Programming Assignment II
Due Friday, February 25, 2005 (11:59 PM)
Individual Assignment

1 Overview

Programming assignments II–V 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. You will have an option of doing your projects in C++ or Java.

For this assignment you are to write a lexical analyzer, also called a *scanner*, using a *lexical analyzer generator* (The C++ tool is called flex; 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 (C++ or Java) for recognizing tokens in Cool programs.

The textbook shows an example Lex specification. The manuals for flex and JLex are available online. The O'Reilly lex and yacc book is also a useful resource. For the Java programmers, the web site www.cs.princeton.edu/~appel/modern/java is useful.

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 flex (JLex) for your own specification.

2 Two installations (and versions) of Cool

I have installed the latest Cool project directory at ~pollock/public/cool02. The version that was used previously is in ~pollock/public/cool. It turns out that there is a slight change in the syntax of the newer Cool; we'll call it Cool02. The difference is in the delimiters for blocks. They have made Cool02 more C-like with { and } as delimiters in place of the old delimiter words.

You may choose to work with Cool, or Cool02. You just need to be careful which coolc compiler you put in your path. Thus, once you decide, you need to place the correct path at the beginning of your path variable to make sure the executables used are the ones the assignments are designed for. You can do this by adding the following to your .login file.

setenv PATH ~pollock/public/cool/bin:$PATH

for the original Cool.

setenv PATH ~pollock/public/cool02/bin:$PATH

for the newer version, Cool02.

If you want to work with Java, you need to use the Cool02. C++ is available for either version of Cool. I have written the instructions for the Cool02, but you can easily replace Cool02 by Cool below, and it will all work the same. Again, the only difference is the syntax and the fact that Java support is only available for Cool02.

3 Files and Directories

To get started, create a directory where you want to do the assignment and execute one of the following commands *in that directory*. For the C++ version of the assignment, you should type

```
gmake -f ~pollock/public/cool02/assignments/PA2/Makefile
```

For Java, type:

```
gmake -f ~pollock/public/cool02/assignments/PA2J/Makefile
```

(notice the "J" in the path name). 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 (in the C++ version) / cool.lex (in the Java version)
  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/jlex 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.

  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 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 lexer does compile and run. There are a number of useful tips in the README file.

All of the software supplied with this assignment is supported on the Solaris machines. However, if you switch platforms be sure to run gmake clean to remove files compiled for the other architecture. A version of the project for Linux is available for downloading on Aiken's web page www.cs.berkeley.edu/~aiken. I make no guarantees about how well this works on Linux. I have not tried it out.

4 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, 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. 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 both C++ and 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 (e.g., a string constant that is too long, or an end-of-file inside of a comment) it is sufficient to return an informative error message (e.g., "String constant too long" or "EOF in comment"). 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 in C++ and Java versions of the assignment.

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 Notes for the C++ version of the assignment

If you are working on the Java version, skip to the following section.

- 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.
- 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 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 string, not a symbol). See previous section for information on how to construct error messages.

6 Notes for the Java version of the assignment

If you are working on the C++ version, skip this section.

- 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. The documentation for the class **java_cup.runtime.Symbol** as well as other supporting code is available on the course web page.
- 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. 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 **TokenConstats.ERROR** and the semantic value is the error message string. See previous section for information on how to construct error messages.

7 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.

8 What to Turn in

When you are ready to turn in the assignment, tar your files and email the TA with your tarred files. You may use the gmke submit-clean but it is restrictive in what you can submit. You may submit more than one test case if you like.

Doctoring the output that is sent is considered cheating. If you want to explain something, do it in the README file. The last turnin you do will be the one graded. Each turnin overwrites the previous one in our mind. We will only grade the latest submission. Read the course syllabus for the late policy. 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.

9 Evaluation Criteria

Correctness: token recognition (50), token attribute handling (20), error detection (15), 85%,
Program Structure 5 %,
Documentation 5%,
Efficiency 5%.