Programming Assignment I
Due Thursday, April 20, 2017 at 11:59pm

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 be implementing the projects in C++.

For this assignment, you are to write a lexical analyzer, also called a *scanner*, using a *lexical analyzer generator* named flex. You will describe the set of tokens for Cool in an appropriate input format, and the analyzer generator will generate the actual C++ code for recognizing tokens in Cool programs.

Online documentation for all the tools needed for the project will be made available on the course website. This includes manuals for flex (used in this assignment), the documentation for bison (used in the next assignment), as well as the manual for the spim simulator.

You may work either individually or in pairs for this assignment.

2 Introduction to flex

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.flex) to C 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. 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*	$0, 1, 2, \ldots$ instances of x.
x+	$1, 2, 3, \ldots$ 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 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 Section 5 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.

3 Files and Directories

To get started, log in to one of the *corn* machines and run the following command:

/usr/class/cs143/bin/pa_fetch PA1 <project_directory>

The project directory you specify will be created if necessary, and will contain a few files for you to edit and a bunch of symbolic links for things you should not be editing. (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
  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 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 edit this file to include your SUNet ID(s) (see Section 7 for details).

Although these files are incomplete as given, the lexer does compile and run. **To build the lexer, type** make lexer.

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 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 happen; 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 tokenizing 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

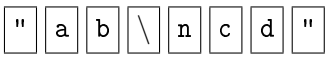
Programs tend to have many occurrences of the same lexeme. For example, an identifier is generally referred to more than once in a program (or else it isn't very useful!). To save space and time, a common compiler practice is to store lexemes in a *string table*. We provide a string table implementation;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

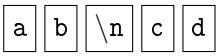
Do *not* test whether integer literals fit within the representation specified in the Cool manual—simply create a **Symbol** with the entire literal's text as its contents, regardless of its length.

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 

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 scanner generated by flex does undesirable things. *Make sure your specification is complete.*

5 Implementation Notes

- 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 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.

7 What to Turn In

Before submitting, please ensure you have done the following:

- It is your responsibility to ensure that the final version you submit does not have any debug print statements and that your lexical specification is complete (every possible input has some regular expression that matches).

- Please remove the instructions section in the README. This section is the one that starts at the beginning, and runs till the following line (including this line):

  ```
  ---8<------8<------8<------8<---cut here---8<------8<------8<------8<---
  ```

  This section of the README is only to provide you additional guidance about which files you need to edit, and you should remove it before actually submitting the assignment.

- Make sure that you have edited the README file so that the first line includes your username in the following format:

  ```
  user: <your_sunet_id>
  ```

  If you are working in a group of two, there should be TWO separate user lines—the submission script will fail if both SUNet IDs are contained on the same line. In particular, if two students with SUNet IDs `abcdef` and `bcdefg` work together, their README should mention the SUNet IDs in the following way:

  ```
  user: abcdef
  user: bcdefg
  ```

  The important thing to note is that the following will NOT work:

  ```
  user: abcdef, bcdefg
  ```

  Your SUNet ID is the name you use to log in to the corn machines (not your 8-digit student ID number), and can be queried with the whoami command.

- Once the README is ready, run the following from your project directory:

  make submit

  This will package up most of the files in your project directory (it will ignore things like core dumps and Emacs backup files) and copy them into the submission folder, along with a time stamp.

  The last version of the assignment you submit will be the one graded. Each submission overwrites the previous one. However, we reserve the right to retain older submissions for reference in case of any disputes. Remember that you have three late days to use for the entire quarter; please refer to the late policy page on the course website for details.

  The burden of convincing us that you understand the material is on you. Obtuse code will have a negative effect on your grade, so take the extra time to make your code readable.

- **Important:** We will only accept assignments turned in via the submit script. Please test-submit your assignment at least 24 hours before the deadline to be sure that the script is working for you, even if you are not finished. It is your responsibility to check for problems ahead of time.