Problem 1

- The output of the program is 4.
- One (of several) correct activation record layouts looks as follows:

```
------------
|   args   |
------------
|   fp1    |
------------    main
|   ra1    |
------------
|    p     |
------------
|    r     |
------------
------------
|   r/y    |
------------
|   p/x    |
------------
|   fp2    |    f
------------
|   ra2    |
------------
|    t     |
------------
------------
|   y/s    |
------------
|   t/r    |
------------    g
|   fp3    |
------------
|   ra3    |
------------
|    4     |
------------
```

  - SP points to 4
  - FP points to fp3
  - control links: fp3 points to fp2, and fp2 points to fp1
  - Return addresses: ra2 points to the end of function main (right after f(p,r) in main); ra3 points to the end of function f (right after g(t, y) in f).
  - Calling convention:
    * args: caller

        * fp: we accepted two possible answers: (1) either, or (2) callee
        * ra: either
        * locals: callee

      Note: fp/ra can be in any order, as long as they are consistent across the whole stack. Locals 'p' and 'r' in main can be in any order as well. "args" for main is optional, as is the "4" in the activation record for g. fp can point to either fp3 or ra3. Control links should point accordingly to where fp points to.

  Point deductions: Missing part of activation record: -1. Incorrect order of stack contents: -1. Incorrect or missing boundaries: -1. Incorrect answer to what caller/callee pushes: -1. SP/FP not shown or incorrectly pointed: -1. Control links missing or inaccurate/inconsistent: -1. Missing pointers from RA's or incorrect pointers: -1. Stack not drawn: -1.

- If sp (whose value changes during the execution of the method) is to be used for accessing AR entries, the compiler must maintain the distance of sp's value from some fixed location in the AR. This value is then used to adjust the offset when generating the code for accessing the AR entry. Note: the generated code isn't more complex than that of the fp-based code; only the code generator is more complex.

Optimizing the Activation Records

- Necessity of control link and return address:
  - Control link is not needed because the location of the activation record can be static, and therefore known at compile time. We will not need a seperate register to store this address.
  - Return address is still needed because the function can be called from multiple call sites in the program. A return address won't be needed if a function has only one point of invocation in the program, meaning that this location can be determined statically.
- Recursion Detection Algorithm:

  There were different kinds of answers for this algorithm. The main idea is the following, a DFS following the method invocations in the program. If at any point we see an already visited method, then there is recursion:

```
Set visited; // maintain a set of visited methods

visit(MethodDeclaration d) {
   visited = {d.name};

   visit all of the body statements.
}

visit(MethodInvocation i) {
   if(i element of visited) {
       signal that we have recursion
     }
```

```
        visited = visited union {i.name};
        visit the body statements of the method declaration for d;
        visited = visited - i.name;
    }
```

How will we be able to get at the body statements of an arbitrary method declaration? An initial pass will be needed to map method names to their bodies, using a symbol table. Many answers confused the structure of the AST between MethodDeclaration nodes which contain the body statements and MethodInvocation, which do not contain body statements (they represent method calls).

Other common errors:

- Assuming that the compiler can "execute" the program, as opposed to analyze it.
- Not handling indirect recursion. Usually this resulted in an algorithm that simply checked to see if a method name appeared in its own body as an invocation.

Problem 2

Why is global dataflow analysis imprecise?

- Fill in the blanks:

  It suffices to make the second if statement execute whenever the first if statement executes. The easiest way to do this is to have the same condition in both if statements, for example `true`. There were many other solutions as well.

- Why is this imprecise / (overly) conservative?

  This analysis is imprecise because in the CFG there are four different paths, one of which contains a path in which there was a potentially uninitialized variable. Since dataflow analysis only considers paths and does not evaluate the conditionals to determine which paths are actually taken, we end up with an imprecise (but safe) analysis.

  Saying that dataflow analysis is conservative was not enough as an answer. You must state 1) what is going on in the CFG, 2) why dataflow analysis is conservative in this specific case.

