Programming Assignment I

1 Overview of the Programming Project

Programming assignments I–IV 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.

On-line documentation for all the tools needed for the project will be made available on the "Project Resources" page of the wiki on the Coursera web site. This includes manuals for flex and jlex (used in this assignment), the documentation for bison and java_cup (used in the next assignment), as well as the manual for the spim simulator.

You must work individually on this assignment (no collaboration in groups).

2 Introduction to Flex/JLex

Flex allows you to implement a lexical analyzer by writing rules that match on user-defined regular expressions and performing a specified action for each matched pattern. Flex compiles your rule file (e.g., "lexer.l") to C (or, if you are using JLex, Java) source code implementing a finite automaton recognizing the regular expressions that you specify in your rule file. Fortunately, it is not necessary to understand or even look at the automatically generated (and often very messy) file implementing your rules.

Rule files in flex are structured as follows:

```
%{
Declarations
%}
Definitions
%%
Rules
%%
User subroutines
```

The Declarations and User subroutines sections are optional and allow you to write declarations and helper functions in C (or for JLex, Java). The Definitions section is also optional, but often very useful as definitions allow you to give names to regular expressions. For example, the definition

```
\DIGIT     [0-9]
```

allows you to define a digit. Here, `DIGIT` is the name given to the regular expression matching any single character between 0 and 9. The following table gives an overview of the common regular expressions

that can be specified in Flex:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	m through n occurrences of x

The most important part of your lexical analyzer is the rules section. A rule in Flex specifies an action to perform if the input matches the regular expression or definition at the beginning of the rule. The action to perform is specified by writing regular C (or Java) source code. For example, assuming that a digit represents a token in our language (note that this is not the case in Cool), the rule:

```
{DIGIT} {
    cool_yylval.symbol = inttable.add_string(yytext);
    return DIGIT_TOKEN;
}
```

records the value of the digit in the global variable `cool_yylval` and returns the appropriate token code. (See Sections 5 and 6 for a more detailed discussion of the global variable `cool_yylval` and see Section 4.2 for a discussion of the `inttable` used in the above code fragment.)

An important point to remember is that if the current input (i.e., the result of the function call to `yylex()`) matches multiple rules, Flex picks the rule that matches the largest number of characters. For instance, if you define the following two rules

```
[0-9]+ { // action 1}
[0-9a-z]+ {// action 2}
```

and if the character sequence `2a` appears next in the file being scanned, then `action 2` will be performed since the second rule matches more characters than the first rule. If multiple rules match the same number of characters, then the rule appearing first in the file is chosen.

When writing rules in Flex, it may be necessary to perform different actions depending on previously encountered tokens. For example, when processing a closing comment token, you might be interested in knowing whether an opening comment was previously encountered. One obvious way to track state is to declare global variables in your declaration section, which are set to true when certain tokens of

interest are encountered. Flex also provides syntactic sugar for achieving similar functionality by using state declarations such as:

```
%Start COMMENT
```

which can be set to true by writing `BEGIN(COMMENT)`. To perform an action only if an opening comment was previously encountered, you can predicate your rule on COMMENT using the syntax:

```
<COMMENT> {
   // the rest of your rule ...
}
```

There is also a special default state called `INITIAL` which is active unless you explicitly indicate the beginning of a new state. You might find this syntax useful for various aspects of this assignment, such as error reporting. We strongly encourage you to read the documentation on Lex written by Lesk and Schmidt linked from the Project Resources section on the class wiki before writing your own lexical analyzer.

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

```
make -f /usr/class/cs143/assignments/PA2/Makefile
```

Note that even though this is the first programming assignment, the directory name is *PA2*. Future assignments will also have directories that are one more than the assignment number–please don't get confused! This situation arises because we are skipping the usual first assignment in this offering of the course. For Java, type:

```
make -f /usr/class/cs143/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. 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 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. Feel free to modify this file to test your scanner.

- README
  This file contains detailed instructions for the assignment as well as a number of useful tips.

Although these files are incomplete as given, the lexer does compile and run. **To build the lexer, you must type `make lexer` both in C++ and Java.**

4 Scanner Results

In this assignment, you are expected to write Flex rules that match on the appropriate regular expressions defining valid tokens in Cool as described in Section 10 and Figure 1 of the Cool manual and perform the appropriate actions, such as returning a token of the correct type, recording the value of a lexeme where appropriate, or reporting an error when an error is encountered. Before you start on this assignment, make sure to read Section 10 and Figure 1 of the Cool manual; then study the different tokens defined in cool-parse.h. Your implementation needs to define Flex/Jlex rules that match the regular expressions defining each token defined in cool-parse.h and perform the appropriate action for each matched token. For example, if you match on a token BOOL_CONST, your lexer has to record whether its value is true or false; similarly if you match on a TYPEID token, you need to record the name of the type. Note that not every token requires storing additional information; for example, only returning the token type is sufficient for some tokens like keywords.

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.

4.1 Error Handling

All errors should be passed along to the parser. You lexer should not print anything. 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 is 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).
- Recall from lecture that this phase of the compiler only catches a very limited class of errors. **Do not check for errors that are not lexing errors in this assignment.** For example, you should *not* check if variables are declared before use. Be sure you understand fully what errors the lexing phase of a compiler does and does not check for before you start.

4.2 String Table

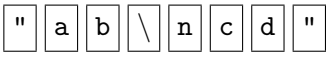
Programs tend to have many occurrences of the same lexeme. For example, an identifier is generally referred to more than once in a program (or else it isn't very useful!). To save space and time, a common compiler practice is to store lexemes in a *string table*. We provide a string table implementation for both C++ and 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.

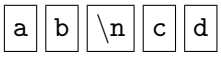
Do *not* test whether integer literals fit within the representation specified in the Cool manual—simply create a Symbol with the entire literal's text as its contents, regardless of its length.

4.3 Strings

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



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



where \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

is allowed but should be converted to the one character

.

4.4 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.

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 both flex and jlex do undesirable things. *Make sure your specification is complete.*

5 Notes for the C++ Version of the Assignment

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

- 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 (e.g., integer literal, semicolon, `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 ",") are represented just by the integer (ASCII) value of the character itself. All of the single character tokens are listed in the grammar for Cool in the Cool manual.

- 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 in 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 the previous section for information on what to put in 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 (e.g., 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. 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**. 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. 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 4 for information on how to construct error messages.

7 Testing the Scanner

There are at least 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. The other way, when you think your scanner is working, is to 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 any test programs.

In addition, for the public version of the class, we will provide the students with scripts that provide automated feedback on what works and what doesn't.

8 What to Turn In

We will post instructions on how to submit your lexer on the Coursera page for this assignment.