Addis Ababa University
ITSE 3242: System Programming

Project 1: A "De-Comment" Program

Due: February 25-27, 2017

Acknowledgement

This assignment was originally written by Robert M. Dondero, Jr. and other COS 217 faculty at Princeton University.

Purpose

The purpose of this assignment is to help you learn or review

(1) the fundamentals of the C programming language,

(2) the details of the "de-commenting" task of the C preprocessor, and

(3) how to use the GNU/Unix programming tools, especially `shell`, `vi/vim`, and `gcc`.

Rules

Form a group of three students and submit the grouping to your instructor.

Background

The C preprocessor is an important part of the C programming system. Given a C source code file, the C preprocessor performs three jobs:

- Merge *physical* lines of source code into *logical* lines. That is, when the preprocessor detects a line that ends with the backslash character, it merges that physical line with the next physical line to form one logical line. More precisely, if the preprocessor detects a backslash character immediately followed by a newline character, then it simply removes both characters.
- Remove comments from ("de-comment") the source code.

- Handle preprocessor directives (`#define`, `#include`, etc.) that reside in the source code.

The second of those jobs -- the de-comment job -- is more substantial than one might think. For example, when de-commenting a program the C preprocessor must be sensitive to:

- The fact that a comment is a token delimiter. After removing a comment, the C preprocessor must make sure that a white space character is in its place.
- Line numbers. After removing a comment, the C preprocessor sometimes must insert blank lines in its place to preserve the original line numbering.
- String and character literal boundaries. The preprocessor must not consider the character sequence (`/*...*/`) to be a comment if it occurs inside a string literal (`"..."`) or character literal (`'...'`).

The Task

Your task is to compose a C program named `decomment` that performs a subset of the de-comment job of the C preprocessor, as defined below.

Functionality

Your program should be a Unix *filter*. A filter is a program that reads characters from the standard input stream, and writes characters to the standard output stream and possibly to the standard error stream. Specifically, your program should

(1) read text, presumably a C program, from the standard input stream,

(2) write that same text to the standard output stream with each comment replaced by a space, and

(3) write error and warning messages as appropriate to the standard error stream. A typical execution of your program from the shell might look like this:

```
decomment < somefile.c > somefileWithoutComments.c 2>
errorsAndWarnings
```

In the following examples a space character is shown as "$_s$" and a newline character as "$_n$".

Your program should replace each comment with a space. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc/*def*/ghi$_n$	abc$_s$ghi$_n$	
abc/*def*/$_s$ghi$_n$	abc$_{ss}$ghi$_n$	

abc$_s$/*def*/ghi$_n$	abc$_{ss}$ghi$_n$	

Your program should define "comment" as in the C90 standard. In particular, your program should consider text of the form (`/*...*/`) to be a comment. It should *not* consider text of the form (`//...`) to be a comment. Example:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc//def$_n$	abc//def$_n$	

Your program should allow a comment to span multiple lines. That is, your program should allow a comment to contain newline characters. Your program should add blank lines as necessary to preserve the original line numbering. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc/*def$_n$ghi*/jkl$_n$mno$_n$	abc$_{sn}$jkl$_n$mno$_n$	
abc/*def$_n$ghi$_n$jkl*/mno$_n$pqr$_n$	abc$_{snn}$mno$_n$pqr$_n$	

Your program should not recognize nested comments. Example:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc/*def/*ghi*/jkl*/mno$_n$	abc$_s$jkl*/mno$_n$	

Your program should handle C string literals. In particular, your program should not consider text of the form (`/*...*/`) that occurs within a string literal (`"..."`) to be a comment. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc"def/*ghi*/jkl"mno$_n$	abc"def/*ghi*/jkl"mno$_n$	
abc/*def"ghi"jkl*/mno$_n$	abc$_s$mno$_n$	
abc/*def"ghijkl*/mno$_n$	abc$_s$mno$_n$	

Similarly, your program should handle C character literals. In particular, your program should not consider text of the form (`/*...*/`) that occurs within a character literal (`'...'`) to be a comment. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc'def/*ghi*/jkl'mno$_n$	abc'def/*ghi*/jkl'mno$_n$	
abc/*def'ghi'jkl*/mno$_n$	abc$_s$mno$_n$	
abc/*def'ghijkl*/mno$_n$	abc$_s$mno$_n$	

Note that the C *compiler* would consider the first of those examples to be erroneous (multiple characters in a character literal). But many C *preprocessors* would not, and your program should not.

Your program should handle escaped characters within string literals. That is, when your program reads a backslash (`\`) while processing a string literal, your program should consider the next character to be an ordinary character that is devoid of any special meaning. In particular, your program should consider text of the form (`"...\"..."`) to be a valid string literal which happens to contain the double quote character. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc"def\"ghi"jkl$_n$	abc"def\"ghi"jkl$_n$	
abc"def\'ghi"jkl$_n$	abc"def\'ghi"jkl$_n$	

Similarly, your program should handle escaped characters within character literals. That is, when your program reads a backslash (`\`) while processing a character literal, your program should consider the next character to be an ordinary character that is devoid of any special meaning. In particular, your program should consider text of the form (`'...\'...'`) to be a valid character literal which happens to contain the quote character. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc'def\'ghi'jkl$_n$	abc'def\'ghi'jkl$_n$	
abc'def\"ghi'jkl$_n$	abc'def\"ghi'jkl$_n$	

Note that the C *compiler* would consider both of those examples to be erroneous (multiple characters in a character literal). But many C *preprocessors* would not, and your program should not.

