Mandatory music: <https://www.youtube.com/watch?v=O87lzhoexyA>

Section A:

1-

1. |?-X=Y,Y=Z,Z=6.   
   X = 6  
   Y = 6  
   Comment: we unified X with Y, then we Unified Y with Z, then we unified Z with 6.

|?-X == Y.

No.  
Comment: we are checking if X unifies with Y, but we are asking for something (X) then we are asking for something else (Y)

|?-X=Y,X==Y.

Yes.  
 Comment: here we are unifying X with Y then checking if X and Y unifies.

|?-Xis3\*4.

X = 12  
 Comment : the is operators calls the arithmetic on the RIGHT side of it and solve it then we unify left side with right side

|?-X=3\*4.   
 X = 3 \* 4.  
 Comment: all we did is unify the right side with X

|?-3\*4 is 12.  
 No.  
 Comment: is evaluate right side then unify with left, so unifying 3 \* 4 with 12 will yield no.

* 1. So the steps are: Explanation  
     Call p(3, [[[a, [3], b]]]). // checking first rule  
     Call p(3, [[a, [3], b]]). // check head second rule  
     Call p(3, [a, [3], b]). // check head second rule   
     Call p(3, a). // check first rule  
     Fail p(3, a).  
     Try Again Call p(3, [a, [3], b]). // check head second rule but since this failed we check next rule   
     Call p(3, [[3], b]). // check head second rule   
     Call p(3, [3]). // check second rule  
     Call p(3, 3). // check second rule  
     Yes. now exit all previous calls, return true (or yes)
  2. X = [[[a, [3]]]].  
     If you force other answers you will get:  
     X = [[a, [3]]]; X = [a, [3]]; X = a; X = [[3]]; X = [3]; X = 3; X = []; X= [], X = [], X= [], no. // the 3 [] as pointed out by Gergely were caused by having empty list followed by another empty list, this is rather weird behaviour…
  3. Because that is a singleton variable and to suppress prolog warning about singleton variables we prefix them with \_. The reason prolog gives warning about singleton variable is because they are clauses that appears only once and most cases you want to use the clauses you have introduced.

1. 1. It will assert 3, 2, 1 in the prolog database.   
      If you call it with r(3, X). you will get 3, forcing other answers will produce 2, 1  
      Steps taking out by program:  
      Call r(3, X).   
      assert(p(3)) // add p(3) to database  
      Call r(2, X).  
      assert(p(2)) // add p(2) to db  
      Call r(1, X).  
      assert(p(1))  
      Call r(0, X)  
      Which in turn will call p(X) and output the stuff stored in database
   2. Will produce X = 3; X = 2; X = 1; and then it will ~~stop~~ loop forever recursively
2. First look at the program, it is looking to see if the element on the left list exist on the right, and if they do check if they contain the same value
   1. ?- q([a=9, b=10], [a=9, c=11, b=10]).  
      Yes. since a =9 and b = 10 in both lists.

| ?- q([a=9, b=10], [a=9, c=11, b=10, b=11]).  
 Yes, again a = 9, b = 10 in both lists. (b = 11 hasn’t been read by the program yet).   
 | ?- q([a=9, b=10], [a=9, c=11, b=11, b=10]).   
 No. because it compares b = 10 to b = 11, and because there is cut before this unification it fails and terminates.   
 | ?- q([a=X, b=X], [a=9, c=11, b=X]).  
 X = 9. Because a = 9 on the right side, then our left side must be a = 9, since a = X and b =X then all X are 9 as well.

| ?- q([a=9, b=10], [a=Y, c=11, b=Y]).  
 No. because we are saying that a = 9, so right side must be a = 9 -> Y = 9, however b also = Y so right side b = 9, but left side is b = 10. So false.

| ?- q([a=9, b=10], [a=Y, c=11]).  
 Y = 9. Because left side a = 9, and right side a = Y, so Y must be 9. We don’t care that b doesn’t exist on the right side.

* 1. In case the program found a contradictory value on the right side, it will try and search for another value that brings it about to true.   
     The statement | ?- q([a=9, b=10], [a=9, c=11, b=11, b=10]). Would then become yes, because if b = 11 fails to unify with b = 10, then we will simply check next element which is b = 10 from right side and will unify nicely with b = 10 from left side.

Section B

2.

1. Context-free grammar tries to work out how to match rules of the right hand side of your rules, without taking into account the surrounding items.   
   Feature-based grammar allow us to attach more information to our grammar such as tenses, position, form, agreement, etc… it allow us to model complex sentences in more powerful way that represent the language accurately   
     
   Example: She love he. He isn’t accounting for its position in the sentence that it came as an object and therefore should be ‘him’   
     
   Author\* note: Sami for example writes in a context-free grammar, most people over the age of 13 writes in a feature-based grammar.
2. With categorial descriptions we can define what a phrase need in its argument rather than a separate rule, this cut down the number of the rewrite rules BY A LOT. so instead of having 3-4 rule for each type of phrases, we have one that requires the arguments to be configured in a certain way.   
     
