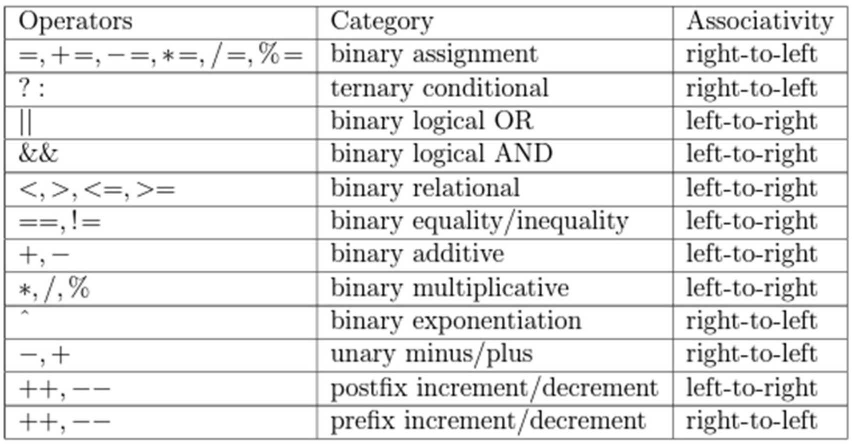
Assignment 2

The purpose of this problem is to introduce you to lexical and syntactic analysis, and to help you gain some experience with Flex, Bison, and C. You will start with a simple integer calculator. Its Flex and Bison specification scan be found here, in the files called, respectively, icalc.l and icalc.y:

https://drive.google.com/drive/folders/1yJPEJxWsMNqqCZX_ncQvkdQnmFbr6PcP?usp=sharing

Your goal is to extend the calculator by adding new lexical specifications and BNF grammar rules for correctly evaluating new types of expressions. The operators and their associativity rules are given below, listed from lowest to highest precedence. Parentheses "()" can be used to override the default precedence rules.



Operators	Category	Associativity
=, +=, −=, *=, /=, %=	binary assignment	right-to-left
? :	ternary conditional	right-to-left
||	binary logical OR	left-to-right
&&	binary logical AND	left-to-right
<, >, <=, >=	binary relational	left-to-right
==, !=	binary equality/inequality	left-to-right
+, −	binary additive	left-to-right
*, /, %	binary multiplicative	left-to-right
ˆ	binary exponentiation	right-to-left
−, +	unary minus/plus	right-to-left
++, −−	postfix increment/decrement	left-to-right
++, −−	prefix increment/decrement	right-to-left

Problem 1

Add twenty-six 32-bit registers, labeled a through z. Users should be able to store values in registers and use them in subsequent expressions in the calculator. A register variable by itself evaluates to the value currently stored in the register. Registers should all be initialized to 0 when the calculator starts.

You should also implement assignment expressions of the form reg = expr, reg += expr, reg -= expr, reg *= expr, reg /= expr, and reg %= expr. Each assignment ex�pression evaluates to the value of the expression on the right-hand side, updates the register on the left-hand side, and prints the result. Note the associativity of the assignment operators (see the table above).

```
icalc: b
0
icalc: b=3
3
icalc: b *= 5 + a
15
icalc: a = b = c = 7
7
icalc: z += a*b + c
56
```

Problem 2

Add C-style unary increment and decrement operators ++ and --, ensuring that they can only be applied to registers (i.e., ++2 is a syntax error). Pay attention to the precedence of these operators vs. binary operators. These operators should have the same semantics as in C.

```
icalc: x=1
1
icalc: x++
1
icalc: x
2
icalc: ++x
3
icalc: ---x
syntax error
icalc: -(--x)
-2
```

Problem 3

Implement built-in constants MAXINT and MININT, assuming signed two's-complement 32-bit integers.

```
icalc: MAXINT
2147483647

icalc: MININT
-2147483648
```

Problem 4

Implement a binary exponentiation operator ˆ that raises the first argument to the power of the second argument (note the associativity of this operator in the table above). Feel free to look at the standard C function pow(x,y), as defined in math.h. It computes x^y up to MAXINT and -x^y up to MININT, and pow(x,0)=1 for all x.

```
icalc: 2^5
32
icalc: 2^0
1
icalc: -2^31
-2147483648
icalc: 2^31
2147483647
icalc: 2^100
2147483647
```

Problem 5

Implement built-in functions abs, min and max as prefix operators which take a comma-separated, parenthesized list of arguments that can be constants, registers, or expressions.

```
icalc: abs(-5)
5
icalc: a
0
icalc: x=abs(a-2*3)
6
icalc: min(abs(-2),min(5,1))
1 icalc:
max(x,2+2)
6
```

Problem 6

Implement relational operators ==, !=, <=, >, >=, logical operators || and &&, and the ternary conditional expression operator ?: Their semantics should be the same as in C.