23 A Java Representation for Predicate Calculus and Unification

Chapter Objectives

A brief introduction to the predicate calculus
- Predicates with:
  - Atoms
  - Variables
  - Functions
- A unification algorithm
  - A `car/crd` recursive tree walk
  - Ignoring occurs check

Techniques proposed for mapping predicate calculus to an object system
- A meta-linguistic interpreter
- Class hierarchy for `PCExpression`

Built `Unifiable` interface

Important discussion on design issues
- Building a representation for predicate calculus based reasoning
- Discussion in Section 23.3 and 23.5

First of three chapters covering predicate calculus
- Representation and unification (Ch 23)
- Reasoning with the predicate calculus (Ch 24)
- A rule-based expert system shell in Java (Ch 25)

Chapter Contents

23.1 Introduction to the Task

Although Java supports classes, inheritance, relations, and other structures used in knowledge representation, we should not think of it as a representation language in itself, but as a general purpose programming language. In AI applications, Java is more commonly used to implement interpreters for higher-level representations such as logic, frames, semantic networks, or state-space search. Generally speaking, representing the elements of a problem domain directly in Java classes and methods is only feasible for well-defined, relatively simple problems. The complex, ill-formed problems that artificial intelligence typically confronts require higher-level representation and inference languages for their solution.

The difference between AI representation languages and Java is a matter of semantics. As a general programming language, Java grounds object-oriented principles in the practical virtual machine architecture – the abstract architecture at the root of Java's platform independence – rather than in mathematical systems or knowledge representation theories. Although Java draws on ideas from knowledge representation, such as class inheritance, its underlying semantics is procedural, defining loops, branches, variable scoping, memory addresses, and other machine constructs. This contrasts with higher-level knowledge representation languages, which draw their semantics from mathematical (formal logic or the lambda calculus), psychological (frames, semantic networks), or neural (genetic and connectionist network) theories of symbols, reference, and reasoning. The power of higher-level representation languages is in addressing the specific problems of reasoning about complex domains. This also simplifies the verification and validation of code, since a theory-based implementation can support code integrity better than the combination of machine semantics and often ill-defined user requirements.

Meta-linguistic abstraction is the technique of using one language to implement an interpreter for another language whose structure better fits a given class of problems. The term, *meta-linguistic*, refers to the use of a language's constructs to represent the elements of the target language, rather than elements of the final problem domain, as seen in Figure 23.1. We can think of meta-linguistic abstraction as a series of mappings, the arrows in Figure 23.1. The elements of a representation language, in this case, predicate calculus, are mapped into Java classes, and the entities in our problem domain are mapped into the representation language. If done carefully, this simplifies both mappings and their implementation – indeed, one of the benefits of this approach is that the theoretical basis of the representation language serves as a well-defined, mathematically grounded basis for the implementation. In turn, this simplifies both the implementation of the representation language, and the development of problem solvers.

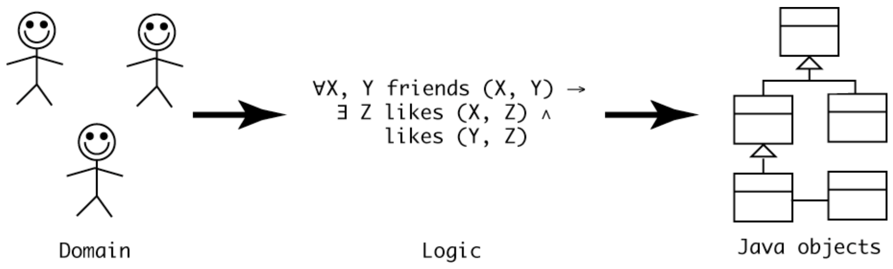

Figure 23.1. Creating an interpreter in Java to represent predicate calculus expressions, which, capture the semantics of a problem domain.

The search engines of Chapter 22 hinted at this technique through their use of solver and state classes to describe general search elements rather than defining classes for a particular problem, e.g, farmers, wolves, goats, and cabbages. Chapters 23, 24 and 25 provide a more sophisticated example of meta-linguistic abstraction, using Java to build an inference engine for first-order predicate calculus.

Implementing a logic-based reasoner in an object-oriented language like Java offers an interesting challenge, largely because the predicate calculus's "flat" declarative semantics is so different from that of Java. In the predicate calculus every predicate is a statement that is either true or false for the domain of discourse; there is no hierarchy within predicate relationships, nor is there inheritance of predicates, variables, or truth-values across predicate expressions. In addition, the scope of variables is limited to a single predicate. What predicate calculus gives us in turn is representational generality and theoretically supported algorithms for logical inference and variable binding through unification.

