Programming Assignment 1
Due September 27 2017 at 11:59pm

1 Overview of the Programming Project

Programming assignments 1-4 will direct you to design and build an interpreter for L. Each assignment will cover one component of the interpreter: lexical analysis, parsing, interpreting L and performing type inference on L.

For this assignment, you are to write a lexical analyzer, also sometimes called a scanner, using a lexical analyzer generator. The tool you will use for this is called `flex`. You will describe the set of tokens for L in an appropriate input format, and the analyzer generator will generate the actual C++ code for recognizing tokens in L programs. On-line documentation for all the tools needed for the project will be made available on the CS345 web site. This includes the manual for flex used in this assignment, and the documentation for bison (used in the next assignment). **You will need to refer to the flex manual to complete this assignment**. 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 (this file is called `lexer.l` in your assignment) 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"`	the character "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
`[\n]`	newline
`<Y>x`	an x where flex is in start condition Y
`x?`	an optional x (0 or 1 instances of x)
`x*`	0,1,2,... instances of x
`x+`	1,2,3,... instances of x
`x|y`	an x or y
`(x)`	and x
`{YY}`	matches the pattern `YY` defined in your definitions section

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, since `COMMA` is a token in L, the rule:

```
"," {
  return TOKEN_COMMA;
}
```

matches the string `","` and returns the token code `TOKEN_COMMA`.

As we have discussed in lecture, with some tokens such as integers, identifiers and strings we also need to record the value or *lexeme* of the token. As an example, assume that every digit is one token with token type `TOKEN_DIGIT`. **This is not the case in L; digits are not tokens in L, this is just an example.**

```
{DIGIT} {
  SET_LEXEME(yytext);
  return TOKEN_DIGIT;
}
```

This rule illustrates two important points of flex. First, it uses the definition of `DIGIT` defined earlier in the rules section. Second, in any rule the string that matched the current pattern in stored in a global variable called `yytext` of type `char*`. To assign the lexeme to the token, simply use the macro `SET_LEXEME` as shown in the example.

An important point to remember is that if the current input 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. For this, `flex` provides 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 will need states for processing comments and strings. We strongly encourage you to read the documentation on Flex on the CS312 website before writing your own lexical analyzer.

3 Tokens in L

For this assignment, you will accept all tokens that are part of the L language. You will need to refer the the L reference manual for what strings correspond to which tokens. The token types defined for you are listed below. You must not define any other token types.

```
TOKEN_READSTRING
TOKEN_READINT
TOKEN_PRINT
TOKEN_ISNIL
TOKEN_HD
TOKEN_TL
TOKEN_CONS
TOKEN_NIL
TOKEN_DOT
TOKEN_WITH
TOKEN_LET
TOKEN_PLUS
TOKEN_MINUS
TOKEN_IDENTIFIER
TOKEN_TIMES
TOKEN_DIVIDE
TOKEN_INT
TOKEN_LPAREN
TOKEN_RPAREN
TOKEN_AND
TOKEN_OR
TOKEN_EQ
TOKEN_NEQ
TOKEN_GT
TOKEN_GEQ
TOKEN_LT
TOKEN_LEQ
```

```
TOKEN_IF
TOKEN_THEN
TOKEN_ELSE
TOKEN_LAMBDA
TOKEN_FUN
TOKEN_COMMA
TOKEN_STRING
TOKEN_IN
TOKEN_ERROR
```

Of special interest here is `TOKEN_ERROR`. Any time you encounter a lexing error, you simply return `TOKEN_ERROR`. This will generate an error message and abort your lexer. You can also specify an error message using `SET_LEXEME` if you choose to, but you are not required to do so for this assignment.

4 Files and Directories

To get started, create a directory where you want to do the assignment on any William & Mary computer science computer and execute the following command in that directory:

```
/projects/cs345.tdillig/PA1/get-assignment
```

This command will copy a number of files to your directory. The only file you will need to modify for this assignment is `lexer.l` This file contains a skeleton for a lexical description for L. 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.

4.1 Building, Running and Testing your Lexer

To compile your lexer, simply type

```
make
```

in your directory. This will build a binary called `lexer` that you can run. For example, to run your lexer on the file `test.L`, you type:

```
./lexer test.L
```

Once your lexer is finished, this should print all tokens and lexemes (if applicable) to the screen. For your reference, you can also run a reference lexer whose binary we provide. To run the reference lexer, type:

```
/projects/cs345.tdillig/lexer test.L
```

A big component of this assignment is to test your lexer thoroughly. It is your responsibility to ensure your lexer accepts all legal tokens in L, rejects all invalid ones and never crashes. You will want to feed many different inputs to your lexer to test it.

4.2 Turning in and Grading

You must hand in the following for this assignment:

- The file `lexer.l` containing your lexer
- Ten interesting test cases to test your lexer in a file called `tests.txt`

For this assignment, you will receive 20% credit for your test cases and 80% for your lexer. We will test your lexer automatically on the best selection of submitted test cases, so it is very much in your interest to have your tests included.

Important: Since we grade your lexer automatically, do not print anything in your final lexer since this will confuse our grading script and potentially result in a bad grade for you.

For submission, please submit the file `lexer.l` as well as the file `tests.txt` with all your test cases. Separate different tests by a newline with three (3) dashes, i.e. `---`.