Tutorial 04: CS31003: Compilers:

[M-05] IC Translation: Arithmetic & Boolean Expressions

Indranil Sengupta
Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

isg@iitkgp.ac.in
ppd@cse.iitkgp.ac.in

October 10, 2020

Doubts from the Week

Problem: Symbol Table: 1

Consider the following code:

```
int n = 1729;

int main() {
    int sum = 0;

    while(n != 0) {
        int remainder;

        remainder = n % 10;
        sum += remainder;
        n = n / 10;
    }

    return 0;
}
```

1. Convert the code to 3-address
2. Show the symbol tables of the code with user-defined as well as compiler-generated names (ignore constants)

Solution: Symbol Table: 1


```
// global initialization
n_g = 1729
main:
    sum = 0
L1:
    if n_g != 0 goto L2
    goto L3
L2:
    t1 = n_g % 10
    remainder = t1
    t2 = sum + remainder
    sum = t2
    t3 = n_g / 10
    n_g = t3
    goto L1
L3:
    return 0
```

ST.glb — Parent: *Null*

Name	Type	Category	Size	Offset
n_g	int	global	4	
main	void → int	func	0	0

ST.main() — Parent: *ST.glb*

Name	Type	Category	Size	Offset
sum	int	local	4	–4

ST.main().while.$1 — Parent: *ST.main()*

Name	Type	Category	Size	Offset
remainder	int	local	4	0
t1	int	temp	4	–4
t2	int	temp	4	–8
t3	int	temp	4	–12

Problem: Symbol Table: 2

Consider the following code:

```
int fact(int num) {
    int k = 1;
    if (num == 0)
        return(k);
    else
        for (int i = 1; i <= num; i++) {
            k = k * i;
        }
    return(k);
}

int ncr(int n, int r) { return fact(n) / (fact(r) * fact(n - r)); }

int n = 7, r = 4;

int main() {
    int ncr_var = ncr(n, r);

    return 0;
}
```

1. Convert the code to 3-address
2. Show the symbol tables of the code with user-defined as well as compiler-generated names (ignore constants)

Solution: Symbol Table: 2


```
fact:
    k = 1
    if num == 0 goto L4
L1: i = 1
L2: if i<= num goto L3
    goto L4
L3: k = k * i
    i = i + 1
    goto L2
L4: return k
ncr:
    param n
    t1 = call fact, 1
    param r
    t2 = call fact, 1
    t3 = n-r
    param t3
    t4 = call fact, 1
    t5 = t2 * t4
    t6 = t1 / t5
    return t6
n_g = 7
r_g = 4
main:
    param n_g
    param r_g
    ncr_var = call ncr,2
    return 0
```

ST.glb				Parent: *Null*
Name	Type	Category	Size	Offset
fact	int → int	func	0	0
ncr	int×int → int	func	0	0
n_g	int	global	4	
r_g	int	global	4	
main	void → int	func	0	0
ST.main()			Parent:	*ST.glb*
ncr_var	int	local	4	–4
ST.ncr()			Parent:	*ST.glb*
n	int	param	4	+8
r	int	param	4	+4
t1	int	temp	4	–4
t2	int	temp	4	–8
t3	int	temp	4	–12
t4	int	temp	4	–16
t5	int	temp	4	–20
t6	int	temp	4	–24
ST.fact()			Parent:	*ST.glb*
n	int	param	4	+4
k	int	local	4	–4
ST.fact().for.$1			Parent:	*ST.fact()*
i	int	local	4	0

Problem: Symbol Table: 3

Consider the following code:

```
class Shape { protected: int width, height; public:
        Shape(int a = 0, int b = 0) : width(a), height(b) { }
        virtual int area() = 0;
};
class Rectangle: public Shape { public:
        Rectangle(int a = 0, int b = 0): Shape(a, b) { }
        int area () { return (width * height); }
};
class Triangle: public Shape { public:
        Triangle(int a = 0, int b = 0): Shape(a, b) { }
        int area () { return (width * height / 2); }
};
int main() {
    Shape *shape;
    Rectangle rec(10,7);
    Triangle  tri(10,5);

    // Calls to methods

    return 0;
}
```

1. Show the type tables
2. Show the symbol tables with user-defined names (ignore constants). You may also ignore compiler-generated names.