In building a predicate calculus problem solver, we begin with simple predicate calculus expressions, and then implement a unification algorithm that determines the variable substitutions that make two expressions equivalent (Luger 2009, Section 2.3.3).

Chapter 24 addresses the representation of more complex logical expressions (`and`, `or`, `not` and `implies`), and then uses the unification algorithm as the basis of a logic problem-solver that searches an and/or graph of logical inferences. This problem solver then implements a depth-first search with backtracking, and constructs a proof tree for each solution found. This can be seen as building a Prolog interpreter in Java.

23.2 A Review of the Predicate Calculus and Unification

The predicate calculus is, first of all, a formal language: it is made up of tokens and a grammar for creating predicate names, variables, and constants. Chapter 2 of Luger (2009) describes predicate calculus in detail, but we offer a brief summary in this section.

The *atomic* unit of meaning in the predicate calculus is the predicate *sentence* or *expression*. A simple predicate expression, or simple sentence, consists of a predicate name, such as likes or friends in the following examples, followed by zero or more arguments. The arguments of predicates can be atoms (represented, by convention, as symbols beginning with a lower case letter), variables (symbols beginning with an upper case letter), or functions (in the same syntactic form as predicates). A function may itself have zero or more arguments, expressions separated by commas and enclosed by parentheses. A function is interpreted in the traditional manner, that is, by replacing it and its arguments, which are taken from its domain of definition, by the unique constant that is the function's evaluation. For example, `father_of(david)` is evaluated to `george`, when `george` is the computed father of `david`. Although our interpreter allows functions in expressions and will match them as patterns, we do not support their interpretation. Examples of simple sentences include:

```
likes(george,  kate).
likes(kate, wine).
likes(david, kate).
likes(kate, father_of(david)).
```

Predicate calculus also allows the construction of complex sentences using the operators, ∧ (and), ∨ (or), ¬ (not), and ← (implies). For example,

```
friends(X, Y) ← likes(X, Z)  ∧ likes(Y, Z)
```

can be interpreted as stating that `X` and `Y` are `friends` if there is some individual `Z` such that `X` and `Y` both have the `likes` relationship with `Z`. When using variables in logical expressions, there are various possibilities for their interpretation. Predicate calculus uses variable quantification to specify the scope of meaning of the variables in a sentence. The above sentence is more properly written using the existential (∃) and universal (∀) quantifiers:

```
∀X, Y (friends(X, Y) ←
  ∃Z (likes(X, Z) ∧ likes(Y, Z)))
```

This can be read: "for all X and Y, X and Y are friends if there exists a Z such that X likes Z and Y likes Z."

Horn Clauses and Unification

In developing an inference engine for predicate calculus, we will follow Prolog conventions and restrict ourselves to a subset of logical expressions called *Horn Clause Logic*. Although their theoretical definition is more complex, for our purposes, we can think of a Horn clause as an implication, or rule, with only a single predicate on the left hand side of the implication, ←. The right hand side, or "body" of the clause can be empty, a simple predicate, or any syntactically well-formed expression made up of simple predicates. Horn clauses may consist of the body only; these are called goal clauses. Examples in propositional form include:

```
p ← q ∧ r ∧ s
p ←
q ∧ r ∧ s
```

Following Prolog conventions, we extend the definition to allow ∨ (or), and ¬ (not) in the body of the Horn clause. In order to accommodate Java syntax, we vary these syntactic conventions in ways that will be evident over the next few chapters.

The power of Horn clauses is in their simplification of logical reasoning. As we will see in the next chapter, restricting the head of implications to a single predicate simplifies the development of a backward chaining search engine. To answer the query of whether two people `X` and `Y` can be found who are `friends`, a search engine must determine if there are any variable substitutions for `X`, `Y`, and `Z` that satisfy the `likes(X, Z)` and `likes(Y, Z)` relationships. In this case, the substitutions `george/X`, `david/Y`, and `kate/Z` lead to the conclusion: `friends(george, david)`. *Unification* is the algorithm for matching predicate calculus expressions and managing the variable substitutions generally required for such matches. Unification, combined with the use of search to try all possible matches, form the heart of a logic problem solver.

