7 Pattern Matcher

Program file for this chapter: `match`

In a *conversational* program, one that carries on a conversation with the user, you may often have occasion to compare what the user types with some expected response. For example, a quiz program will compare the user's response with the correct answer; if you've just asked "how are you," you might look for words like "fine" or "lousy" in the reply. The main tools that Logo provides for such comparisons are `equalp`, which compares two values for exact equality, and `memberp`, which compares one datum with a list of alternatives. This project provides a more advanced comparison tool.

Most of the projects in this book are fairly complicated in their inner workings, but relatively simple in the external appearance of what they do. This project is the reverse; the actual program is not so complex, but it does quite a lot, and it will take a while to explain all of it. Pattern matching is a powerful programming tool, and I hope you won't be put off by the effort required to learn how to use it.

A *pattern* is a list in which some members are not made explicit. This definition is best understood by considering an example. Consider the pattern

```
[Every # is a #]
```

The words `every`, `is`, and `a` represent themselves explicitly. The two number signs, however, are symbols representing "zero or more arbitrary data." Here are some lists that would match the pattern:

```
[Every man is a mortal]
[Every computer programmer is a genius]
[Every is a word]
[Every datum is a word or a list]
```

Here are some lists that would *not* match the pattern:

```
[Socrates is a man]
[Every man is an animal]
[Everyone I know is a friend]
[I think every list is a match]
```

The first of these examples doesn't match the pattern because the word `every` is missing. The second has `an` instead of `a`, while the third has `everyone` instead of `every`. The fourth has the extra words `I think` before the word `every`. This last example *would* match the pattern

```
[# every # is a #]
```

because this new pattern allows for extra words at the beginning.

`Match` is a predicate that takes two inputs. The first input is a pattern and the second input is a sentence. The output is `true` if the sentence matches the pattern, otherwise `false`.

```
? print match [Every # is a #] [Every book is a joy to read]
true
? print match [Every # is a #] [Every adolescent is obnoxious]
false
```

Patterns can be more complicated than the ones I've shown so far. In the following paragraphs I'll introduce the many ways that you can modify patterns to control which sentences they'll match. As you read, you should make up sample patterns of your own and try them out with `match`.

Often, in a conversational program, it's not good enough just to know whether or not a sentence matches a pattern. You also want to know the pieces of the sentence that match the variable parts of the pattern. `Match` meets this requirement by allowing you to tell it the names of variables that you want to receive the matching words from the sentence. Here is an example:

```
? print match [#food is for #animal] [Hay is for horses]
true
? show :food
[Hay]
? show :animal
[horses]
```

```
? print match [#food is for #animal] [C++ is for the birds]
true
? show :food
[C++]
? show :animal
[the birds]
```

Here is a short conversational program using the parts of the pattern matcher we've discussed so far.

```
to converse
local [response name like]
print [Hi, my name is Toby and I like ice cream]
print [Tell me about yourself]
make "response readlist
if match [# my name is #name] :response [do.name strip.and :name]
if match [# i like #like] :response [do.like strip.and :like]
print [Nice meeting you!]
end

to do.name :name
print sentence "Hello, :name
end

to do.like :like
print sentence [I'm glad you like] :like
end

to strip.and :text
local "short
if match [#short and #] :text [output :short]
output :text
end

? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian and I like Chinese food
Hello, Brian
I'm glad you like Chinese food
Nice meeting you!
```

```
? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
I like spaghetti and meat balls
I'm glad you like spaghetti
Nice meeting you!
```

If `match` outputs `false`, there is no guarantee of what values will end up in the variables mentioned in the pattern. `Converse` uses the result of the match only if `match` outputs `true`.

`Converse` looks for each part of the sentence (the name and the thing the person likes) in two steps: first it finds the keywords `my name is` or `I like` and extracts everything following those phrases, then it looks within what it extracted for the word `and` and removes anything following it. For example, when I typed

```
My name is Brian and I like Chinese food
```

the result of matching the name pattern was to give the variable `name` the value

```
Brian and I like Chinese food
```

Then `strip.and` used a second pattern to eliminate everything after the `and`. You might be tempted to extract the name in one step by using a pattern like

```
[# my name is #name and #]
```

but I wanted to avoid that pattern because it won't match a sentence that only contains

```
my name is Mary
```

without expressing any likes or dislikes. The program as I've written it does accept these shorter sentences also. Later we'll see a more complicated pattern that accepts sentences with or without `and` using a single pattern.

