CSE 425 Studio: Syntax II

These studio exercises are intended to expand your experience with ideas and techniques for recognizing and manipulating lexical structures, again implemented 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 sent to the cse425@seas.wustl.edu course e-mail account. 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 encouraged as well. Please also use the links on the course web page as resources.

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 cse425@seas.wustl.edu with "Syntax Studio II" in the subject line. The enrichment exercise is optional but lets you dig into the material a little 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 compile your code as in the previous studio. Develop several new scanning functions to recognize (respectively) a **`LABEL`** token that consists entirely of (uppercase or lowercase) alphabetic characters, a **`LEFTPAREN`** token that is exactly the string **`"(`**, a **`RIGHTPAREN`** token that is exactly the string **`")`**, and a **`COMMA`** token that is exactly the string **`",`**.

Using the functions you just wrote and also the function from the previous studio exercises that can recognize a **`NUMBER`** token that consists entirely of decimal digit characters, write a simple C++ program that can read in the contents of a file and for each line of the file (delimited by the **`\n`** newline character) classify each whitespace-delimited substring of the line using those scanning functions.

On a separate line for each line of the input file, the program should print out all of the input line's tokens that are valid **`NUMBER`**, **`LABEL`**, **`LEFTPAREN`**, **`RIGHTPAREN`**, or **`COMMA`** tokens (according to your scanning functions). For example, if an input line contained **`( echo hello, world )`** the output for that line would be **`( echo world )`** since **`"hello,"`** would be seen as a single invalid token rather than two valid (**`LABEL`** and **`COMMA`**) tokens.

Create a file with several lines of space-delimited tokens (including different combinations of valid and invalid tokens according to the definitions above) and run your program using that file. As the answer to this exercise, show your code, the contents of the file, and the output your program produced.

3. Write a simple recursive descent parsing function that will recognize the following sequence of tokens: **LABEL LEFTPAREN RIGHTPAREN** and when it recognizes them will print out their corresponding strings on a single line. Modify your main program from the previous exercise so it calls this function exactly once per line of the input file (passing the function the entire token sequence for that line).

Update the input file so that some of the lines begin with a valid token sequence and others do not, e.g., the line **hello ( ) world** should produce output **hello ( )** but no output should be produced for the similar line **( ) hello ( ) world**. Run your program with the updated input file, and as the answer to this exercise, show your code, the contents of the file, and the output your program produced.

4. Modify your parsing function from the previous exercise (one good way to do this is to add another recursive descent parsing function that the original one now will call) so that it can recognize **function** expressions according to the following context-free grammar (shown here in BNF format):

function → LABEL LEFTPAREN arg RIGHTPAREN

arg → LABEL | NUMBER

Update the input file so that some of the lines begin with a valid token sequence and others do not (according to this new definition of the grammar) and run your program with it. As the answer to this exercise, show your code, the contents of the file, and the output your program produced.

5.Modify your parsing function from the previous exercise so that it can recognize **function** expressions according to the following revised context-free grammar (shown here in EBNF format where the square brackets indicate an optional sequence of elements):

function → LABEL LEFTPAREN [arg] RIGHTPAREN

arg → LABEL | NUMBER

Update the input file so that some of the lines begin with a valid token sequence and others do not (according to this new definition of the grammar) and run your program with it. As the answer to this exercise, show your code, the contents of the file, and the output your program produced.

6.Modify your parsing function from the previous exercise so that it can recognize **`function`** expressions according to the following revised context-free grammar (shown here in EBNF format where the curly braces indicate zero or more repetitions of a sequence of elements):

`function → LABEL LEFTPAREN [args] RIGHTPAREN`

`args → arg {COMMA arg}`

`arg → LABEL | NUMBER`

Update the input file so that some of the lines begin with a valid token sequence and others do not (according to this new definition of the grammar) and run your program with it. As the answer to this exercise, show your code, the contents of the file, and the output your program produced.

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

7. Extend your code from the previous exercises so that it can isolate tokens without relying on whitespace to delimit them. That is, each scanning function should be able to identify the longest matching prefix of a string of characters given to it, and the scanning portion of the program then should be able to separate that prefix as a distinct token, from the remaining sequence of characters which it should then continue to scan. If the program cannot match a token beginning with the first character, it should try again at the next character, until it finally finds a position at which it can isolate a valid token. As with valid tokens the program should isolate the sequence of such skipped non-whitespace characters as a distinct (invalid) token. For example, the string **`"0x8765bead(*&^ ^&*()"`** would be tokenized as **`"0" "x" "8765" "bead" "(" "*&^" "^&*" "(" ")"`**.

Create a file with several lines tokens (including different combinations of valid and invalid and of space-delimited and non-space-delimited tokens) and run your program using that file. As the answer to this exercise, show your code, the contents of the file, and the output your program produced.