CSE 30 Spring 2022 Programming Assignment #1 - (Vers 1.1) Due April 7, 2022 @ 11:59PM

Assignment – C preprocessor, Separate Compilation, and Test Harnesses

The goals of this PA are:

1. To learn about the C preprocessor.
2. Practice separate compilation techniques.
3. Get familiar with developing software in the Linux environment.
4. Most importantly, introduce the basic concept of using/developing a test harness, a critical part of all commercial software product development processes.

As we have discussed in lecture, the C preprocessor (cpp) processes a C source file, sending its output to the C compiler. In this programming assignment, you will be implementing a simplified version of just one of many tasks that cpp performs: *removing comments from source code*. Removing comments from a source file is a bit more complex than it seems, as there are many special situations (jargon: "corner cases") that you must address while performing this task.

1. A *token* is a single element of a programming language like a keyword, a variable name, etc. Tokens are separated by what are called *token delimiters*. A comment, white space, and newlines are all examples of *token delimiter*s. Compilers (Compiler Course CSE 131) perform operations on a source file such as enforce the syntax of a programming language and depend on token delimiters to find the tokens in a program. When you remove a comment from a program and since the comment is a *token delimiter*, it must be replaced with another token delimiter. For this assignment (and what cpp uses), your program will use a ***single*** *white space (a single blank space)* as the replacement *token delimiter*.

2. While compiling a program and an error is detected, the compiler outputs an error message along with the line number (aside: the first line in the file is numbered 1) in the *original source file* where the fault lies. During the process of removing comments, especially comments that span multiple lines, your program must preserve the line location of each token in the output. Therefore, the number of lines in the comment from *comment start* to *comment end* must be preserved so the output has the same number of lines.

3. There are two kinds of comments that the C preprocessor (and *your program*) must handle, C style comments /*…*/ and C++ style comments //... . C++ style comments start with a // and end up to, *but not including*, **either** a *newline* (the newline is not part of the comment) or when *end of file* is reached on the input stream, whichever comes first. A C++ style comment is replaced with a single space. C style comments may span more than one line and start with the /* delimiter and ending with the */ delimiter. Both delimiters and all the characters between the start (/*) and end (*/) delimiters are part of the comment. However, since a C style comment may include newlines, a C style comment would be

replaced by a single space followed by all the newlines (if any) **found within the comment** so the input and output without the comments will contain the same number of lines.

4. In C, characters inside either a character literal boundary ('...') or a string boundary ("...") must **not be** processed by the C preprocessor (and *your program*). For example, the character sequences (/*...*/) or (//…) are **not** comments when they **occur inside** a string literal ("...") or character literal ('...').

5. In C, **inside** a string or character literal (between the " …" or '…'), the \ when combined with the next character often invokes a special meaning. For example, a "\n" is interpreted as a newline and a "\"" is a string with a single " in it. The C preprocessor is not involved with assigning special meaning to these two characters \ sequences, that is the job of the compiler. When the C preprocessor (and your program) sees a \ **inside of a string or character literal**, it says the next character (after the \) has no special meaning (so skip to the next character). In general, this requires no action on the part of the C preprocessor except when for two conditions, a \' and a \". In this case the \" or \' are just treated as regular characters and do not terminate the string or character literal. For example the string literal "…..\"…." is a single string literal with a \" as part of the literal. \ can also be used outside of literals as a line extension (in front of a \n). In this assignment you only handle \ as a special "escape" character inside of a literal.

6. While most C preprocessors do not check language syntax, some do (and *your program* will) flag a few error situations by outputting an error message (with a line number to standard error) and then returning from `main()` with a EXIT_FAILURE return. Here are the syntax error conditions that your program must handle.
   (1) When *your program* sees the **start** of a *C style comment*, /* and by reaching end of the file on the input stream it has not yet seen the closing */, it will issue an error message. The warning message written to the standard error stream will indicate that a comment starting at line number *N* was not terminated.
   (2) The same error reporting as above occurs for missing terminations for *character or string literals*. However, illegal syntax within either a character literal (more than a single character for example) or a string literal are not flagged as errors (that is the job of the compiler).

At the end of this document are some output samples for normal and error conditions. Also examine the test files and the expected output files in the test harness (see below).

Usage

The program that you write will operate like a Linux command line utility: it will read characters from the standard input stream, write characters to the standard output stream, and write error messages to the standard error stream. If there are no errors in the input stream, the program will return EXIT_SUCCESS from main(). If there are errors in the input stream, the program will return EXIT_FAILURE from main().

