24 A Logic-Based Reasoning System

Chapter Objectives

Predicate calculus representation extended:
- Continued use of meta-linguistic abstraction
- Java interpreter for predicate calculus expressions

Search supported by unification

Proof trees implemented
- Capture logic inferences
- Represent structure of logic-based implications
- Reflect search path for producing proofs

`Tester` class developed
- Tests code so far built

Extensions proposed for user transparency

Java idioms utilized

Implement separation of representation and search
- Use of static structures

Chapter Contents

24.1 Introduction

Chapter 23 introduced *meta-linguistic abstraction* as an approach to solving the complex problems typically found in Artificial Intelligence. That chapter also began a three-chapter (23, 24, and 25) exploration this idea through the development of a reasoning engine for predicate calculus. Chapter 23 outlined a scheme for representing predicate calculus expressions as Java objects, and developed the unification algorithm for finding a set of variable substitutions, if they exist, that make two expressions in the predicate calculus equivalent. This chapter extends that work to include more complex predicate expressions involving the logical operators *and*, ∧, *or*, ∨, *not*, ¬, and implication, ←, and develops a reasoning engine that solves logic queries through the backtracking search of a state space defined by the possible inferences on a set of logic expressions.

24.2 Reasoning in Logic as Searching an And/Or Graph

A *logic-based reasoner* searches a space defined by sequences of valid inferences on a set of predicate logic sentences. For example:

```
likes(kate, wine).
likes(george, kate).
likes(david, kate).
friends(X, Y) ← likes(X, Z) ∧ likes(Y, Z).
```

We can see intuitively that, because both `likes(george, kate)` and `likes(david, kate)` are true, it follows from the "`friends` rule" that `friends(george, david)` is true. A more detailed explanation of this reasoning demonstrates the search process that constructs these conclusions formally. We begin by unifying the goal query, `friends(george, david)`, with the conclusion, or head of the `friends` predicate under the substitutions `{george/X, david/Y}`, as seen in Figure 24.1.

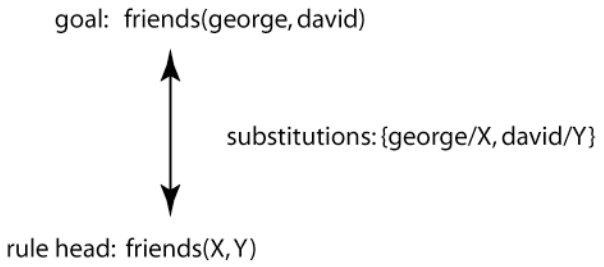

Figure 24.1. The set of variable substitutions, found under unification, by which the two `friends` predicates are identical.

Figure 24.2 illustrates the result of propagating these substitutions through the body of the rule. As the figure suggests, under the inference rule of *modus ponens*, `friends(george, david)` is true if there exists some binding for `Z` such that `likes(george, Z)` and `likes(david, Z)` are true. When viewed in terms of search, this leads to the sub-goal of proving the rule premise, or that the "tail," of the rule is true. Figure 24.2 illustrates this structure of reasoning as a graph. The arc joining the branches between the two `likes` predicates indicates that they are joined by a logical `and`. For the conclusion `friends(george, david)` to be true, we must find a substitution for `Z` under which both `likes(george, Z)` and `likes(david, Z)` are true. Figure 24.2 is an example of a representation called an *and/or* graph (Luger 2009, Section 3.3). And/or graphs represent systems of logic relationships as a graph that can be searched to find valid logical inferences. *And* nodes require that all child branches be satisfied (found to be true) in order for the entire node to be satisfied. *Or* nodes only require that one of the child branches be satisfied.

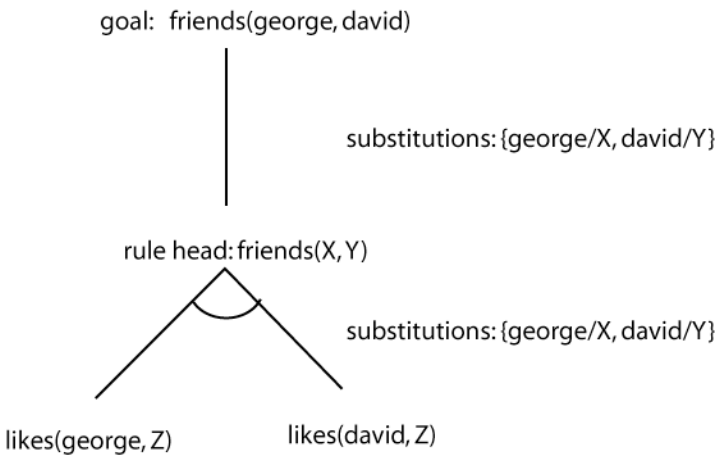

Figure 24.2. Substitution sets supporting the graph search of the `friends` predicate.

As we continue building this graph, the next step is to match the sentence `likes(george, Z)` with the different `likes` predicates. The first attempt, matching `likes(george, Z)` with `likes(kate, wine)` fails to unify. Trying the second predicate, `likes(george, kate)` results in a successful match with the substitution `{kate/Z}`, as in Figure 24.3.

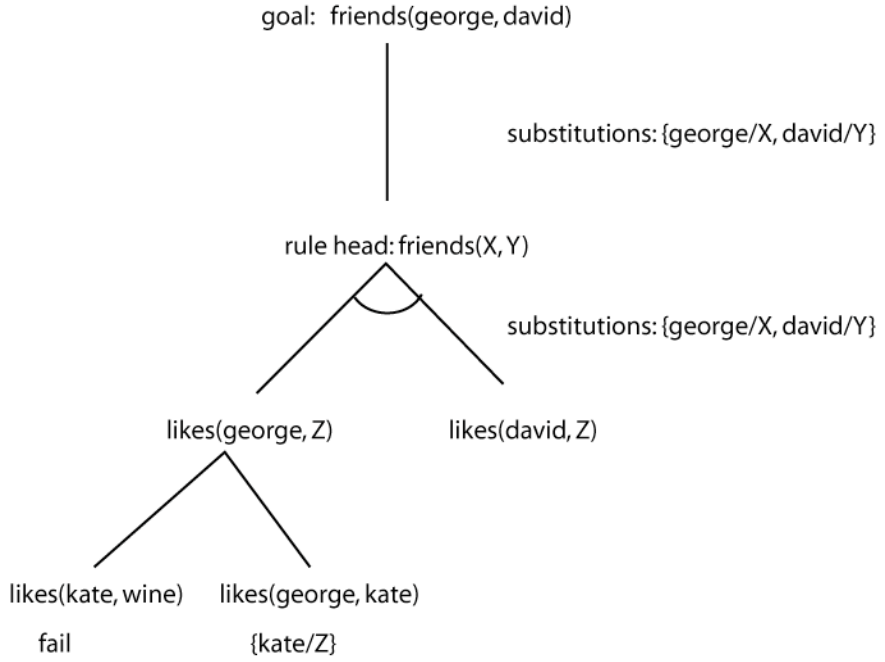

Figure 24.3 Substitution sets supporting the search to satisfy the `friends` predicate.

