CS164 Programming Language and Compilers **Spring 2016**

Programming Assignment II

Assigned: February 2, 2016 **Due:** February 11, 2016 at 11:59pm

1 Overview

Programming assignments II–V will direct you to design and build a compiler for Cool using Java. 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.

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

On-line documentation for all the tools needed for the project is available on the "Resources" page of the CS164 course web site. This includes manuals for jflex (used in this assignment), the documentation for java_cup (used in the next assignment), as well as the manual for the Mars simulator.

You must work in a group for this assignment (where a group consists of one or two people). The submit program will ask you to specify group members when you turn in your assignment.

2 Files and Directories

Everything you need is included in the archive file (PA2.zip) available from the course website. This package contains a number of files. Some of the files (Java files) should not be edited. In fact, if you modify these files, you may find it impossible to complete the assignment. See the instructions in the README file.

2.1 Files to modify

The files that you will need to modify are:

- cool.lex
  This file contains a skeleton for a lexical description for Cool. There are comments indicating where you need to fill in code, but this is not necessarily a complete guide. Part of the assignment is for you to make sure that you have a correct and working lexer. Except for the sections indicated, you are welcome to make modifications to our skeleton. You can actually build a scanner with the skeleton description, but it does not do much. You should read the jflex 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 as well as a number of useful tips. 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 as given, the lexer does compile and run.

2.2 Helper scripts

The assignment package includes several scripts to easily compile and run your lexer (and the reference implementation as well). The following is the list of commands you can run:

- To compile your lexer implementation: `ant lexer`
- To run your lexer on the file test.cl: `ant test`
- To run five examples in tests directory: `ant test-all`
- To run your lexer on an input file 'foo.cl': `python mylexer.py foo.cl`
- To compile a program 'foo.cl' with your lexer and the rest of the reference compiler: `python mycoolc.py foo.cl`
- To run the reference lexer on an input file 'foo.cl' : `python lexer.py foo.cl`
- To compile a program 'foo.cl' with the reference compiler and to generate 'foo.s': `python coolc.py foo.cl`
- To run the compiled program 'foo.s' using the MARS mips simulator: `python runmips.py foo.s`

2.3 Using your own machine

If you want to use your own machine for the development, you will need to install JDK 7, Apache Ant 1.7 (or newer), and Python 2.7. You can get the packages from the following links:

- JDK 7: `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`
- Apache Ant: `http://ant.apache.org/bindownload.cgi`
- Python 2.7.11: `https://www.python.org/downloads/`

After installing a package, please put the path to the bin folder of the installed package in your `PATH` variable. In other words, `java`, `javac`, `ant`, and `python` commands should run on the shell prompt without the exact path to the corresponding bin folder. You also need to set `JAVA_HOME` and `ANT_HOME` environment variables.

Mac and Linux. You can also use a package manager for your system, such as `apt-get` and `brew`, to install a necessary package. Just make sure that you are using the correct version. Package managers do not set `JAVA_HOME` and `ANT_HOME` environment variables for you. So, please make sure to set these variables.

Windows. Please check the following guides to setup a development environment on a Windows machine:

- *JDK installation*: `http://docs.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html`
- *Setting the JAVA_HOME Variable*: `https://confluence.atlassian.com/doc/setting-the-java_home-variable-in-windows-8895.html`
- *Python on Windows*: `https://docs.python.org/2/using/windows.html`
- *Installing Apache Ant*: `http://ant.apache.org/manual/install.html`

3 Scanner Results

You should follow the specification of the lexical structure of Cool given in Section 10 and Figure 1 of the Cool manual. 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. Uncaught exceptions are unacceptable.

3.1 Error Handling

All errors should be passed along to the parser. Your lexer should not print any error messages. Errors are communicated to the parser by returning a special error token called **ERROR**. (Note, you should ignore the token called **error** [in lowercase] for this assignment; it used by the parser in PA3.) There are several requirements for reporting and recovering from lexical errors:

- When an invalid character (one that 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.
- If a string contains an unescaped newline, report that error as `‘‘Unterminated string constant’’` and resume lexing at the beginning of the next line—we assume the programmer simply forgot the close-quote.
- When a string is too long, report the error as `‘‘String constant too long’’` in the error string in the **ERROR** token. If the string contains invalid characters (i.e., the null character), report this as `‘‘String contains null character’’`. In either case, lexing should resume after the end of the string. The end of the string is defined as either

    1. the beginning of the next line if an unescaped newline occurs after these errors are encountered; or
    2. after the closing " otherwise.

- If a comment remains open when EOF is encountered, report this error with the message `''EOF in comment''`. Do *not* tokenize the comment's contents simply because the terminator is missing. Similarly for strings, if an EOF is encountered before the close-quote, report this error as `''EOF in string constant''`.

- If you see "`*)`" outside a comment, report this error as `''Unmatched *)''`, rather than tokenzing it as `*` and `)`.

3.2 String Table

Programs tend to have many occurrences of the same lexeme. For example, an identifier is generally 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 Java. See the following sections for the details.

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.

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.

