Pattern Matching and Its Use
Project P1 of
Common Lisp: An Interactive Approach

Stuart C. Shapiro
Department of Computer Science
State University of New York at Buffalo

August 3, 1999

The goal of this project is to implement a pattern matcher and to use it in three simple ways:

1. a simple rule-based reasoning system;
2. a simple parser;
3. A simple version of Eliza.

1 Patterns

Consider the list

```
(A B C M A B C (A B C) M A B C)
```

We see that this has an interesting *pattern:* there are 3 symbols followed by a fourth symbol; then the 3 symbols are repeated; then they are repeated again inside a list; then the fourth symbol is repeated; then the 3 symbols are repeated again. Here is another list *matching* the same pattern:

```
(JOHN EATS CHOCOLATE THEN JOHN EATS CHOCOLATE
      (JOHN EATS CHOCOLATE) THEN JOHN EATS CHOCOLATE)
```

and here is one that doesn't

```
(JOHN EATS CHOCOLATE THEN JANE EATS CHOCOLATE
      (JOHN LOVES CHOCOLATE) BUT MARY DOESNT)
```

In Project P1, you will implement a way to specify such patterns symbolically, and decide which lists match the patterns and which don't.

The simplest notion of a list matching a pattern is when the pattern is exactly the same as the list. The list

```
(A B C M A B C (A B C) M A B C)
```

certainly matches the pattern

```
(A B C M A B C (A B C) M A B C)
```

but

```
(JOHN EATS CHOCOLATE THEN JOHN EATS CHOCOLATE
      (JOHN EATS CHOCOLATE) THEN JOHN EATS CHOCOLATE)
```

doesn't match this pattern, and neither does

```
(JOHN EATS CHOCOLATE THEN JANE EATS CHOCOLATE
      (JOHN LOVES CHOCOLATE) BUT MARY DOESNT)
```

To make a pattern more flexible, we can replace some of its members by *variables.* This notion of "variable" is not quite the same as the notion of a variable in programming languages, but similar in that a variable can have different values at different times. The simplest kind of a variable is one that just doesn't care what it is matched with. We will call this a *don't care variable,* and use the symbol `?` for it. The list

```
(A B A)
```

matches each of the following patterns:

```
(A B A)
(? B A)
(A ? A)
(A B ?)
(A ? ?)
(? B ?)
(? ? A)
(? ? ?)
```

and the pattern

```
(? B ?)
```

matches any list of three members, the second of which is `B`, including

```
(A B A)
```

and

```
(A B C)
```

The don't care variable is too undiscriminating to specify the complicated pattern we first mentioned above. The best we could do with it would be

```
(? ? ? ? ? ? ? (? ? ?) ? ? ? ?)
```

but this would match any 12 member list, the eighth of which is a 3 member list. We want the four symbols of that first list to be any symbols, but the *same* four symbols whenever they recur. For this we need another kind of variable, called simply a *variable*. We need to have multiple variables, because the four symbols needn't be the same, but we have to recognize when one recurs, so that in must match the same *constant* every time. We will consider any symbol that starts with the character "`?`" to be a variable. So, `?w`, `?x`, `?y`, and `?z`, are four different variables, and `?bill` is another.

Using these variables, we can specify the pattern we are interested in:

```
(?X ?Y ?Z ?W ?X ?Y ?Z (?X ?Y ?Z) ?W ?X ?Y ?Z)
```

Notice that this is a symbolic version of the pattern we first stated in words, and that it matches the first two lists, but not the third.