Note that the branches connecting the goal `likes(george, Z)` to the different attempted matches in the graph are not connected. This indicates an *or* node, which can be satisfied by matching any one of the branches.

The final step is to apply the substitution `{kate/Z}` to the goal sentence `likes(david, Z)`, and to try to match this with the logic expressions. Figure 24.4 indicates this step, which completes the search and proves the initial `friends` goal to be true. Note again how the algorithm tries the alternative branches of the *or* nodes of the graph to find a solution.

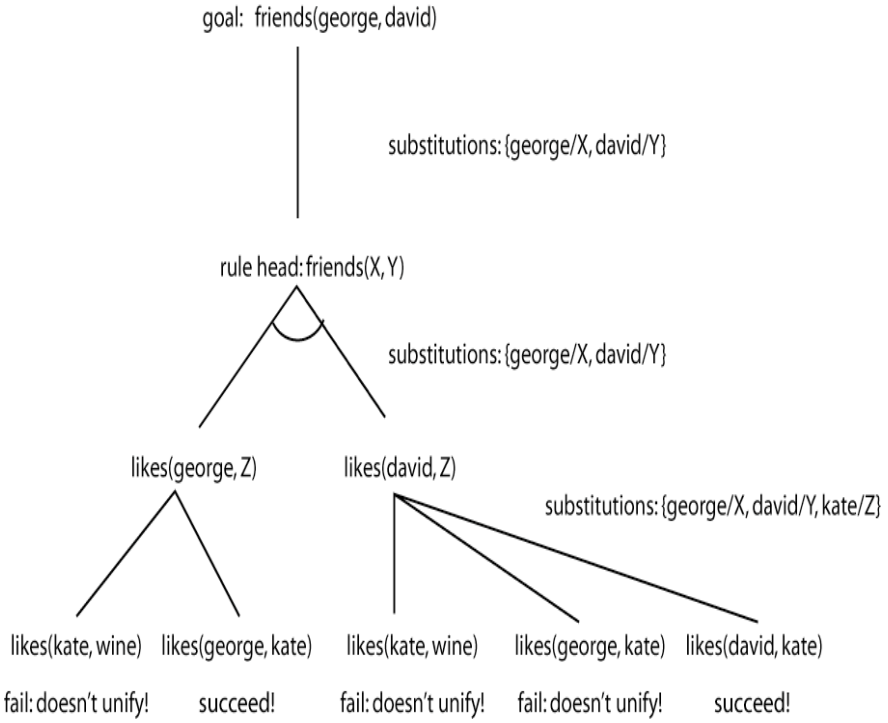

Figure 24.4. A search-based solution of the `friends` relationship.

This process of trying alternative branches of a state space can be implemented as a backtracking search. If a goal in the search space fails, such as trying to match `likes(george, Z)` and `likes(kate, wine)`, the algorithm backtracks and tries the next possible branch of the search space. The basic backtracking algorithm is given in (Luger 2009, Section 3.2) as:

> If some state S does not offer a solution to a search problem, then open and investigate its first child S_1 and apply the backtrack procedure recursively to this node. If no solution emerges from the subtree rooted by S_1 then fail S_1 and apply backtrack recursively to the second child S_2. Continuing on, if no solution emerges from any of the children of S, then fail back to S's parent and apply backtrack to S's first sibling.

Before implementing our logic-based reasoner as a backtracking search of and/or graphs, there is one more concept we need to introduce. That is the notion of a *proof tree*. If we take only the successful branches of our search, the result is a tree that illustrates the steps supporting the conclusion, as can be seen in Figure 24.5. In implementing a logic-based reasoning system, we not only search an and/or graph, but also construct the proof tree illustrating a successful search.

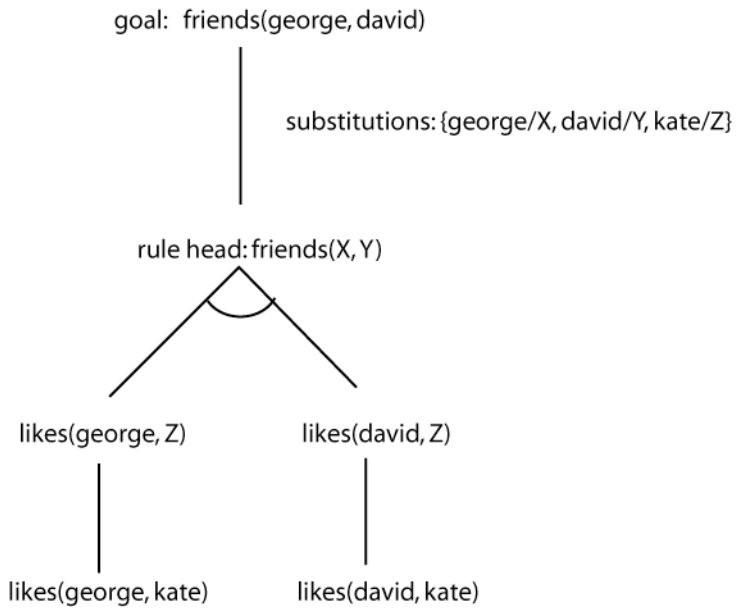

Figure 24.5. A proof tree showing the successful satisfaction of the `friends` predicate.

24.3 The Design of a Logic-Based Reasoning System

The first step in designing a logic-based reasoning system is to create a representation for the logical operators *and*, ∧, *or*, ∨, *not*, ¬, and implication, ⟵. Figure 24.6 begins this process by adding several classes and interfaces to those described in Figure 23.2.

