Programming Assignment II
Due Tuesday, September 26

1 Overview

Programming assignments II–V will lead 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.

For this assignment you are to write a lexical analyzer, also called a *scanner*, using the tool flex. You will describe the set of tokens for Cool in flex input format. If you have not done so already, you should consider buying the reader available for sale from Copy Central on Euclid Ave. In addition to having manual pages for flex and gmake, the reader contains the bison documentation, which will be used in the next assignment.

You must work in a group for this assignment (where a group consists of from one to three people), and in order to turn in the assignment, you must register the group. To register your group, send a message to cs164@cory with the subject PA2 group, and the message consisting of the login names of the people in your group, one name per line, and nothing else. Each member of the group will receive a confirmation via email.

2 Files and Directories

To get started, you should type

```
gmake -f ~cs164/assignments/PA2/Makefile
```

in a directory where you want to do the assignment. 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 token start (no pun intended) at a flex description for Cool. You can actually build a scanner with this description but it does not do much. You should read the man pages for flex to figure out what this description does do. Any auxiliary C++ routines that you wish to write should be added directly to the cool.flex file after the last %%.

- 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 testing strategy.

  You should modify this file with tests that you think adequately exercise your scanner. Our test.cl is actually close 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. 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.

Although these files are incomplete, the two program files do compile and run. There are a number of useful tips on using flex in the README file.

All of the software supplied with this assignment is supported on both the HP and DEC machines. However, if you switch platforms be sure to run gmake clean to remove files compiled for the other architecture.

Important: You should make sure to place cs164/bin at the beginning of your path variable to make sure the executables used are the ones the assignments are designed for. To do this, add the line

```
setenv PATH ~cs164/bin:${PATH}
```

at the end of your .login file.

3 Scanner Results

You should follow the specification of the lexical structure of Cool given in the 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 are unacceptable.

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_yylex** 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_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 ",", among others) are represented just by the integer value of the character itself. All of the single character tokens are listed it the grammar for Cool in the CoolAid.

Programs tend to have many occurrences of the same lexemes. 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 you with a string table package, which is discussed in detail in *A Tour of the Cool Support Code* and documentation in the code. For the moment, we only need to know that the type of string table entries is **Symbol**.

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.

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. The routine **cool_yylex** should return the token **ERROR**. The semantic value is the string of illegal characters, which is stored in the field **cool_yylval.error_msg** (note that this field is an ordinary string, not a symbol). 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 set **cool_yylval.error_msg** to 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.

There is an issue in deciding how to handle the special identifiers for the basic classes, `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 a flex 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 stdout. 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 Testing the Scanner

There are two ways that you can test your scanner. The first is to generate sample inputs and run them using **lextest** with the **-v** option, which will print out the lexeme and line number of every token recognized by your scanner. When you think your scanner is working, you should try gmake parser. This command will link your scanner with the course Cool parser. Once you have successfully linked your scanner with the parser, `mycoolc` will be a complete Cool compiler that you can try on the sample programs and your program from Assignment I.

5 What to Turn in

When you are ready to turn in the assignment, type gmake turnin in the directory where you have prepared your assignment. This action will copy the three files of the assignment (cool.flex, README, test.cl) and test.output (the output of running your program on test.cl) to the reader's directory.

Doctoring the output that is sent is considered cheating; if you want to explain something, do it in the README file.

The last turnin you do will be the one graded. Each turnin overwrites the previous one. Remember that there is a 0.5% penalty per hour for late assignments and that no credit will be given for an assignment after a solution has been provided. 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.