The special symbol `#` in a pattern represents zero or more words. `Match` recognizes other symbols with different meanings:

`#`	zero or more
`&`	one or more
`?`	zero or one
`!`	exactly one

For example, if you'd like the `converse` program to recognize only the first name of the person using it, you could change the relevant pattern to

```
[# my name is !name #]
```

Then a conversation with the program might look like this:

```
? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian Harvey
Hello, Brian
Nice meeting you!
?
```

The word `!name` in the pattern matched just the single word `Brian`, not the multiple words `Brian Harvey` that the original pattern would have selected. (If you modify `converse` in this way, it should be possible to remove the invocation of `strip.and` in computing the input to `do.name`. The single word stored in the variable `name` won't contain any other clauses.)

So far, the patterns we've seen allow two extremes: the pattern can include a single word that must be matched exactly, or it can allow *any* word at all to be matched. It is also possible to write a pattern that calls for words in some specified category—that is, words that satisfy some predicate. Here is an example:

```
to ask.age
local "age
print [How old are you?]
if match [# !age:numberp #] readlist ~
   [print (sentence [You are] :age [years old.]]
end

? ask.age
How old are you?
I will be 36 next month
You are 36 years old.
```

This is a slightly silly example, but it does illustrate the use of a predicate to restrict which words can match a variable part of a pattern. The pattern used in `ask.age` looks for a single word for which `numberp` is `true`, that is, for a number. Any number of words surrounding the number are allowed.

Of course, a predicate used in a pattern need not be a primitive one like `numberp`. You may find it useful to write your own predicates that select categories of words. Such a predicate might have a list built in:

```
to colorp :word
output memberp :word [red orange yellow blue green violet white black]
end
```

Or you could check some inherent property of a word:

```
to ends.y :word
output equalp last :word "y
end
```

In either case, what is essential is that your predicate must take a word as its single input, and must output `true` if you want `match` to accept the word to fill a slot in the pattern.

It is most common to want a predicate like `colorp` above—one that tests its input word for membership in a certain list. A special notation makes it possible to include such a list in the pattern itself, instead of writing a predicate procedure. For example, suppose you are writing a quiz program, and you want to ask the question, "What is the quickest route from Boston to Framingham?" You'd like to accept answers like these:

```
Mass Pike
the Massachusetts Turnpike
the Pike
```

but not `the Ohio Turnpike`! Here is a pattern you could use.

```
[?:in [the] ?:in [Mass Massachusetts] !:in [Pike Turnpike]]
```

The special predicate `in` is a version of `memberp` that knows to look in the pattern, right after the element that invokes `in`, for the list of acceptable words. This pattern accepts zero or one `the`, zero or one of `Mass` or `Massachusetts`, and one of `Pike` or `Turnpike`. That is, the first two words are optional and the third is required.

Earlier I rejected the use of a pattern

```
[# my name is #name and #]
```

because I wanted also to be able to accept sentences without `and` following the name. I promised to exhibit a pattern that would accept both sentence forms. Here it is:

```
[# my name is #name:notand #]
```

This pattern uses a predicate `notand` that allows any word except `and`. It's easy to write this predicate:

```
to notand :word
output not equalp :word "and
end
```

(By the way, the symbols indicating the number of words to match are meant to be mnemonic. The question mark indicates that it's questionable whether or not the word will be in the sentence. The exclamation point looks a little like a digit 1, and also shouts emphatically that the word is present. The number sign means that any number of words (including zero) is okay, and the ampersand indicates that more words are required, namely at least one instead of at least zero.)

We've seen various combinations of quantifiers (that's what I'll call the characters like # that control how many words are matched), variable names, and predicates:

`#`	no variable, no predicate (accept any word)
`#name`	set variable, no predicate
`?:in`	no variable, test predicate
`!age:numberp`	set variable, test predicate

We are now about to discuss some of the more esoteric features of the `match` program. So far, we have always compared a pattern against a *sentence,* a list of words. It is also possible to match a pattern against a structured list, with smaller lists among its members. `Match` treats a sublist just like a word, if you don't want to examine the inner structure of the sublist. Here are some examples.

```
? print match [hello #middle goodbye] [hello is [very much] like goodbye]
true
? show :middle
[is [very much] like]
? print match [hi #middle:wordp bye] [hi and then bye]
true
? show :middle
[and then]
? print match [hi #middle:wordp bye] [hi and [then] bye]
false
```

```
? print match [hi #mid:wordp #dle:listp bye] [hi and [then] bye]
true
? show :mid show :dle
[and]
[[then]]
```

A more interesting possibility is to ask `match` to apply a sub-pattern to a sublist. This is done by using the pattern (that is, a list) in place of the name of a predicate. Here is an example:

```
? print match [a #:[x # y] b] [a [x 111 y] [x 222 y] b]
true
? print match [a #:[x # y] b] [a [x 333 zzz] b]
false
```

It is possible to include variable names in the subpattern, but this makes sense only if the quantifier outside the pattern is `!` or `?` because otherwise you may be trying to assign more than one value to the same variable. Here's what I mean:

```
? print match [a #all:[x #some y] b] [a [x 111 y] [x 222 y] b]
true
? show :all show :some
[[x 111 y] [x 222 y]]
[222]
```

The variable `all` is properly set to contain both of the lists that matched the subpattern, but the variable `some` only contains the result of the second match.

