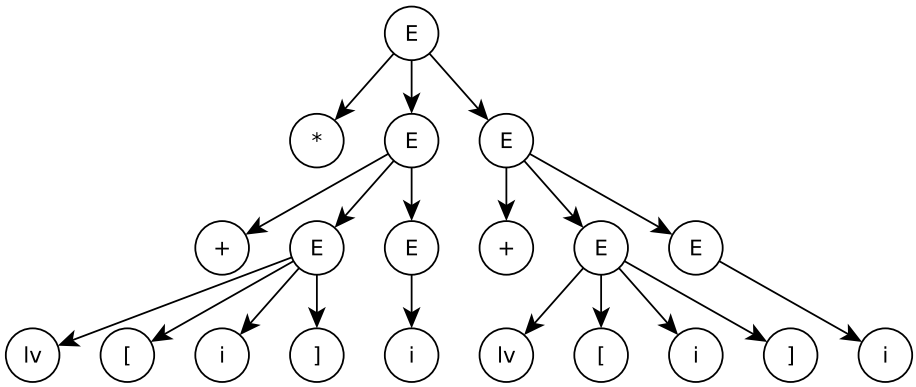
Question 1.

(a) In the derivation, E is *expr* and i is **int-literal**. So, the goal is to derive $\boxed{\texttt{* + lv [i] i + lv [i] i}}$.

Working string	Production
$\underline{\text{E}}$	E → * E E
* $\underline{\text{E}}$ E	E → + E E
* + $\underline{\text{E}}$ E E	E → **lv** [i]
* + **lv** [i] $\underline{\text{E}}$ E	E → i
* + **lv** [i] i $\underline{\text{E}}$	E → + E E
* + **lv** [i] i + $\underline{\text{E}}$ E	E → **lv** [i]
* + **lv** [i] i + **lv** [i] $\underline{\text{E}}$	E → i
* + **lv** [i] i + **lv** [i] i	

Parse tree:



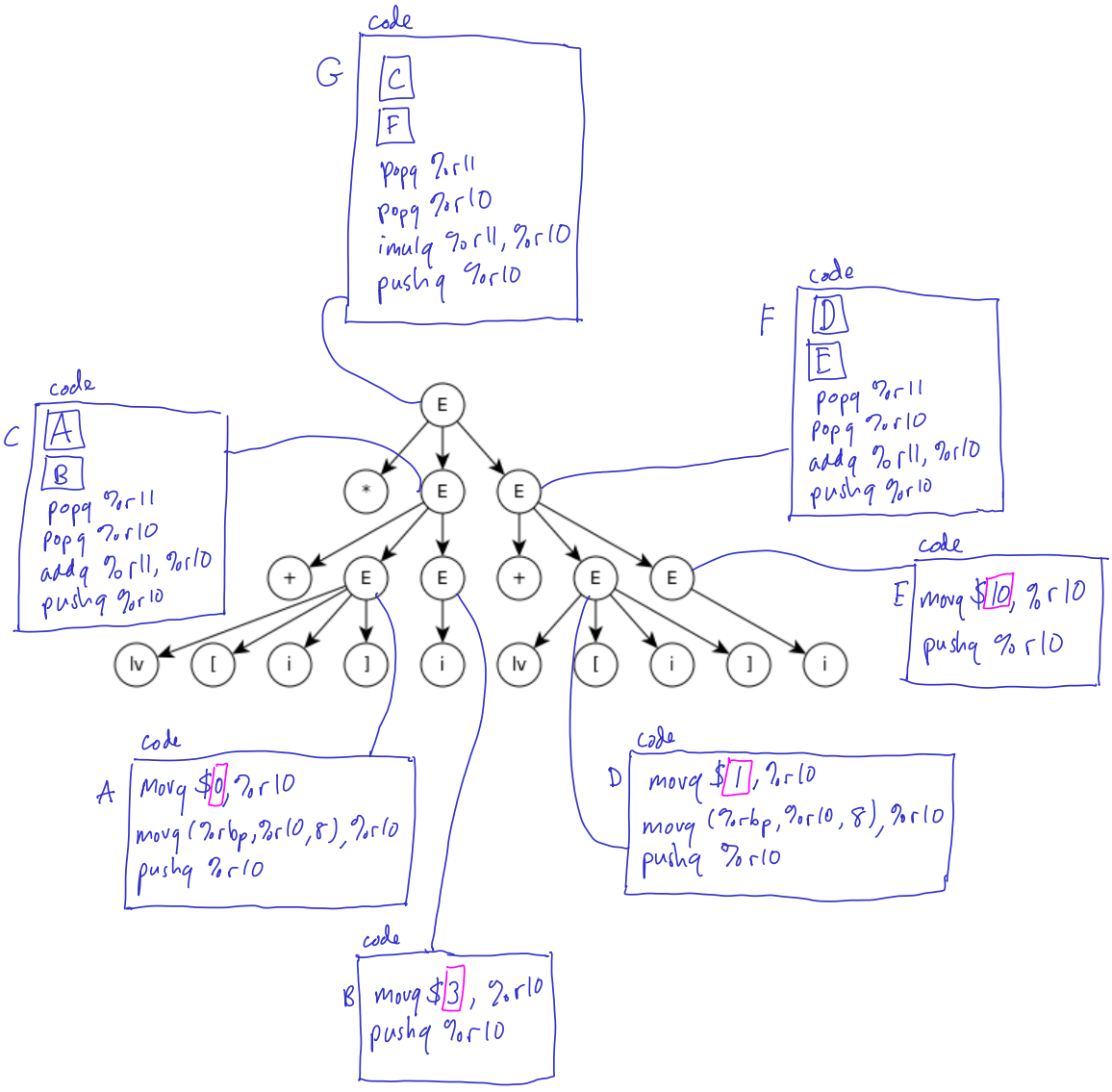
(b) Possible attribute grammar ("code" attribute is a string, + means string concatenation):