Logic-based reasoning requires determining the equivalence of two expressions. For example, consider the reasoning schema, *modus ponens*:

```
Given: q(X) ← p(X) and p(george)
Infer: q(george)
```

We can read this as "if `p` of `X` implies `q` of `X`, and `p(george)` are both true, then we infer that `q(george)` is true as well." This inference is a result of the equivalence of the fact "`p(george)`" and the premise "`p(X)`" in the implication. What makes this difficult for the predicate calculus is the more complex structure of sentences, and, more importantly, the handling of variables. In the example above, the sentence

```
likes(X, Z)  ∧ likes(Y, Z)
```

matched the sentences `likes(george, kate)` and `likes(david, kate)` under the variable bindings `george/X`, `david/Y` and `kate/Z`. Note that, although the variable `Z` appears in two places in the sentence, it can only be *bound* once: in this case, to `kate`. In another example, the expression `likes(X, X)` could not match the sentence since `X` can only be bound to one constant. The algorithm for determining the bindings under which two predicate expressions match is called *unification*.

The unification algorithm determines a minimal set of variable bindings under which two sentences are equivalent. This minimal set of bindings is called the *unifier*. It maintains these bindings in a *substitution set*, a list of variables paired with the expressions (constants, functions or other variables) to which they are bound. It also insures consistency of these bindings throughout their scope. It is also important to recognize that in the above example, where `X` is bound to `george`, it is possible for the variable `X` to appear in different predicates with different scopes and bindings. Unification must manage these different contexts as well.

Luger (2009, Chapter 2) defines the unification algorithm as returning a substitution set if two expressions unify. The algorithm treats the expressions as lists of component expressions, a technique we will use in our own implementation:

```
function unify(E1, E2)
  begin
     case
        both E1 and E2 are constants or empty list:
           if E1 = E2 then
              return the empty substitution set
           else return FAIL;
        E1 is a variable:
           if E1 is bound then
              return unify(binding of E1, E2);
           if E1 occurs in E2 then return FAIL;
           return the substitution set {E1/E2};
        E2 is a variable:
           if E2 is bound then
              return unify(binding of E2, E1);
           if E2 occurs in E1 then return FAIL
           else return the substitution set {E2/E1};
        either E1 or E2 are different lengths:
           return FAIL;
        otherwise:
           HE1 = first element of E1;
           HE2 = first element of E2;
           S1 = unify(E1, E2);
           if S1 = FAIL then return Fail;
```

```
TE1 = apply S1 to the tail of E1;
TE2 = apply S1 to the tail of E2;
S2 = unify (TE1, TE2);
if S2 = FAIL then return FAIL
else return the composition of S1 and S2;
```

Although the algorithm is straightforward, a few aspects of it are worth noting. This is a recursive algorithm, and follows the pattern of head/tail recursion already discussed for Lisp and Prolog. We will retain a recursive approach in the Java implementation, although we will adapt it to an object-oriented idiom. Variables may bind to other variables; in this case, if either of the variables becomes bound to a constant or function expression, then both will share this binding. Finally, note that before binding a variable to an expression, the algorithm first checks if the variable is in the expression. This is called the *occurs check*, and it is necessary because, if a variable binds to an expression that contains it, replacing all occurrences of the variable with the expression will result in an infinite structure. We will omit the occurs check in our implementation, both for efficiency (as with Prolog) and to simplify the discussion. We do, however, leave its implementation as an exercise.

23.3 Building a Predicate Calculus Problem Solver in Java

Representing Basic Predicates

As with any object-oriented implementation, we began with representation, defining the elements of the predicate calculus as Java classes. In the present chapter, we define the classes `Constant`, `Variable`, and `SimpleSentence`. In Chapter 24 we define the classes `And`, and `Rule` (or `Implies)`. We will leave the definition of `Or` and `Not` as an exercise.

We organize `Constant`, `Variable`, and `SimpleSentence` into a hierarchy, with the interface `PCExpression` as its root. As we develop the algorithm, this interface will come to define signatures for methods shared across all predicate calculus expressions.

```
public interface PCExpression {}
```

We also create a child interface, called `Unifiable`, that defines the signature for the `unify(. . .)` method, which is the focus of this chapter. For now, we leave the arguments to `unify` unspecified.

```
public interface Unifiable extends PCExpression
   { public boolean unify(. . .); }
```

The reason for introducing the `Unifiable` interface will become evident as the discussion moves into Chapter 24.

Based on these interfaces, we define the classes shown in the hierarchy of Figure 23.2.

