Programming Assignment II
Due Wednesday, September 18

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. Your programs will be written 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 Cool programs.

The manual for flex is on our home page.

2 Files and Directories

To get started, create a directory where you want to do the assignment and execute one of the following commands in that directory.

```
gmake -f ~soffa/public/html/cs2210/cool/assignments/PA2/Makefile
```

This command will copy a number of files to your directory. Some of the files will be copied read-only (using symbolic links). You should not edit these files. In fact, if you make and modify private copies of these files, you may find it impossible to complete the assignment. See the instructions in the README file. The files that you will need to modify are:

- **cool.flex**
  This file contains a skeleton for a lexical description for Cool. You can actually build a scanner with this 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.

  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. Part of your assignment is to comment your code.
Although these files are incomplete, the lexer does compile and run. There are a number of useful tips in the README file.

You should use any of the elements (linux) machines in the CS Department.

3 Scanner Results

You should follow the specification of the lexical structure of Cool given in Section 10 and Figure 1 of the CoolAid. 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.

Programs tend to have many occurrences of the same lexeme. For example, an identifier generally is 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.

All errors will be passed along to the parser, which is better equipped to handle them. The Cool parser knows about a special error token called ERROR. When an invalid character is encountered, that character and any invalid characters that follow should be gathered together into a string until the lexer finds a character that can begin a new token. This string will be returned as the error message. For errors besides strings of invalid characters (e.g., a string constant that is too long, or an end-of-file inside of a comment) it is sufficient to return an informative error message (e.g., “String constant too long” or “EOF in comment”). Make sure that the error message is informative so that we can understand what you did. The following sections clarify how to actually return the error message in C++.

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.

Finally, if the lexical specification is incomplete (some input has no regular expression that matches) then the generated scanner will invoke a default action on unmatched strings. The default action simply copies the string to the console. Your final scanner should have no default actions. Note that default actions are very bad for mycoolc, which works by piping output from one compiler phase to the next; any extra output will cause errors in downstream phases.

4 Notes

- Your scanner should maintain the global 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.

- Each call on the scanner returns the next token and lexeme from the input. The value returned by the function cool_yy lex is an integer code representing the syntactic category: whether it is an integer literal, semicolon, the 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_yyval, which is of type YYSTYPE. The type YYSTYPE is also defined in cool-parse.h. The tokens for single character symbols (e.g., “;” and “,”, among others) are represented just by the integer value of the character itself. All of the single character tokens are listed in the grammar for Cool in the CoolAid.
• 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 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 previous section
  for information on how to construct error messages.

5 Testing the Scanner

There are 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. When you think your scanner is working, you should 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 the sample programs and your program from Assignment I.

6 What to Turn in

When you are ready to turn in the assignment, create a directory with your name on it. Copy the 3 files
(`cool.flex`, `test.cl`) and `test.output` (the output of running your program on `test.cl`) to this directory. Then
copy the directory to

```
/afs/cs.pitt.edu/ soffa/public/cs2210/PA2
```