of the pattern, following the `*` placeholder that we're trying to match now. `Sent-matched` is the part of the sentence that we're considering as a candidate to match the `*` placeholder. `Sent-unmatched` is

the remainder of the sentence, following the words in `sent-matched`; it must match `pattern-rest`.

Since we're trying to find the longest possible match, `*-longest-match` chooses the entire sentence as the first attempt for `sent-matched`. Since `sent-matched` is using up the entire sentence, the initial value of `sent-unmatched` is empty. The only job of `*-longest-match` is to invoke `*-lm-helper` with these initial arguments. On each recursive invocation, `*-lm-helper` shortens `sent-matched` by one word and accordingly lengthens `sent-unmatched`.

Here's an example in which the `*` placeholder tries to match four words, then three words, and finally succeeds with two words:

```
> (trace match? *-longest-match *-lm-helper)

> (match? '(* days night) '(a hard days night))
(match? (* days night) (a hard days night))
|  (*-longest-match (days night) (a hard days night))
|  |  (*-lm-helper (days night) (a hard days night) ())
|  |  |  (match? (days night) ())
|  |  |  #f
|  |  |  (*-lm-helper (days night) (a hard days) (night))
|  |  |  |  (match? (days night) (night))
|  |  |  |  #f
|  |  |  |  (*-lm-helper (days night) (a hard) (days night))
|  |  |  |  |  (match? (days night) (days night))
|  |  |  |  |  |  (match? (night) (night))
|  |  |  |  |  |  |  (match? () ())
|  |  |  |  |  |  |  #t
|  |  |  |  |  |  #t
|  |  |  |  |  #t
|  |  |  |  #t
|  |  |  #t
|  |  #t
|  #t
#t
#t
```

Combining the Placeholders

We have one remaining placeholder, `&`, which is much like `*` except that it fails unless it can match at least one word. We could, therefore, write a `&-longest-match` that would be identical to `*-longest-match` except for the base case of its helper procedure. If `sent-matched` is empty, the result is `#f` even if it would be possible to match the rest of

the pattern against the rest of the sentence. (All we have to do is exchange the first two clauses of the `cond`.)

```
(define (&-longest-match pattern-rest sent)
  (&-lm-helper pattern-rest sent '()))

(define (&-lm-helper pattern-rest sent-matched sent-unmatched)
  (cond ((empty? sent-matched) #f)
        ((match? pattern-rest sent-unmatched) #t)
        (else (&-lm-helper pattern-rest
                           (bl sent-matched)
                           (se (last sent-matched) sent-unmatched)))))
```

When two procedures are so similar, that's a clue that perhaps they could be combined into one. We could look at the bodies of these two procedures to find a way to combine them textually. But instead, let's step back and think about the meanings of the placeholders.

The reason that the procedures `*-longest-match` and `&-longest-match` are so similar is that the two placeholders have almost identical meanings. `*` means "match as many words as possible"; `&` means "match as many words as possible, but at least one." Once we're thinking in these terms, it's plausible to think of `?` as meaning "match as many words as possible, but at most one." In fact, although this is a stretch, we can also describe `!` similarly: "Match as many words as possible, but at least one, and at most one."

Placeholder	Minimum size	Maximum size
`*`	0	no limit
`&`	1	no limit
`?`	0	1
`!`	1	1

We'll take advantage of this newly understood similarity to simplify the program by using a single algorithm for all placeholders.

How do we generalize `*-longest-match` and `&-longest-match` to handle all four cases? There are two kinds of generalization involved. We'll write a procedure `longest-match` that will have the same arguments as `*-longest-match`, plus two others, one for for the minimum size of the matched text and one for the maximum.

We'll specify the minimum size with a formal parameter `min`. (The corresponding argument will always be 0 or 1.) `Longest-match` will pass the value of `min` down to `lm-helper`, which will use it to reject potential matches that are too short.