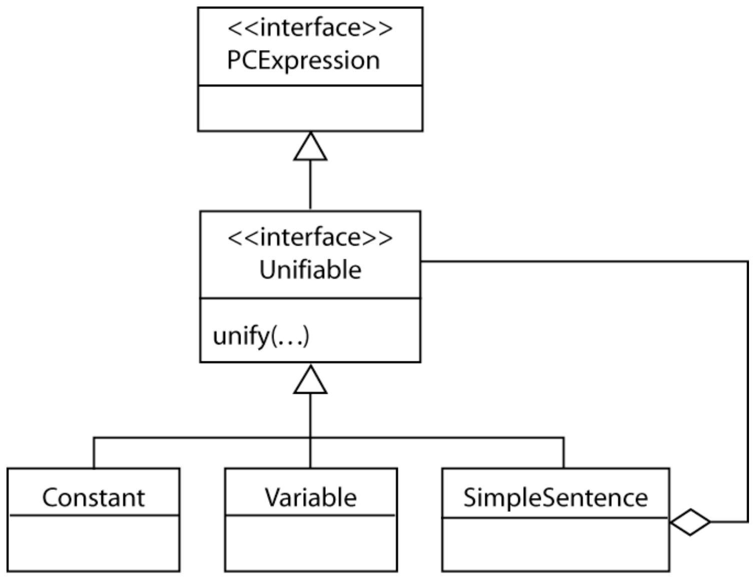

Figure 23.2. The class hierarchy for PCExpression.

`Constant`, `Variable` and `SimpleSentence` are all subclasses of `Unifiable`. Each instance of these classes will have a one-to-one relationship to its corresponding expression. This simplifies testing if constants or variables are the same: they are the same if and only if they are the same Java object. We can test this using the "`==`" operator. A `SimpleSentence` is an assembly of instances of `Unifiable`. This recursive structure allows sentences such as:

```
likes(kate, wine).
likes(david, X).
likes(kate, father_of(david))
```

Note that we do not distinguish between predicate expressions and functions in this implementation. This works for our implementation since we are not evaluating functions, and the syntax of predicate expressions and functions is the same. Introducing evaluable functions into this model is left as an exercise. Under this approach, an initial implementation of `Constant` is straightforward:

```
public class Constant implements Unifiable
{
  private String printName = null;
  private static int nextId = 1;
  private int id;
  public Constant()
  {
     this.id = nextId++;
  }
  public Constant(String printName)
  {
     this();
```

```
            this. printName= printName;
         }
         public String toString()
         {
            if (printName!= null)
               return printName;
            return "constant_" + id;
         }
                          //unify and other functions to be defined shortly.
      }
```

This definition includes a `printName` member variable, which we will use to display constants like `george` or `kate` in our earlier example. This practice of distinguishing the display name of an object from its internal representation is a common object-oriented practice that allows us to optimize internal representation while maintaining a familiar "face" for printing objects. `Variable` has a nearly identical definition:

```
      public class Variable implements Unifiable
      {
         private String printName = null;
         private static int nextId = 1;
         private int id;
         public Variable()
         {
            this.id = nextId++;
         }
         public Variable(String printName)
         {
            this();
            this.printName = printName;
         }
         public String toString()
         {
            if (printName != null)
               return printName + "_" + id;
            return "V" + id;
         }
                          //unify  and other functions to be defined shortly.
      }
```

The `id` member variable in these classes serves several purposes. It can serve as an identifier for unnamed constants or variables. Although, generally speaking, unnamed constants and variables can be confusing, we include them for completeness and to be consistent with similar features in Prolog. A more important use of the `id` variable is in the `Variable` class.

As mentioned earlier, the same variable name may be used in different sentences, where it is treated as different variables. By appending the `id` to the `printName` in the `toString` method, we enable the programmer to more easily distinguish variables in this case, simplifying tracing of program execution.

Also note the introduction of the constructor `Variable(Variable v)`. This pattern is called a *copy constructor*, and serves the same function as the `clone` method. We prefer this approach because the semantics of `clone` are problematic, with programmers frequently redefining it to reflect their own needs. Using a copy constructor emphasizes that the semantics of the copy are specific to the class. In the case of `Variable`, we define copying to use the same `printName` but a different `id`; this will be important for dealing with occurrences of the same variable in different contexts, as presented in Chapter 24.

As we mentioned, for the portion of the definition displayed above, `Constant` and `Variable` are essentially the same, and this part of their definition could be placed in a common parent class. We have chosen not to do so, feeling that the functionality is so simple that a common parent definition would buy us too little in the way of maintainability to justify the added complexity of doing so. However, like many design decisions, this is a matter of taste. We encourage the reader to explore the trade-offs of this decision on her own.