Solution: Symbol Table: 3

ST.glb			Parent: *Null*	
Name	Type	Category	Size	Offset
`main`	void → int	func	0	0
ST_type.glb			Parent: *Null*	
Name	Type	Category	Size	Offset
`Shape`	class Shape		12	
`Rectangle`	class Rectangle		12	
`Triangle`	class Triangle		12	
ST_type.class Shape			Parent: *ST_type.glb*	
Name	Type	Category	Size	Offset
`width`	int	member	4	0
`height`	int	member	4	–4
`vtable`	void *	virtual method table	4	–8
`Shape`	int×int → class Shape	method	0	0
`~Shape`	class Shape * → void	method	0	0
ST_type.class Rectangle			Parent: *ST_type.class Shape*	
Name	Type	Category	Size	Offset
`vtable`	void *	virtual method table	4	0
`Rectangle`	int×int → class Shape	method	0	0
`~Rectangle`	class Rectangle * → void	method	0	0
ST_type.class Triangle			Parent: *ST_type.class Shape*	
Name	Type	Category	Size	Offset
`vtable`	void *	virtual method table	4	0
`Triangle`	int×int → class Shape	method	0	0
`~Triangle`	class Triangle * → void	method	0	0

Solution: Symbol Table: 3

ST_type.class Shape.vTable				Parent: *ST_type.class Shape*
Name	Type	Category	Size	Offset
`class Shape::area()`	void → int	pure func	0	0

ST_type.class Rectangle.vTable				Parent: *ST_type.class Rectangle*
Name	Type	Category	Size	Offset
`class Rectangle::area()`	void → int	func	0	0

ST_type.class Triangle.vTable				Parent: *ST_type.class Triangle*
Name	Type	Category	Size	Offset
`class Triangle::area()`	void → int	func	0	0

ST.main()				Parent: *ST.glb*
Name	Type	Category	Size	Offset
`shape`	class Shape *	local	4	–4
`rec`	class Rectangle	local	8	–8
`tri`	class Triangle	local	8	–16

Problem: Arithmetic Expression: 1

Given following grammar:

$$
\begin{array}{lcl}
L & \rightarrow & L\ S\ \backslash\mathrm{n} \mid S\ \backslash\mathrm{n} \\
S & \rightarrow & \textbf{id} = E \mid E \\
E & \rightarrow & E + E \mid E - E \mid E * E \mid E\ /\ E \mid (E) \mid - E \mid \textbf{num} \mid \textbf{id}
\end{array}
$$

corresponding attributed grammar, and the Bison specs for translation to the three-address codes (as discussed in the Module 5 Lecture), translate the following inputs to the three-address codes. Illustrate with the steps of reduction and the annotated (with attributes) parse tree.

1.
```
a = (2 + 3) * 4
```

2.
```
t = a
a = b
b = t
```

3.
```
a = 1
b = 2
c = -(a + b) / 3
b = c - 5 * a
```

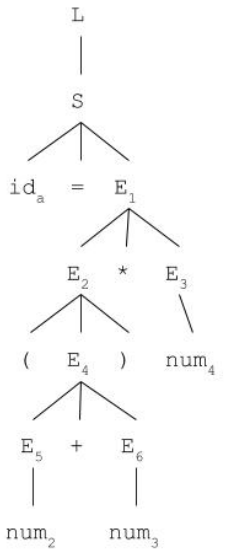
Problem: Arithmetic Expression: 1

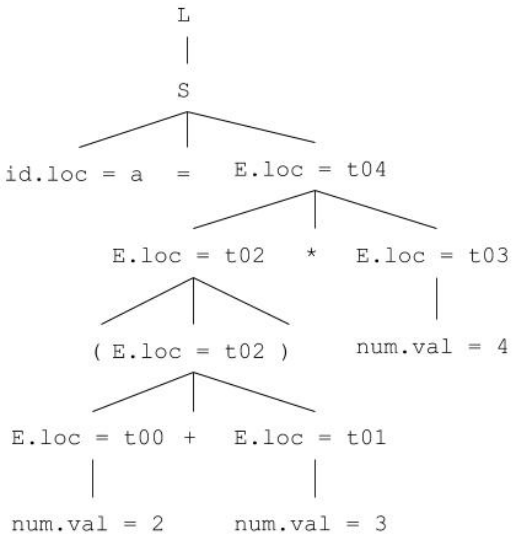

```
TAC

t00 = 2
t01 = 3
t02 = t00 + t01
t03 = 4
t04 = t02 * t03
a = t04

Reductions

E -> num
E -> num
E -> E + E
E -> (E)
E -> num
E -> E * E
S -> id = E
```