Unfortunately, we can't use a number to specify the maximum size, because for `*` and `&` there is no maximum. Instead, `longest-match` has a formal parameter `max-one?` whose value is `#t` only for `?` and `!`.

Our earlier, special-case versions of `longest-match` were written for `*` and `&`, the placeholders for which `max-one?` will be false. For those placeholders, the new `longest-match` will be just like the earlier versions. Our next task is to generalize `longest-match` so that it can handle the `#t` cases.

Think about the meaning of the `sent-matched` and `sent-unmatched` parameters in the `lm-helper` procedures. `Sent-matched` means "the longest part of the sentence that this placeholder is still allowed to match," while `sent-unmatched` contains whatever portion of the sentence has already been disqualified from being matched by the placeholder.

Consider the behavior of `*-longest-match` when an asterisk is at the beginning of a pattern that we're trying to match against a seven-word sentence. Initially, `sent-matched` is the entire seven-word sentence, and `sent-unmatched` is empty. Then, supposing that doesn't work, `sent-matched` is a six-word sentence, while `sent-unmatched` contains the remaining word. This continues as long as no match succeeds until, near the end of `longest-match`'s job, `sent-matched` is a one-word sentence and `sent-unmatched` contains six words. At this point, the longest possible match for the asterisk is a single word.

This situation is where we want to *start* in the case of the `?` and `!` placeholders. So when we're trying to match one of these placeholders, our initialization procedure won't use the entire sentence as the initial value of `sent-matched`; rather, the initial value of `sent-matched` will be a one-word sentence, and `sent-unmatched` will contain the rest of the sentence.

```
(define (longest-match pattern-rest sent min max-one?)  ;; first version
  (cond ((empty? sent)
         (and (= min 0) (match? pattern-rest sent)))
        (max-one?
         (lm-helper pattern-rest (se (first sent)) (bf sent) min))
        (else (lm-helper pattern-rest sent '() min))))
```

```
(define (lm-helper pattern-rest sent-matched sent-unmatched min)
  (cond ((< (length sent-matched) min) #f)
        ((match? pattern-rest sent-unmatched) #t)
        ((empty? sent-matched) #f)
        (else (lm-helper pattern-rest
                         (bl sent-matched)
                         (se (last sent-matched) sent-unmatched)
                         min))))
```

Now we can rewrite `match?` to use `longest-match`. `Match?` will delegate the handling of all placeholders to a subprocedure `match-special` that will invoke `longest-match` with the correct values for `min` and `max-one?` according to the table.

```
(define (match? pattern sent)                          ;; fourth version
  (cond ((empty? pattern)
         (empty? sent))
        ((special? (first pattern))
         (match-special (first pattern) (bf pattern) sent))
        ((empty? sent) #f)
        ((equal? (first pattern) (first sent))
         (match? (bf pattern) (bf sent)))
        (else #f)))

(define (special? wd)                                  ;; first version
  (member? wd '(* & ? !)))

(define (match-special placeholder pattern-rest sent)  ;; first version
  (cond ((equal? placeholder '?)
         (longest-match pattern-rest sent 0 #t))
        ((equal? placeholder '!)
         (longest-match pattern-rest sent 1 #t))
        ((equal? placeholder '*)
         (longest-match pattern-rest sent 0 #f))
        ((equal? placeholder '&)
         (longest-match pattern-rest sent 1 #f))))
```

Naming the Matched Text

So far we've worked out how to match the four kinds of placeholders and return a true or false value indicating whether a match is possible. Our program is almost finished; all we need to make it useful is the facility that will let us find out *which* words in the sentence matched each placeholder in the pattern.

