Printer-Friendly Version (PDF)

# Recursion

CS236 Project Three

## Directly Interpreting Datalog

We can execute a Datalog program by recursively plugging in all the values of the domain and testing each combination. This is a correct way to interpret Datalog programs; however, as you will see later in Projects 4 & 5, it is not the most efficient way.

Although recursively plugging in all values is not the most efficient implementation of a Datalog interpreter, there are nevertheless some useful things to learn by seeing how to code this direct implementation of the meaning of a Datalog program: (1) The implementation is based on an interesting recursive solution. Writing recursive programs is a skill many find hard to master. Taking on a meaningful challenge to write recursive code and seeing that its execution is "not that bad" will hopefully help. (2) Reading someone else's code and working with it is a skill seldom required of students, but it is a skill often needed by professional programmers. Here's a chance to practice this skill as a student. (3) Finally, after doing projects 4 & 5, you should see that a direct implementation of the semantics of a system is not always the best way to implement the system. More efficient implementations are often possible, if you are knowledgeable enough to find them.

## Project Description

Ninety-Nine percent of the code for directly interpreting 236-Datalog has already been written for you. It is already included in the cs236Bundle that you have used for projects 1 and 2. All the code is in *src/project3* directory. Here is a link to the Project3-API (Javadoc). There you will find the source files `PredicateList.java` and *RuleList.java*. We have removed the body of two methods in the class *PredicateList* which you are to implement:

```
boolean createAllCombinations(int i)
boolean checkToSeeIfTrue()
```

This method substitutes all combinations of constants from the domain for the variables in the predicate list. For every combination of constants *createAllCombinations* calls *checkToSeeIfTrue*. The method *checkToSeeIfTrue* checks to see if the predicate with all constants can be proven true by the facts or the rules. You are to modify this method in the class *RuleList*:

*boolean canProve(Predicate predicate)*

You are to modify it so that it can detect infinite recursion. When your code is working, you are to run the pre-defined test cases and record the time it takes to run each test case. Notice that this exercise is not so much about how to write code but about how to understand, modify, and extend code. To assess your understanding about particular aspects of this project, you are to answer the Project3Questions. Note: these instructions are for the CS linux labs in the basement of TMCB. If you wish to port this to any other environment (e.g. eclipse or windows) you are on your own.

## Implementing the *createAllCombinations* Method

The basic idea of the *createAllCombinations* method is to determine which combinations of constants from the domain, when substituted for each unique variable in the predicate list, make the predicate list true. A single combination of constants is called a *possible solution*. A *possible solution* that makes the predicate list true is called a *solution*. Substituting for "each unique variable" means that when we substitute a constant *c* for a variable *v*, we substitute *c* for all occurrences of *v*. If there exists at least one *solution*, then the *createAllCombinations* method returns true, otherwise it returns false.

Before the *createAllCombinations* method is called, a static array, whose name is *variables*, is initialized with all of the variables in the predicate list. The array *variables* is an array of *Identifier* which is a *Token* whose token type is restricted to the *Id* token type. The *createAllCombinations* method has a single formal parameter, *i*. This parameter is an index into the array *variables* and tells us which variable we are going to make substitutions for.

To create all combinations of constants for all variables in the predicate list, we have to be able to get all constants from the domain. This is provided by a static method in the class *Domain* called `iterator()` which returns an *Iterator*<*Constant*>.

For the variable indexed by the parameter *i*, *variables[i]*, we must substitute the constant we get from the domain for *variables[i]*, wherever it occurs in the predicate list. This is done by calling the method

```
void setVariableToValue(Identifier variable, Constant
value)
```

This method is implemented in the class where the method `createAllCombinations(int i)` is defined, the *PredicateList* class.

Once we have a *possible solution*, we must check to see if it is a *solution*. This is done by calling the method:

```
boolean checkToSeeIfTrue()
```

which is also defined in the *PredicateList* class.

In this design, the *Query* class inherits from the *PredicateList* class. The only restriction is that a query is a predicate list with exactly one predicate. Because of inheritance, the *Query* class inherits the *createAllCombinations* method from *PredicateList*. However, there is a small difference between the semantics of the *createAllCombinations* method for predicate lists and the *createAllCombinations* method for queries. The *createAllCombinations* method for a query only stops when it has considered all *possible solutions*. The *createAllCombinations* method for a predicate list stops as soon as it finds the first *solution*. To simplify the design, we define a single, general *createAllCombinations* method in the *PredicateList* class (which you are to write). If written properly, the *Query* class inherits the *createAllCombinations* method without overriding it. To do this we isolate the differences in the semantics of *createAllCombinations* by defining the method:

```
boolean keepOnGoing(boolean validSolutionFound)
```

The method *keepOnGoing* is declared in the *PredicateList* class and overridden in the *Query* class. This method takes in a value which should be the result of the most recent call to *checkToSeeIfTrue.* The *PredicateList* version of *keepOnGoing* returns true if and only if the previous *possible solution* was not a valid solution (i.e. *checkToSeeIfTrue* was false). The *Query* version of *keepOnGoing* always returns true. Both versions of *keepOnGoing* have been implemented for you. You do not have to implement them, but you do have to call the method appropriately in your implementation of *createAllCombinations*.