If a list appears in a pattern without a quantifier before it, `match` treats it as if it were preceded by "`!:`"; in other words, it tries to match the subpattern exactly once.

A pattern element like `#:predicate` can match several members of the target sentence; the predicate is applied to each candidate member separately. For example:

```
? print match [#nums:numberp #rest] [3 2 1 blastoff!]
true
? show :nums show :rest
[3 2 1]
[blastoff!]
```

Sometimes you may want to match several members of a sentence, but apply the predicate to all of the candidates *together* in one list. To do this, use the quantifier `@`:

```
to threep :list
output equalp count :list 3
end

? print match [@begin:threep #rest] [a b c d e]
true
? show :begin show :rest
[a b c]
[d e]
```

In this example, I haven't used the predicate to examine the *nature* of the matching words, but rather to control the *number* of words that are matched. Here is another example that looks "inside" the matching words.

```
to headtailp :list
if (count :list) < 2 [output "false]
output equalp first :list last :list
end

? print match [#front @good:headtailp #back] [a b c x d e f g x h i]
true
? show :front show :good show :back
[a b c]
[x d e f g x]
[h i]
```

Think about all the different tests that `match` has to make to find this match! Also, do you see why the first instruction of `headtailp` is needed?

Some patterns are *ambiguous;* that is, there might be more than one way to associate words from the matched sentence with quantifiers in the pattern. For example, what should the following do?

```
match [#front xx #back] [a b c d xx e f g xx h i]
```

The word `xx` appears twice in the matched sentence. The program could choose to use everything up to the first `xx` as `front`, leaving six words for `back`, or it could choose to use everything up to the second `xx` as `front`, leaving only two words for `back`. In fact, each quantifier, starting from the left, matches as many words as it can:

```
? print match [#front xx #back] [a b c d xx e f g xx h i]
true
? show :front show :back
[a b c d xx e f g]
[h i]
```

If that's not what you want, the quantifier ^ behaves like # except that it matches as *few* words as possible.

```
? print match [^front xx #back] [a b c d xx e f g xx h i]
true
? show :front show :back
[a b c d]
[e f g xx h i]
```

We can use the ^ quantifier to fix a bug in the `converse` program on page 111:

```
? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian and I like bacon and eggs
Hello, Brian and I like bacon
I'm glad you like bacon
Nice meeting you!
```

The problem here is that the pattern used by `strip.and` divided the sentence at the second `and`, just as the earlier example chose the second `xx` when I used # as the quantifier. We can fix it this way:

```
to strip.and :text
local "short
if match [^short and #] :text [output :short]
output :text
end

? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian and I like bacon and eggs
Hello, Brian
I'm glad you like bacon
Nice meeting you!
```

There is just one more special feature of `match` left to describe. It is another special predicate, like `in`, but this one is called `anyof`. When you use `anyof`, the next member of the pattern should be a *list of patterns* to test. `Match` tries each pattern in turn, applied to list members as determined by the quantifier used. In practice, though, `anyof` only makes sense when applied to several members as a group, so the quantifier `@` should always be used. An example may make this clear. I'm going to rewrite the `converse` program to check for names and likes all at once.

```
to converse
local [response name like rest]
print [Hi, my name is Toby and I like ice cream]
print [Tell me about yourself]
make "response readlist
while match [@:anyof [[My name is #name:notand]
                      [I like #like:notand]
                      [&:notand]]
                     ?:in [and] #rest] ~
            :line ~
      [make "response :rest]
if not emptyp :name [print sentence "Hello, :name]
if not emptyp :like [print sentence [I'm glad you like] :like]
print [Nice meeting you!]
end
```

This program uses the `notand` predicate I wrote earlier. It checks for clauses separated by the word `and`. Each clause can match any of three patterns, one for the name, one for the liking, and a general pattern that matches any other clause. The clauses can appear in any order.

```
? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian and I hate cheese
Hello, Brian
Nice meeting you!

? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
I like wings and my name is Jonathan
Hello, Jonathan
I'm glad you like wings
Nice meeting you!
```

Reinventing `Equalp` for Lists

`Match` is a kind of fancy `equalp` with a complicated understanding of what equality means. One way to approach an understanding of `match` is to begin with this question: Suppose Logo's primitive `equalp` only worked for comparing two *words* for equality. (For the remainder of this section, I won't use the word `equalp` at all; I'll call this imaginary primitive `wordequalp` instead.) How would you write a `listequalp` to compare two lists? This is basically a `butfirst`-style recursive operation, but you have to be a little careful about the fact that either input might be smaller than the other.

```
to listequalp :a :b
if emptyp :a [output emptyp :b]
if emptyp :b [output "false]
if wordequalp first :a first :b ~
   [output listequalp butfirst :a butfirst :b]
output "false
end
```

(This procedure contains the instruction `output "false` twice, but it never says `output "true`. How can it ever say that two lists are equal?)

