CS131 Compilers: Programming Assignment 1
Due Tuesday, March 7, 2017 at 11:59pm

Fu Song Zhiming Mei

1 Policy on plagiarism

These are individual homework. While you may discuss the ideas and algorithms or share the test cases with others, at no time may you read, possess, or submit the solution code of anyone else (including people outside this course), submit anyone else's solution code, or allow anyone else to read or possess your source code. We will detect plagiarism using automated tools and will prosecute all violations to the fullest extent of the university regulations, including failing this course, academic probation, and expulsion from the university.

2 Overview of the Project

Project assignments 1-4 will direct you to design and build a compiler for the Classroom Object-Oriented Language (cool), designed by Alexander Aiken for use in an undergraduate compiler course project. Assignments will cover the frond-end of the compiler: lexical analysis, parsing, semantic analysis, and intermediate code generation. Each assignment should be solved using C++ programming language and 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 Flex. You will describe the set of tokens for cool in an appropriate input format, and the analyzer generator Flex will generate the actual code C++ for recognizing tokens in cool programs. Manuals for all the tools needed for the project and provided source codes will be made available on the course's web site. This includes manuals for Flex (used in this assignment), the documentation for Bison and as well as the manual for the MIPS. You have to work **individually**.

This assignment consists of three parts:

1. get familiar with cool, its support code, and the files we provide for this assignment,

2. install and get familiar with Flex `https://github.com/westes/flex`,

3. write a lexical analyzer, also called a scanner, using a lexical analyzer generator Flex.

3 cool and Provided Files

1. To get familiar with cool read the documentation coolAid: The cool Reference Manual, available under on the class web page. Write some simple example programs in cool to get familiar with the language. You don't have to submit those programs, but understanding the language is important for the rest of the project assignments. If you need further examples, there is a link to a set of example cool programs on the class web page, together with a README file and expected output files. Read the README file to learn what the example programs are about, and read the comments in the example files for further help.

2. Download and unpack the PA1 directory from the class web page. This directory contains all the code that you will need for this assignment.

3. Become familiar with the cool support code, which provides several data structures that you will use in writing your cool compiler, including all the AST classes. You will compile this code and link against it for each of the parts of this project. Download the document "A Tour of the cool Support Code" from the class web page read and understand it. If you want to examine the support code source files you can find them in the directories cool-support/include (header files) and cool-support/src (implementation files) within the directory PA1.

4. Familiarize yourself with the files in PA1/src, which contains the main source files for PA1. These files include:

   - Makefile: this file describes how to generate the binaries for scanning. You should not need to modify it. If you would like to understand Makefile, read `https://www.gnu.org/software/make/manual/make.html`.
   - cool.flex: a skeleton Flex input file that you will need to extend to write your lexical analyser.
   - flex_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 write your own examples to make sure your lexer processes the full set of cool tokens.

The compiler consists of several phases. Each phase will be compiled into its own binary file. It will take its inputs from standard input (except the lexer which takes a commandline input filename) and write to standard output. So in the end to start your compiler you will call

$$lexer\ input\_file \mid parser \mid semant \mid cgen > output\_file$$

For this assignment you are making the lexer. You should test your compiler passes extensively with your own tests. Your goal should be that the reference binaries uncover no new bugs. Note that, you might need to add "./" at the beginning of the tool name, e.g.,

$$./lexer\ input\_file \mid ./parser \mid ./semant \mid ./cgen > output\_file$$

4 Scanner

Write a lexical scanner for cool using flex. To do this, first read the documentation on 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."cool.flex") to C source code implementing a finite automaton recognizing the regular expressions that you specify in your rule file.

4.1 Files

The files that you will need to modify are:

1. cool.flex. 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 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).

2. flex_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 flex_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 flex 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.

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

make lexer

in the directory PA1/src. This will start the compilation process and link the support code needed for this phase into your working directory. Start the lexical analyser by typing

lexer input_file

4.2 Scanner Results

Your Flex rules should exactly match on the specification of the lexical structure of cool given in Section 10 and Figure 1 of the cool manual and 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 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.

4.3 Error Handling

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 communicate errors from the lexer 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.

1. 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.

2. If a string contains an unescaped newline, report that, 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.

3. 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

   - the beginning of the next line if an unescaped newline occurs after these errors are encountered; or
   - after the closing " otherwise.

4. If a comment remains open when EOF is encountered, report that with message "EOF in comment". Do not tokenize the comment's contents simply because the terminator is missing. (This applies to strings as well, that reports report this error as "EOF in string constant".)

5. If you see ")" outside a comment, report this error as "Unmatched *)", rather than tokenizing it as * and).

6. Do **not** test whether integer literals fit within the representation specified in the cool manual simply use the add_* functions, which create a Symbol with the entire literal's text as its contents, regardless of its length. Symbol (a typedef defined in stringtab.h) is a pointer to Entry, which is a wrapper around a string.

7. 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.

4.4 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.

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 specification is incomplete (some input has no regular expression that matches) then the scanner generated does undesirable things. Make sure your specification is complete.

4.5 String

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

“	a	b	\	n	c	d	”

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

a	b	\n	c	d

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: \0 is allowed but should be converted to the one character 0.

4.6 Notes

1. 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 (ASCII) value of the character itself. All of the single character tokens are listed in the grammar for cool in the CoolAid.

2. 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.

3. 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*.

4. 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.

5 Testing the Scanner

To test your program, generate sample inputs and run them using lexer, which prints out the line number and the lexeme of every token recognized by your scanner.

6 When, What and How to Hand in

1. When: Make sure 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)

2. What: You have to hand in all files that you modify in this assignment. That is cool.flex. Don't copy and modify any part of the support code! In addition to looking at the test results, we will manually look at your code. 5% of the grade is for commenting your code. Another 5% is for appropriate usage of Flex features.

3. How: Please hand in the packaged file named in the format

   $$StudentID\_Name$$

   to `meizhm@shanghaitech.edu.cn` with subject:

   $$CS131PA1\_StudentID\_Name$$

otherwise, we will not grade your submission.