Derek Schatel
RUID: 032004123

Tokenizer.c Design Document

For this project my goal was to implement a robust tokenizer that would not only split a command line string into individual tokens but would also identify the type of token (Octal, Hexadecimal, Decimal, Float, Zero, or Malformed/Invalid). I implemented it as follows:

Algorithm:

The program creates a new TokenizerT object based on a command line input string. The Tokenizer makes a copy of the input string so as to not make any changes to the original. It also contains two pointers: a “current” pointer that is used to iterate through every character in the string, and a “tokStart” point which is used to indicate the beginning of the new token, and is reset to the current character being analyzed every time the TKGetNextToken function is called. The struct also utilizes a tokLength counter to mark the length of a token, as well as enums to specify the current “type” of token and what state the Finite State Machine (FSM) is in.

TKGetNextToken increments through fullstring using the “current” pointer and analyzes each character as it goes. It uses a series of finite state machines to traverse the string and determine what a token is and whether it is invalid. This way, the program only needs to go through the string once, in linear time. The state machine only ends iterating through a future token once it hits an EndState, denoted by an enumeration type used to hold the state of the FSM. An EndState is reached when a valid token is ended by whitespace or a character that would not belong in a token of the type the FSM is currently evaluating.

Once the FSM finishes parsing a token, it creates a substring of that token using strncpy (denoting the length of the string with another struct variable, tokLength). It then returns the newly created token. I used strncpy rather than strncat because strncat would need to iterate through the whole string in order to concatenate characters on the end, so strncpy seemed like it would be more time efficient so long as I made sure to null terminate the newly created strings.

Some Notes:

Rather than returning malformed data when a valid Float or Hexadecimal is terminated prematurely be an invalid character, I “walked back” the struct variables in order to return a valid token followed by invalid tokens. For example, an input of “345.F”, instead of returning as a malformed Float, instead returns a valid Decimal “345” followed by invalid tokens “.” and “F”.

Invalid tokens are printed out as individual characters in their hexadecimal format.