Finally, we define `SimpleSentence` as an array of type `Unifiable`:

```
public class SimpleSentence implements Unifiable
{
  private Unifiable[] terms;
  public SimpleSentence(Constant predicateName,
        Unifiable... args)
  {
     this.terms = new Unifiable[args.length + 1];
     terms[0] = predicateName;
     System.arraycopy(args, 0, terms, 1,
                args.length);
  }
  private SimpleSentence(Unifiable... args)
  {
     terms = args
  }
  public String toString()
  {
     String s = null;
     for (Unifiable p : terms)
        if (s == null)
           s = p.toString();
```

```
            else
               s += " " + p;
            if (s == null)
            return "null";
         return "(" + s + ")";
      }
      public int length()
      {
         return terms.length;
      }
      public Unifiable getTerm(int index)
      {
         return terms[index];
      }
                        //unify and other functions to be defined shortly.
   }
```

Representing a `simpleSentence` as an array of `Unifiable` simplifies access to its elements using the `length` and `getTerm` methods. These will be important to our implementation of the `unify` method. Although first-order predicate calculus requires that the first term of a simple sentence be a constant, to gain the benefits of using an array of type `Unifiable` to represent simple sentences, we did not make this distinction internally. Instead, we enforce it in the constructor. This approach maintains the integrity of the Predicate Calculus implementation, while giving is the internal simplicity of representing a simple sentence as an array of items of type `Unifiable`.

Defining unify(…) and Substitution Sets

To complete this part of the implementation, we need to define the `unify` method. `unify` has the signature:

```
public SubstitutionSet unify(Unifiable p,
  SubstitutionSet s)
```

A call to unify takes a `Unifiable` expression and a `SubstitutionSet` containing any bindings from the algorithm so far. For example, if `exp1` and `exp2` are both of type `Unifiable`, and `s` is an initial `SubstitutionSet`, we unify the expressions by:

```
exp1.unify(exp2, s)
```

or by:

```
exp2.unify(exp1, s)
```

Both calls are equivalent. If the unification succeeds, unify will return a new substitution set, adding additional variable bindings to those passed in the parameters. If it fails, `unify` will return `null.` In either case, the original `SubstitutionSet` will be unchanged. The class maintains variable bindings as a list of `Variable/Unifiable` pairs, using the `HashMap` collection type:

```
public class SubstitutionSet
{
  private HashMap<Variable, Unifiable> bindings =
     new HashMap<Variable, Unifiable>();
  public SubstitutionSet(){}
  public SubstitutionSet(SubstitutionSet s)
  {                 this.bindings =
        new HashMap<Variable,
           Unifiable>(s.bindings);
  }
  public void clear()
  {
     bindings.clear();
  }
  public void add(Variable v, Unifiable exp)
  {
     bindings.put(v, exp);
  }
  public Unifiable getBinding(Variable v)
  {
     return (Unifiable)bindings.get(v);
  }
  public boolean isBound(Variable v)
  {
     return bindings.get(v) != null;
  }
  public String toString()
  {
     return "Bindings:[" + bindings + "]";
  }
}
```

This is a straightforward implementation that does little more than "wrap" the `HashMap` in the `SubstitutionSet` object (see the design discussion in section 23.3.1 for the reasons behind this approach). Finally, we will define `unify` for each class. Implementing it for `Constant` is straightforward and addresses two cases: if the expression to be matched is equal to the constant, unify returns a new substitution set; if the expression is a variable, it calls `unify` on that variable.

```
public class Constant implements Unifiable
{
  //constructors and other methods as defined above.
```

```
      public SubstitutionSet unify(Unifiable exp,
                                   SubstitutionSet s)
      {
         if (this == exp)
            return new SubstitutionSet(s);
         if (exp instanceof Variable)
            return exp.unify(this, s);
         return null;
      }
   }
```

Defining `unify` for a variable is a bit more complicated, since it must manage bindings:

```
   public class Variable implements Unifiable
   {
      // constructors and other methods as defined above
      public SubstitutionSet unify(Unifiable p,
                  SubstitutionSet s)
      {
         if (this == p) return s;
         if(s.isBound(this))
            return s.getBinding(this).unify(p, s);
         SubstitutionSet sNew = new SubstitutionSet(s);
         sNew.add(this, p);
         return sNew;
      }
   }
```

This definition checks three cases. The first is if the expressions are equal: anything matches itself. Second, it checks if the variable is bound in the substitution set `s`; if it is, it retrieves the binding, and calls `unify` on it. Finally, the variable is unbound and the algorithm adds the binding to a new substitution set and returns it.

