Programming Assignment II
Due Monday, January 24, 2005

1 Overview

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

You may work in groups of two 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.*

```
gmake -f ~c22/cool_project/assignments/PA2/Makefile
```

This will copy a number of files to your directory. Some of the files will be copied read-only (using symbolic links). You should not modify these files in any way. 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. (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. You should also edit this file to include the write-up for your project. You should explain your testing strategy here. It is part of the assignment to clearly and concisely explain things in text as well as to comment your code.

Although these files as given are incomplete, the lexer does compile and run (gmake lexer). There are a number of useful tips in the README file.

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, string constants that are too long, etc.

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.

All errors will be passed along to the parser. The Cool parser knows about a special error token called **ERROR**, which is used to communicate errors from the lexer to the parser. There are several requirements for reporting and recoving from lexical errors:

- When an invalid character (one which 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.
- When a string is too long, or contains invalid characters, that should be reported. Lexing should resume after the end of the string.
- If a string contains an unescaped newline, report that, and resume lexing at the beginning of the next line – we assume the programmer simply forgot the close-quote.
- If a comment remains open when EOF is encountered, report that. Do *not* tokenize the comment's contents simply because the terminator is missing. (This applies to strings as well.)
- If you see "`*)`" outside a comment, report this as an unmatched comment terminator, rather than tokenzing it as `*` and `)`.
- 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.
- Error messages should be clear and informative (e.g. "String constant too long" or "EOF in comment").

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.

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.

Finally, note that if the lexical specification is incomplete (some input has no regular expression that matches) then the scanner generated by flex does undesirable things. Make sure your specification *is* complete.

4 Notes

- 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 (ASCII) 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 C string, not a symbol).

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. 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). Try compiling your program from assignment I.

6 What to Turn in

Name your test cases test*.cl. Create a tarball containing cool.flex, README and the test cases and email it to **c22@cs.northwestern.edu**