Villanova University
CSC 2400: Computer Systems I

A "De-Comment" Program

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 bash, emacs, and gcc.

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.
- Remove comments from ("de-comment") the source code.
- Handle preprocessor directives (#define, #include, etc.) that reside in the source code.

The de-comment job is substantial. For example, 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 whitespace 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 ('...').

Your 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." That is, your program should read characters from the standard input stream, and write 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, and (2) write that same text to the standard output stream with each comment replaced by a space. A typical execution of your program from the shell might look like this:

```
decomment < somefile.c > somefilewithoutcomments.c
```

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
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 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
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
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
abc/*def/*ghi*/jkl*/mno$_n$	abc$_s$jkl*/mno$_n$

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	Error Message
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 should work for standard input lines of any length.

Design

Design your program as a *deterministic finite state automaton (DFA, alias FSA)*. The FSA concept is described in wikipedia, http://en.wikipedia.org/wiki/Finite-state_machine.

Generally, a (large) C program should consist of multiple source code files. For this assignment, you need not split your source code into multiple files. Instead you may place all source code in a single source code file. Subsequent assignments will ask you to write programs consisting of multiple source code files.

We suggest that your program use the standard C **`getchar`**() function to read characters from the standard input stream. Also use **`enum`** or **`define`** constructs for the states of your FSA in your C implementation.

Logistics

You should create your program on tanner using bash, emacs and gcc.

Step 1: Design a DFA

Express your DFA using the traditional "ovals and labeled arrows" notation. More precisely, use the same notation as is used in the examples shown in class. Capture 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.

Step 2: Create Source Code

Use emacs 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.

You should also test your decomment program against its own source code using a command sequence such as this:

```
decomment < decomment.c > output
```

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, as prescribed by the course Policies web page.
- 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

Hand in a printout of your decomment.c file, your readme file, and a hardcopy of your "circles and labeled arrows" DFA. A DFA drawn using drawing software (e.g. Microsoft PowerPoint) would be good, but it is sufficient to submit a neatly hand-drawn DFA.

Grading

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

From the user's point of view, a program has quality if it behaves as it should. The correct behavior of the decomment 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, style is defined by the rules summarized in the Basic Rules of Programming Style document. These additional rules apply:

- Names: You should 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, "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.
- Comments: Each source code file should begin with a comment that includes your name, the number of the assignment, and the name of the file. Include comments in your source code to explain what blocks of code do.

OPTIONAL Functionality (Bonus Points)

You may expand your code to handle string literals and character literals as follows (just like a real preprocessor does). Text of the form (/* ... */) that occurs within a string literal ("...") should not be considered a comment. Examples:

Standard Input Stream	Standard Output 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, text of the form (/* ... */) that occurs within a character literal ('...') should not be considered a comment. Examples:

Standard Input Stream	Standard Output 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**
`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. 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**
`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**
`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**
`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**
`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.

Acknowledgement

This project has been designed by Prof. Jennifer Rexford from Princeton University, slightly modified by Prof. Mirela Damian.