There is one deficiency in the procedure as I've defined it. The problem is that it only works for *sentences*—lists whose members are words. If either list contains a sublist, `listequalp` will try to apply `wordequalp` to that sublist. If you enjoy the exercise of reinventing Logo primitives, you may want to fix that. But for my purposes, the version here is good enough as a basis for further development of the pattern matcher.

A Simple Pattern Matcher

We can extend the idea of `listequalp` slightly to make a pattern matcher that only recognizes the special word `#` to mean "match zero or more words." We won't do any of the fancy things like storing the matching words in a variable.

```
to match :pat :sen
if emptyp :pat [output emptyp :sen]
if emptyp :sen [if equalp first :pat "#
                   [output match butfirst :pat :sen]
                   [output "false]]
if equalp first :pat "# [output or match butfirst :pat :sen
                                   match :pat butfirst :sen]
if equalp first :pat first :sen ~
   [output match butfirst :pat butfirst :sen]
output "false
end
```

The end test is more complicated in this program than in `listequalp` because the combination of an empty sentence and a nonempty pattern can still be a match, if the pattern is something like `[#]` that matches zero or more words.

The really interesting part of this procedure is what happens if a `#` is found in the pattern. The match succeeds (outputs `true`) if one of two smaller matches succeeds. The two smaller matches correspond to two possible conditions: the `#` can match zero words, or more than zero. The first case is detected by the expression

```
match butfirst :pat :sen
```

For example, suppose you want to evaluate

```
match [# cream] [cream]
```

This expression should yield the value `true`, with the `#` matching no words in the sentence. In this example the expression

```
match butfirst :pat :sen
```

is equivalent to

```
match [cream] [cream]
```

which straightforwardly outputs `true`.

On the other hand, the expression

```
match :pat butfirst :sen
```

comes into play when the `#` has to match at least one word. For example, consider the expression

```
match [# cream] [ice cream]
```

Here the `#` should match the word `ice`. The expression

```
match :pat butfirst :sen
```

is here equivalent to

```
match [# cream] [cream]
```

But this is the example that was `true` just above.

If the `#` has to match more than one word, several recursive invocations of `match` are required, each one taking the `butfirst` of the sentence once. For example, suppose we start with

```
match [# cream] [vanilla ice cream]
```

Here is the sequence of recursive invocations leading to a `true` match:

```
match :pat butfirst :sen      match [# cream] [ice cream]
  match :pat butfirst :sen      match [# cream] [cream]
    match butfirst :pat :sen      match [cream] [cream]
```

I have been talking as if Logo only evaluated whichever of the two expressions

```
match butfirst :pat :sen
```

and

```
match :pat butfirst :sen
```

is appropriate for the particular inputs used. Actually, *both* expressions are evaluated each time, so there are many recursive invocations of `match` that come out `false`. However, the purpose of the primitive operation `or` is to output `true` if *either* of its inputs is `true`. To understand fully how `match` works, you'll almost certainly have to trace a few examples carefully by hand.

Efficiency and Elegance

Pattern matching is a complicated task, and even the best-written programs are not blindingly fast. But what is the "best-written" program? In the simple pattern matcher of the last section, the instruction

```
if equalp first :pat "# [output or match butfirst :pat :sen
                                   match :pat butfirst :sen]
```

is extremely compact and elegant. It packs a lot of power into a single instruction, by combining the results of two recursive invocations with `or`. The similarity of the inputs to the two invocations is also appealing.

The trouble with this instruction is that it is much slower than necessary, because it always tries both recursive invocations even if the first one succeeds. A more efficient way to program the same general idea would be this:

```
if equalp first :pat "# ~
   [if match butfirst :pat :sen
       [output "true]
       [output match :pat butfirst :sen]]
```

This new version is much less pleasing to the eye, but it's much faster. The reason is that if the expression

```
match butfirst :pat :sen
```

outputs `true`, then the other recursive invocation is avoided.

