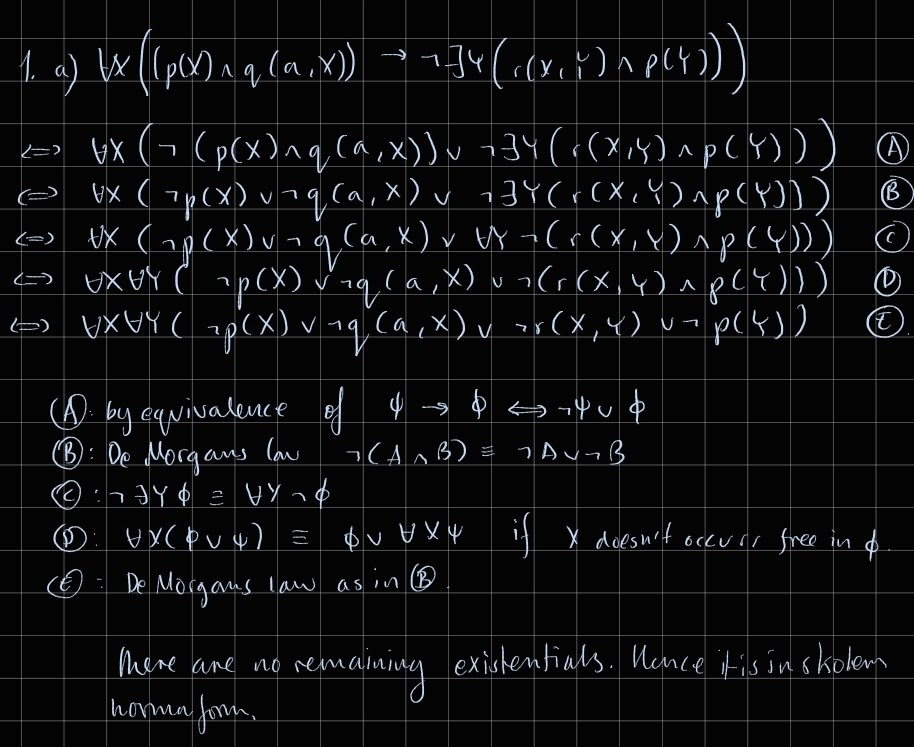
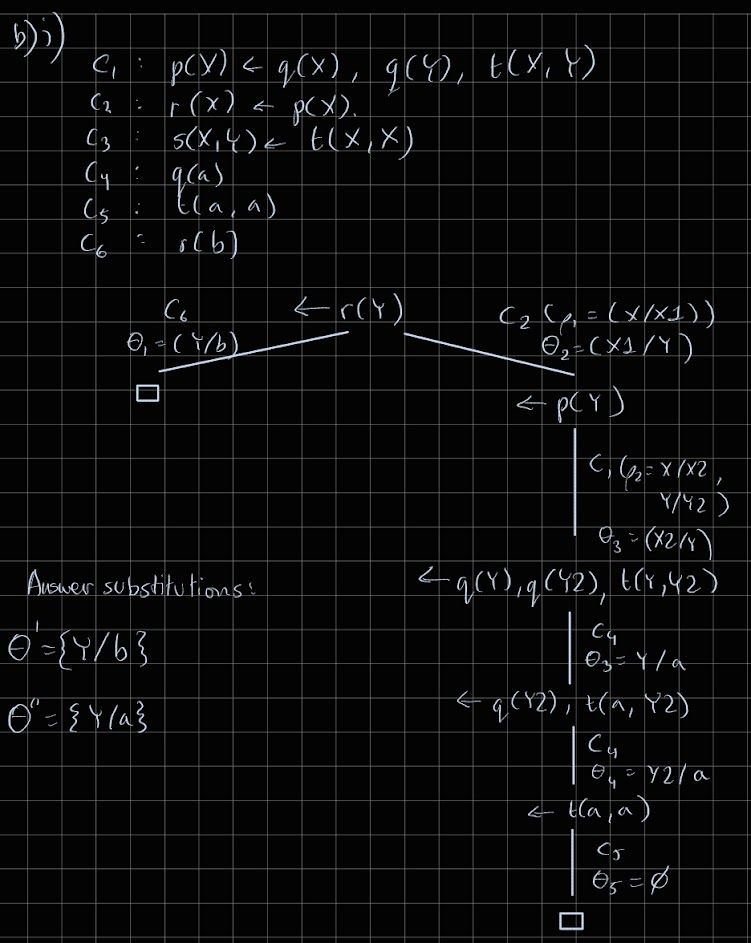
**1.a)** Optional to note that we can remove the universal quantifiers at the end



**1.b.i)**



**1.b.ii)**

**A:** no: p(a) is in the model M1 but r(a) is not, and the program requires r(a) <- p(a).

**B:** yes: repeated applications of the TP operator will produce M2. Applying TP to M2 produces M2, therefore M2 is a model and M2 is minimal.

