COGS 125 / CSE 175
Introduction to Artificial Intelligence
Specification for Programming Assignment #2

David C. Noelle

Due: 5:00 P.M. on Monday, November 18, 2013

Overview

This programming assignment has two main learning goals. First, the assignment will provide you with an opportunity to practice your skills in developing programs in Java™ using the Eclipse integrated development environment. Second, this assignment will provide you with some experience in implementing a central component of a simple logical inference engine, namely the *unify* procedure used in logical pattern matching. Your efforts in implementing this procedure are intended to produce a deeper understanding of the mechanisms of logical reasoning systems. At minimum, implementing *unify* will require an understanding of the constituents of sentences in first-order logic, as well as how sentences can be made equivalent through appropriate variable substitution.

You will be provided with Java™ source code that implements a very simple backward chaining inference algorithm over definite clauses. This code will be nearly functional, missing only a few key components of the *unify* procedure. You will be required to complete the implementation of *unify*, allowing the program to appropriately answer simple logical queries put to it.

Submission for Evaluation

To complete this assignment, you must modify only a single Java™ class source file, called "`BackwardChain.java`". A template for this file will be provided to you, along with additional supporting source code. This file is to implement a very simple backward chaining inference procedure. As provided, this source code file is *not* complete, however. It is missing some versions of the overloaded method called `unify`. In particular, it is missing implementations for `unify` that take two literals, that take two logical terms, and that take two functions. Your task is to provide bodies for these three methods by modifying the "`BackwardChain.java`" file. You should *not* modify this file except to provide these three method implementations. Your implementations should work well with the rest of the provided source code, without any other modifications to that

code. Thus, your modified version of "`BackwardChain.java`" is the only file that you should submit for evaluation.

To submit your completed assignment for evaluation, log onto the "`F13-INTRO TO AI`" site on `UCMCROPS` and navigate to the "Assignments" section. Then, locate "Programming Assignment #2" and select the option to submit your solution for this assignment. Provide your program file as an attachment. Comments to the teaching team should largely appear as comments in your Java™ source code file, but you may also include brief written comments using the assignment submission web form.

Submissions must arrive by 5:00 P.M. on Monday, November 18th. Please allow time for potential system slowness immediately prior to this deadline. You may submit assignment solutions multiple times, and only the most recently submitted version will be evaluated. As discussed in the course syllabus, late assignments will not be evaluated.

If your last submission for this assignment arrives by 5:00 P.M. on Thursday, November 14th, you will receive a 10% bonus to the score that you receive for this assignment. This bonus is intended to encourage you to try to complete this assignment early.

Activities

You are to provide three versions of the first-order logic *unify* procedure, implemented as methods of the provided `BackwardChain` Java™ class. Your provided Java™ source code must be compatible with a collection of provided Java™ classes that implement a very simple backward chaining inference engine. Indeed, your assignment solution will be evaluated by combining your submitted modified `BackwardChain` class file with copies of the provided inference engine utility files and testing the resulting complete program against a variety of test cases. In other words, *your solution must work with the provided utilities, without any modifications to these provided files*.

More specifically, you are to provide the following three public methods, all appearing in the "`BackwardChain.java`" source code file:

- `BindingList unify(Literal lit1, Literal lit2, BindingList bl)`

  This method attempts to find a binding list that allows for the unification of the two provided literals, under the constraints of the given binding list. If the two literals can be unified, a new binding list is returned, augmenting the given binding list with any addition bindings needed to unify the two literals. If unification is not possible, this method should return "`null`".

- `BindingList unify(Term t1, Term t2, BindingList bl)`

  This method attempts to find a binding list that allows for the unification of the two provided terms (which can be constants, variables, or functions), under the constraints of the given binding list. If the two terms can be unified, a new binding list is returned, augmenting the given binding list with any addition bindings needed to unify the two terms. If unification is not possible, this method should return "`null`".

- `BindingList unify(Function f1, Function f2, BindingList bl)`

  This method attempts to find a binding list that allows for the unification of the two provided functions, under the constraints of the given binding list. If the two functions can be unified, a new binding list is returned, augmenting the given binding list with any addition bindings needed to unify the two functions. If unification is not possible, this method should return "`null`".

Note that an empty binding list is not the same as a "`null`" value. For example, two ground terms may be unified without any need for the binding of variables (e.g., "`A`" unifies with "`A`", and "`FatherOf(Dave)`" unifies with "`FatherOf(Dave)`"), and an empty binding list is the appropriate return value for `unify` in that case. The "`null`" return value specifically indicates a *failure* of unification.

