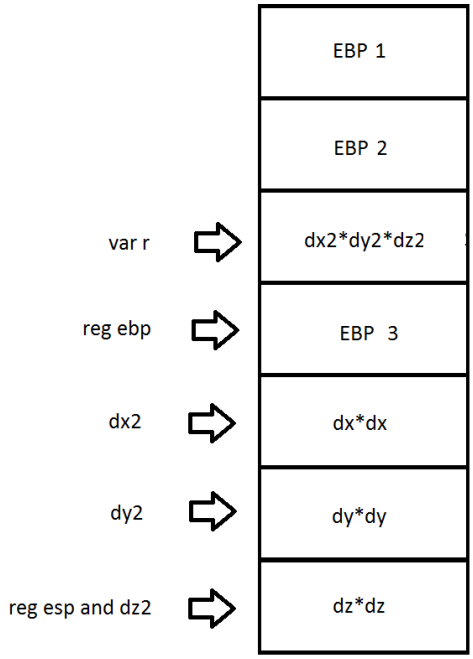
Problem 1, Stack frames (5%)



Problem 2, SDD (7.5%)

a) S-attributed is a subset of L-attributed. A S-attributed SDD requires that every attribute is synthesized. In other words you can determine a attribute by traversing the tree bottom up.

A L-attributed SDD requires that every attribute is synthesized OR inherited with the following limitations: 1) An attribute can only inherit from attributes that is associated with the same productions. 2) It can only inherit from an attribute that is on its left in the production.

b)

i) It is not S-attributed, since in the first production B.b is depended on its parent's attribute, A.a. It is L-attributed, since every attribute is either synthesized or inherited (and abides the two limitations).

ii) It is both S-attributed and L-attributed. All attributes are synthesized.

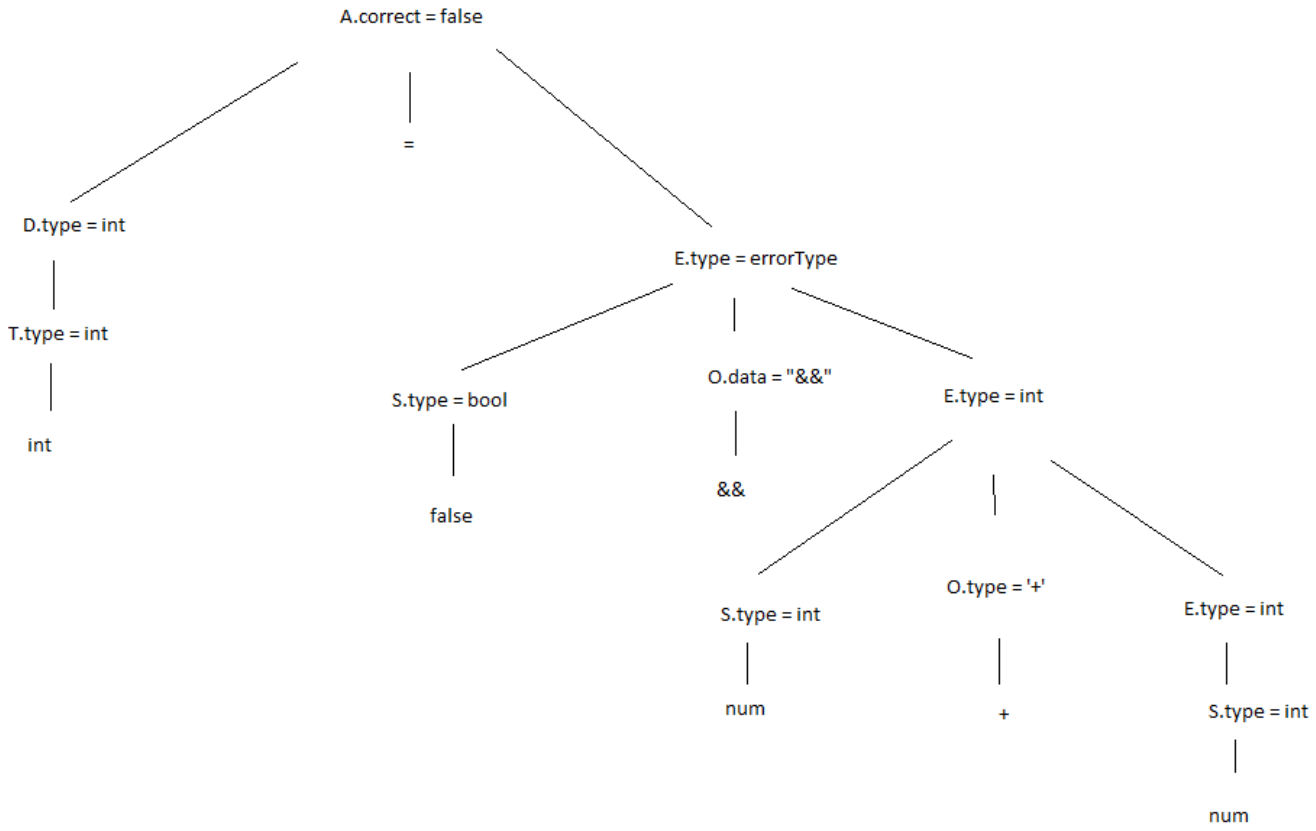
iii) Neither. C.a = C.b + D.a is not synthesized and it violates the 2) limitation for inherited attributes.