## Implementing the *checkToSeeIfTrue* Method

The *checkToSeeIfTrue* method iterates over all the predicates in the predicate list and checks to see if they are true. A predicate is true if it can be proven by the facts or the rules. To prove a predicate *p* true by the facts, issue the following call:

```
Project3.datalogProgram.getFactList().canProve(p)
```

To prove a predicate *p* true by the rules, issue the following call:

```
Project3.datalogProgram.getRuleList().canProve(p)
```

If *p* could be proven true by the facts or the rules then call the method *saveResult()*. In the *PredicateList* class this method does nothing, but in the *Query* class this method is overridden and saves the predicate so it can be printed later.

## Implementing the *canProve* Method

In the *RuleList* class is the method *canProve(Predicate predicate)*. Modify it so that it checks for infinite recursion. There is a HashSet you will need to use called

*previouslySeenPredicates* that holds all predicate instantiations you have previously seen. If predicate *p* is in the HashSet, then return false. Otherwise add *p* to the HashSet. Use the existing code to check to see if there is a rule that can be used to prove *p* true. Finally, remove *p* from the HashSet.

## Compiling Your Code

Ant manages the code production for this project so to compile your code, go to the cs236 directory (where the *build.xml* file is) and type:

```
ant compile3
```

This will compile all of the source files that have been changed since the last compilation. If there are any syntax errors, they will appear on your screen. If there are errors, fix them, and recompile.

## Testing Your Code

We have already written the code that will test what you have written. Use this command to run individual test files:

```
ant run3
```

`Test files are located in test/inputs.` Also, we have automated large tests from a tool called *JUnit*. When you want to run all the tests, go to the cs236 directory again and type:

```
ant testproject3
```

This invokes the prewritten tests. It will take from 2 to 13 minutes. On a slow machine it could take longer. It provides a textual response indicating whether there were any errors, and if so, what they were.

Starting from the cs236 directory, you can find the code for testing Project 3 in *test/test-src*. There you will find a test file for every class in the package *project3*. The error messages will refer to various methods in the test files. If anything goes wrong anywhere in any method of a test file, it will flag that method as having failed. Unfortunately, it will not tell you where in the method it failed

You will find that the main test for Project 3 is in the file *TestProject3.java*. This merely reads inputs from a file and compares the result produced with expected results read in from another file. The input files are found in *test/inputs* and the expected output files are in *test/answers3.*
Because the test cases take so long to execute, you can use comments to eliminate test code you don't want to execute so that it will go faster. If you want to eliminate test

cases for entire classes, go to the file *TestProject3Suite.java* and comment out the lines corresponding to the classes that don't need testing.

If you are so inclined, you can also run the test cases for the lexical analyzer and the parser. This is done by typing either of the following two commands while in the cs236 directory.

```
ant testlex
ant testparser
```

Unless you modify any of the parsing code or lexical analyzer code, you shouldn't have to run these test cases.

For your information, running any of the test cases will automatically recompile any source code or test code.

## Timing the Test Cases

After you have passed all test cases you will need to time them. To do that, go to the cs236 directory and type:

```
ant timetest3
```

If your code is correct, this will return a list of times for each of the test cases in the file `TimeTest3.java`. It will also return a total time. It takes from 20 seconds to 3 minutes to run the test cases. On a slow machine it could take longer.

Later in the semester we will implement Project 3 using relational database technology. This is done in Projects 4 and 5. At the end of Project 5 we will again execute and time the same test cases used here. We will use the timing information to compare the efficiencies of the two implementations.

## Submission and Passoff

Create your project3.jar with:

```
ant jar3
```

This will jar the two files that you edited (*PredicateList.java* & *Rulelist.java*). You then submit the jar as your project3. The TAs will run the '*ant testproject3*' and the '*ant timetest3*' on your code.

**Don't forget to answer the Project 3 Questions !**
**The due date is specified on the class schedule.**

**Note: The two parts are turned in separately.**

**Project 3 can be passed off early, on time or late.**
**The Project3Questions are due on their due date only. Not accepted late.**