We don't have to change the overall structure of the program in order to make this work. But most of the procedures in the pattern matcher will have to be given an additional argument, the database of placeholder names and values that have been matched so far.* The formal parameter `known-values` will hold this database. Its value will be a sentence containing placeholder names followed by the corresponding words and an exclamation point to separate the entries, as in the examples earlier in the chapter. When we begin the search for a match, we use an empty sentence as the initial `known-values`:

```
(define (match pattern sent)
  (match-using-known-values pattern sent '()))

(define (match-using-known-values pattern sent known-values)
  ...)
```

As `match-using-known-values` matches the beginning of a pattern with the beginning of a sentence, it invokes itself recursively with an expanded `known-values` containing each newly matched placeholder. For example, in evaluating

```
(match '(!twice !other !twice) '(cry baby cry))
```

the program will call `match-using-known-values` four times:

pattern	sent	known-values
(!twice !other !twice)	(cry baby cry)	()
(!other !twice)	(baby cry)	(twice cry !)
(!twice)	(cry)	(twice cry ! other baby !)
()	()	(twice cry ! other baby !)

In the first invocation, we try to match `!twice` against some part of the sentence.

* The word *database* has two possible meanings in computer science, a broad meaning and a narrow one. The broad meaning, which we're using here, is a repository of information to which the program periodically adds new items for later retrieval. The narrow meaning is a collection of information that's manipulated by a *database program,* which provides facilities for adding new information, modifying existing entries, selecting entries that match some specified criterion, and so on. We'll see a database program near the end of the book.

Since `!` matches exactly one word, the only possibility is to match the word `cry`. The recursive invocation, therefore, is made with the first words of the pattern and sentence removed, but with the match between `twice` and `cry` added to the database.

Similarly, the second invocation matches `!other` with `baby` and causes a third invocation with shortened pattern and sentence but a longer database.

The third invocation is a little different because the pattern contains the placeholder `!twice`, but the name `twice` is already in the database. Therefore, this placeholder can't match whatever word happens to be available; it must match the same word that it matched before. (Our program will have to check for this situation.) Luckily, the sentence does indeed contain the word `cry` at this position.

The final invocation reaches the base case of the recursion, because the pattern is empty. The value that `match-using-known-values` returns is the database in this invocation.

The Final Version

We're now ready to show you the final version of the program. The program structure is much like what you've seen before; the main difference is the database of placeholder names and values. The program must add entries to this database and must look for database entries that were added earlier. Here are the three most important procedures and how they are changed from the earlier version to implement this capability:

- `match-using-known-values`, essentially the same as what was formerly named `match?` except for bookkeeping details.
- `match-special`, similar to the old version, except that it must recognize the case of a placeholder whose name has already been seen. In this case, the placeholder can match only the same words that it matched before.
- `longest-match` and `lm-helper`, also similar to the old versions, except that they have the additional job of adding to the database the name and value of any placeholder that they match.

Here are the modified procedures. Compare them to the previous versions.

```
(define (match pattern sent)
  (match-using-known-values pattern sent '()))
```

```
(define (match-using-known-values pattern sent known-values)
  (cond ((empty? pattern)
         (if (empty? sent) known-values 'failed))
        ((special? (first pattern))
         (let ((placeholder (first pattern)))
           (match-special (first placeholder)
                          (bf placeholder)
                          (bf pattern)
                          sent
                          known-values)))
        ((empty? sent) 'failed)
        ((equal? (first pattern) (first sent))
         (match-using-known-values (bf pattern) (bf sent) known-values))
        (else 'failed)))

(define (match-special howmany name pattern-rest sent known-values)
  (let ((old-value (lookup name known-values)))
    (cond ((not (equal? old-value 'no-value))
           (if (length-ok? old-value howmany)
               (already-known-match
                  old-value pattern-rest sent known-values)
               'failed))
          ((equal? howmany '?)
           (longest-match name pattern-rest sent 0 #t known-values))
          ((equal? howmany '!)
           (longest-match name pattern-rest sent 1 #t known-values))
          ((equal? howmany '*)
           (longest-match name pattern-rest sent 0 #f known-values))
          ((equal? howmany '&)
           (longest-match name pattern-rest sent 1 #f known-values)))))

(define (longest-match name pattern-rest sent min max-one? known-values)
  (cond ((empty? sent)
         (if (= min 0)
             (match-using-known-values pattern-rest
                                       sent
                                       (add name '() known-values))
             'failed))
        (max-one?
         (lm-helper name pattern-rest (se (first sent))
                    (bf sent) min known-values))
        (else (lm-helper name pattern-rest
                         sent '() min known-values))))
```

```
(define (lm-helper name pattern-rest
                   sent-matched sent-unmatched min known-values)
  (if (< (length sent-matched) min)
      'failed
      (let ((tentative-result (match-using-known-values
                               pattern-rest
                               sent-unmatched
                               (add name sent-matched known-values))))
        (cond ((not (equal? tentative-result 'failed)) tentative-result)
              ((empty? sent-matched) 'failed)
              (else (lm-helper name
                               pattern-rest
                               (bl sent-matched)
                               (se (last sent-matched) sent-unmatched)
                               min
                               known-values))))))
```