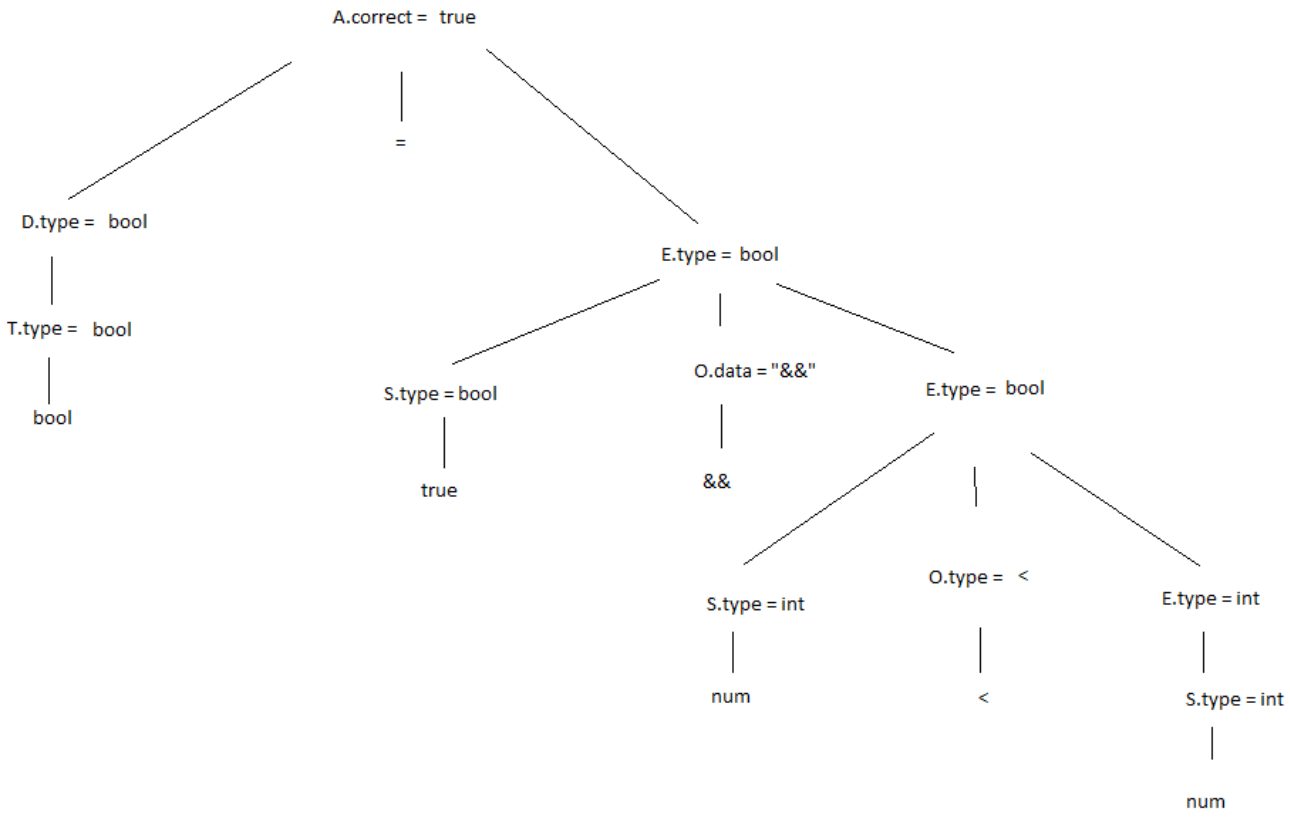
iv) L-attributed. B.b = C.b + D.b and D.b = B.b + C.b is not synthesized, but it is inherited and abides the two limitations.