Production	Attribute rules
$expr_0$ → ***** $expr_1$ $expr_2$	$expr_0$.code ← $expr_1$.code + $expr_2$.code + "popq %r11" + "\n" + "popq %r10" + "\n" + "imulq %r11, %r10" + "\n" + "pushq %r10" + "\n"
$expr_0$ → **+** $expr_1$ $expr_2$	$expr_0$.code ← $expr_1$.code + $expr_2$.code + "popq %r11" + "\n" + "popq %r10" + "\n" + "addq %r11, %r10" + "\n" + "pushq %r10" + "\n"
expr → **int-literal**	*expr*.code ← "movq $" + **int-literal**.lexeme + ", %r10" + "\n" + "pushq %r10" + "\n"
expr → **lv** [**int-literal**]	*expr*.code ← "movq $" + **int-literal**.lexeme + ", %r10" + "\n" + "movq (%rbp,%r10,8), %r10" + "\n" + "pushq %r10" + "\n"

Assumptions:

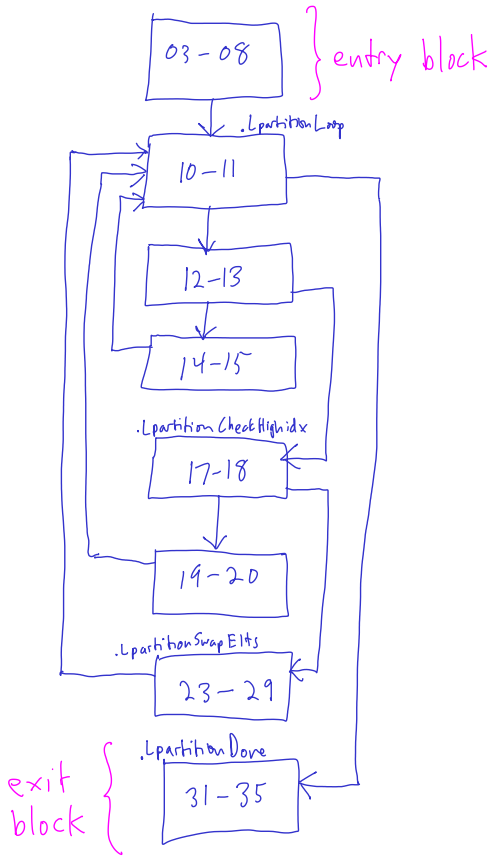
- The %rbp register points to a storage area for local variables (essentially, an array of 64 bit storage locations)
- Integers are 64 bit
- %r10 and %r11 are available for use as temporaries
- Operand values are on the stack
- The result is left as the top value on the stack

(c)



The code attribute is a synthesized attribute, so evaluation proceeds from the bottom up.

Question 2.



Question 3 (628).

There is no inherently right or wrong answer to this question.

If a procedure call is known not to throw any exceptions, then it can be treated as an ordinary instruction. Analyses will need to make conservative assumptions about the effects of calling the instruction, e.g., the values of global variables might change, the values of variables whose addresses are passed might change, etc.

If a procedure call can throw exceptions, then there is a strong argument for treating exceptions as a type of control flow, in which case the call instruction should be the last in the basic block. The control edges leading from the block would include a fall through edge (representing the case where the procedure doesn't throw an exception) and one or more exception edges, which could lead to exception handler blocks, or possibly to a block representing the possibility that the exception is thrown out of the procedure body the control flow graph represents.

It is possible to put calls to exception-throwing procedures in the middle of the block if all local analyses are written with the assumption that an exception could occur in the middle of the block. This will complicate every local analysis, reducing the usefulness of the control graph representation.