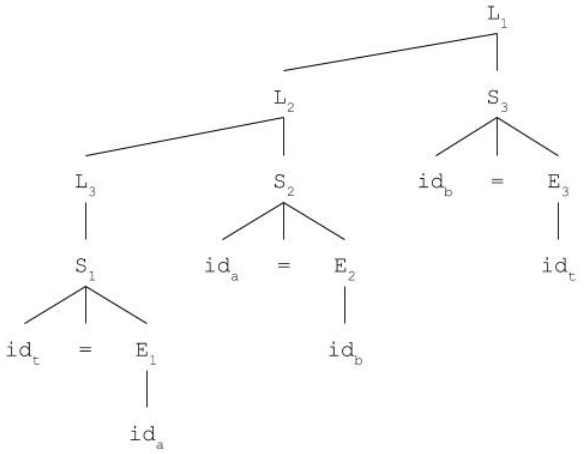
Problem: Arithmetic Expression: 2

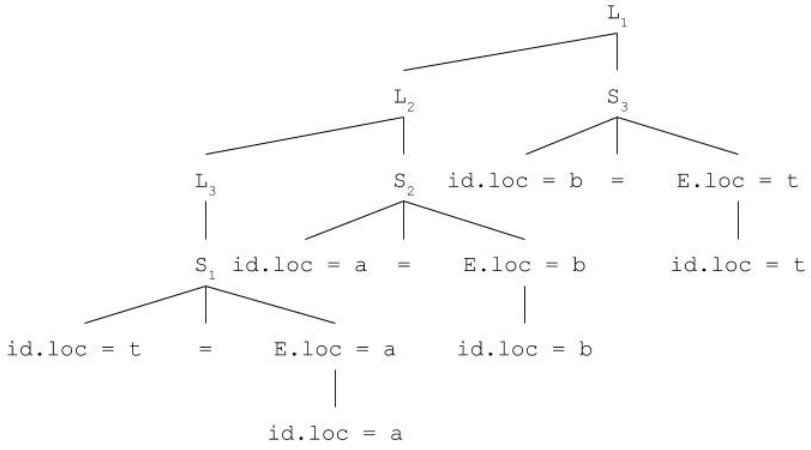
TAC

```
t = a
a = b
b = t
```

Reductions

```
E -> id
S -> id = E
E -> id
S -> id = E
E -> id
S -> id = E
```

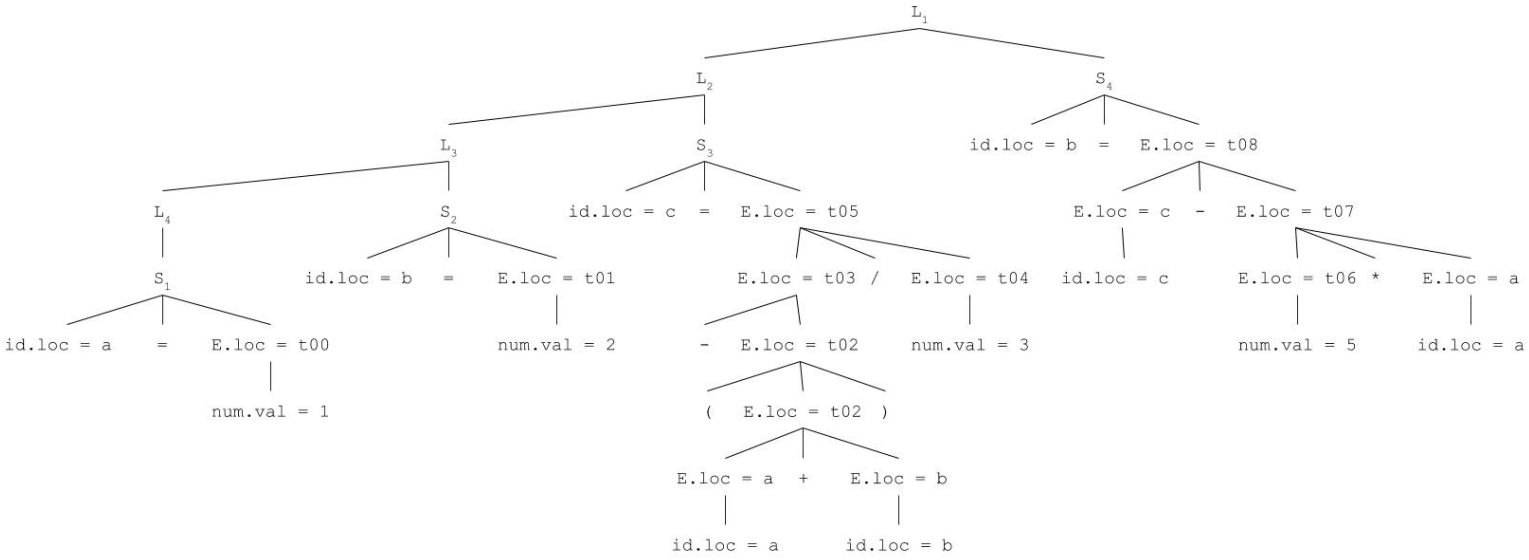




Problem: Arithmetic Expression: 3

TAC	Reductions
t00 = 1	E -> num
a = t00	S -> id = E
t01 = 2	E -> num
b = t01	S -> id = E
t02 = a + b	E -> E + E
t03 = - t02	E -> (E)
	E -> - E

TAC	Reductions
t04 = 3	E -> num
t05 = t03 / t04	E -> E / E
c = t05	S -> id = E
t06 = 5	E -> num
t07 = t06 * a	E -> E * E
t08 = c - t07	E -> E - E
a = t08	S -> id = E



Problem: Boolean Expression: 1

Consider following grammar where **relop** $\in \{==, !=, <, \leq, >, \geq\}$ having equal precedence and left-associativity, and $!(unary) \gtrdot \&\& \gtrdot ||$ with binary operators being left-associative and the unary operator being right associative.