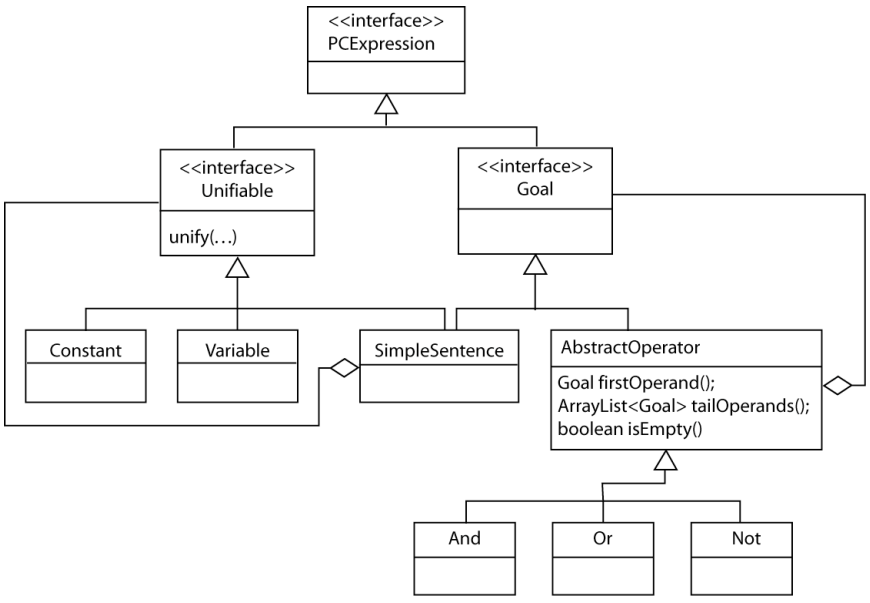

Figure 24.6. Classes and interfaces for a logic-based inference system.

The basis of this extension is the interface, `Goal`. Expressions that will appear as goals in an and/or graph must implement this interface. These include `SimpleSentence`, and the basic logical operators. We will add methods to this interface shortly, but first it is worth looking at a number of interesting design decisions supported by this object model.

The first of these design decisions is to divide `PCExpressions` into two basic groups: `Unifiable`, which defines the basic unification algorithm, and `Goal` which defines nodes of our search space. It is worth noting that, when we were developing this algorithm, our initial approach did not make this distinction, but included both basic unification and the search of logical operators in the `unify` method, which was specified in the top-level interface, `PCExpression`.

We chose to re-factor the code and divide this functionality among the two interfaces because 1) the initial approach complicated the `unify` method considerably, and 2) since the objects `Constant` and `Variable` did not appear in proof trees, we had to treat these as exceptions, complicating both search and construction of proof trees. Note also that `SimpleSentence` implements both interfaces. This is an example of how Java uses interfaces to achieve a form of multiple inheritance.

Another important aspect of this design is the introduction of the `AbstractOperator` class. As indicated in the model of Figure 24.6, an `AbstractOperator` is the parent class of all logical operators. This abstract class defines the basic handling of the arguments of operators through the methods `firstOperand`, `tailOperands`, and `isEmpty`. These methods will enable a recursive search to find solutions to the different operands.

To complete our logical representation language, we need to define Horn Clause rules. Rules do not correspond directly to nodes in an and/or graph; rather, they define relationships between nodes. Consequently, the `Rule` class will be a direct descendant of `PCExpression, as shown in` Figure 24.7, where a rule is a Horn Clause, taking a `SimpleSentence` as its conclusion, or *head*, and a `Goal` as its premise, or *tail*.

This completes the classes defining our logic-based language. The next section gives their implementation, which is fairly straightforward, and Section 24.5 adds new classes for searching the and/or graph defined by inferences on these expressions. This decision to define separate classes for the representation and search reflects common AI programming practice.

24.4 Implementing Complex Logic Expressions

Implementing complex expressions starts with the `Goal` interface. Although Section 24.5 adds a method to this definition, for now, it is a methodless `interface`:

```
public interface Goal extends PCExpression {}
```

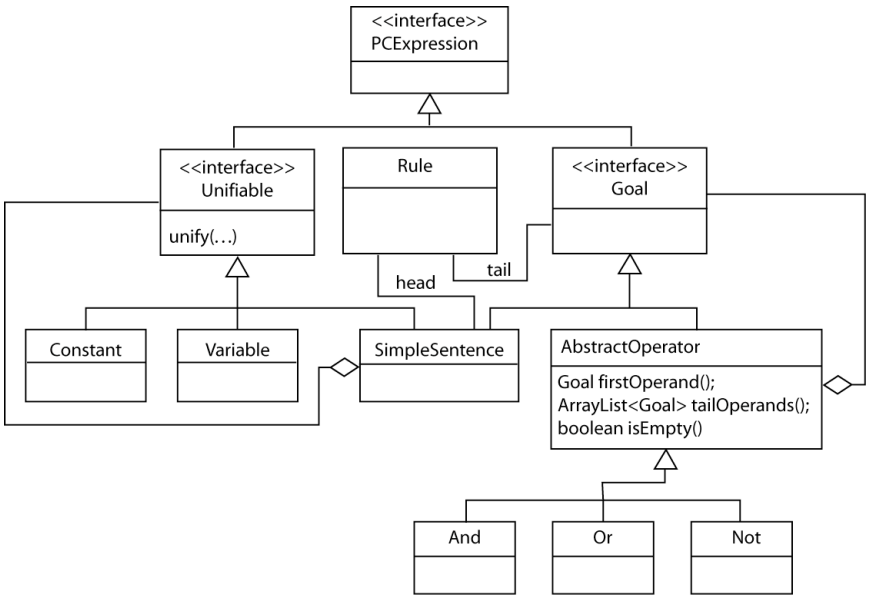

Figure 24.7. A Horn clause `Rule` representation as an instance of `PCExpression`.

Later, we modify `SimpleSentence` to implement this interface, but first, we define a new class, called `AbstractOperator`, that defines the basic methods for accessing the arguments of n-ary operators. In keeping with common Java practice, we implement several patterns for accessing operators, including retrieval of operands by number using the methods `operandCount()` and `getOperand(int i)`. Since we also want to support recursive algorithms for manipulating operands, we implement a *head/tail* approach similar to the `car/cdr` pattern widely used in Lisp. We do this through the methods `firstOperand()`, `getOperatorTail()`, and `isEmpty()`. We also define the `replaceVariables()` method required of all `PCExpressions`, taking advantage of the class' general representation of operands.