We define `unify` for `SimpleSentence` as unifying two lists by moving through them, unifying corresponding elements. If this succeeds, it returns the accumulated substitutions.

```
   public SubstitutionSet unify(Unifiable p,
           SubstitutionSet s)
   {
      if (p instanceof SimpleSentence)
      {
         SimpleSentence s2 = (SimpleSentence) p;
         if (this.length() != s2.length())
            return null;
         SubstitutionSet sNew = new SubstitutionSet(s);
```

```
            for (int i = 0; i < this.length(); i++)
            {
               sNew = this.getTerm(i).unify(s2.getTerm(i),
                         sNew);
               if(sNew == null)
                  return null;
            }
            return sNew;
         }
         if(p instanceof Variable)
            return p.unify(this, s);
         return null;
      }
```

This method tests two cases. If the argument `p` is a simple sentence, it casts `p` to an instance of `SimpleSentence`: `s2`. As an efficiency measure, the method checks to make sure both simple sentences are the same length, since they cannot match otherwise. Then, the method iterates down the elements of each simple sentence, attempting to unify them recursively. If any pair of elements fails to unify the entire unification fails. The second case is if `p` is a `Variable`. If so, the method calls `unify` on `p`.

Testing the unify Algorithm

To simplify testing of the algorithm, we write one more method to replace any bound variable with its binding. We will define this method signature at the level of the interface `PCExpression`:

```
public interface PCExpression
{
   public PCExpression
         replaceVariables(SubstitutionSet s);
}
```

Defining the method for a constant is straightforward:

```
public class Constant implements Unifiable
{
                  //Use constructors and other methods as defined above.
   public PCExpression
      replaceVariables(SubstitutionSet s)
   {
      return this;
   }
}
```

In the case of variables, the method must search the substitution set to find the binding of the variable. Since a variable may be bound to other variables, the method must search until it finds a constant binding or a final, unbound variable:

```
public class Variable implements Unifiable
{
```

```
                  //Use constructors and other methods as defined above.
  public PCExpression replaceVariables(
        SubstitutionSet s)
  {
     if(s.isBound(this))
        return
           s.getBinding(this).replaceVariables(s);
     else
        return this;
  }
}
```

Finally, a `SimpleSentence` replaces variables with bindings in all its terms, and then creates a new sentence from the results:

```
public class SimpleSentence implements Unifiable
{
                  //Use constructors and other methods as defined above.
  public PCExpression
        replaceVariables(SubstitutionSet s)
  {
     Unifiable[] newTerms = new
        Unifiable[terms.length];
     for(int i = 0; i < length(); i++)
        newTerms[i] =
           (Unifiable)terms[i].replaceVariables(s);
     return new SimpleSentence(newTerms);
}
```

Using these definitions of our key objects, an example `UnifyTester` can create a list of expressions and try a series of goals against them:

```
public class UnifyTester
{
  public static void main(String[] args)
  {
     Constant friend = new Constant("friend"),
              bill = new Constant("bill"),
              george = new Constant("george"),
              kate = new Constant("kate"),
              merry = new Constant("merry");
     Variable X = new Variable("X"),
              Y = new Variable("Y");
     Vector<Unifiable> expressions =
           new Vector<Unifiable>();
     expressions.add(new SimpleSentence(friend,
           bill, george));
```

```
            expressions.add(new SimpleSentence(friend,
                  bill, kate));
            expressions.add(new SimpleSentence(friend,
                  bill, merry));
            expressions.add(new SimpleSentence(friend,
                  george, bill));
            expressions.add(new SimpleSentence(friend,
                  george, kate));
            expressions.add(new SimpleSentence(friend,
                  kate, merry));
            //Test 1
            Unifiable goal = new SimpleSentence(friend, X,
                  Y);
            Iterator iter = expressions.iterator();
            SubstitutionSet s;
            System.out.println("Goal = " + goal);
            while(iter.hasNext()){
               Unifiable next = (Unifiable)iter.next();
               s = next.unify(goal, new SubstitutionSet());
               if(s != null)
                  System.out.println(
                        goal.replaceVariables(s));
               else
                  System.out.println("False");
            }
            //Test 2
            goal = new SimpleSentence(friend, bill, Y);
            iter = expressions.iterator();
            System.out.println("Goal = " + goal);
            while(iter.hasNext()){
               Unifiable next = (Unifiable)iter.next();
               s = next.unify(goal, new SubstitutionSet());
               if(s != null)
                  System.out.println(
                     goal.replaceVariables(s));
               else
                  System.out.println("False");
            }
         }
      }
```

`UnifyTester` creates a list of simple sentences, and tests if several goals bind with them. It produces the following output:

```
Goal = (friend X_1 Y_2)
(friend bill george)
(friend bill kate)
(friend bill merry)
(friend george bill)
(friend george kate)
(friend kate merry)
Goal = (friend bill Y_2)
(friend bill george)
(friend bill kate)
(friend bill merry)
False
False
False
```

We leave the creation of additional tests to the reader.

