# Programming Assignment II
# Due Tuesday, March 15 (24 Esfand), 2022 before 10:00 am

## 1 Overview

Programming assignments II–V 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 have to do your projects in C++.

For this assignment, you are to write a lexical analyzer, also called a *scanner*, using a *lexical analyzer generator*. (The C++ tool is called flex.) You will describe the set of tokens for Cool in an appropriate input format, and the analyzer generator will generate the actual C++-code for recognizing tokens in Cool programs.

On-line documentation for all the tools needed for the project will be made available on the homepage of this course. This includes manuals for flex (used in this assignment), the documentation for bison (used in the next assignment), as well as the manual for the spim simulator.

You must work individually for this assignment.

## 2 Files and Directories

To get started, create a directory where you want to do the assignment and execute the following command *in that directory*:

```
make -f /usr/class/assignments/PA2/Makefile
```

The files that you will need to modify are:

- cool.flex
  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 flex manual to figure out what this description does do. 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.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 should modify this file with tests that you think adequately exercise your scanner. Our test.cl 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
  This file contains detailed instructions for the assignment as well as a number of useful tips. You should also 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 part of the assignment to clearly and concisely explain things in text as well as to comment your code.

# 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 returning a special error token called **ERROR**. (Note, you should ignore the token called **error** [in lowercase] for this assignment; it is used by the parser in PA3.) 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 String Table

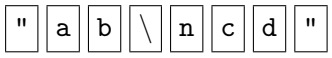
Programs tend to have many occurrences of the same lexeme. For example, an identifier is generally referred to more than once in a program (or else it isn't very useful!). To save space and time, a common compiler practice is to store lexemes in a *string table*. We provide a string table implementation for C++. See the following sections for the details.

There is an issue in deciding how to handle the special identifiers for the basic classes (**Object**, **Int**, **Bool**, **String**), `SELF_TYPE`, and `self`. However, this issue doesn't actually come up until later phases of the compiler—the scanner should treat the special identifiers exactly like any other identifier.

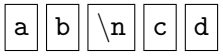
Do *not* test whether integer literals fit within the representation specified in the Cool manual—simply create a Symbol with the entire literal's text as its contents, regardless of its length.

### 3.3 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  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


is allowed but should be converted to the one character


.

### 3.4 Other Notes

Your scanner should maintain the variable **curr_lineno** that indicates which line in the source text is currently being scanned. This feature will aid the parser in printing useful error messages.

You should ignore the token **LET_STMT**. It is used only by the parser (PA3). Finally, note that if the lexical specification is incomplete (some input has no regular expression that matches), then the scanners generated by flex do undesirable things. *Make sure your specification is complete.*

## 4 Notes for the C++

- Each call on the scanner returns the next token and lexeme from the input. The value returned by the function **cool_yylex** is an integer code representing the syntactic category (e.g., integer literal, semicolon, `if` keyword, etc.). The codes for all tokens are defined in the file cool-parse.h. The second component, the semantic value or lexeme, is placed in the global union **cool_yylval**, which is of type YYSTYPE. The type YYSTYPE is also defined in cool-parse.h. The tokens for single character symbols (e.g., ";" and ",") are represented just by the integer (ASCII) value of the character itself. All of the single character tokens are listed in the grammar for Cool in the Cool manual.

- For class identifiers, object identifiers, integers, and strings, the semantic value should be a **Symbol** stored in the field **cool_yylval.symbol**. For boolean constants, the semantic value is stored in the field **cool_yylval.boolean**. Except for errors (see below), the lexemes for the other tokens do not carry any interesting information.
- We provide you with a string table implementation, which is discussed in detail in *A Tour of the Cool Support Code* and in documentation in the code. For the moment, you only need to know that the type of string table entries is **Symbol**.
- When a lexical error is encountered, the routine **cool_yylex** should return the token **ERROR**. The semantic value is the string representing the error message, which is stored in the field **cool_yylval.error_msg** (note that this field is an ordinary string, not a symbol). See the previous section for information on what to put in error messages.

## 5 Testing the Scanner

There are at least two ways that you can test your scanner. The first way is to generate sample inputs and run them using lexer, which prints out the line number and the lexeme of every token recognized by your scanner. The other way, when you think your scanner is working, is to try running mycoolc to invoke your lexer together with all other compiler phases (which we provide). This will be a complete Cool compiler that you can try on any test programs.

## 6 What to Turn In

When you are ready to turn in the assignment, type make submit-clean in the directory where you have prepared your assignment. This action will remove all the unnecessary files, such as object files, class files, core dumps, Emacs autosave files, etc.

The last submission you do will be the one graded. Each submission overwrites the previous one.Take the extra time to clearly (and concisely!) explain your results.