COM S 440/540 Project part 1

Create a lexer (and preprocessor)

1 Requirements for part 1

When executed with a mode of 1, your compiler should read the specified input file(s) and process the tokens contained in the inputs; essentially this is the lexical analysis phase of the compiler (referred to as the "lexer"). The output stream should contain a line for each token, exactly of the form

filename `line` *line number* `text` '*text*' `token` *TOKENNAME*

Nothing else should be written to the output stream (again, you are encouraged to define your own switches for displaying additional information, to help with debugging). Use the token names indicated in Table 1. You will notice that some of the C keywords and operators are missing; you may implement some or all of these if you wish, but do not introduce any new keywords or operators that are not part of the C standard. The input file(s) may be C header files, C source files, or arbitrary text files; in any case, your lexer must split the input stream into tokens and handle the preprocessor features described in the following sections. At minimum, you must handle a single input file specified on the command line. There are extra credit options (see below) for handling several input files. While you can always write a lexer by hand, you are encouraged to use `flex` or `lex` to make your job easier.

2 Errors

Any unknown tokens (such as "`$`") should generate an error message. For this and all remaining parts of the compiler, any error messages caused by some issue in an input file must:

- be written to standard error;
- contain the name of the input file that caused the error;
- contain the line number and optionally the text that caused the error (or as close as possible to the actual error);
- have the format "`Error` (description) `file` (filename) `line` (line number)", optionally followed by "`text` '(text)' ". Additional information may follow on the next line(s), also written to standard error.

You are free to choose the description and other text. Using this format allows the grading script to compare error messages. Warning messages should follow the same format, with "`Warning`" in place of "`Error`". Students are encouraged to check the output and error messages given for the example test files, for all parts of the project.

3 Preprocessor features

The "whitespace" characters (space, tab, carriage return, and newline), except when part of a lexeme, serve only as delimiters to separate tokens, and should be discarded. The only exception to this rule is when a space appears as a character constant or within a string constant. Also, note that the line number is maintained by counting newline characters (automatically by `flex`; see `yylineno`).

Name	Lexeme Description
TYPE	Type name: `void`, `char`, `int`, or `float`
CONST	Keyword `const`
STRUCT	Keyword `struct`
FOR	Keyword `for`
WHILE	Keyword `while`
DO	Keyword `do`
IF	Keyword `if`
ELSE	Keyword `else`
BREAK	Keyword `break`
CONTINUE	Keyword `continue`
RETURN	Keyword `return`
IDENT	Identifier: sequence of letters, underscores, or digits that does not start with a digit, and is not a keyword
INTCONST	Integer constant: one or more digits
REALCONST	Real constant: one or more digits, optional decimal part, optional exponent part, not an integer constant; OR decimal part and optional exponent part
STRCONST	String constant: double quoted string, can have ‘`\"`’ inside
CHARCONST	Character constant: single quote, character other than `\` and `'`, single quote; OR single quote, `\`, character, single quote for escape sequences: `'\a'`, `'\b'`, `'\n'`, `'\r'`, `'\t'`, `'\\'`, `'\''`

Name	Lexeme
LPAR	`(`
RPAR	`)`
LBRACKET	`[`
RBRACKET	`]`
LBRACE	`{`
RBRACE	`}`
DOT	`.`
COMMA	`,`
SEMI	`;`
QUEST	`?`
COLON	`:`

Name	Lexeme
PLUS	`+`
MINUS	`-`
STAR	`*`
SLASH	`/`
MOD	`%`
TILDE	`~`
PIPE	`|`
AMP	`&`
BANG	`!`
DPIPE	`||`
DAMP	`&&`

Name	Lexeme
ASSIGN	`=`
PLUSASSIGN	`+=`
MINUSASSIGN	`-=`
STARASSIGN	`*=`
SLASHASSIGN	`/=`
INCR	`++`
DECR	`--`
EQUALS	`==`
NEQUAL	`!=`
GT	`>`
GE	`>=`
LT	`<`
LE	`<=`

Table 1: Token names to be displayed by your lexer.