The pattern

```
(?X ?Y ?X)
```

matches any 3 element list whose first and third elements are the same, including

```
(A B A)
```

and

```
(M N M)
```

but not

```
(A B C)
```

and the pattern

```
(?X B ?X)
```

matches any 3 element list whose second element is a `B` and whose first and third elements are the same, including

```
(A B A)
```

and

```
(B B B)
```

but not

```
(M N M)
```

We can get even more flexible in our patterns by having *sequence variables.* A sequence variable can match any sequence of objects, including the empty sequence, but if it's repeated, it must match the same thing(s) each time, just like a normal variable. We will have our sequence variables look like our normal variables, but use "`$`" instead of "?". Using sequence variables, the first pattern above can be expressed as

```
($X ?Y $X ($X) ?Y $X)
```

This also matches

```
(A B C M A B C (A B C) M A B C)
```

and

```
(JOHN EATS CHOCOLATE THEN JOHN EATS CHOCOLATE
      (JOHN EATS CHOCOLATE) THEN JOHN EATS CHOCOLATE)
```

but not

```
(JOHN EATS CHOCOLATE THEN JANE EATS CHOCOLATE
      (JOHN LOVES CHOCOLATE) BUT MARY DOESNT)
```

It also matches

```
(A B A (A) B A)
```

and even

```
(NULL () NULL)
```

with `$X` matching the empty sequence and `?Y` matching the symbol `NULL`.

Notice that there are three ways for a list to fail to match a pattern:

1. Corresponding members of the list and the pattern might be different constants. For example, the list

   ```
   (A B C)
   ```

   doesn't match the pattern

   ```
   (?X B D)
   ```

2. A variable or sequence variable might recur in the pattern, but match up with different objects in the list. For example, the list

   ```
   (A B C)
   ```

   doesn't match the pattern

   ```
   (?X B ?X)
   ```

and the list

```
(A B C A D)
```

doesn't match the pattern

```
($X C $X)
```

3. The list might be too short or too long for the pattern. For example, the list

   ```
   (A B)
   ```

   doesn't match the pattern

   ```
   (?X B ?Y)
   ```

   and neither does the list

   ```
   (A B C D)
   ```

2 Substitutions

The pattern

```
(?X ?Y ?Z ?W ?X ?Y ?Z (?X ?Y ?Z) ?W ?X ?Y ?Z)
```

matches both the constant list

```
(A B C M A B C (A B C) M A B C)
```

and the constant list

```
(JOHN EATS CHOCOLATE THEN JOHN EATS CHOCOLATE
      (JOHN EATS CHOCOLATE) THEN JOHN EATS CHOCOLATE)
```

but it matches them in different ways. A *substitution* is a way for us to talk about how a pattern matches a constant.

A substitution is a list of pairs, where each pair is a list of two members, a variable and a constant. We can use substitutions to show how the variables in a pattern matched up with constants in the constant list that the pattern matched. For example, the substitution

```
((?X A) (?Y B) (?Z C) (?W M))
```

shows how the pattern

```
(?X ?Y ?Z ?W ?X ?Y ?Z (?X ?Y ?Z) ?W ?X ?Y ?Z)
```

matches the constant

```
(A B C M A B C (A B C) M A B C)
```

and the substitution

```
((?X JOHN) (?Y EATS) (?Z CHOCOLATE) (?W THEN))
```

shows how it matches

```
(JOHN EATS CHOCOLATE THEN JOHN EATS CHOCOLATE
      (JOHN EATS CHOCOLATE) THEN JOHN EATS CHOCOLATE)
```

If a variable appears in a substitution, we say that the variable is *bound to* the constant it is paired with. For example `?X` is bound to `A` in the first substitution above and to `JOHN` in the second.

Since the only important things about a substitution are what variables are in it and what constants they are bound to, the order of the pairs is not important. (A substitution is a *set* of pairs.) Thus, for example,

```
((?X A) (?Y B) (?Z C) (?W M))
```

and

```
((?Z C) (?W M) (?Y B) (?X A))
```

are the *same* substitution.

Sequence variables may also appear in substitutions, but they, of course, are always bound to lists. For example, the substitution

```
(($X (JOHN EATS CHOCOLATE)) (?Y THEN))
```

shows how the pattern

```
($X ?Y $X ($X) ?Y $X)
```

matches the constant

```
(JOHN EATS CHOCOLATE THEN JOHN EATS CHOCOLATE
      (JOHN EATS CHOCOLATE) THEN JOHN EATS CHOCOLATE)
```

Don't care variables, on the other hand, never appear in substitutions because we never care what they match, and, anyway, a don't care variable may match several different constants in the same list.