We haven't listed all of the minor procedures that these procedures invoke. A complete listing is at the end of the chapter, but we hope that you have enough confidence about the overall program structure to be able to assume these small details will work. In the next few paragraphs we discuss some of the ways in which the procedures shown here differ from the earlier versions.

In the invocation of `match-special` we found it convenient to split the placeholder into its first character, the one that tells how many words can be matched, and the butfirst, which is the name of the placeholder.

What happens if `match-special` finds that the name is already in the database? In this situation, we don't have to try multiple possibilities for the number of words to match (the usual job of `longest-match`); the placeholder must match exactly the words that it matched before. In this situation, three things must be true in order for the match to succeed: (1) The first words of the `sent` argument must match the old value stored in the database. (2) The partial `pattern` that remains after this placeholder must match the rest of the `sent`. (3) The old value must be consistent with the number of words permitted by the `howmany` part of the placeholder. For example, if the pattern is

```
(*stuff and !stuff)
```

and the database says that the placeholder `*stuff` was matched by three words from the sentence, then the second placeholder `!stuff` can't possibly be matched because it accepts only one word. This third condition is actually checked first, by `length-ok?`, and if we pass that hurdle, the other two conditions are checked by `already-known-match`.

The only significant change to `longest-match` is that it invokes `add` to compute an expanded database with the newly found match added, and it uses the resulting database as an argument to `match-using-known-values`.

Abstract Data Types

As you know, a database of known values is represented in this program as a sentence in which the entries are separated by exclamation points. Where is this representation accomplished in the program you've seen? There's nothing like

```
... (sentence old-known-values name value '!) ...
```

anywhere in the procedures we've shown. Instead, the program makes reference to the database of known values through two procedure calls:

```
(lookup name known-values)                    ; in match-special
(add name matched known-values)               ; in longest-match
```

Only the procedures `lookup` and `add` manipulate the database of known values:

```
(define (lookup name known-values)
  (cond ((empty? known-values) 'no-value)
        ((equal? (first known-values) name)
         (get-value (bf known-values)))
        (else (lookup name (skip-value known-values)))))

(define (get-value stuff)
  (if (equal? (first stuff) '!)
      '()
      (se (first stuff) (get-value (bf stuff)))))

(define (skip-value stuff)
  (if (equal? (first stuff) '!)
      (bf stuff)
      (skip-value (bf stuff))))

(define (add name value known-values)
  (if (empty? name)
      known-values
      (se known-values name value '!)))
```

These procedures are full of small details. For example, it's a little tricky to extract the part of a sentence from a name to the next exclamation point. It's convenient that we could write the more important procedures, such as `longest-match`, without filling them with these details. As far as `longest-match` knows, `lookup` and `add` could be Scheme primitive procedures. In effect we've created a new data type, with `add` as its constructor and `lookup` as its selector.

Types such as these, that are invented by a programmer and aren't part of the Scheme language itself, are called *abstract data types*. Creating an abstract data type means drawing a barrier between an idea about some kind of information we want to model in a program and the particular mechanism that we use to represent the information. In this case, the information is a collection of name-value associations, and the particular mechanism is a sentence with exclamation points and so on. The pattern matcher doesn't think of the database as a sentence. For example, it would be silly to translate the database into Pig Latin or find its acronym.