3.1 C style comments

A “C style” comment should be treated the same as a block of whitespace. These comments begin with the character sequence `/*` and end with the character sequence `*/`. Note that these comments do not “nest”. Also note that the starting sequence and ending sequence must appear in the same input file. An error message should be given for an unclosed comment, that gives the location of the start of the comment. Check the C/C++ standards, but I believe the rule for this comment is:

> If we are not currently inside a comment or a string constant, then text between `/*` and the next `*/` is a comment and should be ignored.

3.2 C++ style comments

A "C++ style" comment should be treated the same as a block of whitespace. These comments begin with the character sequence `//` and end with a newline character or end of file. I believe the rule for this comment is:

> If we are not currently inside a comment or a string constant, then text between `//` and the end of the line (or file, whichever comes first) is a comment and should be ignored.

3.3 Minimal implementation

At the very least, your lexer must read lines of the form

```
#include "filename"
#define identifier arbitrary text here all on one line
#undef identifier
#ifdef identifier
#ifndef identifier
#else
#endif
```

where `identifier` is a valid identifier as specified in Table 1. Your lexer may ignore these lines with an appropriate warning message, or you may implement these preprocessor features for extra credit (see below). Note: you do not need to handle include directives of the form

```
#include <filename>
```

To simplify your lexer, you may assume that lines containing preprocessor directives will not contain comments anywhere in the line.

3.4 Extra credit: `#include`

While processing input file `A`, if your lexer encounters a line

```
#include "B"
```

then the lexer should immediately begin processing file `B`. Once the end of file is reached on `B`, the lexer should continue processing file `A` where it left off. If there is an error opening file `B`, then an apprpropriate error message should be generated. Of course, file `B` may have its own `#include` directives, which should be followed also. You may set a reasonable limit to the include depth (say, 256) with an appropriate error message if this depth is exceeded.

For additional extra credit, your lexer should check for "include cycles" (for example, if file `A` includes file `B` which includes file `C` which includes file `A`) and print an appropriate error message when this occurs.

3.5 Extra credit: `#define` substitution

Your lexer maintains a set of defined preprocessor symbols. An input line of the form

```
#define identifier arbitrary text on one line
```

will define a symbol with name "`identifier`" (any legal C identifier may go here). As long as this preprocessor symbol is defined, the text "`identifier`" is replaced by "`arbitrary text on one line`". An error should be generated if the identifier is already in the set of preprocessor symbols. You may set a reasonable limit (say, 4096 characters) on the length of the replacement text, with an appropriate error message generated if this is exceeded. Note that the replacement text could contain some other identifier that should be replaced. You may set a reasonable limit (say, 256) on the maximum "substitution depth", with an appropriate error message generated if this depth is exceeded. An input line of the form

```
#undef identifier
```

causes the symbol with name "`identifer`" to be removed from the set of preprocessor symbols.

For additional extra credit, an appropriate error message should be displayed if an identifier causes a replacement cycle (e.g., replacement for `FOO` contains `BAR`, and replacement for `BAR` contains `FOO`).

3.6 `#ifdef`

Implement the `#ifdef`, `#ifndef`, `#else`, and `#endif` directives. These directives tell the lexer to process or ignore input based on whether a preprocessor symbol has been defined or not. Thus, to implement this, you must also implement `#define` and `#undef`, at least partially, so you can tell if an identifier is in the set of defined preprocessor symbols.

The directive `#ifdef identifier` tells the lexer to process input if and only if `identifier` is a defined preprocessor symbol. Conversely, `#ifndef identifier` tells the lexer to process input if and only if `identifier` is *not* a defined processor symbol. This continues until either a matching `#else` (which is optional), which reverses the directive, or until a matching `#endif`, which causes processing to resume as it was. For example:

```
#define THING
#ifdef THING
  // this will be processed
#else
  // this will be ignored
#endif
  // Normal processing from here
```

There can be at most one `#else` directive for a given `#ifdef` or `#ifndef` directive. The `#ifdef`/`#ifndef`, `#else`, and `#endif` directives must all appear in the same file. Appropriate error messages should be generated for mismatched directives. For a more complete discussion, find a reference on the C preprocessor.

For additional extra credit, your lexer should allow nesting of `#ifdef` and `#ifndef` directives up to some reasonable limit (say, 256):

```
#ifdef FOO
#ifndef BAR
  // processed if FOO is defined, BAR is not
#endif
#else
#ifdef BAR
  // processed if FOO is not defined, BAR is
#endif
#endif
```

Note that you must check for mismatched directives, even inside a block that the preprocessor is discarding.

