6170 Fall 2020
P7 Staff Solutions
Some student solutions included

1a. Each user has exactly one user name

- all u: User | #u.name = 1
- all u: User | one u.name
- ~name.name in iden

1b. User names are unique

- no disj u1, u2: User | u1.name = u2.name
- all n: Name | lone name.n
- all n: Name | #name.n <= 1
- all n : Name | lone n.~name
- all disj u, u': User | u.name = u'.name implies u = u'
- name.~name in iden

2a. The manager of a department is an employee of the department.

- all d: Department | d.manager in d.employees
- manager in employees

2b. An employee's boss (if one exists) manages the department the employee belongs to

- all e: Employee | e.boss in (employees.e).manager
- all e: Employee | some e.boss implies e in e.boss.~manager.employees
- (employees - manager.~boss) in manager

2c. If an employee has no boss, that employee is someone else's boss

- **all e: Employee | no e.boss implies some o: Employee - e | o.boss = e**
- **all e: Employee | no e.boss implies some (boss.e - e)**
- all e : Employee | (no e.boss and some e.~boss) or some e.boss
- all e: Employee - Employee.~boss | some e.~boss
- Employee in boss.Employee + Employee.boss
- all e: Employee | some (e.boss + e.~boss)
- all e: Employee | no e.boss => some e.(~boss)
  [Note: some of these are a bit obscure. We're including them to show you the variety of constraints that students invented. In practice, you will want to use the constraints that most directly the English statement, such as the ones in bold.]

3a. Nobody is their own mother or father

- all p: Person | not(p in p.mother) and not(p in p.father)
- no p: Person | p in p.(mother + father)

- all p: Person | p.mother != p and p.father != p
- mother & iden = none and father & iden = none
- no ((mother + father) & iden)

3b. Nobody is their own ancestor

- let ancestors = ^(mother + father)
  all p: Person | not(p in p.ancestors)
- let ancestors = ^(mother + father)
  no p: Person | p in p.ancestors
- no ^(mother + father) & iden
- ^(mother + father) & iden = none

3c. All people are the descendants of some person (except for that person!). (Interpret this carefully following what it likely means in English, and not the trivial interpretation that would be satisfied by everyone having a mother.)

- let descendants = ~^(mother + father)
  some p: Person | (Person - p) in p.descendants
- let ancestors = ^(mother + father)
  some p: Person | all q: Person - p | p in q.ancestors
- one p:Person |^(mother+father).p = (persons-p)

3d. Each person has exactly one mother and father, and at most two parents

- all p: Person | (#p.mother = 1 and #p.father = 1 and #p.parents =< 2)

3e. For some people, their parents are their biological mother and father

- some p: Person | p.parents = p.mother + p.father
- some p : Person | p.mother in p.parents and p.father in p.parents

3f. For some people, neither parent is a biological mother or father

- some p: Person | no p.parents & (p.mother + p.father)
- some p:Person | not (p.mother+p.father = p.parents)
- some p: Person | p.mother not in p.parents and p.father not in p.parents

4a. If you're my friend, I'm your friend

- friend = ~friend
- all disj p: Person | r in p.friend implies p in r.friend

4b. A person's recommended friends are their friends' friends

- recommended = friend.friend

4c. Nobody is their own friend

- no iden & friend
- all x : Person | not (x in x.friend)
- no (iden[Person] & friend)

4d. No pair of people are more than six degrees of separation apart

- all disj p1, p2: Person | (p2 in p1.friend)
  or (p2 in p1.friend.friend)
  or (p2 in p1.friend.friend.friend)
  or (p2 in p1.friend.friend.friend.friend)
  or (p2 in p1.friend.friend.friend.friend.friend)
  or (p2 in p1.friend.friend.friend.friend.friend.friend)
- all disj p1, p2: Person | p1 in p2.(friend + friend.friend + ...)
- Person -> Person in iden + friend + friend.friend + …

4e. Without using any quantifiers: for each person, their friends' friends are also their friends

- friend.friend in friend

4f. Without using any quantifiers: everyone is a friend of Zuckerberg, directly or indirectly

- Person in Zuckerberg.^friend
- Zuckerberg.^friends=Persons

5a. asYummy is a preorder

- Preorder means reflexive and transitive
- Reflexive means every food item is at least as yummy as itself.
- In logic: iden[Food] in asYummy
- Transitive means that if food f1 is asYummy as food f2 and food f2 is asYummy as food f3, then food f1 is asYummy as food f3
- In logic: asYummy.asYummy in asYummy
- asYummy being a preorder should hold

5b. asYummy is a partial order

- Partial order means reflexive, transitive, and antisymmetric.
- The argument for reflexive and transitive, the argument is the same as for 5a.
- Antisymmetric means that food f1 is asYummy as food f2 then food f2 must be not asYummy as food f1
- In logic: all disj f1, f2: Food | f2 in f1.asYummy implies f1 not in f2.asYummy
  all f,g| g in f.asYummy and f in g.asYummy => f=g

~asYummy & asYummy in iden
no ((asYummy & ~asYummy) - iden)

- asYummy being a partial order should NOT hold. Foods f1 and f2 could be just asYummy as each other

5c. asYummy is a total order

- Total order means antisymmetric, transitive, and a connex relation.
- The argument for antisymmetric and transitive is the same as for 5b.
- Connex relation means that for every pair of foods f1 and f2, either f1 is asYummy as f2 or f2 is asYummy as f1
- In logic: Food->Food in (asYummy + ~asYummy)
  all f,g| (g in f.asYummy or f in g.asYummy))
- asYummy being a total order should NOT hold. 5b highlights how antisymmetry would not make sense.

6a. No node is directly connected to itself

- no n: Node | n in n.connects
- no iden & connects
- all n: Nodes | n not in n.connects

6b. The nodes of the network form a single ring, in which each node is connected to a successor, and so on, back to itself

each node has a single direct successor and also the chain of successors of said node includes the node itself

let successors = connects + connects.connects + ... = ^connects

- all n: Node | some s: Node - n | (n.connects = s) and (n in s.successors)
- all n: Node | n in n.successors

6c. The nodes of the network form one or more rings, in which each node is connected to a successor, and so on, back to itself

every node has some immediate successor and is connected back to itself

- all n: Node | some n.connects and n in n.^connects

6d. There is a network partition in which there are some nodes that cannot communicate with each other

for all pairs of distinct nodes, some pairs won't be connected to each other

let reaches = connects + connects.connects + ... = ^connects

- some disj n1, n2: Node | not(n2 in n1.reaches) and not(n1 in n2.reaches)

7a. The id column is a key (that is, it uniquely identifies a row)

- all r1, r2: Row | r1.id = r2.id implies r1 = r2
- all i:id | lone row.i
- All disj r1, r2: Row | r1.id != r2.id

7b. The email and name columns are a key (ie, no two rows have the same email and name)

- all r1, r2: Row | (r1.email = r2.email and r1.name = r2.name) implies r1 = r2