Implementation of these methods is straightforward, and we do not discuss it other than to present the code:

```
public abstract class AbstractOperator
        implements Goal, Cloneable
{
    protected ArrayList<Goal> operands;
    public AbstractOperator(Goal... operands)
    {
        Goal[] operandArray = operands;
        this.operands = new ArrayList<Goal>();
        for(int i = 0; i < operandArray.length;i++)
        {
            this.operands.add(operandArray[i]);
        }
    }
```

```
public AbstractOperator(ArrayList<Goal>
        operands)
{
    this.operands = operands;
}
public void setOperands(ArrayList<Goal>
        operands)
{
    this.operands = operands;
}
public int operandCount()
{
    return operands.size();
}
public Goal getOperand(int i)
{
    return operands.get(i);
}
public Goal getFirstOperand()
{
    return operands.get(0);
}
public AbstractOperator getOperatorTail()
        throws CloneNotSupportedException
{
    ArrayList<Goal> tail = new
        ArrayList<Goal>(operands);
    tail.remove(0);
    AbstractOperator tailOperator =
        (AbstractOperator)this.clone();
    tailOperator.setOperands(tail);
    return tailOperator;
}
public boolean isEmpty()
{
 return operands.isEmpty();
}
public PCExpression
        replaceVariables(SubstitutionSet s)
        throws CloneNotSupportedException
```

```
        {
            ArrayList<Goal> newOperands =
                new ArrayList<Goal>();
            for(int i = 0; i < operandCount(); i++)
                newOperands.add((Goal)
                    getOperand(i).
                        replaceVariables(s));
            AbstractOperator copy =
                (AbstractOperator) this.clone();
            copy.setOperands(newOperands);
            return copy;
        }
    }
```

The `And` operator is a simple extension to this class. At this time, our implementation includes just the `toString()` method. Note use of the accessors defined in `AbstractOperator()`:

```
public class And extends AbstractOperator
{
    public And(Goal... operands)
    {
        super(operands);
    }
    public And(ArrayList<Goal> operands)
    {
        super(operands);
    }
    public String toString()
    {
        String result = new String("(AND ");
        for(int i = 0; i < operandCount(); i++)
            result = result +
                getOperand(i).toString();
            return result;
    }
}
```

We leave implementation of `Or` and `Not` as exercises.

Finally, we implement `Rule` as a Horn Clause, having a `SimpleSentence` as its conclusion, or *head*, and any `Goal` as its premise, or *tail*. At this time, we provide another basic implementation, consisting of accessor methods and the `replaceVariables()` method required for all classes implementing `PCExpression`. Also, we allow `Rule` to have a head only (i.e., body = null), as follows from the

definition of Horn Clauses. These rules correspond to simple assertions, such as `likes(george, kate)`.

```
public class Rule implements PCExpression
{
    private SimpleSentence head;
    private Goal body;
    public Rule(SimpleSentence head)
    {
        this(head, null);
    }
    public Rule(SimpleSentence head, Goal body)
    {
        this.head = head;
        this.body = body;
    }
    public SimpleSentence getHead()
    {
            return head;
    }
    public Goal getBody()
    {
            return body;
    }
    public PCExpression
            replaceVariables(SubstitutionSet s)
            throws CloneNotSupportedException
    {
        ArrayList<Goal> newOperands =
            new ArrayList<Goal>();
        for(int i = 0; i < operandCount(); i++)
        newOperands.add((Goal)getOperand(i).
                replaceVariables(s));
        AbstractOperator copy =
            (AbstractOperator)this.clone();
        copy.setOperands(newOperands);
        return copy;
    }
    public String toString()
    {
     if (body == null)
       return head.toString();
```

```
            return head + " :- " + body;
        }
    }
```

Up to this point, the implementation of complex expressions has been straightforward, focusing on operators for manipulating their component structures. This is because their more complex semantics is a consequence of how they are interpreted in problem solvers. The next section discusses the design of the logic-based reasoning engine that supports this interpretation.

24.5 Logic-Based Reasoning as And/Or Graph Search

The basis of our implementation of and/or graph search is a set of classes for defining nodes of the graph. These will correspond to simple sentences, and the operators `And`, `Or`, and `Not`. In this section we define nodes for `And` with simple sentences, leaving `Or` and `Not` as exercises. Our approach is to construct an and/or graph as we search. When the search terminates in success, this graph will be the *proof tree* for that solution. If additional solutions are desired, a call to a `nextSolution()` method causes the most recent subgoal to fail, resuming the search at that point. If there are no further solutions from that subgoal, the search will continue to "fail back" to a parent goal, and continue searching. The implementation will repeat this backtracking search until the space is exhausted.

Figure 24.8 illustrates this search. At the top of the figure we begin with an initial and/or graph consisting only of the initial goal (e.g., `friends(george, X)`). A call to the method `nextSolution()` starts a search of the graph and constructs the proof tree, stopping the algorithm. In addition to constructing the proof tree, each node stores its state at the time the search finished, so a second call to `nextSolution()` causes the search to resume where it left off.

This technique is made possible by a programming pattern known as *continuations*. Continuations have had multiple uses but the main idea is that they allow the programmer to save the program execution at any instant (state) in time so that it can be re-started from that point sometime in the future. In languages that support continuations directly, this is usually implemented by saving the program stack and program counter at the point where the program is frozen. Java does not support this pattern directly, so we will implement a simplified form of the continuation pattern using object member variables to save a reference to the current goal, the current rule used to solve it, and the current set of variable bindings in the tree search. Figure 24.9 shows the classes we introduce to implement this approach.

`AbstractSolutionNode` defines the basic functionality for every node of the graph, including the abstract method `nextSolution()`. `AbstractSolutionNode` and its descendants will implement the ability to search the and/or graph, to save the state of the search, and to resume on subsequent calls to `nextSolution()`.

The class `RuleSet` maintains the logic base, a list of rules. The intention is that all nodes will use the same instance of `RuleSet`, with each instance of `AbstractSolutionNode` maintaining a reference to a particular rule in the set to enable the continuation pattern.