3.7 Other preprocessor directives

You are not required to implement, or even recognize, any other preprocessor directives.

4 Examples: minimal implementation

4.1 Input: `hello.c`

```
/*
  The "standard" hello world program written in C.
  Except we use #include "stdio.h" instead of #include <stdio.h>
*/

```

```
6  #include "stdio.h"
7
8  int main() // no arguments
9  {
10   return printf("Hello, world!\n"); // this is legal C
11 }
```

4.2 Output for `mycc -1 hello.c`

```
hello.c line 8 text 'int' token TYPE
hello.c line 8 text 'main' token IDENT
hello.c line 8 text '(' token LPAR
hello.c line 8 text ')' token RPAR
hello.c line 9 text '{' token LBRACE
hello.c line 10 text 'return' token RETURN
hello.c line 10 text 'printf' token IDENT
hello.c line 10 text '(' token LPAR
hello.c line 10 text '"Hello, world!\n"' token STRCONST
hello.c line 10 text ')' token RPAR
hello.c line 10 text ';' token SEMI
hello.c line 11 text '}' token RBRACE
```

4.3 Errors for `mycc -1 hello.c`

```
Warning: ignoring #include directive in hello.c line 6
```

4.4 Input: `tricky.txt`

```
/* Some tricky cases and also random ones // */ 3

// /* This is a C++ comment, not a C comment
4
*/
><>=<=!=
= == === ====
i
in
int
inte
integ
intege
integer
'c'
' '
'\n'
""
" "
"            "
"+ - * / ++ -- "
" /* evil 1 */ "
" // evil 2  "
// " this is a comment

/*  Multi
    line
```

```
28     comment
29  */
30
31  /*/   Still in the comment! */
32
33  /**/    5
34  /***/   6
35  /****/  7
36  /* /* /* /* */ 8 */
37
38  /*
39     Unclosed comment
40     Error message should indicate where it starts
```

4.5 Output for `mycc -1 tricky.txt`

```
tricky.txt line 1 text '3' token INTCONST
tricky.txt line 4 text '4' token INTCONST
tricky.txt line 5 text '*' token STAR
tricky.txt line 5 text '/' token SLASH
tricky.txt line 6 text '>' token GT
tricky.txt line 6 text '<' token LT
tricky.txt line 6 text '>=' token GE
tricky.txt line 6 text '<=' token LE
tricky.txt line 6 text '!=' token NEQUAL
tricky.txt line 7 text '=' token ASSIGN
tricky.txt line 7 text '==' token EQUALS
tricky.txt line 7 text '==' token EQUALS
tricky.txt line 7 text '=' token ASSIGN
tricky.txt line 7 text '==' token EQUALS
tricky.txt line 7 text '==' token EQUALS
tricky.txt line 8 text 'i' token IDENT
tricky.txt line 9 text 'in' token IDENT
tricky.txt line 10 text 'int' token TYPE
tricky.txt line 11 text 'inte' token IDENT
tricky.txt line 12 text 'integ' token IDENT
tricky.txt line 13 text 'intege' token IDENT
tricky.txt line 14 text 'integer' token IDENT
tricky.txt line 15 text ''c'' token CHARCONST
tricky.txt line 16 text '' '' token CHARCONST
tricky.txt line 17 text ''\n'' token CHARCONST
tricky.txt line 18 text '""' token STRCONST
tricky.txt line 19 text '" "' token STRCONST
tricky.txt line 20 text '"          "' token STRCONST
tricky.txt line 21 text '"+ - * / ++ -- "' token STRCONST
tricky.txt line 22 text '" /* evil 1 */ "' token STRCONST
tricky.txt line 23 text '" // evil 2 "' token STRCONST
tricky.txt line 33 text '5' token INTCONST
tricky.txt line 34 text '6' token INTCONST
tricky.txt line 35 text '7' token INTCONST
tricky.txt line 36 text '8' token INTCONST
tricky.txt line 36 text '*' token STAR
tricky.txt line 36 text '/' token SLASH
```