It's a mistake to make efficiency your only criterion for program style. Sometimes it's worth a small slowdown of your program to achieve a large gain in clarity. But this is a case in which the saving is quite substantial. Here is a partial trace of the evaluation of

```
match [cat # bat] [cat rat bat]
```

using the original version of the procedure:

```
match [cat # bat] [cat rat bat]              [cat # bat]   [cat rat bat]
  match butfirst :pat butfirst :sen          [# bat]       [rat bat]
    match butfirst :pat :sen                 [bat]         [rat bat]
    match :pat butfirst :sen                 [# bat]       [bat]
      match butfirst :pat :sen               [bat]         [bat]
        match butfirst :pat butfirst :sen    []            []
*     match :pat butfirst :sen               [# bat]       []
*       match butfirst :pat :sen             [bat]         []
```

The two invocations marked with asterisks are avoided by using the revised version. These represent 25% of the invocations of `match`, a significant saving. (Don't think that the program necessarily runs 25% faster. Not all invocations take the same amount of time. This is just a rough measure.) If there were more words after the # in the pattern, the saving would be even greater.

In this situation we achieve a large saving of time by reorganizing the flow of control in the program. This is quite different from a more common sort of concern for efficiency, the kind that leads people to use shorter variable names so that the program

will be a little smaller, or to worry about whether to use `fput` or `sentence` in a case where either would do. These small "bumming" kinds of optimization are rarely worth the trouble they cause. Figuring out how many times `match` is invoked using each version is a simple example of the branch of computer science called *analysis of algorithms;* a more profound analysis might use mathematical techniques to compare the two versions in general, rather than for a single example.

In the full version of the pattern matcher, listed at the end of this project description, I've taken some care to avoid unnecessary matching. On the other hand, the full version has less flexibility than the simple version because of its ability to assign matching words to variables. Consider a case like

```
match [# #] [any old list of words]
```

Which `#` matches how many words? It doesn't matter if you don't store the result of the match in variables. But if the pattern is `[#a #b]` instead, there has to be a uniform rule about which part of the pattern matches what. (In my pattern matcher, all of the words would be assigned to `a`, and `:b` would be empty. In general, pattern elements toward the left match as many words as possible when there is any ambiguity.) The simple pattern matcher doesn't have this problem, and can be written to match the ambiguous pattern whichever way gives a `true` result most quickly.

By the way, what if the two expressions that invoke `match` recursively were reversed in the revised instruction? That is, what if the instruction were changed again, to read

```
if equalp first :pat "# ~
   [if match :pat butfirst :sen
       [output "true]
       [output match butfirst :pat :sen]]
```

Would this be more or less efficient than the previous version?

Logo's Evaluation of Inputs

The discussion about efficiency started because Logo *evaluates* the inputs to the primitive operation `or` before invoking the procedure. That is, in the example in question, Logo invokes `match` twice before using `or` to check whether either invocation output `true`. This is consistent with the way Logo does things in general: To evaluate an expression that uses some procedure, Logo first evaluates all the inputs for that procedure, and then invokes the procedure with the evaluated inputs. Logo's rule is extremely consistent

(except for the `to` command), but it isn't the only possible way. In Lisp, a language that's like Logo in many ways, each procedure can choose whether or not its inputs should be evaluated in advance.

An example may make it clearer what I mean by this. Lisp has a procedure called `set` that's equivalent to the Logo `make`. You say

```
(set 'var 27)
```

as the equivalent of

```
make "var 27
```

But Lisp also has a version called `setq` whose first input is *not* evaluated before `setq` is invoked. It's as if there were an automatic quote mark before the first input, so you just say

```
(setq var 27)
```

with the same effect as the other examples.

Except for the special format of the `to` command that forms the title line of a procedure, Berkeley Logo and many other Logo dialects do not have any form of automatically-quoted inputs. The design principle was that consistency of evaluation would make the rules easier to understand. Some other versions of Logo do use auto-quoting for certain procedures. For example, in Berkeley Logo, to edit the definition of a procedure named `doit` you type the instruction

```
edit "doit
```

But in some other versions of Logo you instead say

```
edit doit
```

because in those versions, the `edit` command auto-quotes its input. One possible reason for this design decision is that teachers of young children like to present Logo without an explicit discussion of the evaluation rules. They teach the `edit` command as a special case, rather than as just the invocation of a procedure like everything else. Using this approach, auto-quoting the input avoids having to explain what that quotation mark means.

The advantage of the non-auto-quoting version of `edit` isn't just in some abstract idea of consistency. It allows us to take advantage of composition of functions. Suppose you are working on a very large project, a video game, with hundreds of procedures. You

want to edit all the procedures having to do with the speed of the spaceships, or whatever moves around the screen in this game. Luckily, all the procedures you want have the word `speed` as part of their names; they are called `shipspeed` or `asteroidspeed` or `speedcontrol`. You can say

```
edit filter [substringp "speed ?] procedures
```

(`Procedures` is a Berkeley Logo primitive operation that outputs a list of all procedures defined in the workspace; `substringp` is a predicate that checks whether one word appears as part of a longer word.) An auto-quoting `edit` command wouldn't have this flexibility.

