4) To bottom up parse: *(+ 5 6 (* 6 2))*

Current Stack	*Operation*
(	*shift*
(+	*shift*
(+ 5	*shift*
(+ TERM	*REDUCE*
(+ TERM 6	*shift*
(+ TERM TERM	*REDUCE*
(+ TERM FACTOR	*REDUCE*
(+ EXPR	*REDUCE*
(+ EXPR (	*shift*
*(+ EXPR (**	*shift*
(+ EXPR (6*	*shift*
(+ EXPR (TERM*	*REDUCE*
(+ EXPR (TERM 2*	*Shift*
(+ EXPR (TERM TERM*	*REDUCE*
(+EXPR (EXPR	*REDUCE*
(+EXPR (EXPR)	*Shift*
(+EXPR EXPR	*REDUCE*
(+EXPR EXPR)	*Shift*
(EXPR)	*REDUCE*
EXPR	*REDUCE*
PROG	*REDUCE*

5) **Precedence**

```
expr : expr = expr { $1 = $2; }
   | * POST_EXPR {$$ = VALUE($2); }
   ;|

POST_EXPR : SYMBOL++ {$$ = $1; $1 = $1 + 1;}
```

6) **C++ Standard**

a. There are five types of tokens.
   Identifier
   Keyword
   Literal
   Operator
   Separator

b. C++ has a total of 87 keywords.

c.

<floating-literal>	::=	<fractional-constant> {<exponent-part>} {<floating-suffix> | <digit-sequence> <exponent-part> {<floating-suffix>}
<fractional-constant>	::=	{<digit-sequence>} '.' <digit-sequence> | <digit-sequence> '.'
<exponent-part>	::=	'e' { <sign> } <digit-sequence> 'E'{ <sign> } <digit-sequence>
<sign>	::=	+ | -
<digit-sequence>	::=	<digit> | <digit-sequence> <digit>
<floating-suffix>	::=	f | l | F | L

d. No. The grammar for c++ cannot be used as is in a LL parser. C++ grammar cannot handle ambiguous rules.
   for example: x * y can be treated as declaring a pointer y of type x. Or a multiplication of x and y.

7) **Identifiers, Lifetime and Binding**

a. No. It is not possible to have an identifier associated with more than one address. An identifier points to a memory location and hence cannot point to more than one at a time.
b. Yes. We can have two pointers (identifiers) pointing to the same location in memory. Thus this case is possible.
   say, int* a = new int[3];
   int* b = a;
   here, both the address of the array is associated with identifiers a & b at the same time.
c. Yes. From the example above, if we delete the array using pointer b, we have an identifier a pointing to an object that doesn't exist any more. Therefore, a's lifetime is greater than the memory it is binded to.
d. Yes. Suppose after initializing a and b in the above example, we execute the following statements,

a = new int[5];
b = a;

Now, the original array of size 3 is still allocated but is not referenced by any identifier.

delete a;
delete b;

Thus we have a memory whose lifetime is greater than the lifetime of the identifier it is associated with.

8) **Scopes**

a. Static scoping - 10
b. Dynamic scoping - 7

9) **Short circuit evaluation**

a. Yes. The c++ standard mandates the short circuiting of || and && operator. (Refer to page 120 of the standards http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf)
b. The code prints

Output 1 0
Output 2 0
Output 3 0
Output 4 0
Output 5 0

Initial condition:

x and y = 0;

x++ < 5 || y++ < 3 is evaluated. (the post increment is applied after the condition is evaluated)

condition evaluates to true in left side alone. y++ never gers executed.

Maintenance

for all values of x from 1 to 4 the condition holds true. the print statement prints the values 2 to 5.

Termination condition

x is 5 now. The condition fails. The right hand side of the condition y++ > 3 is evaluated, which returns false. Since both condition fails, the print statement is never executed. The loop terminates.

c. Yes. Short circuiting is mandated by Java language specifications too. From the section 15.23 (http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.23), "*If the resulting value is true, the value of the conditional-or expression is true and the right-hand operand expression is not evaluated."* for logical OR and "*If the resulting value is false, the value of the conditional-and expression is false and the right-hand operand expression is not evaluated.*" for logical AND operator. This makes it clear that short circuiting is mandated.