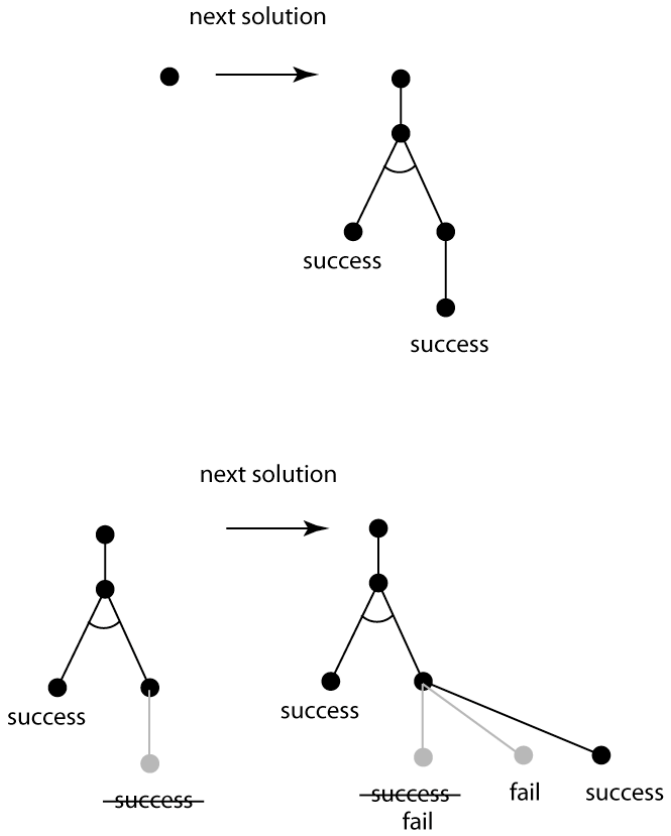

Figure 24.8. An example search space and construction of the proof tree.

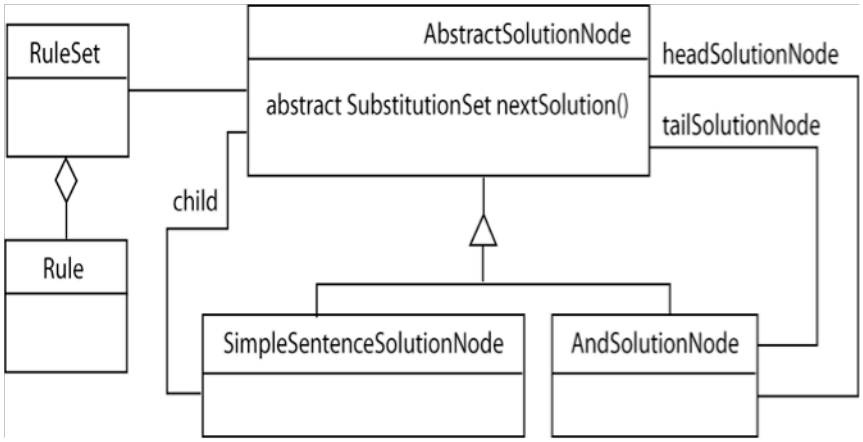

Figure 24.9. The class structure for implementing continuations.

The descendants of `AbstractSolutionNode` maintain references to their children. `SimpleSentenceSolutionNode` represents a simple sentence as a goal, and maintains a reference to its child: the head of a rule. `AndSolutionNode` represents an `and` node, and keeps a

reference to the first branch of the `and` node (the relationship labeled `headSolutionNode`) and the subsequent branches in the `and` node (the relationship labeled `tailSolutionNode`).

We begin implementation with the `RuleSet` class:

```
public class RuleSet
{
    private Rule[] rules;
    public RuleSet(Rule... rules)
    {
        this.rules = rules;
    }
    public Rule getRuleStandardizedApart(int i)
    {
        Rule rule =
            (Rule)rules[i].
                standardizeVariablesApart(
                    new Hashtable<Variable,
                        Variable>());
        return rule;
    }
    public Rule getRule(int i)
    {
        return rules[i];
    }
    public int getRuleCount()
    {
        return rules.length;
    }
}
```

This definition is simple: it maintains an array of rules and allows them to be retrieved by number. The only unusual element is the method `getRuleStandardizedApart(int i)`. This is necessary because the scope of logical variables is the single predicate sentence containing it in a single reasoning step. If we use the same rule again in the search, which is fairly common, we will need to assign new bindings to the variables. A simple way to insure this is to replace the variables in the rule with new copies having the same name. This operation, called "standardizing variables apart" must be defined for all expressions in the rule set. To support this, we will add a new method signature to the interface `PCExpression`. This interface now becomes:

```
public interface PCExpression
{
    public PCExpression
            standardizeVariablesApart(
```

```
                Hashtable<Variable, Variable> newVars);
        public PCExpression
                replaceVariables(SubstitutionSet s);
    }
```

The intention here is that the method will be recursive, with each type of `PCExpression` giving it its own appropriate definition. In the method signature, the hash table of pairs of variables keeps track of the substitutions made so far, since a variable may occur multiple times in an expression, and will need to use the same replacement. Defining this requires changes to the following classes. `AbstractOperator` will define it for all n-ary operators:

```
public abstract class AbstractOperator implements
Goal, Cloneable
{
    // variables and methods as already defined
    public PCExpression
            standardizeVariablesApart(
                Hashtable<Variable,
                Variable>wVars)
            throws CloneNotSupportedException
    {
        ArrayList<Goal> newOperands =
            new ArrayList<Goal>();
        for(int i = 0; i < operandCount(); i++)
            newOperands.add((Goal)getOperand(i).
            standardizeVariablesApart(newVars));
        AbstractOperator copy =
            (AbstractOperator) this.clone();
        copy.setOperands(newOperands);
        return copy;
    }
}
```

We will also define the method for existing classes `SimpleSentence`, `Constant`, and `Variable`. The definition for Constant is straightforward: each constant returns itself.

```
 public class Constant implements Unifiable
{
    // variables and methods as previously defined
    public PCExpression
        standardizeVariablesApart(
            Hashtable<Variable, Variable> newVars)
    {
         return this;
    }
}
```

The definition for `Variable` is also straightforward, and makes use of the copy constructor defined earlier.

```
public class Variable implements Unifiable
{
        // variables and methods already defined.
    public PCExpression standardizeVariablesApart(
            Hashtable<Variable, Variable>
                newVars)
    {
        Variable newVar = newVars.get(this);
                // Check if the expression already has
                    // a substitute variable.
        if(newVar == null)      // if not create one.
        {
            newVar = new Variable(this);
            newVars.put(this, newVar);
        }
        return newVar;
    }
```

SimpleSentence defines the method recursively:

```
public class SimpleSentence
        implements Unifiable, Goal, Cloneable
{
        // variables and methods already defined.
    public PCExpression
            standardizeVariablesApart(
                Hashtable<Variable, Variable>
                    newVars)
            throws CloneNotSupportedException
    {
        Unifiable[] newTerms =
            new Unifiable[terms.length];
                  //create an array for new terms.
        for(int i = 0; i < length(); i++){
            newTerms[i] =
                (Unifiable)terms[i].
                    standardizeVariablesApart(
                        newVars);
                    // Standardize apart each term.
                    // Only variables will be affected.
        }
```

```
            SimpleSentence newSentence =
                  (SimpleSentence) clone();
            newSentence.setTerms(newTerms);
            return newSentence;
        }
```