The implementations for the first and last of these three methods — for the literal case and the function case — should be relatively compact, as they can both leverage a provided implementation of `unify` that compares lists of terms. This provided version of `unify` can be used to attempt to unify the arguments of a literal or of a function. Thus, the most difficult part of this assignment will involve writing the `unify` method for two terms. This must accurately handle all possible terms: constants, variables, and functions. It must also include an "occur check", disallowing the binding of a variable to a function that contains that variable.

If these three `unify` methods are properly implemented, the `main` method in the provided `Ptwo` class should output correct solutions for provided queries. Note that testing this program will require (1) a plain text file containing facts (positive literals containing only ground terms) in the knowledge base, (2) a plain text file containing rules in the knowledge base, and (3) a positive literal query. In all cases, literals are specified with the predicate *inside* of the parentheses, with arguments separated by whitespace. Thus, the standard format "$IsFatherOf(Dave, Wayne)$" is entered as "`(IsFatherOf Dave Wayne)`". Functions also use this format. For example, "$FatherOf(Dave)$" becomes "`(FatherOf Dave)`". Horn clause rules are entered using a special syntax that begins with the "`DEFRULE`" keyword and includes a rule name. For example:

```
(DEFRULE SiblingDefinition
  (IsParentOf ?child1 ?parent)
  (IsParentOf ?child2 ?parent)
=>
  (IsSiblingOf ?child1 ?child2))
```

In general, the "`DEFRULE`" keyword is followed by the rule name, which is followed by a collection of rule antecedents (i.e., positive literals), which is followed by the implication symbol "=>", which is followed by a single positive literal consequent. The whole rule is wrapped in parentheses. A simple example test case will be provided to you.

The Java™ utility classes that you are required to use are provided in a ZIP archive file called "`PA2.zip`" which is available in the "Resources" section of the class `UCMCROPS` site. These utilities include:

- `Symbol` — This object encodes a simple symbol, such as a constant or a variable. Tools are provided for generating symbols with novel names.
- `Constant` — This object encodes a logical constant.
- `Variable` — This object encodes a logical variable. In this system, all variable names begin with a question mark.
- `FunctionName` — This object encodes the symbolic name of a function.
- `Predicate` — This object encodes the symbolic name of logical predicate.
- `Function` — This object encodes a function instantiation, including the function name and a list of arguments.
- `Term` — This object encodes a term, which is any logical constituent that can refer to an object. Thus, it is either a constant, a variable, or a function.
- `Literal` — This object encodes a non-negated atomic sentence. It other words, it is a predicate applied to a list of terms.
- `Rule` — This object encodes a Horn clause rule, including a rule name, a single consequent (or "head"), and a list of antecedents (or the "tail").
- `Binding` — This object encodes an association between a variable and a value, where each value is a term.
- `BindingList` — This object encodes a list of variable bindings.
- `KnowledgeBase` — This object contains a list of positive literal facts and a list of Horn clause rules. The class includes methods for reading facts and rules from plain text files.
- `BackwardChain` — This object embodies a very simple backward chaining inference engine. It includes a knowledge base, and it allows for backward chaining queries through an "`ask`" method.
- `Ptwo` — This object provides a top-level driver that tests your augmented inference engine. Your code must allow `Ptwo` to produce correct output for a variety of test cases.

The contents of these Java™ utility files will be discussed during a course laboratory session, and inline comments in these files should assist in your understanding of the provided code. Questions are welcome, however, and should be directed to the teaching team.

Your implementation will be evaluated primarily for accuracy, with efficiency being a secondary consideration. Your source code *will* be examined, however, and the readability and style of your implementation will have a substantial influence on how your assignment is evaluated. As a rough rule of thumb, consider the use of good software writing practices as accounting for approximately 10% to 20% of the value of this exercise. Note also that, as discussed in the course

syllabus, submissions that fail to successfully compile under the laboratory Eclipse Java™ IDE (i.e., they produce "build errors") will not be evaluated and will receive *no credit*.

As for all assignments in this class, submitted solutions should reflect the understanding and effort of the individual student making the submission. Not a single line of computer code should be shared between course participants. If there is ever any doubt concerning the propriety of a given interaction, it is the student's responsibility to approach the instructor and clarify the situation *prior* to the submission of work results. Also, helpful conversations with fellow students, or any other person (including members of the teaching team), should be explicitly mentioned in submitted assignments (e.g., in comments in the submitted source code files). Failure to appropriately cite sources is called *plagiarism*, and it will not be tolerated!

The members of the teaching team stand ready to help you with the learning process embodied by this assignment. Please do not hesitate to request their assistance.