CSE 425 Studio: Logic Programming II

These studio exercises are intended to give you further experience with logic programming, in C++. In this studio you will again work in self-selected groups of ~3 people, and will report the results of your work on the studio exercises in an e-mail to the course account. Students who are more familiar with the material are encouraged to partner with those who are less familiar with it, and asking questions of your classmates and professor during the studio sessions is highly encouraged as well. Please record your answers as you work through the following exercises. After you have finished please email your answers to the required exercises, and to any of the enrichment exercises you completed, to the cse425@seas.wustl.edu course account, with subject line "Logic Programming Studio II". The enrichment exercises are optional but are a good way to dig into the material a little deeper, especially if you breeze through the required ones. **Please feel free to use code from your previous studio exercises and/or lab assignments in completing the exercises for this studio (and the next one).**

PART I: REQUIRED EXERCISES

1. Form a team of ~3 people of your choice and write down the names of the team members as the answer to this first exercise.

2. Create a new Visual Studio 2013 project for this studio (for example, named **`LogicStudioII`** or something similar that identifies which studio this is for). Modify the signature of the main function in your project's main source file (e.g., **`LogicStudioII.cpp`**) so that it matches the standard portable (e.g., between Windows and Linux) main function signature (and variable names) for a C++ program: **`int main (int argc, char * argv[])`**. Also copy over the source and header files for your parser from the previous into the appropriate directory for this project, and add them to this project.

Extend (or as needed modify) your recursive descent parser from the previous studio, so that it can recognize conjunctions of logic statements as well as individual logic statements, according to the following updated version of the EBNF production for logic statements from last time:

predicate ➔ expression LESSTHAN expression

logic_statement ➔ predicate {AND predicate}

where the AND token is "^"

Whenever the parser detects a valid (simple or conjunctive) logic statement it should print it on its own line, to **`cout`**. In your main function, construct an input stream object (e.g., from a string or file) with a mixture of valid (simple and conjunctive) and invalid logic statements, and repeatedly call your parser over the stream until it reaches the end. Build and run your program, and as the answer to this exercise please show your code and the output from one run of your program over that stream.

3. Add semantic processing for conjunctive logic statements, so that whenever it recognizes a logic statement it "ands together" the results of evaluating the simple logic statements in the conjunction, and prints out the result (in addition to printing out the statement itself). The entire conjunctive logic statement should evaluate to true if and only if all of the simple logic statements within it evaluate to true (and otherwise the entire conjunctive logic statement should evaluate to false). For example, the following conjunction would evaluate to true if the value of x were between 8 and 10 inclusive, and would evaluate to false otherwise:

x < 11 ^ 7 < x

Build your program and run it over the stream from the previous exercise, and as the answer to this exercise please show your code and the output from one run of your program over that stream.

4. Extend your recursive descent parser from the previous studio, so that it can recognize basic Horn clauses, according to the following productions:

horn_clause ➔ head SEPARATOR body

head ➔ predicate

body ➔ logic_statement

Whenever the parser recognizes a valid Horn clause, it should print it out. Add well-formed and badly-formed Horn clauses to the stream of tokens, build and run your program, and as the answer to this exercise please show your code and the output from one run of your program over that stream.

5. Add basic semantic processing for Horn clauses as follows: whenever the parser recognizes a well-formed **predicate**, it should determine whether the predicate evaluates to true or false. Similarly, whenever the parser recognizes a well-formed logic statement it should determine whether the entire statement evaluates to true or false (i.e., it is true if all of its component predicates evaluate to true, and otherwise is false). Finally, whenever the parser recognizes an entire Horn clause, it should determine whether or not the entire Horn clause evaluates to true or false, and should print out true or false accordingly: the Horn clause is true if either the head evaluates to true or the body evaluates to false (if the head evaluates to false and the body evaluates to true, then the entire Horn clause evaluates to false).

Build your program and run it over the stream from the previous exercise, and as the answer to this exercise please show your code and the output from one run of your program over that stream.

PART II: ENRICHMENT EXERCISES (optional, feel free to skip or to try variations that interest you)

6. If you have not done so already, add support for truncated evaluation of conjunctive logic statements. That is, whenever a simple logic statement evaluates to false, semantic processing stops and returns false without evaluating any remaining simple logic statements that are part of the conjunction. As the answer to this exercise please show your code, describe briefly how you implemented that feature.

7. Extend your solution for exercise 5 to allow conjunctive as well as simple logic statements as the head of a Horn clause. As the answer to this exercise please show the grammar productions that would need to be added or changed to achieve this, show your code, describe briefly how you implemented that feature.