   We will need to write one important rule that it says if I need to make a phrase we need to fulfil its argument (one case for when it comes after one case for when it comes before)  
   X0 ⇒ [Y, X1] :-  
    cat@X0 -- cat@X1,   
    args@X1 -- [Y | args@X0],  
    dir@Y -- before.  
     
   X0 ⇒ [X1, Y] :-  
    cat@X0 -- cat@X1,   
    args@X1 -- [Y | args@X0],  
    dir@Y -- after.  
     
   Now we can define the S to be   
   S ⇒ [V | ARGS] :-  
    Cat@S -- s,  
    cat@V -- vp,  
    args@V -- ARGS.  
   VP ⇒ [V | ARGS] :-  
    cat@VP -- vp,  
    cat@V -- verb,  
    args@V -- ARGS.  
   Then need to add rules to deal with iverb and tverbs  
   tverb(V) ;-  
    cat@V -- verb,  
    args@X -- [SUBJ, OBJ],  
    cat@SUBJ -- np,  
    dir@SUBJ -- before,  
    cat@OBJ -- np,  
    dir@OBJ -- after.  
   And a rule for iverbs   
   iverb(V) :-   
    cat@V -- verb,  
    args@X -- [SUBJ],  
    cat@SUBJ -- np,  
    dir@SUBJ -- before.  
   For verb connecting a verb with a sentence >  
   sverb(V) :-

cat@V -- verb,

args@V -- [S, NP],

cat@NP -- np,

cat@S -- s,

dir@NP -- before,

dir@S -- after.

1. Fundamental rule of chart parsing: if you’ve got an X that needs [H | T] to make a complete X, you can combine them to make an X that needs T to complete itself.  
   This is used to save the trace route I have already tried so I don’t have to retrace back as I would do with bottom up.

Note: that not how I would do it in an exam however for completeness sake here is the IDEAL answer.

Adding lexical edge: edge(0, 1, np, he, **[]**, **0**)

Adding edge(0,1,s,he,[vp],1)

because start of s==>[np,vp] has been matched by edge(0,1,np,he,[],0)

Adding lexical edge: edge(1, 2, tverb, knows, [], 2)

Adding edge(1,2,vp,knows,[np],3)

because start of vp==>[tverb,np] has been matched by edge(1,2,tverb,knows,[],2)

Adding lexical edge: edge(1, 2, sverb, knows, [], 4)

Adding edge(1,2,vp,knows,[s],5)

because start of vp==>[sverb,s] has been matched by edge(1,2,sverb,knows,[],4)

Adding lexical edge: edge(2, 3, np, you, [], 6)

Adding edge(2,3,s,you,[vp],7)

because start of s==>[np,vp] has been matched by edge(2,3,np,you,[],6)

Adding edge(1,3,vp,[knows,you],[],8):

edge(1,2,vp,knows,[np],3) wanted edge(2,3,np,you,[],6)

Adding edge(0,3,s,[he,[knows,you]],[],9):

edge(0,1,s,he,[vp],1) wanted edge(1,3,vp,[knows,you],[],8)

Adding lexical edge: edge(3, 4, tverb, saw, [], 10)

Adding edge(3,4,vp,saw,[np],11)

because start of vp==>[tverb,np] has been matched by edge(3,4,tverb,saw,[],10)

Adding lexical edge: edge(4, 5, np, it, [], 12)

Adding edge(4,5,s,it,[vp],13)

because start of s==>[np,vp] has been matched by edge(4,5,np,it,[],12)

Adding edge(3,5,vp,[saw,it],[],14):

edge(3,4,vp,saw,[np],11) wanted edge(4,5,np,it,[],12)

Adding edge(2,5,s,[you,[saw,it]],[],15):

edge(2,3,s,you,[vp],7) wanted edge(3,5,vp,[saw,it],[],14)

Adding edge(1,5,vp,[knows,[you,[saw,it]]],[],16):

edge(1,2,vp,knows,[s],5) wanted edge(2,5,s,[you,[saw,it]],[],15)

Adding edge(0,5,s,[he,[knows,[you,[saw,it]]]],[],17):

edge(0,1,s,he,[vp],1) wanted edge(1,5,vp,[knows,[you,[saw,it]]],[],16)

1. because of fine tuning the grammar too much that can't deal with ambiguities or become prone to them or something dunno

Question 3.

