README.pdf for tokenizer.c

Essential function of the tokenizer program:

tokenizer is a C program whose function is to accept a single string argument to its main function and return each of the the tokens stored within this token stream. The tokens are read from the left-most character in the stream until the null terminator. Importantly, these tokens are NOT separated by a delimiter; rather they are to be separated based upon each token type's unique form (ex. a word token would end at a non-alphanumeric character). However, if a white space is found, this will be considered a delimiter for the next token. Each of the token types are listed below.

Token Types:

Word: A word is defined as a character followed by any number of alphanumeric characters
Decimal: A decimal is defined by any number of digit characters 0 through 9
Hexadecimal: A hexadecimal is defined by a 0x or 0X following any number of hexadecimal characters 0 through 9, a through f or A through F.
Octal: An octal character is defined as a 0 character followed by any number of octal characters 0 through 7. *We will discuss later an ambiguity in these definitions, as seen from the definitions above 003 may either be an octal or a decimal.
Float: A sequence of one or more digit characters 0 through 9 followed by a '.' followed by one or more digit characters 0 through 9. Optionally, these may include the scientific notation character 'e' followed by an optional '-' followed by one or more digit characters 0 through 9.
C-Operator: Any sequence of characters that form a C-Operator defined in the list C-Operators below. *We will discuss later special cases where certain characters may be found in multiple operators.
Comments: There are two types of comments: single line comments, and block comments. Single line comments are the characters // followed by any sequence of characters, including white space, until the first newline (\n) character. Block comments are the sequence of characters “/*” followed by any sequence of characters, including white space, until the appearance of the sequence of characters “*/”. *We will discuss later the special case if these comments do not meet their terminating conditions
Quoted Strings: A double-quoted string is defined as a “ character followed any number of any other characters including white space up until the next “ character. The quotes and all of the contents inside are determined to be a double quoted string. The same is true for single quoted strings except they are identified by the character '. *Special cases where these quoted strings are not properly terminated will be addressed later.
Bad Tokens: Bad tokens are any token (that is sequence of characters not including white space) that is not defined within this program.

Special Exceptions:

During the coding of this program we encountered several special exceptions that arose due to ambiguities when determining how to specify exactly between the multiple token types.

Octal/Decimal tokens and leading 0's: As we know from the above definitions, octal tokens are tokens that start with 0 and have any number of octal digits following. Decimals are the same except they do not require leading 0's and may have any decimal digits following. So then how can we determine if numbers such as 003 are decimal value 3 or octal value 03? We resolved this ambiguity in a way that we believed to be the most effective trade-off. Leading 0's serve no purpose in decimal, whereas they may serve a purpose in octal. For example if you were representing bits in octal the leading 0's may be

useful in representing the leading bits, in say some address, that are 0. So octal maintains its original definition. Whereas decimal's definition was modified so that it *must* start with a non-zero digit unless it is the decimal value 0 itself, or if it is a float whose one's (and also higher) places are 0. For example 0 is the decimal value 0, 0.1 is a legal float, and 00 would be the octal value 0. It follows then that 003 would be octal, but 009 would be octal 00 followed by decimal 9. The provided test cases will demonstrate clearer this functionality.

C-Operators varying lengths: Since some C-operators are certain operators appended to others (for example the += operator could also be read as a + operator and an = operator) we decided to find the *longest* possible C-operator located within the token stream. This, we believed, to be the most effective trade-off since, if we do not search for the longest possible operator, the operator += would never be identified at all. It would have always been interpreted as a + and an =. We felt this was enough justification to search for the longest operator possible in a given stream (unless they are separated by white-space). How this functionality effects the identification of certain tokens will be explored thoroughly in our test cases.

Comments: We decided to not simply identify the character sequences “//”, “/*”, and “*/” as comment identifiers, but also identify the following text in the token stream as a comment as well. For example, if our tokenizer comes across a “//” in the token stream, it willl interpret all following characters and white-space as part of the comment token until it reaches a new line character ('\n') (The new line character can be inserted into the terminal window by pressing shift-enter). So, for example, if the tokenizer reads //comment\n word, it will return //comment as a comment type token and word as a word type token. The same functionality was added for block comments except, rather than terminate at a new line, they will terminate when they reach the sequence of characters “*/”. The most interesting part of this functionality is that our tokenizer can interpret multiple lines to be a single block comment. For instance the tokenizer will read /*block\ncomment*/ as a single, two line, block comment.

Non-terminated Comments: If a comment never meets its terminating condition in the token stream, for example if after a “/*” the tokenizer never finds a “*/”, then everything up until the null terminator '\0' will be interpreted as a comment. This was meant to mimic the way how, in actual C programming, if you never terminate your block comment with a */, the entirety of your code from the /* will be read as a comment.

Non-terminated Quotes: If a double or single quoted string never finds its terminating quote symbol (for example if the tokenizer reads “string not closed) it will read up until the null terminator '\0' and return everything from the first single or double quote as a Bad Token type. This was meant to mimic the way how, in actual C programming, if you forget to close your string using a “ it will interpret all subsequent text in the code as part of that string.

Additional notes for the user:

Delimiters are not required but bear in mind that some tokens may not tokenize how you expect them. For example abc123 will be read as a word abc123 and not as a word abc and decimal 123. Please investigate our definitions further to see how our tokenizer will tokenize your token stream if you choose not to separate them using white space.

Also, bear in mind that only one argument may be passed to tokenizer, so be sure to quote your entire token stream to be read as a single argument. For example use tokenizer “word word word” to pass all 3 words as a single argument. Passing the wrong number of args will return an error.

Finally if you choose to test the double quoted string functionality please be sure to escape your double quotes so they are not read as the end of your string argument. For example using tokenizer “\“Test Quotes\”” will return “Test Quotes” as a double quoted string type.

Thank you for using tokenizer.