1. Sol$^n$:

```
do {
if (j > 50) continue;
j = i + j;
if (j < 10) break;
i += 2; } while (i < 100);
```

The operational semantics for the above program is:

```
loop: if expr1 > 50 continue
        expr2;
        If expr3 < 10 goto out
        expr4;
        while expr4 < 100
out:
          ….
```

2. Compute the weakest precondition for the following sequence of assignment statements with the given postcondition. If the given precondition is {b < - 10}, does the following code have the desired semantics?

Sol$^n$:

```
a = 20 + 2 * b;
if (a > 0)
          b = a * 5 + 10;
else
          b = a * -5 - 10;
{b > 10}
```

| Case (i) | | Case (ii) | |
|---|---|---|---|
| | b > 10 | | b > 10 |
| or, | a * 5 + 10 > 10 | or, | a * -5 - 10 > 10 |
| or, | a > 0 | or, | a < -4 |
| or, | 20 + 2 * b > 0 | or, | 20 + 2b < - 4 |
| or, | b > -20 / 2 | or, | b < -24 / 2 |
| or, | b > -10 | or, | b < -12 |

Therefore the required answer is   b > -10 V  b < -12

No. If precondition is b < -10, the code doesn't have the desired semantics.
( if we take b = -11, we do not end up with the required postcondition)

3. Write the denotational semantics mapping function $M_l$(**do** L **while** B, s) for Java do-while statements. You can (and should) use $M_{sl}$ and $M_b$ of the meaning in Section 3.5.2.5.

Sol$^n$:

$M_{sl}$ maps lists and states to states
$M_b$ maps boolean expn. to boolean values

Denotational semantics mapping function $M_l$ (do L while B, s):

$M_l$ (**do** L **while** B, s) Δ = if $M_{sl}$ ( L, s) == **error**
then **error**
else
s' = $M_{sl}$ (L, s)
if $M_b$ (B, s) == **undef**
then **error**
else if $M_b$(B, s) == **false**
then s'
else $M_l$ (**do** L **while** B, s')

4. Show the output of the lexical analyzer (as in the example on page 171) of front.c in Section 4.2 of the textbook for the expression (100 + j) / i.

Sol$^n$:

(100 + j) / i :

Next token is: 25 Next lexeme is (
Next token is: 10 Next lexeme is 100
Next token is: 21 Next lexeme is +
Next token is: 11 Next lexeme is j
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is i
Next token is: -1 Next lexeme is EOF

5. Consider the following grammar.

a) Modify the recursive-descent program on page 176 ~ 178 in the textbook so that it can parse the expressions generated by the grammar given below. Show the subprograms for the nonterminals and <term> and <bool>.

b) Show the output of the program (as in the example on page 178) for parsing the expression of a ^ b * c.
Use the following macro definitions for the & and ^ operator.

#define AND_OP 27

#define XOR_OP 28

Note: the parentheses in the last rule are terminal symbols, while those in other rules are metasymbols.

<expr> → <term> { (+ | -) <term>}
<term> → <bool> { (& | ^) } <bool> }
<bool> → <factor> { (* | /) } <factor> }
<factor> → id | int_constant | ( <expr> )

Sol$^{n}$:

a.

Here is the complete recursive-descent program (I found the instructions little ambiguous, so modified the whole thing to be on the safe side)

```
void expr () {
        printf("Enter <expr>\n");
        term();
        while (nextToken == ADD_OP || nextToken == SUB_OP) {
                lex();
                term();
        }
        printf("Exit <expr>\n");
}

void term() {
        printf("Enter <term>\n");
        bool();
        while (nextToken == AND_OP || nextToken == XOR_OP) {
                lex();
                bool();
        }
        printf("Exit <term>\n");
}

void bool() {
        printf("Enter <bool>\n");
        factor();
        while (nextToken == MULT_OP || nextToken == DIV_OP) {
                lex();
                factor();
        }
```

```
        printf("Exit <bool>\n");
    }

    void factor() {
        printf("Enter <factor>\n");
        if (nextToken == IDENT || nextToken == INT_LIT)
            lex();
        else {
            if (nextToken == LEFT_PAREN) {
                lex();
                expr();
                if (nextToken == RIGHT_PAREN)
                    lex();
                else
                    error();
            } else
                error();
        }
        printf("Exit <factor>\n");
    }
```

b.

Parsing output for expr: a ^ b * c:

```
Next token is: 11 Next lexeme is a
Enter <expr>
Enter <term>
Enter <bool>
Enter <factor>
Next token is: 27 Next lexeme is ^
Exit <factor>
Exit <bool>
Next token is: 11 Next lexeme is b
Enter <bool>
Enter <factor>
Next token is: 23 Next lexeme is *
Exit <factor>
Next token is: 11 Next lexeme is c
Enter <factor>
Next token is -1 Next lexeme is EOF
Exit <factor>
Exit <bool>
Exit <term>
```

Exit <expr>

6. Remove left recursion in the following BNF grammar.

<id> → <str><sep><bin>
<str> → <str2><char> | <str><char> | <char>
<str2> → <str><char>
<char> → x | y | z
<sep> → * | #
<bin> → <bin> 0 | <bin> 1 | 0 | 1

Sol$^{n}$:

By inspection, the only rules that have the left recursion problem are highlighted in bold:

**<str> → <str2><char> | <str><char> | <char>**

**Group:**

=> <str> → <str><char> | <str><char><char> | <char>

**Replace:**

<str> → <char><str’>
<str’> → <char><str’> | <char><char><str’> | ε

**<bin> → <bin> 0 | <bin> 1 | 0 | 1**

**Group:**

<bin> → <bin> 0 | <bin> 1 | 0 | 1

**Replace:**

<bin> → 0 <bin’> | 1 <bin’>
<bin’> → 0 <bin’> | 1 <bin’> | ε

**Hence the new grammar is:**

<id> → <str><sep><bin>
<str> → <char><str’>
<str’> → <char><str’> | <char><char><str’> | ε
<char> → x | y | z
<sep> → * | #
<bin> → 0 <bin’> | 1 <bin’>
<bin’> → 0 <bin’> | 1 <bin’> | ε

7. Does the nonterminal C in the following grammar pass the pairwise disjointness test? Why or

why not? (a, b, c are terminal symbols)

$$A \rightarrow aCB \mid ba$$
$$B \rightarrow bBC \mid cba$$
$$C \rightarrow Abc \mid BaC \mid c$$

Sol$^n$:

Here for C:

$FIRST(\alpha_i) = \{a, b\}$

$FIRST(\alpha_j) = \{b, c\}$

$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \{ b \}$

Hence, C doesn't pass the pairwise disjointness test. [the intersection is not empty above]

8. Revise the following grammar so that it can pass the pairwise disjointness test.

$S \rightarrow aAB \mid bBC$
$A \rightarrow aA \mid x$
$B \rightarrow bB \mid bC \mid b$
$C \rightarrow aB \mid AC \mid c$

Sol$^n$:

Here, rewriting the above grammar, we get,

$S \rightarrow aAB \mid bBC$
$A \rightarrow aA \mid x$
$B \rightarrow bB \mid bC \mid b$
$C \rightarrow aB \mid aAC \mid xC \mid c$

Using left refactoring method, we get,

$S \rightarrow aAB \mid bBC$
$A \rightarrow aA \mid x$
$B \rightarrow bB'$
$B' \rightarrow B \mid C \mid \varepsilon$
$C \rightarrow aC' \mid xC \mid c$
$C' \rightarrow B \mid AC \mid \varepsilon$,

which is the required grammar.