Your program should handle newline characters in C string literals without generating errors or warnings. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc"def$_n$ghi"jkl$_n$	abc"def$_n$ghi"jkl$_n$	
abc"def$_n$ghi$_n$jkl"mno/*pqr*/stu$_n$	abc"def$_n$ghi$_n$jkl"mno$_s$stu$_n$	

Note that a C *compiler* would consider those examples to be erroneous (newline character in a string literal). But many C *preprocessors* would not, and your program should not.

Similarly, your program should handle newline characters in C character literals without generating errors or warnings. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc'def$_n$ghi'jkl$_n$	abc'def$_n$ghi'jkl$_n$	
abc'def$_n$ghi$_n$jkl'mno/*pqr*/stu$_n$	abc'def$_n$ghi$_n$jkl'mno$_s$stu$_n$	

Note that a C *compiler* would consider those examples to be erroneous (multiple characters in a character literal, newline character in a character literal). But many C *preprocessors* would not, and your program should not.

Your program should handle unterminated string and character literals without generating errors or warnings. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc"def/*ghi*/jkl$_n$	abc"def/*ghi*/jkl$_n$	
abc'def/*ghi*/jkl$_n$	abc'def/*ghi*/jkl$_n$	

Note that a C *compiler* would consider those examples to be erroneous (unterminated string literal, unterminated character literal, multiple characters in a character literal). But many C *preprocessors* would not, and your program should not.

Your program should detect an unterminated comment. If your program detects end-of-file before a comment is terminated, it should write the message "Error: line X: unterminated comment" to the standard error stream. "X" should be the number of the line on which the unterminated comment begins. Examples:

Standard Input Stream	Standard Output Stream	Standard Error Stream
abc/*def$_n$ghi$_n$	abc$_{snn}$	Error:$_s$line$_s$1:$_s$unterminated$_s$comment$_n$
abcdef$_n$ghi/*$_n$	abcdef$_n$ghi$_{sn}$	Error:$_s$line$_s$2:$_s$unterminated$_s$comment$_n$
abc/*def/ghi$_n$jkl$_n$	abc$_{snn}$	Error:$_s$line$_s$1:$_s$unterminated$_s$comment$_n$
abc/*def*ghi$_n$jkl$_n$	abc$_{snn}$	Error:$_s$line$_s$1:$_s$unterminated$_s$comment$_n$
abc/*def$_n$ghi*$_n$	abc$_{snn}$	Error:$_s$line$_s$1:$_s$unterminated$_s$comment$_n$
abc/*def$_n$ghi/$_n$	abc$_{snn}$	Error:$_s$line$_s$1:$_s$unterminated$_s$comment$_n$

Your program (more precisely, its `main` function) should return `EXIT_FAILURE` if it was unsuccessful, that is, if it detects an unterminated comment and so was unable to remove comments properly. Otherwise it should return `EXIT_SUCCESS` or, equivalently 0.

Your program should work for standard input lines of any length.

You may assume that the final character in the standard input stream is the newline character. That is, if the standard input stream contains any characters at all, then you may assume that the last one of them is the newline character.

Your program may assume that the backslash-newline character sequence does not occur in the standard input stream. That is, your program may assume that logical lines are identical to physical lines in the standard input stream. So your de-comment program need not perform the first of the three jobs described above in the "Background" section.

Design

Design your program as a *deterministic finite state automaton* (*DFA*, alias *FSA*).

Your program should not consist of one large `main` function. Instead your program should consist of multiple small functions, each of which performs a single well-defined task. In this program you should create one function to implement each state of your DFA, as described in lectures.

Generally, a (large) C program should consist of multiple source code files. For this assignment, you need can split your source code into multiple files or you may place all source code in a single source code file.

We suggest that your program use the standard C `getchar` function to read characters from the standard input stream.

Logistics

You should create your program on Linux (Ubuntu) using `bash`, `vi/vim`, and `gcc`.

Step 1: Design a DFA

Express your DFA using the traditional "labeled ovals and labeled arrows" notation. More precisely, use the same notation as is used in the examples from Section 7.3 of the Sedgewick and Wayne book. Let each oval represent a state. Give each state a descriptive name. Let each arrow represent a transition from one state to another. Label each arrow with the character, or class of characters, that causes the transition to occur. We encourage (but do not require) you also to label each arrow with action(s) that should occur (e.g. "print the character") when the corresponding transition occurs.

Express as much of the program's logic as you can within your DFA. The more logic you express in your DFA, the better your grade on the DFA will be.

To properly report unterminated comments, your program must contain logic to keep track of the current line number of the standard input stream. You need not show that logic in your DFA.

Create a textual representation of your DFA in a file named `dfa`.