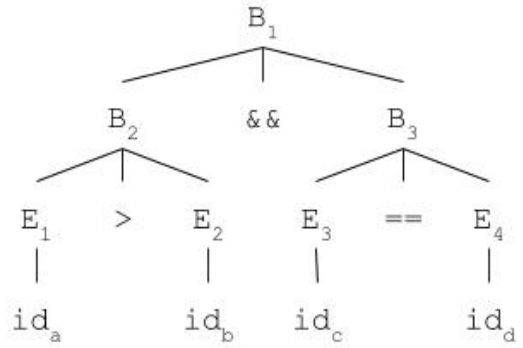
$$
\begin{array}{lcl}
B & \rightarrow & B \,||\, B \mid B \,\&\&\, B \mid !B \mid (B) \mid E \textbf{ relop } E \mid \textbf{true} \mid \textbf{false} \\
E & \rightarrow & \textbf{num} \mid \textbf{id}
\end{array}
$$

Also, consider the corresponding attributed grammar, and the Bison specs for translation to the three-address codes (as discussed in the Module 5 Lecture). Translate the following inputs **by value** to the three-address codes. Illustrate with the steps of reduction and the annotated (with attributes) parse tree.

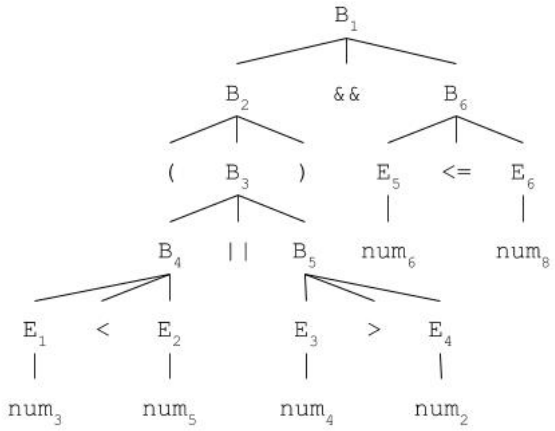
1. `a > b && c == d`
2. `(3 < 5 || 4 > 2) && 6 <= 8`
3. `true && false || true`
4. `true || false && true`

