Lab 2b – Matching Parentheses

Learning Goals

1) Develop your ability to program in C and use a Unix shell.
2) Develop your understanding of the stack data structure.

Your Task

In any well-formed document, program, or math expression, the parentheses and brackets match. Each opening parenthesis has a matching closing parenthesis, and so forth.

Your task is to write a program that checks if all of the parentheses (), brackets [], and curly brackets {} in a document match. The program reads from standard input. The input can contain multiple lines and arbitrary text between and among the opening and closing grouping symbols.

An example legal document is:

```
 [ Hello { } ( [ 5+6])]
```

An example document with a mismatching brace is:

```
 ( [ { ] } )
```

Note: We will ignore the problem of commented or quoted braces for now.

Design

A stack is a very good data structure for solving this type of problem. Stacks have the property that the last element pushed onto the stack will be the first element popped off of the stack. (LIFO – Last In First Out)

In our case, if the last opening grouping symbol we saw was a parenthesis, then the only current valid closing grouping symbol is a parenthesis. Or, alternatively, we might see a different opening grouping symbol which would then need to be closed before our parenthesis can close.

Remember: getchar() returns an int that contains the next character from standard input (stdin). The value EOF indicates when the end of the file have been reached.

Starting Point

In “Part A” of this lab, you implemented a stack. If it isn’t perfect yet, finish that first!

Then, do the following:

1) Make a copy of your “charStack.c” and name that copy “parenCheck.c”
2) Delete the contents of main()
3) Implement main() to check that all grouping symbols match. You should utilize push() and pop(); printStack() is useful for debugging. You do not need to create

any additional functions. See the sample output for details on how the various error conditions should be reported.

Sample Outputs

1) If the input is well formed, the program should output “Well formatted input”.

```
$ ./paren < valid2.in
Well formatted input.
```

2) If an invalid character is found, the program must report the line number and position of the error. The program exits after hitting the error.

```
$ ./paren < invalid1.in
Line 2, Char 28: Found ), expected }

$ ./paren < invalid4.in
Line 1, Char 27: Found }, expected ]
```

3) If the stack is full, the program must report the error and exit.

```
$ ./paren < invalid3.in
Error: Stack Full!
```

4) If the stack is empty and a closing token is read, an error must also be reported.

```
$ ./paren < invalid2.in
Line 3, Char 12: Found ). No matching character.
```

5) If the end of the input is reached and the stack is not empty, an error must be reported.

```
$ ./paren < invalid5.in
Error: Expecting ], found end of input.
```

Grading

This assignment will be scored out of 100 points:

50 pts – Invalid test cases all correctly identify the case as invalid. (1 pt each)

10 pts – Message for invalid cases accurately reports the line, position, and expected characters.

30 pts – Valid test cases all correctly execute.

10 pts – Output is free from spurious content. (Execution should only produce 1 result and then exit. A string of results should not be printed.)

Extra 5 pts: Ignore braces inside of double quotes.