23.4 Design Discussion

Although simple, the basic `unify` method raises a number of interesting design questions. This section addresses these in more detail.

Why define the substitutionSet class?

The `SubstitutionSet` class has a very simple definition that adds little to the `HashMap` class it uses. Why not use `HashMap` directly? This idea of creating specialized data structures around Java's general collection classes gives us the ability to address problem specific questions while building on more general functionality. A particular example of this is in detecting problem specific errors. Although the `SubstitutionSet` class defined in this chapter works when used properly, it could be used in ways that might lead to subtle bugs. Specifically, consider the `add()` method:

```
public void add(Variable v, Unifiable exp)
{
  bindings.put(v, exp);
}
```

As defined, this method would allow a subtle bug: a programmer could add a binding for a variable that is already bound. Although our implementation makes sure the variable is not bound before adding a binding, a more robust implementation would prevent any such errors, throwing an exception if the variable is bound:

```
public void add(Variable v, Unifiable exp)
{
  if(isBound(v)
     //Throw an appropriate exception.
  else
     bindings.put(v, exp);
}
```

Finishing this method is left as an exercise.

Why is unify a method of unifiable?

An alternative approach to our implementation would make `unify` a method of `SubstitutionSet`. If we assume that `exp1` and `exp2` are `Unifiable`, and `s` is a `SubstitutionSet`, unifications would look like this:

```
s.unify(exp1, exp2);
```

This approach makes sense for a number of reasons. `SubstitutionSet` provides an essential context for unification. Also, it avoids the somewhat odd syntax of making one expression an argument to a method of another (`exp1.unify(exp2, s)`), even though they are equal arguments to what is intuitively a binary operator. Although harmless, many programmers find this asymmetry annoying.

In preparing this chapter, we experimented with both approaches. Our reasons for choosing the approach we did is that adding the `unify` method to the `SubstitutionSet` made it more than a "unification-friendly" data structure, giving it a more complex definition. In particular, it had to test the types of its arguments, an action our approach avoids. A valuable design guideline is to attempt to make all objects have simple, well-defined behaviors as this can reduce the impact of future changes in the design. Exercise 4 asks the reader to implement and evaluate both approaches.

Why introduce the interface definition: unifiable?

Early in the chapter, we introduced the `Unifiable` interface to define the `unify` method signature, rather than making `unify` a method of `PCExpression`. This raises a design question, since, as defined in (Luger 2009), the unification algorithm can apply to all Predicate Calculus expressions (including expressions containing *implies*, *and*, *or* and *not*).

The short answer to this question is that we could have placed the unification functionality in `PCExpression`. However, as we move into expressions containing operators, we encounter the added problems of search, particularly for implications, which can be satisfied in different ways. We felt it better to separate simple unification (this chapter) from the problems of search we present in Chapter 24) for several reasons.

First, since unifying two expressions with operators such as *and* decomposes into unifying their component terms, we can isolate the handling of variable bindings to simple sentences.

Second, our intuitions suggest that adding search to our problem solver naturally creates a new context for our design. Separating search from unification simplifies potential problems of adding specialized search capabilities to our problem solver.

Finally, our goal in a logic problem solver is not only to find a set of variable bindings that satisfy a goal, but also to construct a trace of the solution steps. This trace, called a *proof tree*, is an important structure in expert systems reasoning, as presented in Chapter 25. For simplicity sake, we did not want to include atomic items, e.g., constants and variables, into the proof tree. As we will see in the next chapter, introducing the `Unifiable` interface helps with this.

23.5 Conclusion: Mapping Logic into Objects

In the introduction to this chapter, we argued for the benefits of meta-linguistic abstraction as an approach to developing large-scale problem solvers, knowledge systems, learning programs, and other systems common to Artificial Intelligence. There are a number of advantages of taking this approach including:

Reuse. The most obvious benefit of meta-linguistic abstraction is in reuse of the high-level language. The logic reasoner we are constructing, like the Prolog language it mirrors, greatly simplifies solving a wide range of problems that involve logical reasoning about objects and relations in a domain. The Prolog chapters of this book illustrate the extent of logic's applicability. By basing our designs on well-structured formalisms like logic, we gain a much more powerful foundation for code reuse than the ad-hoc approaches often used in software organizations, since logic has been designed to be a general representation language.

