The course project is to build a compiler for a small language. This is a "living" document will be revised throughout the semester until it is a complete, if sometimes informal, language specification. Revisions may include additions, removals, changes, or clarifications to meet pedagogical goals and to ensure internal consistency.

SECTION 1: Lexical structure

The lexical structure of the language is defined (informally) as follows; part of your job is to define an input file for use with flex or ML-lex to generate a lexical analyzer for the language.

Parentheses are used for grouping. For example, ('e' | 'E') means either the lower case letter 'e' or the upper case letter 'E' (but not both).

'?' indicates optionality (zero or one occurrence). For example, ('+'|'-')? means either the plus sign '+' or the minus sign '-' can appear, but neither is required.

'+' indicates one or more occurrence. For example, digit+ means one or more digits.

Legal identifiers must begin with an upper or lower case letter or '_', followed by an arbitrarily long string of upper or lower case letters, '_' and digits.

The language has five primitive types (literal values described after ':'):

integer – 32-bit wide two's complement numbers: ('+'|'-')? digit+

real – 64-bit wide IEEE 754 numbers: ('+'| '-')? digit+ '.' digit+ (('e' | 'E') integer)?

Boolean – the two values: true and false

character – 8-bit ASCII characters: literals are characters in single quotes, e.g. 'a', including typical '\' escaped special characters (like '\n', '\t' and '\\')

string – a sequence of values of type character, of arbitrary length, enclosed in double quotes, but not spanning more than one line. For example:

"this is a legal \"string\" that contains \n \t several escaped characters"

"this is not a legal string,
because strings cannot span more than one line"

Keywords:

the names of the primitive types (given above),
true, false,
null, reserve, release

for, while,
if, then, else, switch, case, otherwise,
type, function, closure

Punctuation:
'(', ')', '[', ']', '{', '}', ';', ':', ',', '->', '"', '', '\'

Operators:
'+', '-', '*', '/', '%', '.', '<', '=', ':=', 'i2r', 'r2i', 'isNull', '!', '&', '|'

All keywords and operator names are reserved.

Whitespace and comments:
space, tab, newline
comments are delimited by '(*' and '*)'

To standardize the output from every team's lexer, use the following constants to represent your tokens:

```
// identifier
#define ID          101

// type names
#define T_INTEGER   201
#define T_REAL      202
#define T_BOOLEAN   203
#define T_CHARACTER 204
#define T_STRING    205

// constants
#define C_INTEGER   301
#define C_REAL      302
#define C_CHARACTER 303
#define C_STRING    304
#define C_TRUE      305
#define C_FALSE     306

// keywords
#define NULL_PTR    401
#define RESERVE     402
#define RELEASE     403
#define FOR         404
#define WHILE       405
#define IF          406
#define THEN        407
#define ELSE        408
#define SWITCH      409
#define CASE        410
#define OTHERWISE   411
#define TYPE        412
#define FUNCTION    413
#define CLOSURE     414
```

```
// punctuation - grouping
#define L_PARENTHESIS       501
#define R_PARENTHESIS       502
#define L_BRACKET   503
#define R_BRACKET   504
#define L_BRACE     505
#define R_BRACE     506
#define S_QUOTE     507
#define D_QUOTE     508

// punctuation - other
#define SEMI_COLON  551
#define COLON       552
#define COMMA       553
#define ARROW       554
#define BACKSLASH   555

// operators
#define ADD         601
#define SUB_OR_NEG  602
#define MUL         603
#define DIV         604
#define REM         605
#define DOT         606
#define LESS_THAN   607
#define EQUAL_TO    608
#define ASSIGN      609
#define INT2REAL    610
#define REAL2INT    611
#define IS_NULL     612
#define NOT         613
#define AND         614
#define OR          615

// comments
#define COMMENT     700
```

At this point simply have your lexer print out (to standard output) the numeric value representing the token, a space, the text that matched the token, a space, the starting line number or the token, a space, the starting column number of the token, followed by a new line character, using code along these lines:

```
printf("%3d %s %d %d\n", token, text, lineNumber, columnNumber)
```

You might find the following on-line tutorial helpful:

http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf

If you are using ML-Lex rather than flex, your syntax will differ but **the output format must be the same.**

You may run into compilation or other issues. Do make full use of Piazza and office hours to help to get these issues resolved, sooner rather than later.

SUBMISSION & GRADING:

Submit your code using Autolab – details to come. Submissions are due no later than 5:00 PM on Monday February 19.

Your submission will be graded based on whether it produces the correct output for various input files, as compared to known good output files (using 'diff').

Sample input and corresponding output are shown on the next page.
Sample input file:

```
(* Type definition *)
type unaryIntFunction: (integer: x) -> integer

(* This is a function definition.
   It uses the above type definition.
*)
function square : unaryIntFunction {
    square := x * x;
}

(* This is the main block of the program.
   Execution begins in this block.
*)
{
  [ integer: input := 7, expected := 49, actual ; Boolean: result ]
  actual := square(input);
  result := expected = actual;
}
```

Sample output:

```
700 (* Type definition *) 1 1
412 type 2 1
101 unaryIntFunction 2 6
552 : 2 22
501 ( 2 24
201 integer 2 25
552 : 2 32
101 x 2 34
502 ) 2 35
554 -> 2 37
201 integer 2 40
700 (* This is a function definition.
  It uses the above type definition.
*) 4 1
413 function 7 1
101 square 7 10
552 : 7 17
101 unaryIntFunction 7 19
```

```
505 { 7 36
101 square 8 5
609 := 8 12
101 x 8 15
603 * 8 17
101 x 8 19
551 ; 8 20
506 } 9 1
700 (* This is the main block of the program.
  Execution begins in this block.
*) 11 1
505 { 14 1
503 [ 15 3
201 integer 15 5
552 : 15 12
101 input 15 14
609 := 15 20
301 7 15 23
553 , 15 24
101 expected 15 26
609 := 15 35
301 49 15 38
553 , 15 40
101 actual 15 42
551 ; 15 49
203 Boolean 15 51
552 : 15 58
101 result 15 60
504 ] 15 67
101 actual 16 3
609 := 16 10
101 square 16 13
501 ( 16 19
101 input 16 20
502 ) 16 25
551 ; 16 26
101 result 17 3
609 := 17 10
101 expected 17 13
608 = 17 22
101 actual 17 24
551 ; 17 30
506 } 18 1
```