Problem: Boolean Expression: 1.1-2

TAC:	Reductions:
	E -> id
	E -> id
t00 = a > b	B -> E > E
	E -> id
	E -> id
t01 = c == d	B -> E == E
t02 = t00 && t01	B -> B && B



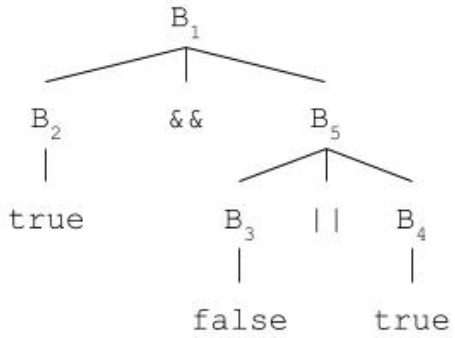
TAC:	Reductions:
t00 = 3	E -> num
t01 = 5	E -> num
t02 = t00 < t01	B -> E < E
t03 = 4	E -> num
t04 = 2	E -> num
t05 = t03 > t04	B -> E > E
t06 = t02 || t05	B -> B || B
	B -> (B)
t07 = 6	E -> num
t08 = 8	E -> num
t09 = t07 <= t08	B -> E <= E
t10 = t06 && t09	B -> B && B



Problem: Boolean Expression: 1.3-4

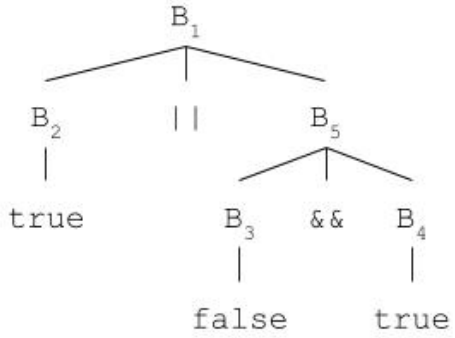

```
TAC:                Reductions:

t00 = true          B -> true
t01 = false         B -> false
t02 = true          B -> true
t03 = t01 || t02    B -> B || B
t04 = t00 && t03    B -> B && B
```




```
TAC:                Reductions:

t00 = true          B -> true
t01 = false         B -> false
t02 = true          B -> true
t03 = t01 && t02    B -> B && B
t04 = t00 || t03    B -> B || B
```



Problem: Boolean Expression: 2

Consider following grammar where **relop** $\in \{==, !=, <, \leq, >, \geq\}$ having equal precedence and left-associativity, and $!(\textit{unary}) \gtrdot \&\& \gtrdot ||$ with binary operators being left-associative and the unary operator being right associative.

$$\begin{array}{lcl} B & \rightarrow & B \,||\, B \mid B \,\&\&\, B \mid !B \mid (B) \mid E \textbf{ relop } E \mid \textbf{true} \mid \textbf{false} \\ E & \rightarrow & \textbf{num} \mid \textbf{id} \end{array}$$

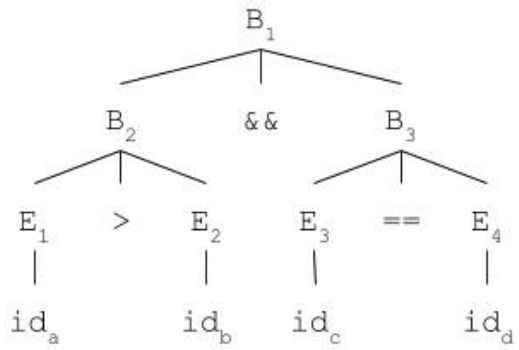
Also, consider the corresponding attributed grammar, and the Bison specs for translation to the three-address codes (as discussed in the Module 5 Lecture). Translate the following inputs **by control** to the three-address codes. Illustrate with the steps of reduction and the annotated (with attributes) parse tree.

