3: A Lexer for FORTH, written in C++

CSCI 6636 / 4536

1 Overview

A FORTH system reads the code that is entered into the window or loaded from a file. The stream of input characters must be lexed before anything else can happen. The instructions that follow ask you to write a primitive lexer for FORTH in C++. The result will be very much like a real FORTH lexer.

When the lex is done, your output file will contain a symbol table, with one copy of each distinct symbol that occurred in your code, with the symbol's type.

2 Types

1. Define an enumerated type `TokenType` with symbols to represent the three kinds of tokens: words, numbers, and strings.

2. Define an enumerated type `StateType` with symbols to represent starting out, done, acquiring a token, acquiring a comment, and acquiring a string.

3 Classes

3.1 The Token Class

1. Data members include the name of the token (a string), its type (one of the TokenTypes) and its reference count (initially 1).

2. The Token constructor should accept a parameters for the name and type and initialize all three fields.

3. The print(oftream&) function should print all members. Use one neat line that will form a nice table when several Tokens are printed.

4. Define a function that will increment the counter.

3.2 The Lexer Class

1. Data members include a variable to hold the state (initially START), an ifstream, an ofstream, a map<string, Token>, one char for the current input, a string for collecting the characters of a token (initially empty), a TokenType variable for the current token, and anything else you think you need.

   The map is used like a dictionary to look up each input word that is found and to store each word, with its type. If it is a duplicate of a previous word, it is not added to the map a second time, instead, the occurrence counter of that symbol is incremented.

2. The Lexer constructor should accept a string parameter (the name of the input file) and open the input and output streams. Write titles to the output file: your name, assignment number, and the name of the file you are lexing. Then write the heading "Comments", for the comments that will soon be printed.

3. Define a destructor that closes your files.

4. Define a function `doLex()`, according to the instructions below.
5. Define a `print()` function that will print the contents of the map to the output file, in a readable layout.
6. `doToken(string name, TokenType tt)`: Define a helper function to be used when a token is identified:
   - Search the tokenMap for your token's name. If it is there, increment its counter.
   - Otherwise, declare a local temporary Token with the name and type you have identified, and insert it into the map.
   - Set the token-string to empty.
7. `doStart():` Define a helper function
   - If currentChar is whitespace, break out of the switch.
   - Else if currentChar is a backslash, set the state to slashPending, break.
   - Else if the current char is a '(' , set the state to parenPending, break.
   - Else you have a token that belongs in the symbol table. Change the state variable to *acquiringToken*. Put the current char into your token-string, break.

4 Detailed Instructions for doLex()

This function will implement the diagram of Forth Lexical Structure that accompanies this assignment.

The body of doLex() consists entirely of a loop (stop looping at end of file). Inside the loop is a switch with one case for each state in the StateType enumeration. Each case will change state or call one of your helper functions. DO NOT try to put the entire logic inside the switch. The helper functions you need are: doStart(), doToken(),

- Read a character (currentChar) each time around the loop. If the ifstream has reached end of file, terminate the loop[1].
- Switch to one of several cases, depending on the state:
  1. In the *start* state, call doStart():
  2. In the *slashPending* state:
     - If currentChar is whitespace, change the state to `acquiringSlash`.
     - Otherwise, change state to `acquiringToken`, break.
  3. In the *acquiringSlash* state:
     - You have found a whole-line comment, not a token.
     - Read the rest of the line and write each char, including the newline, to your output file. Comments and backslashes are not tokens and need no further processing.
     - Change the state to start, break.
  4. In the *parenPending* state:

[1] (Assume that there will be a newline on the end of the last line of code. The code will not end with an unterminated token.)

  - If currentChar is whitespace, change the state to `acquiringParen`.
  - Otherwise, change state to *acquiringToken*, break.

5. In the *acquiringParen* state:
   - You have found a partial-line comment, not a token.
   - Read and echo-print all characters until currentChar is a ')' . Comments and ')' are not tokens and need no further processing.
   - Change the state to *start*, break.

6. In the *acquiringToken* state:
   - Read input chars until you reach a whitespace. Append each to your token-string.
   - If the characters in the token are all digits, 0. . . 9, you have found a number. Call `doToken(name, NUMBER)` to process it. Set the state to *start*, break[2].
   - Else if the token is a dot quote (."), you have found the beginning of a quoted string. Call `doToken(name, WORD)` to process the dot-quote token. Set the state to *acquiringString*, break.
   - Otherwise, call `doToken(name, WORD)` to handle the token. Set the state to *start*, break.

7. In the *acquiringString* state:
   - Read input chars until you reach a double quote. Append each character (but not the double quote) to your token-string as you read it.
   - Call doToken(name, STRING to process your string.
   - Change the state to *start*, break.

5 In your main program...

- Instantiate the Lexer class.
- Call the doLex() function described above to process the input file.
- When you come to the end of the input file, write a line of dashes to the output file, followed by the list of tokens, in any convenient order. Use the print() function from your Token class. Make a neat table with columns.

[2] The actual FORTH lexer converts the string to a binary number at this point.