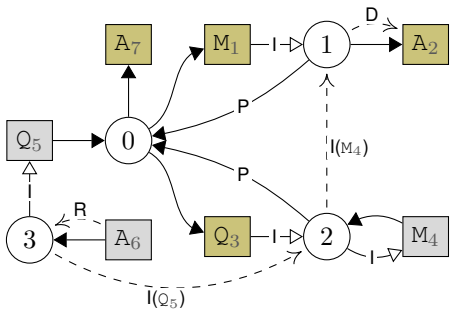
Exercise 1: Name Resolution

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

> **Solution:** The main aspects are:
> - The name binding structure of a program is represented as a graph of scopes, references, and declarations.
> - Resolving a reference is finding a path in the graph from the reference to a matching declaration.
> - The paths used for resolution must be well-formed, which means the labels must match a given well-formedness regular expression.
> - Disambiguation defines which declarations are visible if multiple matching declarations can be reached. Disambiguation is specified using an ordering on path labels.

(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.



> **Solution:** There is one reachable path for A_6, which is shown with the dashed arrows in the figure above. The path is well-formed, since its labels II match the well-formedness P^*I^*. Because there is only one reachable declaration, it is also the visible declaration.

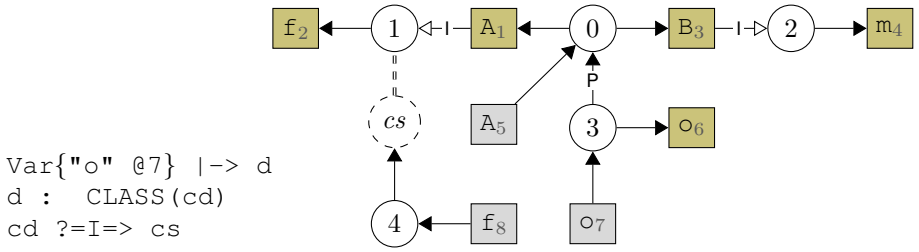
(c) Given the following Java program,

```
public class A1 {
  int f2;
}

public class B3 {
  int m4(A5 o6) {
    return o7.f8;
  }
}
```

give a scope graph that describes its name binding structure. Use the subscripts on identifiers to distinguish different occurrences of the same name. For the type-dependent field name, you also need to list the constraints that are necessary.

> **Solution:** The scope graph and constraints for the program are given by:




```
Var{"o" @7} |-> d
d :  CLASS(cd)
cd ?=I=> cs
```

Note that we kept the variable scope explicit in the graph, and indicated the substitution with the double dashed line. This makes it easier to make the connection with the constraints, than if scope 4 was directly connected to scope 1.

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.

Solution: Type checking with constraints is based on the idea that the property of well-typedness of a program can be stated as a constraint problem. If the constraint problem is satisfied, the program is well-typed. This is implemented in two phases:

1. Constraint *generation* maps the program to a constraint problem. This step is language-dependent, constraint generation rules are defined over the syntax of the language.
2. Constraint *solving* checks if the constraint problem is satisfiable, and tries to find values for the constraint variables. Constraint solving depends only on the constraint language, but is independent of the object language that is being checked.

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

Solution: Soundness and completeness are properties that relate the semantics of constraints to the solver algorithm.

Soundness means that any answer the solver gives, is indeed a valid solution to the constraints, according to the constraint semantics. In the context of type checking, this means that if a program is ill-typed, the equivalent constraint problem must be unsatisfiable, and the solver reports an error that it cannot find a valid solution.

Completeness means that if a valid solution exists, the solver finds it. In the context of type checking, this means that if a program is well-typed, the constraint problem is satisfiable, and the solver returns a valid solution, i.e. a valid assignments for all the constraint variables.

(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.

Solution: A possible unifying substitution is:

$a \mapsto c$ $\quad$ $b \mapsto \texttt{f}(c)$ $\quad$ $d \mapsto \texttt{f(h())}$ $\quad$ $e \mapsto \texttt{h()}$

(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).

Solution: A given substitution s is a principal unifier if every other unifier s' is an instance of s. A unifier s' is an instance of s if a substitution s'' exists, such that s'' applied to s yields s.

In the example, any substitution that fixes a term value t for c is still a unifier for the constraints. However, it would not be principal anymore. The substitution s'' in this scenario would be given by $s \mapsto t$.

Possible additions:

- Principal unifiers are not necessarily unique. For example, the variables a and c could be swapped in our example solution and it would remain principle.
- In type systems there is a similar concept: principle types. These are types that are supertypes of all other valid types for a program. For example, the identity function `\x -> x` may have type **`Int -> Int`** or **`String -> String`**, but its principal type is `forall a. a -> a`.

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.

Solution: The substitution

$$\mathit{ty1} \mapsto \texttt{INT()} \qquad\qquad \mathit{ty2} \mapsto \texttt{STRING()}$$

cannot satisfy the constraints. The function T enforces that every declaration has only one type. If we pick $T(\texttt{Var\{a @1\}}) = \texttt{INT()}$, the third constraint is unsatisfiable. If we pick $T(\texttt{Var\{a @1\}}) = \texttt{STRING()}$, the first constraint is unsatisfiable. If we pick any other value, both will be unsatisfiable.