Just as we distinguish the *primitive* procedures that Scheme knows all along from the *compound* procedures that the Scheme programmer defines, we could use the names "primitive data type" for types such as numbers and Booleans that are built into Scheme and "compound data type" for ones that the programmer invents by defining selectors and constructors. But "compound data type" is a bit of a pun, because it also suggests a data type built out of smaller pieces, just as a compound expression is built of smaller expressions. Perhaps that's why the name "abstract data type" has become generally accepted. It's connected to the idea of abstraction that we introduced earlier, because in order to create an abstract data type, we must specify the selectors and constructors and give names to those patterns of computation.

Backtracking and `Known-Values`

What happens to the database in cases that require backtracking, where a particular recursive call might be "on the wrong track"? Let's trace `match-using-known-values` and see what happens. (We'll use the little-people model to discuss this example, and so we're annotating each invocation in the trace with the name of its little person.)

```
> (trace match-using-known-values)
> (match '(*start me *end) '(love me do))
(match-using-known-values (*start me *end) (love me do) ())          Martha
|  (match-using-known-values (me *end) () (start love me do !))      Mercutio
|  failed
|  (match-using-known-values (me *end) (do) (start love me !))       Masayuki
|  failed
|  (match-using-known-values (me *end) (me do) (start love !))       Mohammad
|  |  (match-using-known-values (*end) (do) (start love !))          Mae
|  |  |  (match-using-known-values () () (start love ! end do !))    Merlin
|  |  |  (start love ! end do !)
|  |  (start love ! end do !)
|  (start love ! end do !)
(start love ! end do !)
(START LOVE ! END DO !)
```

Martha, the first little person shown, has an empty `known-values`. She makes three attempts to match `*start` with parts of the sentence. In each case, a little person is hired with the provisional match in his or her `known-values`. (Actually, Martha does not directly hire Mercutio and the others. Martha hires a `match-special` little person, who in turn hires a `longest-match` specialist, who hires an `lm-helper` specialist, who hires Mercutio. But that added complexity isn't important for the point we're focusing on right now, namely, how backtracking can work. Pretend Martha hires Mercutio.)

If you don't use the little-people model, but instead think about the program as if there were just one `known-values` variable, then the backtracking can indeed be very mysterious. Once a provisional match is added to the database, how is it ever removed? The answer is that it doesn't work that way. There isn't a "the" database. Instead, each little person has a separate database. If an attempted match fails, the little person who reports the failure just stops working. For example, Martha hires Mercutio to attempt a match in which the name `start` has the value `love me do`. Mercutio is unable to complete the match, and reports failure. It is Martha, not Mercutio, who then hires Masayuki to try another value for `start`. Martha's database hasn't changed, so Martha gives Masayuki a database that reflects the new trial value but not the old one.

Not every hiring of a little person starts from an empty database. When a match is partially successful, the continuation of the same attempt must benefit from the work that's already been done. So, for example, when Mohammad hires Mae, and when Mae hires Merlin, each of them passes on an extended database, not an empty one. Specifically, Mae gives Merlin the new match of the name `end` with the value `do`, but also the match of `start` with `love` that she was given by Mohammad.

So as you can see, we don't have to do anything special to keep track of our database when we backtrack; the structure of the recursion takes care of everything for free.

How We Wrote It

For explanatory purposes we've chosen to present the pieces of this program in a different order from the one in which we actually wrote them. We *did* implement the easy placeholders (`!` and `?`) before the harder ones. But our program had provision for a database of names from the beginning.

