CSE 425 Studio: Logic Programming I

These studio exercises are intended to give you 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 I". 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 **`LogicStudioI`** 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., **`LogicStudioI.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[])`**.

Write a simple recursive descent parser that can parse basic assignment statements according to the following syntax (in EBNF) from a whitespace-delimited stream of (C++ style) **`string`** tokens:

assignment_statement ➔ VARIABLE ASSIGN expression

expression ➔ VALUE | VARIABLE

Where a VARIABLE token consists of all alphabetic characters (uppercase or lowercase), an ASSIGN token is a single equal sign ("="), and a VALUE token consists of all numeric characters.

For example, a valid assignment statement in that syntax would be:

x = 15

While the following statements would be invalid:

57 = x

12 = 12

Whenever the parser detects a valid assignment 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 and invalid statements in that syntax, 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 a stream that contains both well-formed and badly-formed statements.

3. Add semantic processing to your program, so that the program remembers (e.g., in a **`set`** or **`map`**) the name of each variable that it sees on the left hand side of the ASSIGN token in any syntactically valid statement. Modify your program so that if a variable appears on the right hand side of a statement, but has not been seen previously on the left hand side of an earlier valid statement, the statement being processed is considered (semantically) invalid and is simply skipped rather than printing it out.

Build your program and run it over a stream containing a mix of (syntactically and semantically) valid and (syntactically and/or semantically) invalid statements, 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 the semantic processing from the previous exercise, so that it remembers the value as well as the name of each variable it sees in any valid (syntactically well-formed) statement. Whenever a variable appears on the left hand side of a valid statement, the program should give it the value of the expression that appears on the right hand side of the ASSIGN token, as follows: if a variable appears on the right hand side, the program should use its most recently stored value; if a VALUE token appears on the right hand side, the program should reduce it to its corresponding decimal integer value.

Whenever the program prints out a valid statement, it also should print out the value that is being assigned to the variable that appears on the left hand side. Build and run your program, and as the result of this exercise show the code you added for it and the output of one run of your program over the token stream.

5. Extend your approach to support parsing of statements that perform a logical comparison based on the following additional production for a statement:

logic_statement ➔ expression LESSTHAN expression

Whenever the parser sees a (syntactically) well-formed logic statement, it should print it to **`cout`**, along with the (syntactically and semantically) well-formed assignment statements it sees. Add (syntactically) well-formed and badly-formed logic statements to the stream of tokens, build and run your program, and as the answer to this exercise please show the new code you wrote for it and the output from one run of your program over the updated token stream.

6. Add semantic processing for the logic statements that were added the previous exercise, so that the program (1) computes values of the expressions on either side of the LESSTHAN token, (2) determines whether or not the value to the left of the LESSTHAN token is less than the value to the right; and then (3) in addition to printing the statement, prints out **`"true"`** if it is, or **`"false"`** if it is not.

Build and run your program, and as the result of this exercise show the code you added for it and the output of one run of your program over the token stream.

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

7. Add support for parsing and evaluating additional logical operators (greater than, equal, not equal, etc.) to your program. Add statements with valid and invalid uses of those operators to the stream of tokens, and build and run your program. As the answer to this exercise please show your code, describe briefly what you did, and show the output from one run of your program over the updated token stream.

8. Add support for parsing and evaluating addition operators (and/or other arithmetic operators) within expressions. Add statements with valid and invalid uses of those expressions to the stream of tokens, and build and run your program. As the answer to this exercise please show your code, describe briefly what you did, and show the output from one run of your program over the updated token stream.