Once `RuleSet` has been defined, the implementation of `AbstractSolutionNode` is, again, fairly straightforward.

```
public abstract class AbstractSolutionNode
{
    private RuleSet rules;
    private Rule currentRule = null;
    private Goal goal= null;
    private SubstitutionSet parentSolution;
    private int ruleNumber = 0;
    public AbstractSolutionNode(Goal goal,
            RuleSet rules,
            SubstitutionSet parentSolution)
    {
        this.rules = rules;
        this.parentSolution = parentSolution;
        this.goal = goal;
    }
    public abstract SubstitutionSet nextSolution()
            throws CloneNotSupportedException;
    protected void reset(SubstitutionSet
            newParentSolution)
    {
        parentSolution = newParentSolution;
        ruleNumber = 0;
    }
    public Rule nextRule() throws
            CloneNotSupportedException
    {
        if(hasNextRule())
            currentRule =
                rules.getRuleStandardizedApart(
                      ruleNumber++);
        else
            currentRule = null;
        return currentRule; }
    protected boolean hasNextRule()
    {
        return ruleNumber < rules.getRuleCount();
    }
```

```
    protected SubstitutionSet getParentSolution()
    {
        return parentSolution;
    }
    protected RuleSet getRuleSet()
    {
        return rules;
    }
    public Rule getCurrentRule()
    {
        return currentRule;
    }
    public Goal getGoal()
    {
        return goal;
    }
}
```

The member variable `rules` holds the rule set shared by all nodes in the graph. `RuleNumber` indicates the rule currently being used to solve the goal. `ParentSolution` is the substitution set as it was when the node was created; saving it allows backtracking on resuming the continuation of the search. These three member variables allow the node to resume search where it left off, as required for the continuation pattern.

The variable `goal` stores the goal being solved at the node, and `currentRule` is the rule that defined the current state of the node. `Reset()` allows us to set a solution node to a state equivalent to a newly created node. `NextRule()` returns the next rule in the set, with variables standardized apart. The definition also includes the signature for the `nextSolution()` method. The remaining methods are simple accessors.

The next class we define is `SimpleSentenceSolutionNode`, an extension of `AbstractSolutionNode` for simple sentences.

```
public class SimpleSentenceSolutionNode extends
          AbstractSolutionNode
{
    private SimpleSentence goal;
    private AbstractSolutionNode child = null;
    public SimpleSentenceSolutionNode(
                SimpleSentence goal,
                RuleSet rules,
                SubstitutionSet parentSolution)
            throws CloneNotSupportedException
    {
        super(goal, rules, parentSolution);
    }
```

```
    public SubstitutionSet nextSolution()
    {
        SubstitutionSet solution;
        if(child != null)
        {
            solution = child.nextSolution();
            if (solution != null)
                return solution;
        }
        child = null;
        Rule rule;
        while(hasNextRule() == true)
        {
            rule = nextRule();
            SimpleSentence head = rule.getHead();
            solution = goal.unify(head,
                getParentSolution());
            if(solution != null)
            {
                Goal tail = rule.getBody();
                if(tail == null)
                    return solution;
                child = tail.getSolver
                        (getRuleSet(),solution);
                SubstitutionSet childSolution =
                         child.nextSolution();
                if(childSolution != null)
                    return childSolution;
            }
        }
        return null;
    }
    public AbstractSolutionNode getChild()
    {
        return child;
    }
}
```

This class has one member variable: `child` is the next node, or subgoal in the state space. The method `nextSolution()` defines the use of

these variables, and is one of the more complex methods in the implementation, and we will list the steps in detail.

1. The first step in `nextSolution()` is to test if `child` is `null`. If it is not, which could be the case if we are resuming a previous search, we call `nextSolution()` on the child node to see if there are any more solutions in that branch of the space. If this returns a non-null result, the method returns this solution.
2. If the child node returns no solution, the method sets `child` to `null`, and resumes trying rules in a while-loop. The loop gets each rule from the `RuleSet` in turn, and attempts to unify the goal with the rule head.
3. If the goal matches a rule head, the method then checks if the rule has a tail, or premise. If there is no tail, then this match represents a solution to the goal and returns that substitution set.
4. If the rule does have a tail, the method calls `getSolver()` on the rule tail to get a new child node. This is a new method, which we will discuss shortly.
5. Finally, the method calls `nextSolution()` on the new child node, returning this solution if there is one, and continuing the search otherwise.
6. If the while-loop exhausts the rule set, the node returns `null`, indicating there are no further solutions.

We have not discussed the method `getSolver()`mentioned in step #4. This is a new method for all classes implementing the `Goal` interface that returns the type of solution node appropriate to that goal. By letting each goal determine the proper type of solver for it, we can implement `nextSolution()` in general terms. The revised definition of `Goal`:

```
public interface Goal extends PCExpression
        throws CloneNotSupportedException
{
    public AbstractSolutionNode getSolver(
            RuleSet rules,
            SubstitutionSet parentSolution);
}
```

To complete the search implementation, we define the class, `AndSolutionNode`. Our approach to this implementation is to define a new `And` node for each argument to the `And` operator and the remaining operators. Figure 24.10 illustrates this approach. At the top of the figure is a portion of an And/Or graph for the goal `p ∧ q ∧ r ∧ s`, indicating that the top-level goal will be satisfied by a set of variable substitutions that understands all four of its child goals.