The reason all this discussion is relevant to the pattern matcher is that the Lisp versions of `or` and `and` have auto-quoted inputs, which get evaluated one by one. As soon as one of the inputs to `or` turns out to be `true` (or one of the inputs to `and` is `false`), the evaluation stops. This is very useful not only for efficiency reasons, as in the discussion earlier, but to prevent certain kinds of errors. For example, consider this Logo instruction:

```
if not emptyp :list [if equalp first :list 1 [print "one]]
```

It would be pleasant to be able to rewrite that instruction this way:

```
if and (not emptyp :list) (equalp first :list 1) [print "one]
```

The use of `and`, I think, makes the program structure clearer than the nested `if`s. That is, it's apparent in the second version that something (the `print`) is to be done if two conditions are met, and that that's all that happens in the instruction. In the first version, there might have been another instruction inside the range of the first (outer) `if`; you have to read carefully to see that that isn't so.

Unfortunately, the second version won't work in Logo. If `:list` is in fact empty, the expression

```
(equalp first :list 1)
```

is evaluated before `and` is invoked; this expression causes an error message because `first` doesn't accept an empty input. In Lisp, the corresponding instruction *would* work, because the two predicate expressions would be evaluated serially and the second wouldn't be evaluated if the first turned out to be false.

The serial evaluation of inputs to `and` and `or` is so often useful that some people have proposed it for Logo, even at the cost of destroying the uniform evaluate-first rule.

But if you want a serial `and` or `or`, it's easy enough to write them, if you explicitly quote the predicate expressions that are its inputs:

```
to serial.and :pred1 :pred2
if not run :pred1 [output "false]
output run :pred2
end

to serial.or :pred1 :pred2
if run :pred1 [output "true]
output run :pred2
end
```

Here's how you would use `serial.and` to solve the problem with the nested `if`s:

```
if (serial.and [not emptyp :list] [equalp first :list 1]) [print "one]
```

Similarly, you could use `serial.or` instead of `or` to solve the efficiency problem in the first version of the pattern matcher:

```
output serial.or [match butfirst :pat :sen] [match :pat butfirst :sen]
```

These procedures depend on the fact that the predicate expressions that are used as their inputs are presented inside square brackets; that's why they are not evaluated before `serial.and` or `serial.or` is invoked.

Indirect Assignment

From now on, I'll be talking about the big pattern matcher, not the simple one I introduced to illustrate the structure of the problem. Here is the top-level procedure `match`:

```
to match :pat :sen
local [special.var special.pred special.buffer in.list]
if or wordp :pat wordp :sen [output "false]
if emptyp :pat [output emptyp :sen]
if listp first :pat [output special fput "!: :pat :sen]
if memberp first first :pat [? # ! & @ ^] [output special :pat :sen]
if emptyp :sen [output "false]
if equalp first :pat first :sen ~
   [output match butfirst :pat butfirst :sen]
output "false
end
```

As you'd expect, there are more cases to consider in this more featureful version, but the basic structure is similar to the simple matcher. The instructions starting `if emptyp`, `if memberp`, `if emptyp`, and `if equalp` play the same roles as similar instructions in the other version. (The `memberp` test replaces the comparison against the word `#` with a wider range of choices.)

The first `if` instruction tests for errors in the format of the pattern or the sentence to be matched, in which a word is found where a list was expected. It's not important if you use well-formed inputs to `match`. The `listp` test essentially converts a pattern like

```
[foo [some # pattern] baz]
```

to the equivalent form

```
[foo !:[some # pattern] baz]
```

The interesting new case comes when `match` sees a word in the pattern that starts with one of the six special quantifier characters. In this case, `match` invokes `special` to check for a match.

One of the interesting properties of `special` is that it has to be able to assign a value to a variable whose name is not built into the program, but instead is part of the *data* used as input to the program. That is, if the word

```
?howmany:numberp
```

appears in the pattern, `special` (or one of its subprocedures) must assign a value to the variable named `howmany`, but there is no instruction of the form

```
make "howmany ...
```

anywhere in the program. Instead, `match` has *another* variable, whose name is `special.var`, whose *value* is the *name* `howmany`. The assignment of the matching words to the pattern-specified variable is done with an instruction like

```
make :special.var ...
```

Here the first input to `make` is not a quoted word, as usual, but an expression that must be evaluated to figure out which variable to use.

