

# Programming Assignment 2

## Due Thursday, February 14

### 1 Overview

The remaining programming assignments 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, with the latter two components each taking up two assignments.

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.

Undergraduates may do this assignment individually or with another person. In rare cases, we may permit a group of three. To submit work as a group, place your finished project in the PA2 directory of one of the team members. Please include both group members' names at the top of all files being graded. Recall that you can give someone else permission to access the PA2 directory of your AFS volume by using the `fs` command:

```
/usr/afsws/bin/fs sa PA2 someoneelse write
```

### 2 Files and Directories

To get started, you should type

```
make -f /afs/cs.uwm.edu/users/classes/cs654/assignments/PA2/Makefile
```

in a directory where you want to do the assignment. We recommend the PA2 directory of your afs volume. 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 the very beginnings of 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 will hold your sample input for testing. It starts off almost empty. It should have both legal and illegal tokens. You should modify this file with tests that adequately exercise your scanner. In particular it should test every keyword in a variety of spellings (to ensure it permits keywords to be matched case insensitively). It should test different forms of numbers. It should test the rules about identifiers (case, underscores, digits, etc). It should test strings. It should test special characters permitted or not permitted in strings. It should test comments, and all kinds of whitespace. The test should be complicated enough to usually detect all bugs in scanners (both incorrectly rejecting something legal and incorrectly accepting something not legal). Approximately 30% of the grade for this assignment depends on the test case.

- 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 varieties of Solaris. I have installed libraries for MacOSX on Intel. I have some libraries for linux but they apparently depend on an old version of g++'s libraries. However, if you switch platforms be sure to run `make clean` to remove files compiled for the other architecture.

### 3 Scanner Results

You should follow the specification of the lexical structure of Cool given in the Section 10 of the CoolAid. Your scanner should be robust—it should work for any conceivable input. Core dumps are unacceptable. Use flex for all input reading. Do not call `yyinput()` yourself.

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. Make sure this value is properly incremented where a newline occurs (for example, escaped in a string). Do not increment it when other things are encountered (for example, form feeds or a backslash followed by an 'n').

When a string literal is encountered, the scanner should translate escapes (as well as removing the beginning and ending quotes). Thus the string literal `"\n"` should be converted into a single character string. Remember that the sequence of two characters `\n` is not the same as a newline character! It is simply a convention that this two character sequence is replaced by newline. The replacing is done by the compiler, in other words, you! You will not be able to use C's regular string routines in this part since the strings may include NUL characters. However, the string table does handle strings with NUL characters.

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 shown in the railroad diagrams for Cool in the CoolAid.

The `native` keyword is a little strange, as it is only legal in `basic.cl`. Outside of `basic.cl`, occurrences of `native` should be treated as errors (see below). To distinguish whether the scanner is in `basic.cl`, the global variable `current_basic_status` has been defined. It is an enumeration with possible values `Basic` and `NotBasic`. For this assignment, you will not need to set this variable; you will, however, need to use it to distinguish between legal and illegal instances of `native`.

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 *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`.

Make sure to put string literals in the **stringtable**, identifiers (both object and type identifiers) in **idtable** and integer literals in **inttable**. The only member function of these tables that you need to use is **add\_string(char\*,int)**; ignore everything else.

For class identifiers, object identifiers, integers and strings, the semantic value should be a **Symbol** stored in the field **cool\_yylval.symbol**. For boolean literals, 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 can be handled using the macro **LEX\_ERROR**. Call **LEX\_ERROR** with an informative error message (e.g., “Unterminated string literal”). The macro will store the semantic value of the error (e.g. the string of illegal characters) in the field **cool\_yylval.error\_msg** (note that this field is an ordinary string, not a symbol) and directly calls the parser’s error-reporting function (**yyerror()**). It does not cause **cool\_yylex** to return a token. A sequence of illegal characters is single lex error; do not report it as a sequence of separate lex errors. An unterminated string literal error token should consume all the tokens up to where it is missing its termination. Make sure your scanner doesn’t backtrack and try to re-scan the contents of the unterminated string literal.

There is an issue in deciding how to handle the special identifiers for the basic classes, 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 **make 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.

As reported above, approx. 30% of the assignment grade will be on your test suite. You should systematically test every situation. You must test all borderline and strange cases possible. For example, you must test what happens when a string literal is interrupted at the end of the file. (What should happen in this case?) By default, **emacs** prevents you from ending a file in the middle of a line. In order to do so, you will need to type the following:

```
(ESC):(setq require-final-newline nil)(RETURN)
```

This is just one of the errors that can occur.

For instance, you need to test all sorts of legal and illegal characters, including control characters. These can be input into a program in Emacs by typing Control-Q and then the special character. The following table gives C string equivalents for control characters:

C code	Control character
<code>\0</code>	Control-@
<code>\b</code>	Control-H
<code>\t</code>	Control-I
<code>\n</code>	Control-J
<code>\f</code>	Control-L
<code>\r</code>	Control-M
<code>\v</code>	Control-K

The `\v` doesn't occur in Cool strings, but Control-K is legal Cool whitespace.

## 5 What to Turn in

When you are ready to turn in the assignment, copy the three files of the assignment (`cool.flex`, `README`, `test.c`) to the PA2 directory of your AFS volume. Each file should list the names of all group members.

Only the appropriately named files in the PA2 directory will be graded. Make sure that the files you place there are the versions you wish to have graded. Remember that assignments are not accepted after the deadline. 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.