1. `a > b && c == d`
2. `(3 < 5 || 4 > 2) && 6 <= 8`
3. `true && false || true`
4. `true || false && true`
5. Compare these solutions with the solutions by value

Problem: Boolean Expression: 2.1

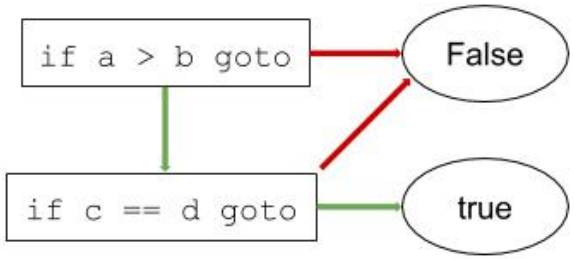
TAC:

```
100: if a > b goto 102
101: goto 105
102: if c == d goto 104
103: goto 105
104: goto 000 ( true )
105: goto 000 ( false )
```



Reductions:

```
E -> id
E -> id
B -> E > E
E -> id
E -> id
B -> E == E
B -> B && B
```



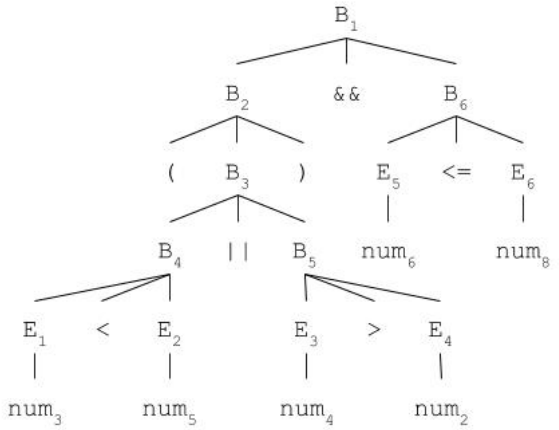
Problem: Boolean Expression: 2.2

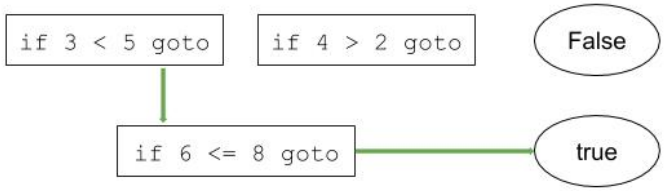
TAC:

```
100: t00 = 3
102: t01 = 5
103: if t00 < t01 goto 109
104: goto 105
105: t03 = 4
106: t04 = 2
107: if t03 > t04 goto 109
108: goto 114
109: t07 = 6
110: t08 = 8
111: if t07 <= t08 goto 113
112: goto 114
113: goto 000 ( true )
114: goto 000 ( false )
```

Reductions:

```
E -> num
E -> num
B -> E < E
E -> num
E -> num
B -> E > E
B -> B || B
B -> ( B )
E -> num
E -> num
B -> E <= E
B -> B && B
```

