COM S 440/540 Project part 1

Create a lexer for C*

1 Overview

When executed with a mode of 1, your compiler should read the specified input file and divide it into tokens; 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, showing the file name, line number, and text corresponding to the token. More details (including the precise output and error formats, and the definitions for the tokens) are given below. The input file may be a C header file, a C source file, or an arbitrary text file. There are opportunities for extra credit.

All implementation must be C, C++, Java, or Python. Instructor permission is required to use anything beyond the standard libraries for these languages. Submissions will be graded on `pyrite.cs.iastate.edu`, and therefore must build and run correctly there.

Students are *strongly* encouraged to encapsulate the functionality of this phase of the compiler, so that later parts of the project can easily examine and consume tokens from an input stream.

2 Tokens in our subset of C

Your lexer must recognize the tokens given in Table 1. A useful list of integer constants for tokens may be found in `tokens.h`, distributed with the materials for this part of the project. Note that single-character symbols use their ASCII codes; all other tokens have integer values above 300. Whitespace (spaces, tabs, and carriage returns) serves only to separate tokens, and should otherwise be discarded. Any characters that are not part of a lexeme, such as `$`, should generate an error message.

You may notice that some of the C keywords and operators are missing. Students are welcome to implement additional language features if desired, but any extra keywords or operators must be part ot the C standard.

2.1 C style comments

C-style comments, beginning with `/*` and ending with `*/`, should be discarded and treated the same as a space. A comment with no matching `*/` is closed at the end of file, and an error message (indicating the starting point of the comment) should be displayed for this case. The formal rule for C-style comments is:

> When not currently inside a comment or string literal, the text `/*` indicates the start of a comment, and all text is ignored until the next `*/` or the end of file.

2.2 C++ style comments

C++-style comments, beginning with `//` and ending with a newline or end of file, should be discarded and treated the same as a newline character. No error message is necessary if the comment is terminated by the end of file. The formal rule for C++-style comments is:

> When not currently inside a comment or string literal, the text `//` indicates the start of a comment, and all text is ignored until the next newline character or the end of file.

*Actually, it's a subset of C, but students are always free to implement more of the C standard.

Single character symbols

Value	Lexeme	Value	Lexeme	Value	Lexeme
33	`!`	45	`-`	63	`?`
37	`%`	46	`.`	91	`[`
38	`&`	47	`/`	93	`]`
40	`(`	58	`:`	123	`{`
41	`)`	59	`;`	124	`|`
42	`*`	60	`<`	125	`}`
43	`+`	61	`=`	126	`~`
44	`,`	62	`>`		

Operators with two characters

Value	Lexeme	Value	Lexeme	Value	Lexeme
351	`==`	355	`++`	361	`+=`
352	`!=`	356	`--`	362	`-=`
353	`>=`	357	`||`	363	`*=`
354	`<=`	358	`&&`	364	`/=`

Keywords

Value	Lexeme	Value	Lexeme	Value	Lexeme
401	`const`	406	`if`	410	`return`
402	`struct`	407	`else`	411	`switch`
403	`for`	408	`break`	412	`case`
404	`while`	409	`continue`	413	`default`
405	`do`				

Lexemes with attributes

Value	Lexeme	Definition
301	type	`void` or `char` or `int` or `float`
302	character literal	a character enclosed in single quotes; escape sequences are extra credit
303	integer literal	a sequence of one or more digits
304	real literal	two sequences each of one or more digits, separated by `.`
305	string literal	a sequence of zero or more characters (including escape sequences, except for `\\` and `\"`, which are extra credit) enclosed in double quotes
306	identifier	a letter or underscore, followed by zero or more letters, underscores, or digits, that is not a keyword or type

Table 1: Tokens to be recognized by the lexer.

2.3 Literals

The token definitions for integers, reals, characters, and strings are simpler than the C standard. Some extensions to these may be implemented for extra credit; see below.

3 Formatting

3.1 Compiler output

For mode 1, the output stream should contain a line for each token, exactly of the form

`File` *filename* `Line` *line number* `Token` *token number* `Text` *lexeme*

where

- *filename* is the name of the input file that contains the token,
- *line number* is the line number of the input file that contains the token,
- *token number* is the integer corresponding to the token, given in `tokens.h` and Table 1, and
- *lexeme* is the matched text that produced the token.

Nothing else should be written to the output stream.

The output stream should be written to a file, based on the input file name given on the command line, with extension replaced by `lexer`. For example, an input file named `infile.c` or `infile.h` should produce an output file named `infile.lexer` in the current working directory.

3.2 Error messages

Error messages generated by the lexer must be written to standard error, and have exactly the form

> `Lexer error in file` *filename* `line` *line number* `at text` *lexeme*
> *Description*

The *Description* should appear on the next line(s) and give a helpful description of the error. The error stream must be empty if no errors are generated. Students are *strongly* encouraged to define their own switches to display any additional information, to help with debugging.

Example inputs, outputs, and errors are given below. Additionally, students are encouraged to check the output and error messages given for the example test files, for all parts of the project. For test files with errors, code is considered correct if the first error message occurs at the expected input line and text, and the error messages describe the same error (as determined by a human).

4 Extra features

4.1 Character literals

In addition to ordinary character literals, e.g., `' '`, your lexer should allow the escape sequences `'\a'`, `'\b'`, `'\n'`, `'\r'`, `'\t'`, `'\\'`, and `'\''`.