Specifically, your program (which will be called **omit**) must perform the following steps in a **loop until end of file is reached** on the input stream:

1) Read text from the standard input stream (like a C source file).
2) Write that same text to the standard output stream with each comment replaced with a space, (handling comments, character and string literals as described below).
3) Write error and warning messages (as specified below) to the standard error stream (and return EXIT_FAILURE from main()). A typical execution of your program from the shell might look like this:

```
./omit < input.c > input_nocomments.c 2> ERRORS
```

You would use the following to test omit on actual C source to make sure the code still compiles properly. The following line would compile the file omit.c to the object file omit.o.

```
./omit < omit.c | gcc -I. -Wall -Wextra -Werror -c -xc –
```

Coding and Program Development Requirements

1. You will write your program only in C to run in a Linux environment.
2. The program part of the assignment will consist of three (3) files, `omit.c`, `subs.c` and `omit.h`. The contents of these files are described below.
3. You will use the supplied Makefile to compile your program.
4. You must test your program on the pi cluster.
5. You must ***first*** use the supplied test files (the MVP[1] tests) to develop your code.
6. You must pass all supplied MVP tests ***before submitting*** to Gradescope for the private tests.
7. Your code will determine each next state transition in a function (one for each current state) that returns the next state (either the same state or a new state) based on the input character passed to it as a single argument (just like we did in lecture).
8. You may **only use** the following library functions**. IMPORTANT: The use of any other library functions in your code will result in a score of *zero* (0) on the assignment**:

   `getchar()`, `putchar()`, `printf()`, and `fprintf()`

Your code, **minus comments**, should be under 250 lines of C. If your code gets larger than this, see a tutor, a TA, or the instructor for help as your approach may be getting overly complex.

[1] Minimum Viable Product

How to Approach This Programming Assignment

Step 1: Design your DFA[2]

The first step of the assignment is to design an DFA that implements the logic of the omit program. Your will create the states and state transitions that accept each of the different sequences. You have the following sequences to recognize: (1) C++ style comment, (2) C style comment, (3) string literal, and (4) character literal.

Design your merged DFA using the circles and labeled arrows representation as was shown in lecture. In this program your output on the labeled arrows will be either to print the input character to standard output or to not print it (when removing comments). Since your program will operate until the end of file, you will not have any accepting end states. Use the examples shown during the lecture as your guide.

Although your program will operate until the end of file (EOF) is reached, do not show any EOF transitions in your DFA.

To properly report unterminated comments and literals, your program must keep track of the current line number of the standard input stream; do not show this processing in your DFA.

YOU MUST TURN IN AS PART OF YOUR GRADE the following: Draw this DFA and create a copy of it in a file called **`dfa.pdf`**. You will submit this with the rest of your assignment.

Step 2: Starting the code, define your states and state handlers

1. In the file `omit.h` you will need to place the `enum` definition for each of the states in your DFA. Specifically, like shown in lecture, you will define an enum that lists all the states in your combined DFA. Something like:
   ```
   enum typestate {START, …};
   ```
2. Next, for each state place the function prototype for each of the state *function handlers* in omit.h. As we showed in lecture there will be a *state function handler* for each current state that when passed the next input it will return the next state. You should not place a function prototype for main() in omit.h as it is not needed (nor is it good practice to do so).
3. For example, say the current state is START (`state == START`). So, you might specify a function prototype where the `int` parameter is the return value from a `getchar()`; note that it is ok to pass any additional parameters if needed to the function but you should not have to do so:
   ```
   enum typestate STARTsub(int);
   ```
4. Review the `CSE30_C_Style` document. In this document are the guidelines on which your program will be graded for style points.

[2] Deterministic finite automaton

Step 3: Write your main function and put it in omit.c

In `main()` you will write your main loop that processes input, one character at a time until EOF is reached like was shown in lecture. Make sure within your loop you keep track of the total line count and the starting line count for comments and literals, so you can print the correct starting line number if there is an error.

When EOF is reached, the loop will be exited. Then, based on the last state the program was in at the point the EOF was reached:

1. Decide if there was an error or not.
2. Print any final character(s) if any to standard out.
3. Print any standard error messages including starting line count for the error.
4. Finally, return either `EXIT_SUCCESS` (for no errors), or `EXIT_FAILURE` (for errors), from `main()`.

Step 4: Write your state function handlers and put all of them in `subs.c`

Write the functions for handling the next state determination for each state in your DFA. If you had any other helper functions (not needed for this assignment), you would also put them here.

Step 5: Compile your program with `make`

The Makefile will compile your program and create an executable called `omit`.
You are required to write code that compiles cleanly (using the compile options already specified in the Makefile – do not alter the Makefile or override the options to gcc).

Step 6: Test your program and debug

The starter files contain a rudimentary test harness (software used to test the operational correctness of your program) to run the Minimum Viable Product Tests (public tests) on your program. (See the section below on testing.) Test harnesses are a standard part of commercial software development. In industry you would write both your code and test harness code. For this assignment, we supply a basic test harness and some test cases for you. **You must use the test harness to test your program before making your first submission to gradescope**. You are encouraged to add any additional tests you feel necessary to pass the hidden tests that gradescope will use.

