

# Programming Assignment 2

## Due Tuesday, 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 *coollex*. You will describe the set of tokens for Cool in JFlex input format.

### 2 Files and Directories

To get started, you should type

```
make -f $CLASSHOME/src/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.lex**  
This file contains the very beginnings of a *coollex* description for Cool. You can actually build a scanner with this description but it does not do much. You should read the documentation for JFlex to figure out what this description does do. Any auxiliary Cool routines that you wish to write should be added directly to the *Cool.lex* file with the ones already given to you, but our solution makes no use of any more helper routines.
- **README**  
You should edit this file to include the write-up for your project. You should explain design decisions, why your code is correct. 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 scanner can be generated, compiled and run.

### 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 without crashing with an uncaught exception.

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! The instructor found this easiest to do using “initial states.”

Each call on the scanner’s `yylex()` method returns the next token and lexeme from the input. You can see the code that the instructor has written to convert this input into an iterator interface. You will return an object of type `YYToken` as defined in `CoolTokens.scala`:

- Each keyword has a “case class” for it. For keyword `key`, execute `return KEY()`;

- Each single character operator or punctuation symbol should be returned wrapped in `YYCHAR(.)`. (As you can see, Extended Cool has character literals.)
- Each multiple-character literal has a “case class” for it defined in `CoolTokens.scala`.
- The end-of-file condition is already handled for you.
- Erroneous tokens should be matched and returned using `ERROR`.
- The `UNARY` token is a parser artifact: do not ever return a token with this type.

The keywords (and illegal keywords) are listed in the CoolAid. All of the literal operators 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.cool`. Outside of `basic.cool`, occurrences of `native` should be treated as errors (see below). To distinguish whether the scanner is in `basic.cool`, the Boolean attribute `in_basic_file` is defined. The `native` keyword should only be permitted if this variable is true; otherwise an `ERROR` token should be returned.

Cool also has a very large number of illegal keywords. The scanner should return an `ERROR` token indicating the problem if someone uses one of the illegal keywords.

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*. You will use Cool's built-in `Symbol` type for all identifiers, string literals and integer literals.

All errors will be passed along as token `ERROR` to the parser, which is better equipped to handle them. 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. 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.

Finally, if a JFlex specification is incomplete (some input has no regular expression that matches) then the generated scanner will crash with an exception fault. The skeleton specification we have given to you changes this default behavior to echo illegal characters to standard output. Your scanner is not properly defined unless this rule is never invoked. The `coollex` tool will let you know when this is the case!

## 4 Testing the Scanner

You can test your scanner by using the `test` target of the makefile. This will run your scanner on the file `test.cool`. The `test-b` target runs the scanner on the file assuming it substitutes for the file `basic.cool` (i.e., `native` is legal). You can also run the scanner “by hand” directly using the `scala` command:

```
scala -cp .:PA2.jar TestScanner [-b] filename
```

You can use your test case, our solution test case (released after Homework 1 is due), or any other test case you wish.

## 5 What to Turn in

When you are ready to turn in the assignment, make sure the two files of the assignment (`Cool.lex` and `README`) are in the PA2 directory of your AFS volume.

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.