Programming Assignment 2
Due **Thursday, 25 August'16** at 11:59pm

1 Overview

The projects in this course will direct you to design and build a compiler for Cool. Each assignment will cover one component of the compiler: lexical analysis, parsing, semantic analysis, and code generation. Each assignment will ultimately result in a working compiler phase which can interface with other phases. You will be doing your projects in Java.

For this assignment, you are to write a lexical analyzer, also called a *scanner*, using a *lexical analyzer generator*. The particular Java based tool that we will be using is called is called ANTLR. You will describe the set of tokens for Cool in an appropriate input format, and the analyzer generator will generate the actual code (Java) for recognizing tokens in Cool programs.

You will work individually for this assignment.

2 Files and Directories

To get started, create a directory where you want to do the assignment and extract lexer.tar.gz in it.

The files that you need to modify are:

- CoolLexer.g4
  This file contains a skeleton for a lexical description for Cool. There are comments indicating where you need to fill in code, but this is not necessarily a complete guide. Part of the assignment is for you to make sure that you have a correct and working lexer. Except for the sections indicated, you are welcome to make modifications to our skeleton. You can actually build a scanner with the skeleton description, but it does not do much. You should read the ANTLR documentation to figure out what this description does. Any auxiliary routines that you wish to write should be added directly to this file in the appropriate section (see comments in the file).

- test_cases/helloworld.cl
  This file contains some sample input to be scanned. It does not exercise all of the lexical specification, but it is nevertheless an interesting test. It is not a good test to start with, nor does it provide adequate testing of your scanner. Part of your assignment is to come up with good testing inputs and a testing strategy. (Don't take this lightly — good test input is difficult to create, and forgetting to test something is the most likely cause of lost points during grading). You are free to modify this file with tests that you think adequately exercise your scanner. You have to add more test files (at least 5) in the `test_cases` directory. Make sure that your test files have the extension `.cl`. The given test file is similar to a real Cool program, but your tests need not be. You may keep as much or as little of our test as you like.

- README.md
  You should edit this file to include the write-up for your project. You should explain design decisions, why your code is correct, and why your test cases are adequate. It is a part of the assignment to clearly and concisely explain things in text as well as to comment your code.

  Although these files are incomplete as given, the lexer does compile and run (make).

3 Scanner Results

You should follow the specification of the lexical structure of Cool given in Section 10 and Figure 1 of the Cool manual. Your scanner should be robust—it should work for any conceivable input. For example, you must handle errors such as an EOF occurring in the middle of a string or comment, as well as string constants that are too long. These are just some of the errors that can occur; see the manual for the rest.

You must make some provision for graceful termination if a fatal error occurs. Core dumps or uncaught exceptions are unacceptable.

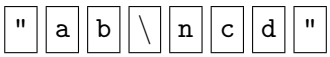
3.1 Error Handling

All errors should be passed along to the parser. Your lexer should not print anything. Errors are communicated to the parser by reporting the error token using `reportError("Message")`. There are several requirements for reporting and recovering from lexical errors:

- When an invalid character (one that can't begin any token) is encountered, a string containing just that character should be returned as the error string. Resume lexing at the following character.
- If a string contains an unescaped newline, report that error as ``` ``Unterminated string constant'' ``` and resume lexing at the beginning of the next line—we assume the programmer simply forgot the close-quote.
- When a string is too long, report the error as ``` ``String constant too long'' ``` in the error string in the **ERROR** token. If the string contains invalid characters (i.e., the null character), report this as ``` ``String contains null character'' ```. In either case, lexing should resume after the end of the string. The end of the string is defined as either
  1. the beginning of the next line if an unescaped newline occurs after these errors are encountered; or
  2. after the closing " otherwise.
- If a comment remains open when EOF is encountered, report this error with the message ``` ``EOF in comment'' ```. Do *not* tokenize the comment's contents simply because the terminator is missing. Similarly for strings, if an EOF is encountered before the close-quote, report this error as ``` ``EOF in string constant'' ```.
- If you see "`*)`" outside a comment, report this error as ``` ``Unmatched *)'' ```, rather than tokenzing it as `*` and `)`.

3.2 Strings

Your scanner should convert escape characters in string constants to their correct values. For example, if the programmer types these eight characters:



your scanner would return the token **STR_CONST** whose semantic value is these 5 characters:

where \n represents the literal ASCII character for newline.

Following specification on page 15 of the Cool manual, you must return an error for a string containing the literal null character. However, the sequence of two characters

\ 0

is allowed but should be converted to the one character

0 .

4 Testing the Scanner

First, you need to compile your grammar and generate the required Java files. We have done the hardwork for you and are providing you with a *Makefile*. Just go the root of the assignment directory and type `make` on the terminal. This will compile your grammar and generate the required Java files (and compile them too).

You can run your generated lexer by the following command:

```
./lexer <test-file>
```

The file `lexer` is present in the director **src/grammar/**. The file will output the list of tokens generated for the given file/s.

5 What to Turn In

You need to submit your grammar file (src/grammar/CoolLexer.g4), a readme file (README.md), and a bunch of test cases (src/test_cases/*.cl) as a **tar ball**. To generate the **tar ball** go to the starting directory of the assignment directory tree and type the following:

```
./submit <roll-number>
```

This will generate a file by the name "**Asn0<your-roll-number>.tar.gz**". You need to submit this file in the Google Classroom page for this course.

The last submission you do will be the one graded. The late penalty announced earlier (10% penalty per day for a maximum of one week) will be applied. The burden of convincing us that you understand the material is on you. Obtuse code, output, and write-ups will have a negative effect on your grade. Take the extra time to clearly (and concisely!) explain your results.