`Special`, then, has two tasks. First it must divide a word like

```
?howmany:numberp
```

into its component parts; then it must carry out the matching tasks that are the *meaning* of those parts. These two tasks are like a smaller version of what a programming language interpreter like Logo does. Finding the meaningful parts of an instruction is called the *syntax* of a language, and understanding what the parts mean is called the *semantics* of the language. `Special` has two instructions, one for the syntax and one for the semantics:

```
to special :pat :sen
set.special parse.special butfirst first :pat "
output run word "match first first :pat
end
```

To *parse* something is to divide it into its pieces. `Parse.special` outputs a list of the form

```
[howmany numberp]
```

for the example we're considering. Then `set.special` assigns the two members of this list as the values of two variables. The variable named `special.var` is given the value `howmany`, and the variable named `special.pred` is given the value `numberp`. This preliminary work is what makes possible the indirect assignment described earlier.

Defaults

What happens if the pattern has a word like

```
?:numberp
```

without a variable name? What happens when the program tries to assign a value to the variable named in the pattern? `Set.special` contains the instruction

```
if emptyp :special.var [make "special.var "special.buffer]
```

The effect of this instruction is that if you do not mention a variable in the pattern, the variable named `special.buffer` will be used to hold the results of the match. This variable is the *default* variable, the one used if no other is specified.

It's important, by the way, that the variable `special.buffer` is declared to be local in procedure `match`. What makes it important is that `match` is recursive; if you use a pattern like

```
[a # b # c]
```

then the matching of the second # is a subproblem of the matching of the first one. Match invokes special, which invokes match#, which invokes #test, which invokes match on the butfirst of the pattern. That butfirst contains another #. Each of these uses the variable special.buffer to remember the words it is trying as a match; since the variable is declared local, the two don't get confused. (This means, by the way, that you can really confuse match by using the same variable name twice in a pattern. It requires a fairly complicated pattern to confuse match, but here is an example. The first result is correct, the second incorrect.

```
? print match [a #x b &y ! c] [a i b j c b k c]
true
? show :x show :y
[i]
[j c b]
? print match [a #x b &x ! c] [a i b j c b k c]
false
```

The only change is that the variable name x is used twice in the second pattern, and as a result, match doesn't find the correct match. You'll know that you really understand how match works if you can explain why it *won't* fail if the ! is removed from the pattern.)

When writing a tool to be used in other projects, especially if the tool will be used by other people, it's important to think about defaults. What should the program do if some piece of information is missing? If you don't provide for a default explicitly, the most likely result is a Logo error message; your program will end up trying to take first of an empty list, or something like that.

Another default in the pattern matcher is for the predicate used to test matches. For example, what happens when the word

```
?howmany
```

appears in the pattern, without a predicate? This case is recognized by parse.special, in the instruction

```
if emptyp :word [output list :var "always]
```

The special predicate always is used if no other is given in the pattern. Always has a very simple definition:

```
to always :x
output "true
end
```

Program as Data

The instruction in `special` that carries out the semantics of a special pattern-matching instruction word is

```
output run word "match first first :pat
```

If the pattern contains the word

```
?howmany:numberp
```

then this instruction extracts the quantifier character `?` (the first character of the first word of the pattern) and makes from it a procedure name `match?`. That name is then `run` as a Logo expression; that is, `special` invokes a procedure whose name is `match?`.

Most programming languages do not allow the invocation of a procedure based on finding the name of the procedure in the program's data. Generally there is a very strict separation between program and data. Being able to manipulate data to create a Logo instruction, and then run it, is a very powerful part of Logo. It is also used to deal with the names of predicates included in the pattern; to see if a word in the sentence input to `match` is a match for a piece of the pattern, the predicate `try.pred` contains the instruction

```
output run list :special.pred quoted first :sen
```

This instruction generates a list whose first member is the name of the predicate found in the pattern and whose second and last member is a word from the sentence. Then this list is run as a Logo expression, which should yield either `true` or `false` as output, indicating whether or not the word is acceptable.

Parsing Rules

When you are reading the program, remember that the kind of pattern that I've written as

```
[begin !:[smaller # pattern] end]
```

is read by Logo as if I'd written

```
[begin !: [smaller # pattern] end]
```

That is to say, this pattern is a list of four members. I think of the middle two as a unit, representing a single thing to match. The sublist takes the place of a predicate name after the `!` quantifier. But for Logo, there is no predicate name in the word starting with the exclamation point; the pattern is a separate member of the large list. That's why `set.special` uses the expression

```
emptyp :special.pred
```

to test for this situation, rather than `listp`. After `parse.special` does its work, all it has found is a colon with nothing following it. `Set.special` has to look at the next member of the pattern list in order to find the subpattern.