The test harness can be invoked from the Makefile. You can use the make target ***test*** to run the public tests on your program. This target will compile your program if required and then run all the tests.

```
$ make test
```

Testing your software is just as important as developing it. Invoking the test harness has its own target in the Makefile so it is easy to run tests after you change your code. This is more important for software that is being maintained (after it has shipped) as simple code changes can break things (called a regression) and you want to run the test suite (the set of tests) to make sure the code still is fully functional. In the test section below this is a description of some of the Minimum Viable Product tests and how the simple test harness works.

Step 7: Comment and check your code follows the CSE30_C_Style document

To get all the style points, read over the style document and clean up your code to follow the guidelines. In PA1 style will not be heavily enforced, so use it as a chance to practice.

Step 8 : Submit to Gradescope under the assignment titled "A1: C preprocessor"

After you pass all the MVP tests in the test harness and only then, will you submit to gradescope the following files: `omit.c omit.h subs.c dfa.pdf`

You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint. Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all the files and upload the .zip to the assignment. Ensure that the files you submit are not in a nested folder.

After submitting, the autograder will:

1. Check all files were submitted.
2. That the program compiles without errors.
3. Rerun the public tests and then the private tests.

Make sure to check the autograder output after submitting!

Grading

The assignment will be graded out of 60 points.

- 5 points on programming style and comments. **Review the CSE30_C_Style document.**
- 40 points on passing the MVP public tests (these are the same ones as **`make test`** runs).
- 15 points on passing the Field Trial private tests.

Development and Testing

This PA will demonstrate to you how a very ***simplistic test harness*** works and how to develop your own test files without depending on gradescope. This is a skill you will need to develop software when you enter the commercial development environment as nobody is going to write the initial program tests for you (nor will you have gradescope to your as a test harness).

Important: In this PA, we give you a simple test harness with the 30 Minimum Viable Product (MVP) tests. In future PAs you will be expected to write more of your own tests and expand on the actual test harness as needed. Study the concept of a test harness by looking at the simple one supplied with this PA.

You are **strongly encouraged** to **add tests of your own** as there are still the hidden tests in the ***Field Trial Tests*** that you will be graded on. Try to think about test cases not covered in the supplied MVP tests that might be included in the ***Field Trial Tests*** and add them to the test harness.

As part of this assignment, you will practice separate compilation techniques. Your program will be developed across two .c files and one .h file, compiled independently (the .c files) and linked together by the Makefile.

The starter files are contained in the `cs30sp22` public folder (`../public` relative to your home directory) in a tarball (a common way to distribute files in Linux). From your home directory, run the following to create the directory PA1 in your home directory:

```
tar -xvf ../public/PA1.tar.gz
```

Inside the directory PA1 at the top-level you will find the following files (and one directory):

`Makefile`: The Makefile you will use to develop your program. Please examine the Makefile to understand the various targets, especially the test target. You should not have any reason to edit the Makefile.

`omit.c`: This is the file where you will put your `main()` function.

`subs.c`: This is the file where you will put your state functions. There should be one function per DFA state. You should not need any additional helper functions.

`omit.h`: This is the function prototype file for all the state function handlers in subs.c. Put only the function prototypes and the `enum` declaration for the DFA states in this file. **There should be no code, functions, or variable definitions in this file.**

`tests/`: A directory described below.

The Test Harness

A test harness works by running your program with a known set of inputs (the tests) and checking the output against a set of expected outputs for each test. If your program output differs from the expected output, there is a flaw in your program.

In the same PA1 directory you will find another directory called **`tests`**. This is the directory where the test harness is located. There is a shell script in this directory called **`runtests`** (running make test in the PA1 directory or in the test directory will call this script). This shell script runs the tests. In industry this script is more complex (and may be written in another scripting language like python) and often include executable code written to test your program.

The directory ***in:*** contains the *Minimum Viable Product* (public tests) input files for this programming assignment. There you will find files labeled *test1* through *test30*. To add tests to the harness you would add them as files **consecutively numbered** *test31* and higher (do not leave out a number) in this directory. There is a line in the runtests scripts that specifies the number range for the public tests, you would edit the ending number in the line below (replacing the 30 with the number of the last test you added).

```
for index in {1..30}
```

The line above will run the script ***runtests*** using files with the names test1 through test30.