There is no "right" way to approach a programming problem. Our particular approach was determined partly by our past experience. Each of us had written similar programs before, and we had preconceived ideas about the easy and hard parts. You might well start at a different point. For example, here is an elegant small program we'd both been shown by friends:

```
(define (match? pattern sent)
  (cond ((empty? pattern) (empty? sent))
        ((empty? sent)
         (and (equal? (first pattern) '*) (match? (bf pattern) sent)))
        ((equal? (first pattern) '*)
         (or (match? pattern (bf sent))
             (match? (bf pattern) sent)))
        (else (and (equal? (first pattern) (first sent))
                   (match? (bf pattern) (bf sent))))))
```

What's appealing about this is the funny symmetry of taking the `butfirst` of the pattern *or* of the sentence. That's not something you'd naturally think of, probably, but once you've worked out how it can work, it affects your preconceptions when you set out to write a pattern matcher yourself.

Based on that inspiration, we might well have started with the hard cases (such as `*`), with the idea that once they're in place, the easy cases won't change the program structure much.

Complete Program Listing

```
(define (match pattern sent)
  (match-using-known-values pattern sent '()))
```

```
(define (match-using-known-values pattern sent known-values)
  (cond ((empty? pattern)
         (if (empty? sent) known-values 'failed))
        ((special? (first pattern))
         (let ((placeholder (first pattern)))
           (match-special (first placeholder)
                          (bf placeholder)
                          (bf pattern)
                          sent
                          known-values)))
        ((empty? sent) 'failed)
        ((equal? (first pattern) (first sent))
         (match-using-known-values (bf pattern) (bf sent) known-values))
        (else 'failed)))

(define (special? wd)
  (member? (first wd) '(* & ? !)))

(define (match-special howmany name pattern-rest sent known-values)
  (let ((old-value (lookup name known-values)))
    (cond ((not (equal? old-value 'no-value))
           (if (length-ok? old-value howmany)
               (already-known-match
                  old-value pattern-rest sent known-values)
               'failed))
          ((equal? howmany '?)
           (longest-match name pattern-rest sent 0 #t known-values))
          ((equal? howmany '!)
           (longest-match name pattern-rest sent 1 #t known-values))
          ((equal? howmany '*)
           (longest-match name pattern-rest sent 0 #f known-values))
          ((equal? howmany '&)
           (longest-match name pattern-rest sent 1 #f known-values)))))

(define (length-ok? value howmany)
  (cond ((empty? value) (member? howmany '(? *)))
        ((not (empty? (bf value))) (member? howmany '(* &)))
        (else #t)))

(define (already-known-match value pattern-rest sent known-values)
  (let ((unmatched (chop-leading-substring value sent)))
    (if (not (equal? unmatched 'failed))
        (match-using-known-values pattern-rest unmatched known-values)
        'failed)))
```

```
(define (chop-leading-substring value sent)
  (cond ((empty? value) sent)
        ((empty? sent) 'failed)
        ((equal? (first value) (first sent))
         (chop-leading-substring (bf value) (bf sent)))
        (else 'failed)))

(define (longest-match name pattern-rest sent min max-one? known-values)
  (cond ((empty? sent)
         (if (= min 0)
             (match-using-known-values pattern-rest
                                       sent
                                       (add name '() known-values))
             'failed))
        (max-one?
         (lm-helper name pattern-rest (se (first sent))
                    (bf sent) min known-values))
        (else (lm-helper name pattern-rest
                         sent '() min known-values))))

(define (lm-helper name pattern-rest
                   sent-matched sent-unmatched min known-values)
  (if (< (length sent-matched) min)
      'failed
      (let ((tentative-result (match-using-known-values
                               pattern-rest
                               sent-unmatched
                               (add name sent-matched known-values))))
        (cond ((not (equal? tentative-result 'failed)) tentative-result)
              ((empty? sent-matched) 'failed)
              (else (lm-helper name
                               pattern-rest
                               (bl sent-matched)
                               (se (last sent-matched) sent-unmatched)
                               min
                               known-values))))))

;;; Known values database abstract data type

(define (lookup name known-values)
  (cond ((empty? known-values) 'no-value)
        ((equal? (first known-values) name)
         (get-value (bf known-values)))
        (else (lookup name (skip-value known-values)))))
```

```
(define (get-value stuff)
  (if (equal? (first stuff) '!)
      '()
      (se (first stuff) (get-value (bf stuff)))))

(define (skip-value stuff)
  (if (equal? (first stuff) '!)
      (bf stuff)
      (skip-value (bf stuff))))

(define (add name value known-values)
  (if (empty? name)
      known-values
      (se known-values name value '!)))
```

Exercises about Using the Pattern Matcher

16.1 Design and test a pattern that matches any sentence containing the word `C` three times (not necessarily next to each other).