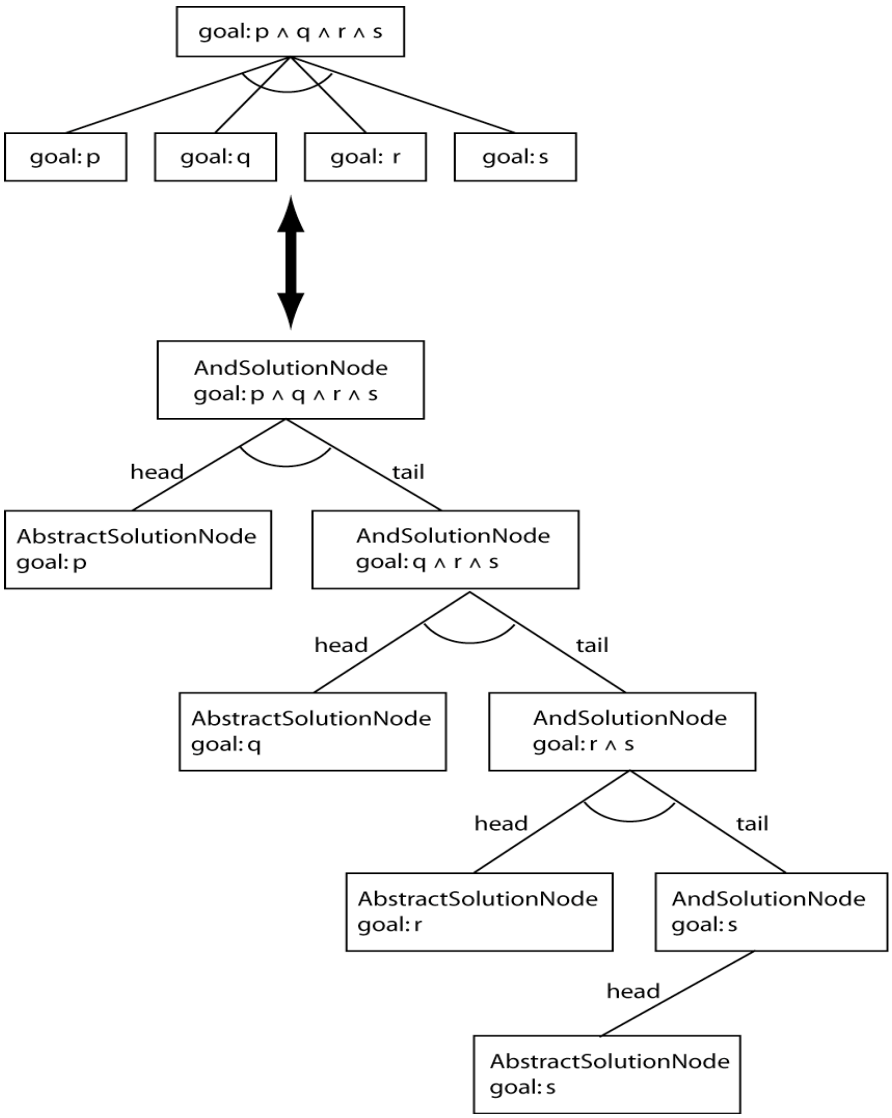

Figure 24.10 A conjunctive goal (top) and the search tree used for its solution.

The bottom of Figure 24.10 indicates the approach we will take. Instead of allowing multiple children at an `and` node, we will make each node binary, consisting of the `and` of the solution for the first operand (the head) and the subsequent operands (the tail). This supports a recursive algorithm that simplifies our code. We leave it to the student to demonstrate (preferably through a formal proof) that the two approaches are equivalent. An additional exercise to implement `and` nodes by using an iterator across a list of child nodes.

`AndSolutionNode` follows the structure of Figure 24.10:

```
public class AndSolutionNode extends
        AbstractSolutionNode
{
    private AbstractSolutionNode
        headSolutionNode = null;
```

```
private AbstractSolutionNode
     tailSolutionNode = null;
private AbstractOperator operatorTail = null;
public AndSolutionNode(And goal,
             RuleSet rules,
             SubstitutionSet parentSolution)
         throws CloneNotSupportedException
{
     super(goal, rules, parentSolution);
     headSolutionNode =
             goal.getFirstOperand().
     getSolver(rules, parentSolution);
     operatorTail = goal.getOperatorTail();
}
protected AbstractSolutionNode
     getHeadSolutionNode()
{
     return headSolutionNode;
}
protected AbstractSolutionNode
     getTailSolutionNode()
{
     return tailSolutionNode;
}
public SubstitutionSet nextSolution()
     throws CloneNotSupportedException
{
     SubstitutionSet solution;
     if(tailSolutionNode != null)
     {
          solution =
               tailSolutionNode.nextSolution();
          if(solution != null) return solution;
     }
     while(solution =
               headSolutionNode.nextSolution())
               != null)
     {
          if(operatorTail.isEmpty())
               return solution;
          else
          {
               tailSolutionNode =
                        operatorTail.getSolver(
                        getRuleSet(), solution);
```

```
                    SubstitutionSet tailSolution =
                              tailSolutionNode.
                              nextSolution();
                    if(tailSolution != null)
                         return tailSolution;
               }
          }
          return null;
     }
}
```

The constructor creates a solution node, `headSolutionNode`, for the first argument of the `And` operator, and also sets the member variable, `operatorTail`, for the rest of the arguments if they exist. Note that it does not create a solution node for the tail at this time. This is an efficiency concern: if there are no solutions to the head subgoal, the entire `and` operator will fail, and there is no need to try the rest of the operators.

As with `SimpleSolutionNode`, the `nextSolution()` method implements the search and the supporting continuation pattern. It begins by testing if `tailSolutionNode` is non-null. This is true only if there are remaining arguments `(operatorTail != null)`, and we have found at least one solution to the head goal. In this case, the continuation must first check to see if there are additional solutions to the tail goal.

When this fails, the algorithm enters a loop of testing for further solutions to the head goal. When it finds a new solution to the head, it checks if there is a tail goal; if not, it returns the solution. If there is a tail goal, it will acquire the child node, a subclass of `AbstractSolutionNode` using the `getSolver` method, and then tries for a solution to the tail goal.

This completes the implementation of the search framework for the `And` operator. We leave implementation of `Or` and `Not` to the reader.

24.6 Testing the Reasoning System

Below is a simple `Tester` class for the reasoning system. It uses a recursive rule for reasoning about ancestor relationships. This is a simple test harness and is not suitable for end users. Finishing the reasoner would involve allowing the representation of rules in a more friendly syntax, such as Prolog, and an interactive query engine. We leave this as an exercise. We also encourage the reader to modify this simple `Tester` to further explore the code.

```
public class Tester
{
    public static void main(String[] args)
    {                                  //Set up the knowledge base.
         Constant parent = new Constant("parent"),
              bill = new Constant("Bill"),
              audrey = new Constant("Audrey"),
              maria = new Constant("Maria"),
```

```
    tony = new Constant("Tony"),
    charles = new Constant("Charles"),
    ancestor = new Constant("ancestor");
Variable X = new Variable("X"),
    Y = new Variable("Y"),
    Z = new Variable("Z");
RuleSet rules = new RuleSet(
    new Rule(new SimpleSentence(parent,
            bill, audrey)),
    new Rule(new SimpleSentence(parent,
            maria, bill)),
    new Rule(new SimpleSentence(parent,
            tony, maria)),
    new Rule(new SimpleSentence(parent,
            charles, tony)),
    new Rule(new SimpleSentence(ancestor,
            X, Y),
    new And(new SimpleSentence(parent,
            X, Y))),
    new Rule(new SimpleSentence(ancestor,
            X, Y),
    new And(new SimpleSentence(parent,
            X, Z),
    new SimpleSentence(ancestor, Z, Y))));
  // define goal and root of search space.
SimpleSentence goal =
            new SimpleSentence(ancestor,
            charles, Y);
AbstractSolutionNode root =
           goal.getSolver(rules,
           new SubstitutionSet());
SubstitutionSet solution;
                    // print out results.
System.out.println("Goal = " + goal);
System.out.println("Solutions:");
try
{
    while((solution = root.nextSolution())
             != null)
    {
        System.out.println("     " + goal.
                 replaceVariables(
                 solution));
    }
}
catch (CloneNotSupportedException e)
```

```
            {
                System.out.println(
                    "CloneNotSupportedException:" + e);
            }
        }
    }
```