3.3 Strings

Your scanner should convert escape characters in string constants to their correct values. For example, if the programmer types these eight characters:

$$\boxed{\texttt{"}}\,\boxed{\texttt{a}}\,\boxed{\texttt{b}}\,\boxed{\backslash}\,\boxed{\texttt{n}}\,\boxed{\texttt{c}}\,\boxed{\texttt{d}}\,\boxed{\texttt{"}}$$

your scanner would return the token **STR_CONST** whose semantic value is these 5 characters:

$$\boxed{\texttt{a}}\,\boxed{\texttt{b}}\,\boxed{\backslash\texttt{n}}\,\boxed{\texttt{c}}\,\boxed{\texttt{d}}$$

where $\boxed{\backslash\texttt{n}}$ represents the literal ASCII character for newline.

Following specification on page 15 of the Cool manual, you must return an error for a string containing the literal null character. However, the sequence of two characters

$$\boxed{\backslash}\,\boxed{\texttt{0}}$$

is allowed but should be converted to the one character

$$\boxed{\texttt{0}}\,.$$

3.4 Other Notes

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. Do not use the "yyline" feature of jflex - it counts non-newline characters as changing the line number.

You should ignore the token **LET_STMT**. It is used only by the parser (PA3). Finally, note that if the lexical specification is incomplete (some input has no regular expression that matches), then the scanners generated by jflex do undesirable things. *Make sure your specification is complete.*

4 Notes for Implementation

- Each call on the scanner returns the next token and lexeme from the input. The value returned by the method **CoolLexer.next_token** is an object of class **java_cup.runtime.Symbol**. This object has a field representing the syntactic category of a token (e.g., integer literal, semicolon, the `if` keyword, etc.). The syntactic codes for all tokens are defined in the file TokenConstants.java. The second component, the semantic value or lexeme (if any), is also placed in a **java_cup.runtime.Symbol** object. The documentation for the class **java_cup.runtime.Symbol** as well as other supporting code is available on the course web page. Examples of its use are also given in the skeleton.

- For class identifiers, object identifiers, integers, and strings, the semantic value should be of type **AbstractSymbol**. For boolean constants, the semantic value is of type **java.lang.Boolean**. The cases for integers and booleans are given in the skeleton as examples. Except for errors (see below), the lexemes for the other tokens do not carry any interesting information. Since the **value** field of class **java_cup.runtime.Symbol** has generic type **java.lang.Object**, you will need to cast it to a proper type before calling any methods on it.

- We provide you with a string table implementation, which is defined in AbstractTable.java. The documentation for this class is also available on the course web page. For the moment, you only need to know that the type of string table entries is **AbstractSymbol**.

- When a lexical error is encountered, the routine **CoolLexer.next_token** should return a **java_cup.runtime.Symbol** object whose syntactic category is **TokenConstants.ERROR** and whose semantic value is the error message string. See Section 3 for information on how to construct error messages.

5 Testing the Scanner

There are at least three ways that you can test your scanner. The first way is to generate sample inputs and run them using mylexer.py, which prints out the line number and the lexeme of every token recognized by your scanner. The second way, when you think your scanner is working, is to try running mycoolc.py 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. The other way is to simply execute `ant test-all` command. This will run your lexer implementation using 5 test cases and tell you whether the outputs are correct.

6 What to Turn In

When you are ready to turn in the assignment, first upload your assignment to the instructional account, then type `ant submit-clean` in the directory where you have uploaded your assignment. This action will remove all the unnecessary files, such as object files, class files, Emacs autosave files, etc. Following `ant submit-clean`, type `submit PA2` which will ask you for the names of the partners doing the assignment and will then send README, cool.lex, test.cl and test.output (the output of running your program on test.cl) to the reader. The submit program will also ask you if you want to turn in any other files in the project directory. Your default answer should be "no", unless you really want us to see those files.

Doctoring the output that is sent is considered cheating (and not effective, since we test your program ourselves). If you want to explain something, do it in the README file.

The last submission you do will be the one graded. Each submission overwrites the previous one. Remember that there is a 1% penalty per hour for late assignments. 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.

7 Grading (out of 50)

The point breakdown for PA2 is as follows:

- 38 points for autograder tests (1 point per correct test / 40 tests with a maximum score of 38)
- 4 points for the README
  - 4 : thorough discussion of design decisions (including handling of strings and comments) and choice of test cases; a few paragraphs of coherent English sentences should be fine
  - 2 : vague or hard to understand; omits important details
  - 0 : little to no effort
- 4 points for test.cl
  - 4 : wide range of test cases added, stressing most Cool features and most of the error conditions discussed in the handout
  - 2 : added some tests, but the scope not sufficiently broad
  - 0 : little to no effort
- 4 points for code cleanliness
  - 4 : code is mostly clean and well-commented
  - 2 : code is sloppy and/or poorly commented in places
  - 0 : little to no effort to organize and document code

References

- **JFLex**
  - JfLex manual (`http://jflex.de/manual.html`)
  - Discussion notes #2 (and possibly later notes)

- Repository: We strongly recommend to use source repository to manage your source code.
  - Github free repository (`https://github.com`)
  - Git screencast: Git in Action (`http://vimeo.com/16395537`)
  - Git from the bottom up (`http://ftp.newartisans.com/pub/git.from.bottom.up.pdf`)