16.2 Design and test a pattern that matches a sentence consisting of two copies of a smaller sentence, such as `(a b a b)`.

16.3 Design and test a pattern that matches any sentence of no more than three words.

16.4 Design and test a pattern that matches any sentence of at least three words.

16.5 Show sentences of length 2, 3, and 4 that match the pattern

```
(*x *y *y *x)
```

For each length, if no sentence can match the pattern, explain why not.

16.6 Show sentences of length 2, 3, and 4 that match the pattern

```
(*x *y &y &x)
```

For each length, if no sentence can match the pattern, explain why not.

16.7 List *all* the sentences of length 6 or less, starting with `a b a`, that match the pattern

```
(*x *y *y *x)
```

Exercises about Implementation

16.8 Explain how `longest-match` handles an empty sentence.

16.9 Suppose the first `cond` clause in `match-using-known-values` were

```
((empty? pattern) known-values)
```

Give an example of a pattern and sentence for which the modified program would give a different result from the original.

16.10 What happens if the sentence argument—not the pattern—contains the word `*` somewhere?

16.11 For each of the following examples, how many `match-using-known-values` little people are required?

```
(match '(from me to you) '(from me to you))
(match '(*x *y *x) '(a b c a b))
(match '(*x *y *z) '(a b c a b))
(match '(*x hey *y bulldog *z) '(a hey b bulldog c))
(match '(*x a b c d e f) '(a b c d e f))
(match '(a b c d e f *x) '(a b c d e f))
```

In general, what can you say about the characteristics that make a pattern easy or hard to match?

16.12 Show a pattern with the following two properties: (1) It has at least two placeholders. (2) When you match it against any sentence, every invocation of `lookup` returns `no-value`.

16.13 Show a pattern and a sentence that can be used as arguments to `match` so that `lookup` returns `(the beatles)` at some point during the match.

16.14 Our program can still match patterns with unnamed placeholders. How would it affect the operation of the program if these unnamed placeholders were added to the database? What part of the program keeps them from being added?

16.15 Why don't `get-value` and `skip-value` check for an empty argument as the base case?

16.16 Why didn't we write the first `cond` clause in `length-ok?` as the following?

```
((and (empty? value) (member? howmany '(? *))) #t)
```

16.17 Where in the program is the initial empty database of known values established?

16.18 For the case of matching a placeholder name that's already been matched in this pattern, we said on page 268 that three conditions must be checked. For each of the three, give a pattern and sentence that the program would incorrectly match if the condition were not checked.

16.19 What will the following example do?

```
(match '(?x is *y !x) '(! is an exclamation point !))
```

Can you suggest a way to fix this problem?

16.20 Modify the pattern matcher so that a placeholder of the form `*15x` is like `*x` except that it can be matched only by exactly 15 words.

```
> (match '(*3front *back) '(your mother should know))
(FRONT YOUR MOTHER SHOULD ! BACK KNOW !)
```

16.21 Modify the pattern matcher so that a `+` placeholder (with or without a name attached) matches only a number:

```
> (match '(*front +middle *back) '(four score and 7 years ago))
(FRONT FOUR SCORE AND ! MIDDLE 7 ! BACK YEARS AGO !)
```

The `+` placeholder is otherwise like `!`—it must match exactly one word.

16.22 Does your favorite text editor or word processor have a search command that allows you to search for patterns rather than only specific strings of characters? Look into this and compare your editor's capabilities with that of our pattern matcher.