Problem 3, Type checking (7.5%)

Production	Semantic rule
A::= D = E	A.correct = (D.type == E.type)
D::= T **id**	D.type = T.type
T.type = **int**	T.type = int
T.type = **bool**	T.type = bool
E::= SOE	if (O.data == '<' && S.type == E.type == num) E.type = bool else if (O.data == "&&" && S.type == E.type == bool) E.type = bool else if (O.data == '+' && S.type ==E.type==num) E.type = num else E.type = errorType
E ::= S	E.type = S.type
S::= num	S.type = num
S::= true | false	S.type = bool
O::= +	O.data = '+'
O::= <	O.data = '<'
O::= &&	O.data = "&&"

b)


A.correct = false
=
D.type = int
E.type = errorType
T.type = int
int
S.type = bool
O.data = "&&"
E.type = int
false
&&
S.type = int
O.type = '+'
E.type = int
num
+
S.type = int
num


A.correct = true
=
D.type = bool
T.type = bool
bool
E.type = bool
S.type = bool
true
O.data = "&&"
&&
E.type = bool
S.type = int
num
O.type = <
<
E.type = int
S.type = int
num

Problem 4, Optimization (5%)

High-IR level:

1) **Eliminate dead code.** Reduces space used.
2) **Code motion**. Avoids doing the same computation several times over in a loop, when we can just create a constant outside the loop.

Low-IR level:

1) **Reduce number of temporaries**. Thus reduced the number of accesses/instructions you have to preform.
2) **Don't generate multiple adjacent label instructions**. If the label do the same thing, we'll only use more space by generating them several times.
3) **Eliminate jumps to unconditional jumps**. If we know a condition is always met, we will make the program faster by not checking the condition every time we run the code.

Problem 5, Assembly (5%)

My compiler provides this code:

```
_addFive:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   %ebp
        movl    %esp,%ebp
        pushl   $1
        movl    %ebp,%ebx
        movl    (%ebx),%ebx
        pushl   8(%ebx)
        call    _addFour
        pushl   %eax
        popl    %ebx
        popl    %eax
        addl    %ebx,%eax
        pushl   %eax
        popl    %eax
        movl    %ebp,%esp
        popl    %ebp
        leave
```

```
        ret
_addFour:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   %ebp
        movl    %esp,%ebp
        pushl   $4
        movl    %ebp,%ebx
        movl    (%ebx),%ebx
        pushl   8(%ebx)
        popl    %ebx
        popl    %eax
        addl    %ebx,%eax
        pushl   %eax
        popl    %eax
        movl    %ebp,%esp
        popl    %ebp
        leave
        ret
```