The directory ***good***: contains the expected standard out and standard error output for each test. The files in the good directory are numbered with the same number as the corresponding test file that generates the output. For example, for in/test14 a properly operating omit program, the expected standard output would be found in good/out14 and the expected standard error would be found in the file good/err14. When you add tests, you need to add the two expected output files (one for standard out and standard error) to the good directory. As an aside, to create an empty file (like there is no output expected) you can use the command ***touch*** to create a new empty file. Example: *`touch err31`* (the file err31 must not exist prior to running touch to make sure you get an empty file). Also, if you try to run this script `runtests` on a different OS than on the pi's, you may have to alter the path to bash in the shebang at the top line if the script fails to run. For example, on Linux or MacOS the path is /bin/bash.

The directory **`out:`** contains the standard output and standard error output from your program (put there by the script runtests). When running test14 for example, your program's standard output would be in the file out/out14, and your program's standard error would be in the file out/err14.

The script **`runtests`** will execute your program on each test, capture the standard and standard error output into files and then uses the Linux program **`diff`** to compare the expected output (from the good directory) against your programs output (in the out directory), printing the differences (if any). To read the output of `diff` (also see the man page – `man diff` on the pi cluster):

The script runs

```
diff expected_output your_programs_output

diff expected_error your_programs_error
```

Lines from diff that start with a **<** is the expected output, lines that start with **>** is your output

The numbers are the line numbers in the files being compared. The diffs are run both on the standard out and standard error.

For example:

```
diff good/out3 out/out3
1c1
< abc  ghi
---

> abc  gh i
```

In the example 1c1 states that line 1 in good/out3 is changed to line 1 in out3. If you are properly preserving line counts the numbers on both sides of the C code should be the same (see the man page for diff).

Your program passes a test if diff does not print any differences.

To look at an output or expected file you can use Linux programs like `od -c filename`, `hexdump -c filename` (or `hd filename` for short) Be aware that od uses byte offsets in octal (default) and hexdump the byte offsets are in hexadecimal (default).

If you want to look at the file using an offset in decimal and with the characters in a single column, some people like to use (see man hexdump on how to program different outputs using hexdump):

```
hexdump -v -e '1 "%_ad\t"' -e '/1 "%_c\n"' filename | less
```

Output Examples

In the tables below, we describe the **exact** behavior of *your program* on a few of the Minimum Viable Product tests. You can use these tables when designing your program. The column *Read Standard Input* describes the content of the input stream (all the contents of a file for example). The column *Write Standard Out* describes the output that your program will emit in response. Only when there is a string in the column *Write Standard Error*, does it indicate there was an error in the input and what the output to the standard error stream should be in response to that error.

A descending "$_b$" is a single blank and a descending "$_n$ " is a single newline.

Read Standard Input	Write Standard Output	Write Standard Error
`abc/*def*/ghi`$_n$	`abc`$_b$`ghi`$_n$	
`abc/*def*/`$_b$`ghi`$_n$	`abc`$_{bb}$`ghi`$_n$	
`abc`$_b$`/*def*/ghi`$_n$	`abc`$_{bb}$`ghi`$_n$	
`abc//ghi`$_n$	`abc`$_{bn}$	
`abc/*def`$_n$`ghi*/jkl`$_n$`mno`$_n$	`abc`$_{bn}$`jkl`$_n$`mno`$_n$	
`abc/*def`$_n$`ghi`$_n$`jkl*/mno`$_n$`pqr`$_n$	`abc`$_{bnn}$`mno`$_n$`pqr`$_n$	
`abc/*def/*ghi*/jkl*/mno`$_n$	`abc`$_b$`jkl*/mno`$_n$	

Some of these may be illegal in C, but that is the compiler's job (not the pre-processor).

Read Standard Input	Write Standard Output	Write Standard Error
abc"def/*ghi*/jkl"mno$_n$	abc"def/*ghi*/jkl"mno$_n$	
abc/*def"ghi"jkl*/mno$_n$	abc$_b$mno$_n$	
abc'def/*ghi*/jkl'mno$_n$	abc'def/*ghi*/jkl'mno$_n$	
abc"def$_n$ghi"jkl$_n$	abc"def$_n$ghi"jkl$_n$	
abc"def\"ghi"jkl$_n$	abc"def\"ghi"jkl$_n$	
abc'def\'ghi'jkl$_n$	abc'def\'ghi'jkl$_n$	

Error conditions with error formatting specifications.

Read Standard Input	Write Standard Output	Write Standard Error
abc$_n$"def/*ghi*/jkl$_n$	abc$_n$"def/*ghi*/jkl$_n$	Error:$_b$line$_b$2:$_b$unterminated$_b$quote(")$_n$
abc$_{nn}$'def/*ghi*/jkl$_n$	abc$_{nn}$'def/*ghi*/jkl$_n$	Error:$_b$line$_b$3:$_b$unterminated$_b$quote(')$_n$
$_n$abc/*def$_n$ghi$_n$	$_n$abc$_{bnn}$	Error:$_b$line$_b$2:$_b$unterminated$_b$comment$_n$