2.1 The `match` Function

Since we are seldom interested in whether a pattern matches a constant without also being interested in how it matches, the function `match`, which you will first write for Exercise 17.31, takes a pattern and a constant list as arguments and returns a substitution as value. For example,

```
> (match '(?x ?y ?z ?w ?x ?y ?z (?x ?y ?z) ?w ?x ?y ?z)
         '(a b c m a b c (a b c) m a b c))
((?W M) (?Z C) (?Y B) (?X A) (T T))

> (match '(?x ?y ?z ?w ?x ?y ?z (?x ?y ?z) ?w ?x ?y ?z)
         '(john eats chocolate then john eats chocolate
           (john eats chocolate) then john eats chocolate))
((?W THEN) (?Z CHOCOLATE) (?Y EATS) (?X JOHN) (T T))

> (match '($x ?y $x ($x) ?y $x)
         '(john eats chocolate then john eats chocolate
           (john eats chocolate) then john eats chocolate))
((?Y THEN) ($X (JOHN EATS CHOCOLATE)) (T T))
```

The pair `(T T)` appears in these substitutions for a specific technical reason. Consider the two problems

```
> (match '(a b) '(a b))
```

and

```
> (match '(a b) '(x y))
```

The first matches, but since there are no variables in the pattern, the substitution showing the match will be the empty set of pairs; the second doesn't match at all, so `match` should return False. Unfortunately, the empty set and False are normally represented by COMMON LISP by the same thing—`NIL`. In order to distinguish the "perfect" match from match failure, we must choose something other than `NIL` either to represent the empty substitution, or to represent match failure. I have chosen to use `((T T))` to represent the empty substitution and leave `NIL` representing failure because that has fewer implications for how we will write our functions than the alternative, though popular, use of `NIL` to represent the empty substitution and `FAIL` to represent match failure.

2.2 Applying Substitutions

Subtitutions may be used by *applying* them to patterns. To apply a substitution to a pattern is to replace each variable in the pattern with the constant it is bound to in the substitution. We will use the function

`(substitute` ***pattern substitution***`)`

which you will first write for Exercise 17.32, to represent this operation. For example,

```
> (substitute '($x ?y $x ($x) ?y $x)
              '((?y then) ($x (john eats chocolate)) (t t)))
(JOHN EATS CHOCOLATE THEN JOHN EATS CHOCOLATE
      (JOHN EATS CHOCOLATE) THEN JOHN EATS CHOCOLATE)
```

It should be clear that if ***pattern*** matches ***constant***, then the identity

`(substitute` ***pattern*** `(match` ***pattern constant***`))` = ***constant***

will always be satisfied.

However, a substitution could also be applied to a pattern other than the one originally used to create it. For example,

```
> (substitute '(so ?y $x)
              '((?y then) ($x (john eats chocolate)) (t t)))
(SO THEN JOHN EATS CHOCOLATE)
```

3 Rules

A *rule* is a list of two patterns. We will call the first pattern the *left-hand side* of the rule, and the second pattern the *right-hand side* of the rule. The function `(apply-rule constant rule)`, which you will write for Exercise 18.27, does the following:

```
if the left-hand side of rule matches constant,
   apply the substitution returned by that match
        to the right-hand side of the rule
   and return the result
else return the constant
```

For example,

```
> (apply-rule '(JOHN EATS CHOCOLATE THEN JOHN EATS CHOCOLATE
                (JOHN EATS CHOCOLATE) THEN JOHN EATS CHOCOLATE)
              '(($X ?Y $X ($X) ?Y $X) (So ?Y $X)))
(SO THEN JOHN EATS CHOCOLATE)
```

4 A Simple Rule-Based Reasoning System

In Exercise 18.27, you will test `apply-rule` on a tiny, one example, rule-based reasoning system. This example is based on the syllogism

$$\begin{array}{c}\text{Socrates is a man.}\\ \text{All men are mortal.}\\ \hline \text{Socrates is mortal.}\end{array}$$