**C:** yes: M2 is a subset of M3, and no atoms appear in M3 which break the rules of the program therefore M3 is a model. However, M3 contains q(b) which does not appear in M2, therefore M3 is not a minimal model. Also r(b) appears twice for some reason.

**1.c.i)**

**Ground atoms:**

light(1), light(2), light(3), alarmOn, doorslocked,

holds(lightOff, 1), holds(lightOff, 2), holds(lightOff, 3),

holds(lightOn, 1), holds(lightOn, 2), holds(lightOn, 3)

**P:**

holds(lightOff, X) ← light(X), not holds(lightOn, X).

holds(lightOn, X) ← light(X), not holds(lightOff, X).

{alarmOn, doorslocked} ← light(X), not holds(lightOn, X).

← not alarmOn, not doorslocked.

**Ground(P):**

holds(lightOff, 1) ← light(1), not holds(lightOn, 1).

holds(lightOff, 2) ← light(2), not holds(lightOn, 2).

holds(lightOff, 3) ← light(3), not holds(lightOn, 3).

holds(lightOn, 1) ← light(1), not holds(lightOff, 1).

holds(lightOn, 2) ← light(2), not holds(lightOff, 2).

holds(lightOn, 3) ← light(3), not holds(lightOff, 3).

{alarmOn, doorslocked} ← light(1), not holds(lightOn, 1).

{alarmOn, doorslocked} ← light(2), not holds(lightOn, 2).

{alarmOn, doorslocked} ← light(3), not holds(lightOn, 3).

← not alarmOn, not doorslocked.

**1.c.ii)**

M = {holds(lightOff, 1), holds(lightOff, 3), holds(lightOn, 2), light(1), light(2), light(3), alarmOn}

R = relevant ground instantiation of P

PM = {head(c) <- body+ (c). | c ∈ R, (body- (c) ∩ M) = ∅}

∪ {<- body+ (c) | c ∈ R, c ∈ constraint(R), (body- (c) ∩ M) = ∅}

∪ {<- body+ (c) | c ∈ R, c ∈ choice(R), !(M |= head(c)), (body- (c) ∩ M) = ∅}

∪ {ai <- body+ (c) | c ∈ R, c ∈ choice(R), M |= ai, ai ∈ (head(c) ∩ M), (body- (c) ∩ M) = ∅}

Let’s look through each item in ground(P) to find the reduct. We can think of this process as:

Set<Atom> model = M; Set<Clause> reduct = new Set<>();

for (Clause c : ground(P)) {

Set<Atom> negatives = body-(c);

if (!model.containsAny(negatives)) {

if (constraint(c))

reduct.add(new Clause(head=EMPTY, body = body+(c)));

else if ( choice(c) && !model.containsAny(head(c)) )

reduct.add(new Clause(head=EMPTY, body = body+(c)));

else if ( choice(c) && model.containsAny(head(c)) ) {

Set<Atom> heads = Set.intersection(head(c), model);

for (Atom a : heads)

reduct.add(new Clause(head=a, body = body+(c)));

}

else

reduct.add(new Clause(head=head(c), body = body+(c)));

}

}

where

Set<Atom> body+(Clause c) returns the set of atoms that occur positively in c’s body

Set<Atom> body-(Clause c) returns the set of atoms that occur negatively in c’s body

Set<Atom> head(Clause c) returns the set of atoms that occur in the head of c

boolean constraint(Clause c) returns true iff c is a constraint, false otherwise.

boolean choice(Clause c) returns true iff c is a choice rule, false otherwise.

Take c = “holds(lightOff, 1) ← light(1), not holds(lightOn, 1).” c is not a constraint or choice rule.

head(c) would return {holds(lightOff, 1)},

body+(c) would return {light(1)}, and

body-(c) would return {holds(lightOn, 1)}.

Therefore negatives = body-(c) = {holds(lightOn, 1)}. As model does not contain any elements in negatives, we add the clause “holds(lightOff, 1) <- light(1).” to the reduct.

reduct = {

holds(lightOff, 1) ← light(1).

holds(lightOff, 3) ← light(3).

holds(lightOn, 2) ← light(2).

alarmOn ← light(1).

alarmOn ← light(3).

