Phase 2: The Scanner

Due Date: March 5th.
Teamwork: Forbidden.

1 Overview

The assignments will direct 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. 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 JLex.) 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. The JavaDoc documentation for the Cool code is available on the course web page.

You should do this assignment individually. That means that you should not be working in groups with the same lex specification, but learning how to use JLex for your own specification.

If you have any questions on the project, send an email to the TA or visit the TA during office hours.

2 Tasks

1. Read the JLex specification manual at
   http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html.

2. The file lexer/CoolLexer.lex contains a skeleton for a lexical description for Cool. You can actually build a scanner with this description but it does not do much. Modify this file to be a JLex specification file for COOL. Any auxiliary routines that you wish to write should be added directly to this file in the appropriate section (see comments in the file). This is the main task of this assigment.

3. The file README_Phase1.txt 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 and 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.

3 Files and Directories

The repositor now contains two more folders: examples and testCasesLexer. The first contains a number of little cool programs. If your lexer can handle all of those, great! The second folder contains a number of odd, many times incorrect cool programs. For full credit, your lexer should be able to handle most of the files in this folder as well.

In your own folder you will find skeleton files for the class project. Don't be overwhelmed by the mass of files – most of them will not be used in this Phase. Instead, you should focus on the files in the lexer-package.

Do not modify any java-file.

4 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.

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 for Java. See the following sections for the details.

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. This string will be returned as the error message. For errors besides strings of invalid characters it is sufficient to return an informative error message Make sure that the error message is informative so that we can understand what you did. The following sections clarify how to actually return the error message.

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.

Finally, if the lexical 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 the console. 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.

5 Brief Discussion of the Skeleton Code

- 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 — whether it is an integer literal, semicolon, the `if` keyword, etc. The syntactic codes for all tokens are defined in the file TokenConstants.java. The component, the semantic value or lexeme (if any), is also placed in a **java_cup.runtime.Symbol** object. We suggest you read the comments in the java file **src/java_cup/runtime/Symbol.java**.

- 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**. 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.

- 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 the semantic value is the error message string. See previous section for information on how to construct error messages.

6 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. When you think your scanner is working, you should try running mycoolc 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.

7 Submission

Doctoring the output that is sent is considered cheating. If you want to explain something, do it in the README_Phase1.txt file. 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.

8 Evaluation Criteria

The rubric for Phase 2 (rubric-phase2.txt) will be posted on course web site. The differences between the work expected as a undergrad versus a graduate student are clearly described in the rubric. You should also look at this rubric to see what is expected of you and to see the point break down for the different components of this phase.