4.6 Errors for `mycc -1 tricky.txt`

```
Error near tricky.txt line 38
       Unclosed comment
```

5 Examples: all extra credit implemented

5.1 Input files for `#include` example

```
// File: a.h
int A;
#include "b.h"
int AA;
```

```
// File: b.h
int B;
#include "c.h"
int BB;
```

```
// File: c.h
int C;
```

```
// File: d.c
#include "a.h"
int D()
{
  return A + B + C;
}
#include "e.h"
// The above causes an include cycle error, so there is no
// guarantee that the compiler will continue past here.
int BYE;
```

```
// File: e.h
#include "f.h"
```

```
// File: f.h
#include "e.h"
```

5.2 Output for `mycc -1 d.c`

```
a.h line 2 text 'int' token TYPE
a.h line 2 text 'A' token IDENT
a.h line 2 text ';' token SEMI
b.h line 2 text 'int' token TYPE
b.h line 2 text 'B' token IDENT
b.h line 2 text ';' token SEMI
c.h line 2 text 'int' token TYPE
c.h line 2 text 'C' token IDENT
c.h line 2 text ';' token SEMI
b.h line 4 text 'int' token TYPE
b.h line 4 text 'BB' token IDENT
b.h line 4 text ';' token SEMI
a.h line 4 text 'int' token TYPE
```

```
a.h line 4 text 'AA' token IDENT
a.h line 4 text ';' token SEMI
d.c line 3 text 'int' token TYPE
d.c line 3 text 'D' token IDENT
d.c line 3 text '(' token LPAR
d.c line 3 text ')' token RPAR
d.c line 4 text '{' token LBRACE
d.c line 5 text 'return' token RETURN
d.c line 5 text 'A' token IDENT
d.c line 5 text '+' token PLUS
d.c line 5 text 'B' token IDENT
d.c line 5 text '+' token PLUS
d.c line 5 text 'C' token IDENT
d.c line 5 text ';' token SEMI
d.c line 6 text '}' token RBRACE
d.c line 10 text 'int' token TYPE
d.c line 10 text 'BYE' token IDENT
d.c line 10 text ';' token SEMI
```

5.3 Errors for `mycc -1 d.c`

```
Error near f.h line 2 text '"e.h"'
       #include cycle:
       File e.h includes f.h
       File f.h includes e.h
Error near f.h line 2 text '"e.h"'
       Couldn't open included file e.h
```

5.4 `#define` substitution example

```
// File: defines.c

/* Not too fancy substitution */

#define PI 3.14159
#define N 100

PI + N;

#undef M
// There is no M to undefine. This is OK but a warning is also OK.
M;
#define M 6
M;
#define M 7
// Should give a warning because we're redefining M
M;

/* Recursive substitution */

#define TWOPI (2.0 * PI)
#define NP1 (N + 1)
#define NP1SQ NP1 * NP1

```

```
25  TWOPI + NP1SQ;
26
27  #undef N
28
29  NP1SQ; // Still ok, but uses N instead of 100
30
31  /* Substitution cycle */
32
33  #define BAD hi THING
34  #define THING hi BAD
35
36  BAD;
```

5.5 Output for `mycc -1 defines.c`

```
defines.c line 8 text '3.14159' token REALCONST
defines.c line 8 text '+' token PLUS
defines.c line 8 text '100' token INTCONST
defines.c line 8 text ';' token SEMI
defines.c line 12 text 'M' token IDENT
defines.c line 12 text ';' token SEMI
defines.c line 14 text '6' token INTCONST
defines.c line 14 text ';' token SEMI
defines.c line 17 text '7' token INTCONST
defines.c line 17 text ';' token SEMI
defines.c line 25 text '(' token LPAR
defines.c line 25 text '2.0' token REALCONST
defines.c line 25 text '*' token STAR
defines.c line 25 text '3.14159' token REALCONST
defines.c line 25 text ')' token RPAR
defines.c line 25 text '+' token PLUS
defines.c line 25 text '(' token LPAR
defines.c line 25 text '100' token INTCONST
defines.c line 25 text '+' token PLUS
defines.c line 25 text '1' token INTCONST
defines.c line 25 text ')' token RPAR
defines.c line 25 text '*' token STAR
defines.c line 25 text '(' token LPAR
defines.c line 25 text '100' token INTCONST
defines.c line 25 text '+' token PLUS
defines.c line 25 text '1' token INTCONST
defines.c line 25 text ')' token RPAR
defines.c line 25 text ';' token SEMI
defines.c line 29 text '(' token LPAR
defines.c line 29 text 'N' token IDENT
defines.c line 29 text '+' token PLUS
defines.c line 29 text '1' token INTCONST
defines.c line 29 text ')' token RPAR
defines.c line 29 text '*' token STAR
defines.c line 29 text '(' token LPAR
defines.c line 29 text 'N' token IDENT
defines.c line 29 text '+' token PLUS
defines.c line 29 text '1' token INTCONST
defines.c line 29 text ')' token RPAR
defines.c line 29 text ';' token SEMI
defines.c line 36 text 'hi' token IDENT
defines.c line 36 text 'hi' token IDENT
defines.c line 36 text ';' token SEMI
```

