Assignment 0: C Tokenizer
Timur Misirpashayev

README

This program includes the following C libraries:

```
<stdlib.h>
<stdio.h>
<string.h>
<ctype.h>
```

My program consists of a main, a `tokenize` function, and a number of other specialized functions that are called by `tokenize`. Keeping track of its location in the input string with an index i, `tokenize` runs several tests on the ith character and chooses which functions are suitable for determining/printing the corresponding token. These functions return an updated value of i to inform `tokenize` of the location of the beginning of the next token in the input string. It should be noted that with this design, `tokenize` only looks at the *first* character of a token.

To clarify, suppose that the input string were "123stuff". Beginning at i = 0 (the zeroth character), `tokenize` runs several tests to determine how to proceed:

- If the current character is a white space character, move on to the next character.
- Else, if the current character could mark the beginning of a comment, check if there is a comment. If there is a comment, ignore everything inside.
- Else, if the current character is either a single (') or double (") quotation mark, check if there is a quote string (see Extra Credit below). Send i to `quotePrint`.
- Else, if the current character is alphabetical, the current token must be a word (or C keyword). Send i to `word`.
- Else, if the current character is a digit, send i to `positiveDigits` if the digit is non-zero and `zeroHandler` otherwise. (These functions then run their own tests and call other functions to determine whether the token should be a decimal integer, octal integer, hexadecimal integer, or floating point.)
- Else, the current character is an operator. Send i to `operator`. (Except in the case of "sizeof", which I have included with the C keywords.)

In this case, it is determined that i=0 marks the beginning of the decimal integer "123". Once this is printed, `tokenize` receives the updated value of i, which is now equal to 3 -- the location of the beginning of the next token, which is the word "stuff". Essentially, there is an interchange of indices between `tokenize` and the other functions, with `tokenize` sending out the index that marks the beginning of a token and receiving an updated index for the next one. In simple terms, `tokenize` can be likened to an administrative clerk that determines which experts to call in for a particular job.

Here is a list of my function prototypes:

```
void tokenize(char * inputString);
int word(char * inputString, int i);
int positiveDigits(char * inputString, int i);
int floatFinder(char * inputString, int i);
int floatPrint(char * inputString, int i);
int decimalPrint(char * inputString, int i);
int numPrint(char * inputString, int i);
int hexCheck(char * inputString, int i);
int zeroHandler(char * inputString, int i);
int hexPrint(char * inputString, int i);
int octalCheck(char * inputString, int i);
int octalPrint(char * inputString, int i);
int operator(char * inputString, int i);
int wordPrint(char * inputString, int i);
int singleLineComment(char * inputString, int i);
int multiLineComment(char * inputString, int i);
int multiLineCommentChecker(char * inputString, int i);
int quoteFinder(char * inputString, int i);
int quotePrint(char * inputString, int i);
```

Different functions are responsible for printing their own token types. For instance, `octalPrint` will only print octal integers; `hexPrint` will only print hexadecimal integers. See tokenizer.c for more information.

With the exception of `tokenize`, every function takes in the original input string and the index currently under consideration. The five functions `floatFinder`, `hexCheck`, `octalCheck`, `multiLineCommentChecker` and `quoteFinder` serve as boolean functions; that is, they return 1 if the token can be a floating point, hexadecimal integer, octal integer, multiline comment or quote string and 0 otherwise. The rest of the functions return an updated index which is to be given back to `tokenize`.

I tried to design the program so that it would be efficient in space and time. I decided to make strings out of words to more easily determine whether they were C keywords (`strcmp` is handy for this.) Without additional space, I would have had to compare each individual character manually, which would have made the code slightly more verbose and difficult to read. With regard to efficiency, my program takes one pass through the input string, and depending on the input, may scan ahead several characters from the current character to determine the token type (float, octal, decimal, etc.).

Extra Credit:

- My program detects C keywords. There are 32 of them: do, if, for, int, auto, case, char, else, enum, goto, long, void, break, const, float, short, union, while, double, extern, return, signed, static, sizeof, struct, switch, default, typedef, continue, register, volatile, and unsigned. These tokens are printed in the form <keyword>: "<keyword>" (e.g. sizeof: "sizeof", int: "int"). Input strings like "whilea", "double07" and "abcif" will be considered words, not C keywords + words. However, C keywords are present in input strings such as "07double" and "(volatile)".
- My program recognizes both single- and multi-line comments. Single-line comments are terminated when either the end of the input string or '\n' is reached. Multi-line comments are terminated when the "*/" closing character sequence is reached. Everything in between is ignored.
- My program recognizes tokens between ' and " marks as single strings. I have decided to call them "quote strings". The output is in the form quote string: <' or ">sample string<' or ">. I have included some examples in testcases.txt. (Note that when typing C escape characters into bash, the string should be surrounded by single quotes and prepended with $.)