In Predicate Logic, this syllogism is expressed

$$\begin{array}{c}Man(Socrates)\\ \forall x[Man(x) \Rightarrow Mortal(x)]\\ \hline Mortal(Socrates)\end{array}$$

Using `apply-rule`, we can do

```
> (apply-rule '(Man Socrates)
              '((Man ?x) (Mortal ?x)))
(MORTAL SOCRATES)
```

5 A Simple Parser

Parsing involves applying a set of *grammar rules* to a purported sentence to see if it obeys the rules. For example, consider the following grammar.

$$\begin{array}{lll} S & ::= & N\ VP \\ VP & ::= & V\ N \\ VP & ::= & V \\ N & ::= & \texttt{John} \\ N & ::= & \texttt{Mary} \\ V & ::= & \texttt{loves} \end{array}$$

This grammar could be used in the forward direction to generate a sentence as follows

$$\begin{array}{c} S \\ N\ VP \\ N\ V\ N \\ \texttt{John}\ V\ N \\ \texttt{John loves}\ N \\ \texttt{John loves Mary} \end{array}$$

or we could use the grammar in the backward direction to parse the sentence. If we constructed a Common Lisp version of the parse tree as we went, it would look like this:

```
              John loves Mary
            (N John) loves Mary
          (N John) (V loves) Mary
        (N John) (V loves) (N Mary)
      (N John) (VP (V loves) (N Mary))
    (S (N John) (VP (V loves) (N Mary)))
```

Our rule applier can do this if we repeatedly apply an appropriate rule to the results of a previous rule application. The function `apply-rules`, which you will first write for Exercise 24.9 does this repeated rule application. For Exercise 26.14, you will have `apply-rules` do the kind of parsing shown above using the rule set

```
(((($x John $y)       ($x (n John) $y))
 (($x loves $y)       ($x (v loves) $y))
 (($x Mary $y)        ($x (n Mary) $y))
 (($x (v $y) (n $z)) ($x (vp (v $y) (n $z))))
 (($x (v $y))         ($x (vp (v $y))))
 (((n $x) (vp $y))    (s (n $x) (vp $y))))
```

which is just a rewriting and reordering of the grammar rules shown above.

6 A Simple Version of Eliza

ELIZA is a program written in the early 1960s by Joseph Weizenbaum[1] to simulate a Rogerian psychotherapist by engaging its user, the simulated patient, in conversation. ELIZA works simply by matching the user's input sentences to patterns and responding with transformations of those sentences. We can do this with our rule applier. For Example, for Exercise 18.27, you will see `apply-rule` do this:

```
> (apply-rule '(I am depressed)
              '((I am ?x) (Why are you ?x ?)))
(WHY ARE YOU DEPRESSED ?)
```

For Exercise 29.32, you will write a more complete, but still simplified version of ELIZA.

7 Summary of P1 Exercises

Here is a list of all the p1 exercises, with some additional comments.

12.1–4 Do as specified on p. 81.

13.13–14 Do as specified on p. 87.

14.7–8 Do as specified on p. 92.

16.15 Do as specified on p. 108.

17.29–30 Do as specified on p. 120. Note that, since you are allowed to use `lisp:boundp` in your definition of `match::boundp`, both `match::boundp` and `match::bound-to` can have extremely short definitions.

17.31 Do as specified on pp. 120–121, but also note that if *pat* and *list* are of different lengths, `match` should return `NIL`. The third argument of `match1`, called `pairs` in the book, is to be a substitution.

17.32 Do as specified on p. 121.

18.25–27 Do as specified on pp. 138–139.

24.9–10 Do as specified on p. 180.

26.8–14 Do as specified on pp. 193–195.

29.32 Do as specified on pp. 227–228.

[1] J. Weizenbaum, "ELIZA—A computer program for the study of natural language communication between man and machine," *Communications of the ACM 9*, 1 (January 1966), 36–45.