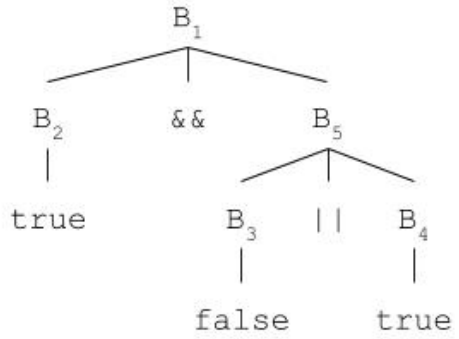




Problem: Boolean Expression: 2.3

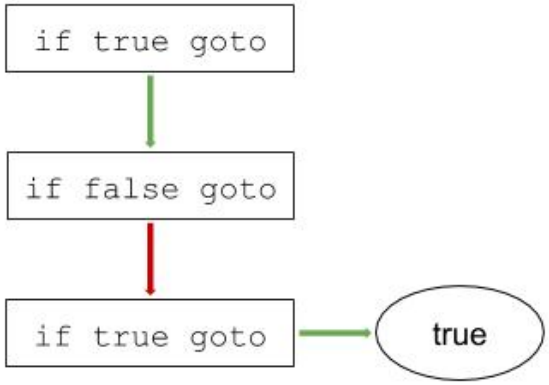

```
TAC:

100: t00 = true
101: if t00 goto 103
102: goto 110
103: t01 = false
104: if t01 goto 109
105: goto 106
106: t02 = true
107: if t02 goto 109
108: goto 110
109: goto 000 ( true )
110: goto 000 ( false )
```




```
Reductions:

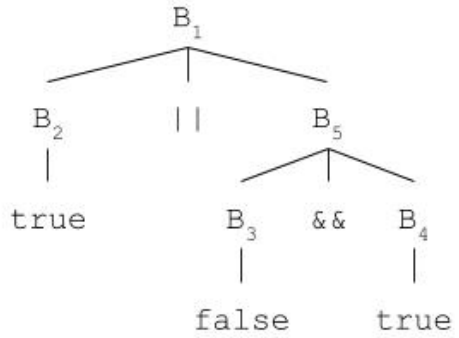
B -> true
B -> false
B -> true
B -> B || B
B -> B && B
```



Problem: Boolean Expression: 2.4

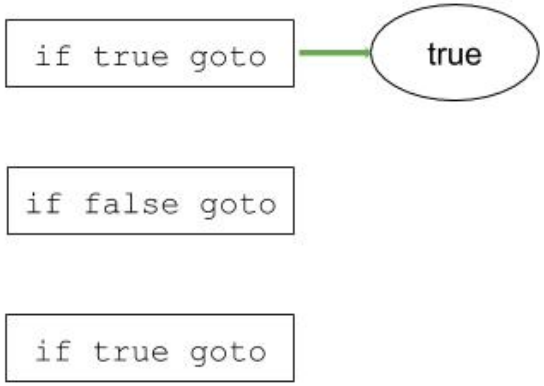

```
TAC:

100: t00 = true
101: if t00 goto 109
102: goto 103
103: t01 = false
104: if t01 goto 106
105: goto 110
106: t02 = true
107: if t02 goto 109
108: goto 110
109: goto 000 ( true )
110: goto 000 ( false )
```




```
Reductions:

B -> true
B -> false
B -> true
B -> B && B
B -> B || B
```



Problem: Boolean Expression: 3

Given the exclusive-OR operator ($\hat{}$), write the actions for the following production rule:

$$B \rightarrow B \hat{} B$$

Solution: Boolean Expression: 3

$$
\begin{aligned}
B \quad \rightarrow \quad & B_1 \hat{} M \ B_2 \\
& \{ \ backpatch(B_1.truelist, nextinstr); \\
& \ \ emit(B_1.loc, " = ", true); \\
& \ \ emit("goto", \ M.instr); \\
& \ \ backpatch(B_1.falselist, nextinstr); \\
& \ \ emit(B_1.loc, " = ", false); \\
& \ \ emit("goto", \ M.instr); \\
& \\
& \ \ B.truelist = makelist(nextinstr); \\
& \ \ backpatch(B_2.falselist, nextinstr); \\
& \ \ emit("if", B_1.loc, \ "goto", \ "....."); \\
& \ \ B.falselist = makelist(nextinstr); \\
& \ \ emit("goto", \ "....."); \\
& \\
& \ \ temp = makelist(nextinstr); \\
& \ \ B.falselist = merge(B.falselist, temp); \\
& \ \ backpatch(B_2.truelist, nextinstr); \\
& \ \ emit("if", B_1.loc, \ "goto", \ "....."); \\
& \ \ temp = makelist(nextinstr); \\
& \ \ B.truelist = merge(B.truelist, temp); \\
& \ \ emit("goto", \ "....."); \ \}
\end{aligned}
$$

Practice Problems

1. Translate the following to 3 address:
   1. ```
      x = 6
      y = x + x / 2
      x = -y + x
      y = x * x
      ```
   2. `4 > 5 && 7 == 8 || 6 > 4`
   3. `6 > 4 || 4 > 5 && 7 == 8`
2. Write semantic actions for translating the following production rules where B is a Boolean expression and $\overline{\bigwedge}$ & $\overline{\bigvee}$ are NAND & NOR operators respectively:
   1. $B \rightarrow B \overline{\bigwedge} B$
   2. $B \rightarrow B \overline{\bigvee} B$
      ```