Sanjana Dodley

CODE IMPLEMENTATION OVERVIEW:
The original input string is scanned as a parameter to TKCreate where it stores the input string as a parameter of TokenizerT. Then, in TKGetNextToken, a CurrToken pointer is made to traverse along the string to identify tokens through various switch statements and if/else branching. A CurrToken also includes a integer called currState that keeps track of what state the token is currently at (decimal, integer, word, etc) as it's being scanned through the conditional branches. The organization of such states is done using enum.

STRUCTS AND ENUM:

- **enum States** - specifies the various states that a token can traverse through (decimal, integer, word, illegal, hex, octal, etc.)
- **TokenizerT** - holds the input argument in a variable called inputString. The variable int pos keeps track of where the pointer is.
- **CurrToken** - will hold the current token and the state, which is a small portion of the original string(int TokenizerT)

FUNCTIONS:

- **TokenizerT * TKCreate (char*ts)**- creates the TokenizerT from the input string
- **CurrToken * Construct ()**- is a basic constructor method for the CurrToken struct.
- **int dataType(char s)** - return an integer corresponding to the correct state (declared in enum States) of the next character when traversing TokenizerT.
    - **BIG O** - makes at most 16 comparisons inside this function. This is because it is a series of if else statements and in each if statement there are multiple boolean expressions.
- **int isLegal(CurrToken * token, int nextData, char next)** - utilizing the information of the current character and next character from traversal, this method appends each token and changes states dynamically till it reaches a split point or delimiter.
    - **BIG O** - takes advantage of switch tables to get about O(1) run time.
- **char* TKGetNextToken (TokenizerT * tk)** - TKGetNextToken returns the next token from the token stream as a character string. This string is then passed to isLegal to identify its state, and then returned to TKGetNextToken where it is passed to the printToken method where it is then printed with the correct identifier.
- **TK Destroy(TokenizerT *ts)** - memory allocated for TokenizerT is freed here, and then called by the main method at the end.
    - **BIG O** - since there is only 1 TokenizerT struct throughout the entire program, this function will just free that struct. O(1) run time

ALGORITHM/LOGIC OVERVIEW

- When scanning input, the numbers starting with 0 were considered octal until turned to hex (indicated by x/X), decimal integer (int>7), or decimal (presence of '.'). From there, the number would traverse through various conditionals to find the proper final state. Similarly, if the number was scanned and wasn't 0, it was given an initial state of integer, an operator was given an initial

state of operator, and an alpha character was given an initial state of string. All others were considered invalid.

NOTE ON IDENTIFYING TYPES:

- This code splits tokens at escape characters, as well as bad tokens.
- single character alphanumeric bad tokens are printed out in hex, otherwise in invalid.
- the code follows the fsm, and when a bad token is reached, the fsm traversal is restarted. (this is why strings would break into each character being a bad token, and decimal values with more than one '.' break accordingly)
- While the test cases provide more detail on this; listed below are few examples:
    - 990x123
        - integer: 990
        - [0x78]
        - integer: 123
    - 0.3.3.3.3
        - float: 0.3
        - invalid: '.'
        - float: 3.3
        - invalid: '.'
        - integer: 3
    - "hi 12.3e 3.4"
        - [0x68]
        - [0x69]
        - invalid: 12.3e
        - float: 3.4