Step 2: Create Source Code

Use `emacs/ vi/ vim/nano` to create source code in a file named `decomment.c` that implements your DFA.

Step 3: Preprocess, Compile, Assemble, and Link

Use the `gcc` command to preprocess, compile, assemble, and link your program. Perform each step individually, and examine the intermediate results to the extent possible.

Step 4: Execute

Execute your program multiple times on various input files that test all logical paths through your code.

Step 5: Create a `readme` File

Use `emacs` to create a text file named `readme` (not `readme.txt`, or `README`, or `Readme`, etc.) that contains:

- Your name and the assignment number.
- A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated.
- An indication of how much time you spent doing the assignment.
- Your assessment of the assignment: Did it help you to learn? What did it help you to learn? Do you have any suggestions for improvement? Etc.
- Any information that will help us to grade your work in the most favorable light. In particular you should describe all known bugs.

Descriptions of your code should not be in the `readme` file. Instead they should be integrated into your code as comments.

Your `readme` file should be a plain text file. Don't create your `readme` file using Microsoft Word or any other word processor.

Step 6: Submit

There are two submissions for this project.

1. The first part of the submission will be 1 week after the project is given (Before Friday February 8, 2019). You are expected to review the given lab material and design your DFA and submit
2. The second submission will be a zipped file of your work (dfa, decomment.c, readme) to your instructor before February 25, 2019 (Three weeks after assignment is given).

A project presentation day will be arranged. You are expected to present the project implementation and answer basic questions about the program you designed based on the schedule. All group members should be present during the project submission and presentation.

Grading

The project will be evaluated out of 20 points. The grading will have group evaluation and individual evaluation based on the work submitted and the presentation.

We will grade your work on two kinds of quality: quality from *the user's* point of view, and quality from *the programmer's* point of view. From the user's point of view, a program has quality if it behaves as it should. The correct behavior of your program is defined by the previous sections of this assignment specification.

From the programmer's point of view, a program has quality if it is well styled and thereby easy to maintain. In part, good style is defined by the rules given in *The Practice of Programming* (Kernighan and Pike), as summarized by the Rules of Programming Style document.

These additional more course-specific rules apply:

- **Indentation**: Indent using spaces, not tabs. `emacs` does that when using the `.emacs` configuration file that we have provided.
- **Line lengths**: Limit line lengths in your source code to 72 characters. Doing that allows us to print your code in two columns, and so saves paper. When using the .emacs configuration file that we have provided, `emacs` indicates lines that exceed 72 characters. Specifically, `emacs` uses a green background to mark the character in column 72, and a gray background to mark the character in column 73.
- **Names**: Use a clear and consistent style for variable and function names. One example of such a style is to prefix each variable name with characters that indicate its type. For example, the prefix `c` might indicate that the variable is of type `char`, `i` might indicate `int`, `pc` might mean `char*`, `ui` might mean `unsigned int`, etc. But it is fine to use another style -- a style that does not include the type of a variable in its name -- as long as the result is a clear and readable program.
- **File Comments**: Begin each source code file with a comment that includes your name, the number of the assignment, and the name of the file.
- **Variable Comments**: Compose a comment for each global variable. The comment should appear immediately before the definition/declaration of the global variable. Compose a comment for each local variable definition/declaration whose purpose is unclear. The comment should appear immediately before the definition/declaration of the local variable, or at the end of the line containing the definition/declaration.
- **Function Comments**: Compose a comment for each function. A function definition/declaration's comment should immediately precede the function's definition/declaration.
    - A function's comment should describe *what the function does* from the point of view of the function's callers. A function's comment should refer explicitly to the function's parameters (by name) and the function's return value. A function's comment should state what, if anything, the function reads from standard input or any other stream, and what, if anything, the function writes to standard output, standard error, or any other stream. In short, a function's comment should describe the flow of data into and out of the function.
    - A function's comment should *not* describe how the function works. If a programmer cannot reasonably determine how the function works by reading the function's definition, then either (1) rewrite the function definition so it is clearer or, if that is impossible, (2) add some local comments (that is, comments with the function definition) to explain the code.

To encourage good coding practices, we will deduct points if `gcc` generates warning messages.

Avoiding Global Variables

Suppose a program contains functions `f` and `g`, and that `f` calls `g`. Further suppose that `f` wishes to pass some values to `g`. `f` can do that using ordinary parameters.

Now suppose `g` wishes to pass some values back to `f`, its caller. `g` can pass the first value back via its return value. But how can `g` pass additional values back to `f`?

One approach is to use global variables, where a *global* variable is one which is defined outside of all functions. `g` could assign the additional values to global variables; `f` then could fetch those values by accessing the global variables.

However, generally you should avoid using global variables. Instead all communication of values into and out of a function should occur via the function's parameters and its return value.

Indeed in your `decomment` program you should avoid using global variables. You can do that using either of these two approaches:

Some notes:

- Although you should avoid defining *variables* at the global level, it's fine to define *data types* at the global level. For example, code of this form:
    - `enum SomeEnum {SOMEVALUE1, SOMEVALUE2, ...};`

  at the global level is fine.