ECE220: Computer Systems & Programming **Fall 2022 ZJUI**
Lab 2 **Tuesday 20 September**

Matching and Balancing Expressions with a Stack*

In today's lab, your group must write a program to check whether a line of input contains balanced parentheses, brackets, and braces. The goal of this exercise is to help you make use of subroutines and a stack to solve a problem, and to introduce you to the idea of lexing: translating raw input into tokens that represent patterns in the input. MP1 only requires you to write subroutines, but understanding how to use a stack will help you with MP3. Lexing is for fun, but the code provide also gives you examples of look-up tables, which you may also want to use in MP1.

Begin by updating your Git repository to obtain the **`lab2`** subdirectory. The directory contains two LC-3 assembly files. The **`example.asm`** file allows you to type input and translates each token into a description of the token, allowing you to see how the tokenization works. The **`lab2.asm`** file is the starting point for your code.

You must write the main program in **`lab2.asm`**. The subroutine needed for lexing has been provided along with a starting point for your stack and all of the message strings that you need to complete your code. My solution took 30 instructions.

The Task

The user will enter text using the keyboard. Your program must check whether parentheses—(and), brackets—[and], and braces—{ and } are matched and balanced on each line of the user's text. For example,

```
( [ This { line is } valid ] -- ( ) the [ { symbols } match and ] balance. )
[ does not match }
( [ The parentheses are not balanced here. ]
Closing before opening is also not allowed. }
```

Token	Meaning
-4	other character
-3	}
-2	]
-1	)
0	end of line (<Enter>)
1	(
2	[
3	{

To help you write your program, a subroutine called **`TOKENIZE`** has been provided. The **`TOKENIZE`** subroutine reads a keystroke from the user and returns a token in **`R0`**. The token has one of eight possible values, as shown to the right. Your main program must be an infinite loop that reads a token and processes it, possibly printing a message in response to each token.

To determine whether a line is valid or invalid, your program should use a stack (the stack base has already been loaded into **`R6`** for you). Each token read from the user must be processed as follows:

- Characters not corresponding to opening and closing symbols (token value -4) should be ignored (read another token).
- When the user presses an opening symbol (positive token values), push the token on to the stack and go back to read another token.

* I took the idea for this lab from a similar lab in the regular section of ECE 220, but rewrote everything from scratch.

- When the user presses a closing symbol (token values -3 to -1), first check whether the stack is empty. If so, print the error message given by the string at the label **`NOOPEN`**. Otherwise, check whether the closing symbol matches the opening symbol on top of the stack. If so, pop the matching opening symbol off of the stack. In the case of a mismatch, print the error message at the label **`MISMAT`** and reset the stack to the base. Regardless of the outcome, continue reading tokens.
- Finally, if the user presses <Enter> (token 0), your program must check whether the line is balanced. Fortunately, doing so is easy: for a balanced line, the stack is empty. So check whether the stack is empty. If so, print the message at the label **`VALID`**. If not, print the message at the label **`INVALID`** and reset the stack to the base for the next line. Then go back to reading tokens.

You can use the example lines from the first page to test your program, or you can test it by typing a few lines yourself.

If you finish early, you may want to read the **`TOKENIZE`** subroutine and/or the **`UNPACK`** subroutine to see how they work. Both use look-up tables and preserve most of the registers (the registers are callee-saved), which may give you some ideas for your MP1 implementation.