}

Applying TP to M with respect to the reduct calculated above gives:

M = {holds(lightOff, 1), holds(lightOff, 3), holds(lightOn, 2), light(1), light(2), light(3), alarmOn}

TP(M) = M = {holds(lightOff, 1), holds(lightOff, 3), holds(lightOn, 2), light(1), light(2), light(3), alarmOn}

Therefore M is a stable model.

**1.c.iii)**

A model M is minimal iff there are no strict subsets of M that are also models. If a model is not minimal, then there is at least one strict subset of that model that is also a model.

**2.a)**

C1: ¬a v ¬b v ¬c

C2: ¬a v c v d v e

C3: d v ¬e v ¬f

**2.b.i)**

Level 0

BCP : no unit resolution possible

Level 1

Decide : assign a -> true :: ‘a’ is unassigned appears most in the not-yet-satisfied clauses

BCP : no unit resolution possible

Level 2

Decide : assign b -> true :: ‘b’ ties by DLIS assignment with !b, c, !c but appears first alphabetically and is positive therefore choose b

BCP : no unit resolution possible

Level 3

Decide : assign c -> true :: ‘c’ ties with !c by DLIS again but c is positive therefore assign c

BCP : assign d -> true as C2 is unit clause ‘d’

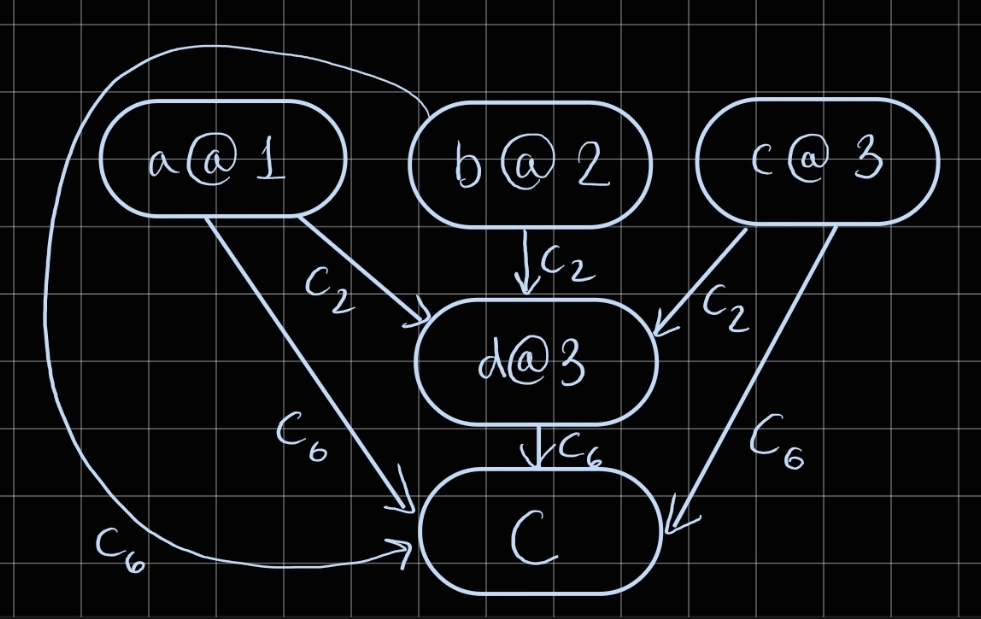
Conflict! - C6 is now UNSAT under this assignment

Alternatively:

BCP : assign d -> false as C6 is unit clause ‘!d’

Conflict! - C2 is now UNSAT under this assignment

**2.b.ii)**



**2.b.iii)**

The conflict clause is selected by making a separating cut after the first UIP. The first UIP is (c@3) and making a cut gives conflict clause ¬a v ¬b v ¬c

**2.b.iv)**

The level is the second highest decision level of any literal appearing in the conflict clause. Hence backtrack to level 2

**2.c)**

If F is valid, then F is satisfiable. If F and F’ are equisatisfiable, then (F is satisfiable) implies (F’ is satisfiable). Therefore, if F is valid, then F’ is satisfiable. However, the same cannot be said for the validity of F’: equisatisfiability makes no guarantee that a particular satisfying arrangement for F will satisfy F’, only that if at least one satisfying arrangement exists for F then at least one will exist for F’ as well. Therefore we cannot guarantee that F’ is valid simply because F is valid.

**2.d)**

F’ would have the form p ^ X where p is the representation of a subformula and X is the CNF transformed subformula. Therefore there will always be an assignment which can make p false (namely p -> false), and therefore can make F’ false. Thus the statement is vacuously true, and since no valid F’ formulae exist, this is not a useful method for finding if F is valid.

F’