5.6 Errors for `mycc -1 defines.c`

```
Error near defines.c line 15 text ' 7'
       re-defining preprocessor symbol M
Error near defines.c line 36 text 'BAD'
       #define substitution cycle:
       BAD macro contains THING
       THING macro contains BAD
```

5.7 `#ifdef` example

```c
// File: ifdefs1.c

#define A
#define B
#define C

#ifdef A
#ifdef B
#ifdef C
     7
#else
     6
#endif
#else
#ifdef C
     5
#else
     4
#endif
#endif
#else
#ifdef B
#ifdef C
     3
#else
     2
#endif
#else
#ifdef C
     1
#else
     0
#endif
#endif
#endif
```

```c
// File: ifdefs2.c

```

```
 3  #define F 1
 4
 5  #ifndef F
 6  #define G
 7  #endif
 8
 9  #ifdef G
10    8
11  #else
12    9
13  #endif
14
15  #ifdef ERROR
16    X
17  #else
18    Y
19  #else
20    Z
21  #endif
```

5.8 Output for `mycc -1 ifdefs1.c ifdefs2.c`

```
ifdefs1.c line 10 text '7' token INTCONST
ifdefs2.c line 12 text '9' token INTCONST
ifdefs2.c line 18 text 'Y' token IDENT
ifdefs2.c line 20 text 'Z' token IDENT
```

5.9 Errors for `mycc -1 ifdefs1.c ifdefs2.c`

```
Error near ifdefs2.c line 19 text '#else'
       second #else (first on line 17), ignoring
```

6 Grading

Points		**Description**
20		**Documentation**
	5	`README.txt`
		How to build the compiler and documentation. Updated to show which part 1 features are implemented.
	15	`developers.pdf`
		New section for part 1, that explains the purpose of each source file, the main data structures used, and gives a high-level overview of how the various features are implemented.
10		**Ease of building**
		How easy was it for the graders to build your compiler and documentation from the `README` file. For full credit, simply running "`make`" should build both the documentation and the compiler executable, and running "`make clean`" should remove all generated files.
10		**Still works in mode 0**

5		**The `-o` switch works**
40		**Basic lexer**
	6	Correct line numbers and output format
	4	Keywords
	4	Types and identifiers
	6	Integer, real, string, character constants
	4	Invalid characters
	6	Comments
	10	Symbols
5		**Extra credit: Multiple input files**
		If more than one input file is specified on the command line, then the input files are processed, one at a time, in that order.
5–15		**`#include`, one of:**
	5	Ignored with a warning
	10	Extra credit: works, does not check cycles
	15	Extra credit: works, cycles are discovered
5–15		**`#define`, one of:**
	5	Ignored with a warning
	10	Extra credit: defines are substituted, does not check cycles
	15	Extra credit: defines are substituted, cycles are discovered
5–15		**`#ifdef`, one of:**
	5	Ignored with a warning
	10	Extra credit: implemented, not nestable
	15	Extra credit: implemented, nestable
100		**Total for students in 440 (max points is 120)**
120		**Total for students in 540**

7 Submission

Be sure to commit your source code and documentation to your git repository, and to upload (push) those commits to the server so that we may grade them. In Canvas, indicate if you would like us to re-grade your part 0 submission for reduced credit, or only grade part 1. If nothing is indicated, we will grade part 1 only.