Joseph A. Boyle

Given an input string, the tokenizer will print each token on a separate line, in the format of: *Token_Type: Token_Text*. In effect, outputs will look like:

| **Input:** "0x44  0123  123  1.23  " |
| --- |
| Hexadecimal: 0x44<br>Octal: 0123<br>Decimal: 123<br>Float: 1.23 |

## Program Functionality (How it works)

The program essentially splits the string into smaller strings (ie: tokens), by defining a token as a sequence of characters that have an ASCII value greater than 32 (ASCII 32 is a space).
**TKGetNextToken** works as follows:
**Input:** A TokenizerT struct, which has a string *original* and integer *currentPos*

1. Starting at *currentPos*, loop through the string until encounter a character that is greater than 32.
2. Mark this position as *beginningIndex*.
3. Continue looping through the string until encounter a character <= 32, or end of string (whichever occurs first).
4. Create a substring, starting at *original*[*beginningIndex*], that is (*i* - *beginningIndex*) characters.
5. Copy over the contents between *beginningIndex* and *i* into the substring, and return it.

The program will pull a token from **TKGetNextToken** and, assuming it's not zero, pass it to **processToken**. **processToken** is a recursive function that will handle all of the formatted printing described above. The function works as follows:
**Input:** A character array representing a token, *token*.

1. Call **TKIdentifyToken**(*token*) (described in the next paragraph).
2. If the result is positive, the result is describing the index at which the token is invalid
    a. Call **processToken** on the left side of the described index
    b. Print [Err] [hex], where hex is the hex representation of *token*[index]
    c. Call **processToken** on the right side of the described index
3. Otherwise, print out the type's name followed by the token's text.

The program uses a verification model for purposes of maintainability, as adding more types is a matter of adding additional type verifications. Modifying existing type definitions doesn't interfere with the type definitions of an existing type. If we successfully verify a type, we return the enum constant for that type, all of which are less than zero. Instead of verifying every type, we make

educated guesses as to what type the token is and then verify to save on computations. In effect, our guesses look as such:
**Input:** A character array representing a token, *token*.

1. If the first character of *token* is 0
    a. If the second character of *token* is x or X, **verifyHex**(*token*).
    b. Otherwise, If the second character of token is a decimal, **verifyFloat**(*token*).
    c. Otherwise, **verifyOctal**(*token*).
2. Otherwise, **verifyDecimal**(*token*).
    a. If the token is not a decimal, check if the token contains a decimal, e, or E
        i. If so, **verifyFloat**(*token*)
    b. Otherwise, **verifyDecimal**(*token*).

Each call to the verification method will yield either -1, which signals that the token is of the indicated type, or a positive (or zero) integer, which is the first index of *token* which makes the token not a valid instance of the indicated type. Verification works by iterating over the character array of a token until a discrepancy is met. For example, the verification of a hexadecimal token looks like:
**Input:** A character array representing a token, *token*.

1. If the first character isn't 0, return 0 (*token*[0] is the invalid character).
2. If the second character isn't either x or X, return 1 (*token*[1] is the invalid character).
3. Let *i* = 2, loop through until the end of the string:
    a. If *token*[*i*] isn't a digit, or a-f/A-F, return *i* (*token*[*i*] is the invalid character).
4. Return -1 (this token is valid hex).

## Valid & Invalid Sequences

If we encounter an input, 0xl4, we expect the following output:
**Decimal: 0**
**[Err] [0x78]**
**[Err] [0x6c]**
**Decimal: 4**

This occurs since l is an invalid character. After interpretting it as an error, we are left to interpret 0x and 4. 0x by itself is invalid, as there must be a character to follow the x - 0x0 is valid, but 0x is not. Thus, we find that the left side, 0 is a valid decimal, and x is an error sequence. 4 by itself is also a valid decimal, and thus our output looks as above. Considering this input: 1.2.3.4, we expect the following output:
**Float: 1.2**
**[Err] [0x2e]**
**Float: 3.4**

Since 1.2 is valid, but 1.2. is not, 1.2 we separate our input at the second decimal, and thus we now have two tokens: 1.2 and 3.4, while . is an error. The left half is a valid float, as is the right half.

## Program Decisions

As previously mentioned, the verification model is used for maintainability purposes. We make educated guesses as to what the token will be to save on computations: if we were to verify every type until we found one that worked, we'd expect a worst case of O(5N) (~O(N)). Differently, we encounter O(N) with our implementation in terms of identifying what the token is.