24.7 Design Discussion

In closing out this chapter, we would like to look at two major design decisions. The first is our separation of representation and search through the introduction of `AbstractSolutionNode` and its descendants. The second is the importance of static structure to the design.

Separating Representation and Search

The separation of representation and search is a common theme in AI programming. In Chapter 22, for example, our implementation of simple search engines relied upon this separation for generality. In the reasoning engine, we bring the relationship between representation and search into sharper focus. Here, the search engine serves to define the semantics of our logical representation by implementing a form of logical inference. As we mentioned before, our approach builds upon the mathematics of the representation language – in this case, theories of logic inference – to insure the quality of our representation.

One detail of our approach bears further discussion. That is the use of the method, `getSolver(RuleSet rules, SubstitutionSet parentSolution)`, which was defined in the `Goal` interface. This method simplifies the handling of the search space by letting search algorithms treat them independently of their type (simple sentence, node, etc). Instead, it lets us treat nodes in terms of the general methods defined by `AbstractSolutionNode`, and to rely upon each goal to return the proper type of solution node. This approach is beneficial, but as is typical of object-oriented design, there are other ways to implement it.

One of these alternatives is through a *factory* pattern. This would replace the `getSolver()` method of `Goal` with a separate class that creates instances of the needed node. For example:

```
Class SolutionNodeFactory
{
    public static AbstractSolutionNode
        getSolver(Goal goal,
                  RuleSet rules,
                  SubstitutionSet parentSolution)
    {
        if (goal instanceof SimpleSentence)
            return new SimpleSentenceSolutionNode(
                      goal, rules, parentSolution);
        if (goal instanceof And)
            return new AndSolutionNode(goal, rules,
                      parentSolution);
    }
}
```

There are several interesting trade-offs between the approaches. Use of the `Factory` sharpens the separation of representation and search. It even allows us to reuse the representation in contexts that do not involve reasoning without the difficulty of deciding how to handle the `getSolver` method required by the parent interface. On the other hand, the approach we did use allows us to get the desired solver without using `instanceof` to test the type of goal objects explicitly. Because the `instanceof` operator is computationally expensive, many programmers consider it good style to avoid it. Also, when adding a new operator, such as `Or`, we only have to change the operator's class definition, rather than adding the new class and modifying the `Factory` object. Both approaches, however, are good Java style. As with all design decisions, we encourage the reader to evaluate these and other approaches and make up their own mind.

The Importance of Static Structure

A more important design decision concerns the *static* structure of the implementation. By static structure, we mean the organization of classes in a program. We call it static because this structure is not changed by program execution. As shown in Figures 24.6, 24.7, and 24.9, our approach has a fairly complex static structure. Indeed, in developing the reasoner, we experimented with several different approaches (this is, we feel, another good design practice), and many of these had considerably fewer classes and simpler static structures. We chose this approach because it is usually better to represent as much of the program's semantic structure as is feasible in the class structure of the code. There are several reasons for this:

1. It makes the code easier to understand. Although our static structure is complex, it is still much simpler than the dynamic behavior of even a moderately complex program. Because it is static, we can make good use of modeling techniques and tools to understand the program, rather than relying on dynamic tracing to see what is going on in program executions.
2. It simplifies methods. A well-designed static structure, although it may be complex, does not necessarily add complexity to the code. Rather, it moves complexity from methods to the class structure. Instead of a few classes with large complex methods, we tend to have more, simpler methods. If we look at the implementation of our logic-based reasoner, the majority of the methods were surprisingly simple: mostly setting or retrieving values from a data structure. This makes methods easier to write correctly, and easier to debug.
3. It makes it easier to modify the code. As any experienced programmer has learned, the lifecycle of useful code inevitably involves enhancements. There is a tendency for these enhancements to complicate the code, leading to increased problems with bugs as the software ages. This phenomenon has been called software entropy. Because it breaks the program functionality down into many smaller methods spread among many classes, good static structure can simplify

code maintenance by reducing the need to make complex changes to existing methods.

This chapter completes the basic implementation of a logic-based reasoner, except for certain extensions including adding the operators for or and not. We leave these as an exercise. The next chapter will add a number of enhancements to the basic reasoner, such as asking users for input during the reasoning process, or replacing true/false values with quantitative measures of uncertainty. As we develop these enhancements, keep in mind how class structure supports these extensions, as well as the implementation patterns we use to construct them.

Exercises

1. Write a method of `AbstractSolutionNode` to print out a proof tree in a readable format. A common approach to this is to indent each node's description c * level, where level is its depth in the tree, and c is the number of spaces each level is indented.

2. Add classes for the logical operators `Or` and `Not`. Try following the pattern of the chapter's implementation of `And`, but do so critically. If you find an alternative approach you prefer, feel free to explore it, rewriting `Or` and `Not` as well. If you do decide on a different approach, explain why.

3. Extend the "user-friendly" input language from exercise 8 of chapter 22 to include `And, ∧`, `Or, ∨`, `Not,¬,` and `Rule, ⇐`.

4. Write a Prolog-style interactive front end to the logical reasoner that will read in a logical knowledge-base from a file using the language of exercise 2, and then enter a loop where users enter goals in the same language, printing out the results, and then prompting for another goal.

5. Implement a factory pattern for generating `solutionNodes`, and compare it to the approach taken in the chapter. A factory would be a class, named `solutionNodeFactory` with a methods that would take any needed variables and return an instance of the class `solutionNodes`.

6. Give a logical proof that the two approaches to representing And nodes in Figure 24.10 are equivalent.

7. Modify the `nextSolution()` method in `AndSolutionNode` to replace the recursive implementation with one that iterates across all the operators of an And operator. Discuss the trade-offs between efficiency, understandability, and maintainability in the two approaches.