Further Explorations

Chapter 9 is a large program that uses `match`. It may give you ideas for the ways in which this tool can be used in your own programs. Here, instead of talking about applications of `match`, I'll discuss some possible extensions or revisions of the pattern matcher itself.

There are many obvious small extensions. For example, to complement the special `in` primitive, you could write `notin`, which would accept all *but* the members of the following list. You could allow the use of a number as the predicate, meaning that exactly that many matching words are required. That is, in the example for which I invented the predicate `threep`, I would instead be able to use

```
[@begin:3 #rest]
```

as the pattern.

There is no convenient way to say in a pattern that some subpattern can be repeated several times, if the subpattern is more than a single word. That is, in the second version of `converse`, instead of having to use `while` to chop off pieces of the matched sentence into a variable `rest`, I'd like to be able to say in the pattern something like

```
[@@: [@:anyof [[my name is #name:notand]
               [i like #like:notand]
               [&:notand]]
      ?:in [and]]]
```

Here the doubled atsign (`@@`) means that the entire pattern that follows should be matched repeatedly instead of only once.

For other approaches to pattern matching, you might want to read about the programming languages Snobol and Icon, each of which includes pattern matching as one of its main features.

Program Listing

```
to match :pat :sen
local [special.var special.pred special.buffer in.list]
if or wordp :pat wordp :sen [output "false]
if emptyp :pat [output emptyp :sen]
if listp first :pat [output special fput "!: :pat :sen]
if memberp first first :pat [? # ! & @ ^] [output special :pat :sen]
if emptyp :sen [output "false]
if equalp first :pat first :sen
   [output match butfirst :pat butfirst :sen]
output "false
end

;; Parsing quantifiers

to special :pat :sen
set.special parse.special butfirst first :pat "
output run word "match first first :pat
end

to parse.special :word :var
if emptyp :word [output list :var "always]
if equalp first :word ": [output list :var butfirst :word]
output parse.special butfirst :word word :var first :word
end

to set.special :list
make "special.var first :list
make "special.pred last :list
if emptyp :special.var [make "special.var "special.buffer]
if memberp :special.pred [in anyof] [set.in]
if not emptyp :special.pred [stop]
make "special.pred first butfirst :pat
make "pat fput first :pat butfirst butfirst :pat
end
```

```
to set.in
make "in.list first butfirst :pat
make "pat fput first :pat butfirst butfirst :pat
end

;; Exactly one match

to match!
if emptyp :sen [output "false]
if not try.pred [output "false]
make :special.var first :sen
output match butfirst :pat butfirst :sen
end

;; Zero or one match

to match?
make :special.var []
if emptyp :sen [output match butfirst :pat :sen]
if not try.pred [output match butfirst :pat :sen]
make :special.var first :sen
if match butfirst :pat butfirst :sen [output "true]
make :special.var []
output match butfirst :pat :sen
end

;; Zero or more matches

to match#
make :special.var []
output #test #gather :sen
end

to #gather :sen
if emptyp :sen [output :sen]
if not try.pred [output :sen]
make :special.var lput first :sen thing :special.var
output #gather butfirst :sen
end

to #test :sen
if match butfirst :pat :sen [output "true]
if emptyp thing :special.var [output "false]
output #test2 fput last thing :special.var :sen
end
```

```
to #test2 :sen
make :special.var butlast thing :special.var
output #test :sen
end

;; One or more matches

to match&
output &test match#
end

to &test :tf
if emptyp thing :special.var [output "false]
output :tf
end

;; Zero or more matches (as few as possible)

to match^
make :special.var []
output ^test :sen
end

to ^test :sen
if match butfirst :pat :sen [output "true]
if emptyp :sen [output "false]
if not try.pred [output "false]
make :special.var lput first :sen thing :special.var
output ^test butfirst :sen
end

;; Match words in a group

to match@
make :special.var :sen
output @test []
end

to @test :sen
if @try.pred [if match butfirst :pat :sen [output "true]]
if emptyp thing :special.var [output "false]
output @test2 fput last thing :special.var :sen
end
```

```
to @test2 :sen
make :special.var butlast thing :special.var
output @test :sen
end

;; Applying the predicates

to try.pred
if listp :special.pred [output match :special.pred first :sen]
output run list :special.pred quoted first :sen
end

to quoted :thing
if listp :thing [output :thing]
output word "" :thing
end

to @try.pred
if listp :special.pred [output match :special.pred thing :special.var]
output run list :special.pred thing :special.var
end

;; Special predicates

to always :x
output "true
end

to in :word
output memberp :word :in.list
end

to anyof :sen
output anyof1 :sen :in.list
end

to anyof1 :sen :pats
if emptyp :pats [output "false]
if match first :pats :sen [output "true]
output anyof1 :sen butfirst :pats
end
```