(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.

Solution: The following constraints (for some declaration d) are a counter example:

```
type-of(d, INT()), type-of(d, STRING()), type-of(d, INT())
```

The constraints specify two different types for the declaration. Therefore, the solver should not be able to solve these constraints. To show this rule is unsound, we need to show a derivation that accepts these constraints. A possible derivation is:

```
     type-of(d, INT()), type-of(d, STRING()), type-of(d, INT())
<=> type-of(d, STRING()), INT() == INT()
<=> type-of(d, STRING())
```

The first step applies the given rule to the first and last constraint, simplifying it to an equality. The second step discharges the equality, which is obviously satisfied. Since no more rules can be applied, the solver concludes the constraint problem is satisfied.

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

Solution: A sound version of the rule is

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

Instead of checking consistency only between pairs, we now check consistency of all constraints. For example, the derivation we had before will now look as follows:

```
     type-of(d, INT()), type-of(d, STRING()), type-of(d, INT())
<=> type-of(d, INT()), type-of(d, STRING()), INT() == INT()
<=> type-of(d, INT()), type-of(d, STRING())
<=> INT() == STRING()
<=> false
```

Now the solver ends with an equality that cannot be satisfied, and the problem is reported as unsatisfiable.

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.

Solution: Step by step application of the rules to the program gives:

```
   [[ LetRec([Bind("x"1,IntLit(42)),Bind("y"2,VarRef("x"3))],VarRef("y"4)) ^ (s) : ty ]]
=> new s_let, s_let ---> s,
   Map1[[ [Bind("x"1,IntLit(42)),Bind("y"2,VarRef("x"3))] ^ (s_let) : ty ]],
```

```
   [[ VarRef("y"4) ^ (s_let) : ty ]]
=> new s_let, s_let ---> s,
   [[ Bind("x"1,IntLit(42)) ^ (s_let) ]],
   [[ Bind("y"2,VarRef("x"3)) ^ (s_let) ]],
   [[ VarRef("y"4) ^ (s_let) : ty ]]
=> new s_let, s_let ---> s,
   Var{"x"1} <- s_let, Var{"x"1} : ty1, [[ IntLit(42) ^ (s_let) : ty1 ]],
   [[ Bind("y"2,VarRef("x"3)) ^ (s_let) ]],
   [[ VarRef("y"4) ^ (s_let) : ty ]]
=> new s_let, s_let ---> s,
   Var{"x"1} <- s_let, Var{"x"1} : ty1, ty1 == INT(),
   [[ Bind("y"2,VarRef("x"3)) ^ (s_let) ]],
   [[ VarRef("y"4) ^ (s_let) : ty ]]
=> new s_let, s_let ---> s,
   Var{"x"1} <- s_let, Var{"x"1} : ty1, ty1 == INT(),
   Var{"y"2} <- s_let, Var{"y"2} : ty2, [[ VarRef("x"3) ^ (s_let) : ty2 ]].
   [[ VarRef("y"4) ^ (s_let) : ty ]]
=> new s_let, s_let ---> s,
   Var{"x"1} <- s_let, Var{"x"1} : ty1, ty1 == INT(),
   Var{"y"2} <- s_let, Var{"y"2} : ty2, Var{"x"3} -> s_let, Var{"x"3} |-> d1, d1 : ty2,
   [[ VarRef("y"4) ^ (s_let) : ty ]]
=> new s_let, s_let ---> s,
   Var{"x"1} <- s_let, Var{"x"1} : ty1, ty1 == INT(),
   Var{"y"2} <- s_let, Var{"y"2} : ty2, Var{"x"3} -> s_let, Var{"x"3} |-> d1, d1 : ty2,
   Var{"y"4} -> s_let, Var{"y"4} |-> d2, d2 : ty
```

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

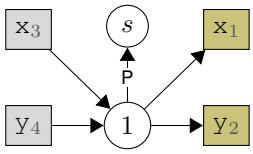
Solution: A solution must consists of all the components of the constraint semantics: a scope graph G, a typing function T, and a substitution s. The substitution s is given by:

$$
\begin{aligned}
s_{\text{let}} &\mapsto (1) \\
ty &\mapsto \texttt{INT()} \\
ty1 &\mapsto \texttt{INT()} \\
ty2 &\mapsto \texttt{INT()} \\
d1 &\mapsto \texttt{Var"x"}_1 \\
d2 &\mapsto \texttt{Var"y"}_2
\end{aligned}
$$

The declarations to types function T is given by:

$$
\begin{aligned}
&\texttt{Var\{"x"}_1\texttt{\}} : \texttt{INT()} \\
&\texttt{Var\{"y"}_2\texttt{\}} : \texttt{INT()}
\end{aligned}
$$

The scope graph G is given by:



(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.

Solution: Only the rules for let and bind have to be changed. The rules are changed in such a way that the expressions in the binds are not checked in the let scope, but in the surrounding lexical scope. These are the rules for parallel lets:

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

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