Expressiveness. The expressiveness of any language involves two questions. What can we say in the language? What can we say easily? Formal language theories have traditionally addressed the first question, as reflected in the Chomsky hierarchy of formal languages (Luger 2009, Section 15.2), which defines a hierarchy of increasingly powerful languages going from regular expressions to Turing complete languages like Java, Lisp, or Prolog. Predicate calculus, when coupled with the appropriate interpreter is a complete language, although there are benefits to less expressive languages. For example, regular expressions are the basis of many powerful string-processing languages. The second question, what can we say easily, is probably of more practical importance to Artificial Intelligence. Knowledge representation research has given us a large number of languages, each with their own strengths. Logic has unique power as a model of sound reasoning, and a well-defined semantics. Semantic networks and Frames give us a psychologically plausible model of memory organization. Semantic networks also support reasoning about relationships in complex linguistic or conceptual spaces. Genetic algorithms implement a powerful heuristic for searching the intractable spaces found in learning and similar problems by unleashing large populations of simple, hill-climbing searches throughout the space. Although knowledge representation research is not in its infancy, it is still a young field that will continue to provide formalisms for the design of meta-languages.

Support for Design. Although languages like logic are very different from an object-oriented language like Java, object-orientation is surprisingly well suited to building interpreters for meta-linguistic abstraction. The reason for this is that, as formal languages, representation schemes have clearly defined objects and relations that support the standard object-oriented design process of mapping domain objects, relations, and behaviors into Java classes and methods. Although, as discussed above, design still involves hard choices with no easy answer, objects provide a strong framework for design.

Semantics and Interpretation. In this chapter, we have focused on the representation of predicate calculus expressions. The other aspect of meta-linguistic abstraction is semantic: how are these expressions interpreted in problem solving. Because many knowledge representation languages, when properly designed, have their foundations in formal mathematics, they offer a clear basis for implementing program behavior. In the example of predicate calculus, this foundation is in logic reasoning using inference rules like modus ponens and resolution. These provide a clear blueprint for implementing program behavior. As was illustrated by our approach to unification in this chapter, and logic-based reasoning in the next, meta-linguistic abstraction provides a much sounder basis for building quality software in complex domains.

Exercises

1. In the friends example of Section 23.1, check whether there are any other situations where friends(X, Y) is true. How many solutions to this query are there? How might you prevent someone from being friends of themselves?

2. Review the recursive list-based unification algorithm in Luger (2009, Section 2.3.3). Run that algorithm on the predicate pairs of friends(george, X, Y), friends(X, fred, Z), and friends(Y, bill, tuesday). Which pairs unify, which fail and why? The unification algorithm in this chapter is based on the Luger (2009, Section 2.3.3) algorithm without the backtrack component and occurs check.

3. Section 23.3.1 suggested augmenting the `SubstitutionSet` data structure with problem-specific error detection. Do so for the class definition, beginning with the example started in that section. Should we define our own exception classes, or can we use built-in exceptions? What other error conditions could we present?

4. Rewrite the problem solver to make `unify` a method of `SubstitutionSet`, as discussed in 23.3.2. Compare this approach with the chapter's approach for ease of understanding, ease of testing, maintainability, and robustness.

5. As defined, the unify method creates a new instance of `SubstitutionSet` each time it succeeds. Because object creation is a relatively expensive operation in Java, this can introduce inefficiencies into the code. Our reasons for taking this approach include helping the programmer avoid inadvertent changes to the `SubstitutionSet` once we introduce unification into the complex search structures developed in the next chapter. What optimizations could reduce this overhead, while maintaining control over the `SubstutionSet`?

6. Add a class `Function` to define evaluable functions to this model. A reasonable approach would be for the class `Function` to define a default evaluation method that returns the function as a pattern to be unified. Subclasses of `Function` can perform actual evaluations of interest. Test your implementation using functions such as simple arithmetic operations, or an `equals` method that returns `true` or `false`.

7. Representing predicate calculus expressions as Java objects simplifies our implementation, but makes it hard to write the expressions. Write a "front end" to the problem solver that allows a user to enter logical expressions in a friendlier format. Approaches could include a Lisp or Prolog like format or, what is more in the spirit of Java, an XML syntax.