Machine Problem 1

CS 426 — Compiler Construction
Fall Semester 2014

Handed Out: September 2, 2014. Due: September 16, 2014, 10:00 P.M.

In this MP, you will write a lexical scanner for *cool* in *C++*.

1 Provided files

Familiarize yourself with the files in *mp1/src*, which contains the main source files for MP1. These files include:

- *Makefile:* this file describes how to generate the binary for scanning your source files. You should not need to modify it.
- *cool-lexer.cc:* a skeleton lexer that you will need to change completely to write your lexical analyzer.
- *lexer_test.cl:* a very simple cool program for the first scanner test. As this file doesn't cover all elements of *cool*, you need to augment it with further examples to make sure your lexer processes the full set of *cool* tokens.

The files that you will need to modify are:

- *cool-lexer.cc*
  This file contains a skeleton lexical analyzer for *cool*. Right now, it does not do much. It returns an error string for each character it encounters till it reaches EOF (end of file).

  The function `cool_yylex()` should return the next token. You are free to design your *lexer* using any suitable method. However, you may not use a tool to generate the lexer.

  Functions `getNext()` and `lookNext()` return the next character from the file, the former moves the pointer to the next character, while the latter does not. You can use the functions as provided.

  Feel free to modify anything in this file, but make sure it works as expected with other provided files.

- *lexer_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 *lexer_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.

Note that you will not hand in your modified *lexer_test.cl* file. However, it is important to make a good test to ensure that your lexer is working properly.

The supplied file *lextest.cc* contains the `main` program for the lexer. You may not modify this file, but it may help you to see how your lexer will be called.

2 Compiling, running and testing the scanner

To build the lexer type

$ make lexer

in the directory *mp1/src*. This will start the compilation process and link the support code needed for this phase into your working directory.

Run the lexical analyzer by typing

$ lexer input_file

We will provide a reference binary for *lexer* one week before the due date (September 09, 2014) and an extensive set of tests a few days before the due date (September 12, 2014). Try to use them only after you are really done with your own testing. Your goal should be that the reference binaries uncover no new bugs.

3 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 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.

All errors will be passed along to the parser. The *cool* parser knows about a special error token called `ERROR` which carries an error message to the parser. There are several requirements for reporting and recovering from lexical errors.

Most of these situations should be reported by returning an error token with some human-readable message describing the problem as the error string.

- When an invalid character (one which 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.

- When a string is too long (¿ 1024 characters), report the error "String constant too long". Lexing should resume after the end of the string. Do not produce a string token before the error token.
- When a null character is encountered in a string constant, report "Null in string constant", and report "EOF in string constant" for EOF. Lexing should resume at the end of the string.
- If a string contains an unescaped newline, report the error "Unterminated string constant", and resume lexing at the beginning of the next line — we assume the programmer simply forgot the close-quote. Do not produce a string token before the error token.
- If a comment remains open when EOF is encountered, report "EOF in comment". Do **not** tokenize the comment's contents simply because the terminator is missing.
- If you see "`*)`" outside a comment, report this as an unmatched comment terminator with the error "Unmatched *)", rather than tokenzing it as `*` and `)`.
- 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.

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.

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.

Finally, note that if the lexical analyzer is incomplete, then the resulting lexer does undesirable things (returns a stream of `ERROR` tokens Make sure your specification **is** complete.

4 Notes

- Each call to 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 (ASCII) 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 what to put in error messages.
- To simplify your implementation, you can assume that the identifiers and integer constants will not be longer than 1024 characters.
- Your results will be validated against the provided reference binary character by character.

5 Further ways to use your Makefile

- *make* builds the lexer.
- *clean* removes all compiled files.
- *realclean* is like clean, but also removes all links. Use this if you really want to start with just your source files.

Documentation on *make* can also be downloaded from the course web page.

6 What and how to hand in

You have to hand in the file *cool-lexer.cc* that you modify in this MP.

Don't copy and modify any part of the support code! The provided files are the ones that will be used in the grading process.

The details on the hand in procedure will be posted on Piazza and the Assignments page on the course website.