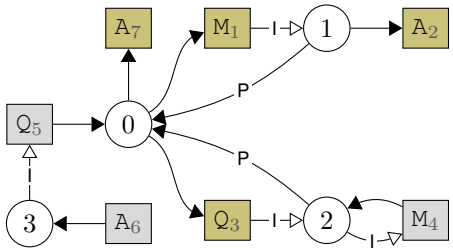
Exercise 1: Name Resolution

(a) Explain the main aspects of name resolution with scope graphs.

(b) Given the following scope graph, list all reachability paths for reference A_6. Also indicate the path that is chosen according to the visibility rules.



(c) Given the following Java program, give a scope graph that describes its name binding structure. Use the subscripts on identifiers to distinguish different occurrences of the same name.

```
public class A1 {
  int f2;
}

public class B3 {
  int m4(A5 obj6) {
    return obj7.f8;
  }
}
```

Exercise 2: Typing Constraints

(a) Explain the main aspects of type checking with constraints. Discuss how the different parts of your answer are related to the language the checker is for.

(b) Explain the concepts of soundness and completeness in terms of your previous answer.

(c) Given the following type equalities,

```
f(a) == b, g(b,d) == g(f(c),f(e)), h() == e
```

give a substitution that satisfies the equalities.

(d) Explain the concept of principality in terms of your previous answer. Illustrate your answer with a different unifier that would also satisfy the constraints of (c).

Exercise 3: Constraint Semantics

Consider the constraint $d : t$ that gives the type t of a declaration d. The semantic rule for this constraint is given by

$$T, G, s \vDash d : t \qquad \text{where } T(s(d)) = s(t)$$

where $T : \mathit{Decl} \to \mathit{Type}$ is a function from types to declarations, G is a scope graph, and s is a variable substitution.

(a) Given the constraints

```
Var{a @1} : ty1, ty1 == INT(), Var{a @1} : ty2
```

give a substitution s for the variables *ty1* and *ty2*, such that the constraints are *not* satisfiable, according to the semantic rule given above.

(b) We use `type-of(d, ty)` as the textual representation of $d : t$ constraints. The following rule specifies a possible solver for `type-of` constraints:

```
type-of(d,ty1), type-of(d, ty2) <=> ty1 == ty2.
```

Show using a counter-example that this rule is unsound. Explain how the rule is applied in your example.

(c) Modify the rule above to make it sound, and explain why your counter-example is now rejected.

Exercise 4: Constraint Rules

(a) Given the following constraint rules

```
[[ LetRec(binds) ^ (s) : ty ]] :=
  new s_let, s_let ---> s,
  Map1[[ binds ^ (s_let) ]],
  [[ e ^ (s_let) : ty ]].

[[ Bind(x, e) ^ (s) ]] :=
  Var{x} <- s, Var{x} : ty,
  [[ e ^ (s) : ty ]].

[[ IntLit(_) ^ (s) : INT() ]].

[[ VarRef(x) ^ (s) : ty ]] :=
  Var{x} -> s, Var{x} |-> d, d : ty.
```

and the program represented by the following abstract syntax tree,

```
LetRec([
  Bind("x"1,IntLit(42)),
  Bind("y"2,VarRef("x"3)),
], VarRef("y"4))
```

show the constraints that would be generated by applying these rules to the program. Feel free to rename variables if necessary to avoid capture.

(b) Give a solution for the constraints you generated in (a), or explain why a solution is not possible.

(c) The rules from (a) model a recursive let. An alternative semantics for let, where the bodies of the bindings cannot refer to the variables that are defined in the let, but only to variables defined outside the let. Give a set of modified rules that implement these let semantics.