4.2 Real literals

In addition to numbers with fractional portions, e.g., `3.14159`, your lexer should allow *exponents*, of the form `e` or `E` followed by an integer with optional sign. If an exponent is present, then the fractional part is optional. For example, the following should be recognized as real literals:

```
31.41592
6.02214e23
1.60217653E-19
1e+100
```

4.3 String literals

Allow escape sequences `\\` and `\"`, in addition to the others, inside string literals. For example, the following should be recognized as string literals:

```
"Hello, world!\n"
"I said, \"Hello, world!\""
"This is evil \\"
```

4.4 Size limits for lexemes

Your lexer should generate an appropriate error message (and terminate, if you wish) if the following size limits are exceeded.

- Integer literals: 48 characters
- Real literals: 48 characters
- Identifiers: 48 characters
- String literals: 1024 characters

Note that there is **no limit** for the length of a line, the length of a comment, or the length of an input file.

4.5 Output file on error

For efficiency, your lexer should make one pass through the input file, generating output for each token as it is encountered. If any error message is generated, then your lexer should remove the output file (it will be ignored anyway during grading).

4.6 `#include` directives

The `include` directive has the form

`# include` *filename*

where *filename* is a string literal. We will only test for include files in double quotes. In other words, we will test for directives of the form

```
#   include   "stack.h"
# include "../mylib/mylib.h"
#include"/usr/include/gcc/includes/stdio.h"
```

where the files are either absolute pathnames or pathnames relative to the current working directory. We will not test for "system" includes of the form

```
#include <stdio.h>
```

although you are free to implement that if you wish.

Each include directive switches the token input stream to the indicated file, with an appropriate error message generated if the file cannot be opened. When the token stream from that file is exhausted, it reverts back to the original file. Of course, the included file may itself have include directives, which should be followed recursively. You may set a reasonable limit to the include depth (say, 256) with an appropriate error message if this depth is exceeded.

5 Basic example

5.1 Input: `hello.c`

```
1  /*
2   *  The usual hello world program, but without
3   *  #include <stdio.h> because the include
4   *  directive is optional.
5   *
6   */
7
```

```
 8  int main()
 9  {
10    return printf("Hello, world!\n");
11  }
```

5.2 Output: `hello.lexer`

```
File hello.c Line     8 Token 301 Text int
File hello.c Line     8 Token 306 Text main
File hello.c Line     8 Token  40 Text (
File hello.c Line     8 Token  41 Text )
File hello.c Line     9 Token 123 Text {
File hello.c Line    10 Token 410 Text return
File hello.c Line    10 Token 306 Text printf
File hello.c Line    10 Token  40 Text (
File hello.c Line    10 Token 305 Text "Hello, world!\n"
File hello.c Line    10 Token  41 Text )
File hello.c Line    10 Token  59 Text ;
File hello.c Line    11 Token 125 Text }
```

6 The grading script

The grading script, `LexTest.sh`, is published on the course git repository, along with other materials for the project. Students are *strongly* encouraged to test their code using the script, especially on `pyrite.cs.iastate.edu`, before submission. The script should be invoked as follows.

- You will want to invoke the script in the directory containing the script, as the script makes assumptions about where files are located.
- The first argument to the script should be the command used to run your compiler, enclosed in double quotes (although, these are not necessary if you just need to run an executable, for example with `./mycc`).
- The remaining arguments to the script should be the input files you wish to test. These should be located in directory `INPUTS/`. Use wildcards as appropriate.
- The script will compare your output (or error message) with the instructor's output (or error message) for both a basic and an extra credit implementation, and give an appropriate summary. These files are located in directories `OUTPUTS_B/` and `OUTPUTS_X/`, respectively. The script will not run a test if there is no corresponding output or error file in `OUTPUTS_B/`.
- Students are encouraged, but not required, to use the same (or very similar) error messages to the instructor's, to make grading easier.

For example, running

```
./LexTest.sh "java -jar ./mycc.jar" INPUTS/tricky?.c
```

would test a compiler written in Java (with jar file in the current directory), on input files in directory `INPUTS` whose names match `tricky?.c`. Running

```
./LexTest.sh ~/ComS440/Part1/mycc INPUTS/*
```

will check all input files (with instructor-generated outputs in directory `OUTPUTS_B`), for a compiler executable located in directory `~/ComS440/Part1`.

7 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. You are encouraged to use `Makefile`s under the `Source/` and `Documentation/` directories so that running "`make`" will build everything and running "`make clean`" will remove all generated files.
5	**The executable works also in mode 0**
60	**Basic lexer**
10	Correct line numbers and output format
15	Keywords, types, identifiers
14	Integer, real, string, character literals
6	Comments
15	Symbols
30	**Extra features**
3	Character literals with escapes (c.f. Sec. 4.1)
3	Real literals with exponents (c.f. Sec. 4.2)
3	String literals with escapes (c.f. Sec. 4.3)
3	Errors for very long lexemes (c.f. Sec. 4.4)
3	Errors for invalid characters
3	Output file removed on error (c.f. Sec. 4.5)
12	`#include` directives (c.f. Sec. 4.6)
100	**Total for students in 440 (Max score is 120 points)**
110	**Total for students in 540**

8 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.