CSE 425 Studio: Syntax I

These studio exercises and the next ones are intended to acquaint you with basic ideas and techniques for recognizing and manipulating lexical structures, which you will implement in C++. There is also a new (as of C++11) regular expression library **`<regex>`** with which you may want to become familiar, in part because (for assigned labs) it automates much of what you'll implement manually (for experience) today.

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 sent to the course account cse425@seas.wustl.edu. Students who are more familiar with the material are encouraged to partner with and help those who are less familiar with it, and asking questions of your classmates, professor and teaching assistant during the studio sessions is highly encouraged as well. Links on the course web page to helpful web sites also may be useful as resources.

Please record your answers as you work through the following exercises. After you have finished please send your answers to the required exercises, and to any of the enrichment exercises you completed, in an e-mail to cse425@seas.wustl.edu with "Syntax Studio I" in the subject line. The enrichment exercise is optional but lets you dig into the material a bit deeper, especially if you breeze through the required ones.

PART I: REQUIRED EXERCISES

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

2. Create a new Visual Studio 2013 C++ Console Application project to develop and compile your code, as in the previous studio. Write a basic function that can take a (C++ style rather than C style – please ask if you don't understand the difference) string of characters and return a list of string tokens by explicitly identifying whitespace characters that separate the tokens (**hint:** using the C++ istringstream class can help with this).

Write a simple C++ program that can read in the contents of a file and use each line of the file (delimited by the **`\n`** newline character) to form a string of characters, pass it to the function you just wrote, and then print out a line containing the sequence of tokens that the function produced. Create a file with several lines of space-delimited tokens and run your program using that file. As the answer to this exercise, show your program, the contents of the file, and the output your program produced.

3. Write a scanning function that will take a string token and return a Boolean value that is true if and only if the token is the string **`"+"`**. Write a similar function that will take a string token and return a Boolean value that is true if and only if the token is entirely numeric, containing only decimal digits (**`0`** through **`9`**), and either is exactly **`"0"`** or begins with a non-zero decimal digit (**`1`** through **`9`**). Modify your program from the previous exercise so that it only prints out tokens that are matched by either of those two functions. Update your file from the previous exercise so that some lines have only tokens that those functions will match, some lines have only tokens that those functions will not match, and some lines have a mixture of the two. Compile and run your program using the updated file, and as the answer to this exercise, show your updated program, your updated file, and the output your program produced.

4. Using the terminal symbols **+** to represent an addition operator and **NUMBER** to represent a numeric token that a scanning function can recognize (as in the previous exercise), write a grammar in BNF for a prefix addition expression, which is defined either as a **NUMBER** or (recursively) as a **+** symbol followed by two prefix addition expressions. As the answer to this exercise, show BNF notation for the grammar you developed, give a few examples of token sequences that would or would not match that grammar, and for each example explain briefly why it would or would not.

5. Develop a C++ function that can take a list of tokens and use the scanning functions from exercise 3 above to recognize whether or not that sequence of tokens begins with a valid prefix addition expression according to the grammar from the previous exercise. The function should return a list that contains two lists of tokens (either of which may be empty): the tokens that make up the valid prefix addition expression and then the tokens that make up the rest of the list. Call this function with lists built from the different valid and invalid token sequences from the previous exercise, print out the different lists that it produces, and confirm that it correctly differentiates well-formed from badly-formed prefix addition expressions. As the answer to this exercise, show the function you developed and describe briefly the cases you tried running it on and the results of each case.

6. Update the main program from exercise 3 above so that it repeatedly tries to find the longest matching (well-formed) prefix addition expression it can in each line of the input file, and each time it finds (well-formed, non-empty) prefix addition expression it should print it out on its own line (so it's easy to identify all of the valid expressions the program matched).

The program should first try to match a prefix addition expression starting with the first token in the line: if it succeeds it should try again starting with the first token of the rest of the line, or if it fails it should move to the next token after the one at which it failed and try again.

Update your input file from the previous exercises so that it contains some lines that contain only a single valid prefix addition expression (and nothing else), some lines that have no valid prefix addition expressions in them at all, and some lines that have different combinations of valid and invalid token sequences for prefix addition expressions (including lines that have multiple valid sequences). Compile and run your program using the updated input file, and as the answer to this exercise, show both the input file and the output your updated program produced.

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

7. Extend exercises 4 through 6 above to add a prefix multiplication operator (given by the symbol *****) in addition to the prefix addition operator. As the answer to this exercise show the resulting grammar and code, and describe briefly your observations about them (especially regarding questions of ambiguity, precedence, and/or associativity which we'll discuss further in the next studio – you may want to review the slides for next time which are posted on the course web site and/or read ahead in the text book).