Problem 3

$$\frac{O \vdash e_1 : int \qquad O \vdash e_2 : int}{O \vdash e_1 + e_2 : int}\ Correct \qquad\qquad \frac{O \vdash e_1 : int}{O \vdash e_1 + e_2 : int}\ Incorrect$$

The incorrect rule is unsound. It allows incorrect programs like `3 + false` to pass.

$$\frac{O \vdash e_0 : int \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash T_0\ x = e_0\ ;\ e_1 : T_1}\ Incorrect \qquad\qquad \frac{O \vdash e_0 : T_0 \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash T_0\ x = e_0\ ;\ e_1 : T_1}\ Correct$$

The incorrect rule is both unsound and incomplete. It is unsound because it allows incorrect programs like `boolean x = 3; x` to pass, and incomplete because it does not allow safe programs like `boolean x = false; x` to pass (only one program example was necessary for full credit).

$$\frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \ldots \\ O, M \vdash e_n : T_n \\ M(T_0, f) = (T_1', T_2', \ldots, T_n', T_r) \\ T_i = T_i' \text{ (for } 1 \leq i \leq n) \end{array}}{O, M \vdash e_0.f(e_1, \ldots, e_n) : T_r} \; Incorrect \qquad \frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \ldots \\ O, M \vdash e_n : T_n \\ M(T_0, f) = (T_1', T_2', \ldots, T_n', T_r) \\ T_i \leq T_i' \text{ (for } 1 \leq i \leq n) \end{array}}{O, M \vdash e_0.f(e_1, \ldots, e_n) : T_r} \; Correct$$

The incorrect rule is incomplete. Suppose we have classes `A` and `B`, with `B` $\leq$ `A`. If we have some method `void foo(A a)` $\{ \ \ldots \ \}$, the method call `foo(new B())` will not be allowed to pass, even though the code is safe.

Problem 4

The `.....` should be filled as follows (in order): `push(peek(-1));`, `TypeDeclaration`, `push(peek(-4));`, `-1`, `peek(-1)`, `0`. Here is the complete, filled-in specification.

```
PROLOGUE: [|
    private AST ast = new AST(new HashMap());

    private SimpleName makeSimpleName(int offset) {
         return ast.newSimpleName(((Token)peek(offset)).getLexeme()); }
|]

PROGRAM -> _ [| push(new ArrayList()); |] CLASSDECLLIST <EOF>;

CLASSDECLLIST   -> CLASSDECL
    [|
        ((List)peek(-1)).add(peek(0));
        push(peek(-1));
    |]
  CDTAIL ;

CDTAIL -> CLASSDECLLIST | _ ;

CLASSDECL -> <CLASS> <IDENTIFIER>
    [|
        TypeDeclaration td = ast.newTypeDeclaration();
        td.setName(makeSimpleName(0));
        push(td);
    |]
  <LBRACE>
    [|
        push(((TypeDeclaration)peek(-1)).bodyDeclarations());
    |]
  FIELDDECLLIST <RBRACE>
    [|
        push(peek(-4));
    |] ;

FIELDDECLLIST -> FIELDDECL
    [|
       ((List)peek(-1)).add((FieldDeclaration)peek(0));
       push(peek(-1));
    |]
  FIELDDECLLIST | _ ;

FIELDDECL ->  <INT> FIELDID <SEMICOLON>
    [|
        ((FieldDeclaration)peek(-1)).setType(ast.newPrimitiveType(PrimitiveType.INT));
        push(peek(-1));
    |] ;

FIELDID ->  <IDENTIFIER>
    [|
        VariableDeclarationFragment vdf = ast.newVariableDeclarationFragment();
        vdf.setName((SimpleName)ast.newSimpleName(((Token)peek(0)).getLexeme()));
        push(ast.newFieldDeclaration(vdf));
    |] ;
```