1. Allan has grey hair. // attribute, a second order entity   
   Saw her yesterday. // time   
   Most students will pass the year. // defaults  
   I believe it is raining. // believe   
   I know it is raining. // knowledge  
   All of those are hard to model in first order logic and requires hacks for them to work.
2. Particular set of words, configured in a particular way, always makes the same contribution, though what you do with that contribution in a given context is up to you.   
     
   A. structural ambiguity: depends on how I structure this sentence it could mean many things. “I don’t know what the mode of combination is”  
   B. lexical ambiguity: jam here isn’t used as jam that we eat. “I don’t know what the parts are”   
   C. Scope ambiguity: I want to buy a bike, is there a specific bike in my mind? Will any bike do? Does the bike have to be from certain place, etc… “I know the words, I know the structure I still don’t know the meaning”.
3. We look at our words: every is det, we replace it with det and put the meaning, then we have man, replace it with noun and we have its meaning, we have a rule that combines the meaning of DET by applying N to it, this will give us np. So first we construct the NP   
     
   **NP)**  
   lambda(P,   
    lambda(Q, forall(X, (P:X => Q:X))))  
   :lambda(U, man(U))  
     
   lambda(Q, forall(X, (lambda(U, man(U)):X => Q:X))))  
     
   lambda(Q, forall(X, (man(X) => Q:X))))  
     
   Now ‘will’ is an aux, we have its meaning, die is a verb which has rule that makes it a vp, we apply the meaning of vp to aux aux and verb to form a vp ,  
     
   **VP)**  
   lambda(A,   
    lambda(B, future(A:B)))  
   :lambda(B, B:(lambda(X, exists(Z, die(Z) & patient(Z, X)))))  
     
   lambda(B,   
    future(lambda(B, B:(lambda(X, exists(Z, die(Z) & patient(Z, X))))):B))  
     
   lambda(B,   
    future(B:(lambda(X, exists(Z, die(Z) & patient(Z, X))))))  
   Then to make sentence we apply the NP to VP   
     
   **VP:NP**  
   lambda(B,   
    future(B:(lambda(X, exists(Z, die(Z) & patient(Z, X))))))  
   :lambda(Q, forall(X, (man(X) => Q:X)))  
   future(lambda(Q, forall(X, (man(X) => Q:X))):(lambda(X, exists(Z, die(Z) & patient(Z, X)))))  
   future(forall(X, (man(X) => lambda(X, exists(Z, die(Z) & patient(Z, X))):X)))  
   future(forall(X, man(X) => exists(Z, die(Z) & patient(Z, X))))  
     
     
   Thanks Oreste for lambda simplification :)
4. In case of a and c, I am an experiencer; my senses are being stimulated without much interference from me.  
   In case of b and d, I am an agent; I am actively trying to engage in watching/listening, so I am an agent.

Question 4:



**horn(P) :- P.**Here when we get a predicate we try to prove it using the predicate itself.

**horn(P or Q) :- horn(P); horn(Q).**

If we have a 2 predicate with disjunction between them, proving either of them to be true will result in the disjunction being true.

**horn(P) :- nonvar(P), Q==>P, horn(Q).**

If we have a predicate implying something else we first prove that if Q holds then P will also hold, and then we also prove that left side of implication is also horn (we can’t for example have Nice or Naughty => Kid)

**prove(P) :- horn(P).**   
Tries to prove given proposition using horn logic.

**prove(P) :- (R or S), cprove(R ==> P), cprove(S ==> P).**

Failing to use horn logic to prove a proposition we try to see if it has disjunction in the antecedent of the rule, if it does then we try to see if we can bring it to true, if it can be proven, then we need to show that if R holds then P will hold, and if S holds P will also hold.

**cprove(P ==> Q) :- nonvar(Q), assert(P), (prove(Q) -> retract(P); (retract(P), fail)).**

Here assert P to be true, then we try to prove Q, per the rule that says if we have True => something, that something has to be true else it is a false statement. The assert and retract lines are there to first make what’s on our left side of implication true, and hence we can prove Q, and retract is to make sure after we finish with our prove we return our world to its normal state where P as it was before.   
  
  
  
**To derive r(2) from {p(X) or q(X) ==> r(X), p(2), q(2)} we call prove(p(X) or q(X) ==> r(X) & p(2) & q(2))**   
Here is the call stack:  
prove(p(X) or q(X) ==> r(X) & p(2) & q(2))   
horn((p(X) or q(X)) ==> r(X) & p(2) & q(2)))  
p(X) or q(X) ==> r(X) & p(2) & q(2)  
cprove((p(X) or q(X) ) ⇒ r(X) & p(2) & q(2))   
// here we are assuming that p(X) or q(X) holds)  
prove(r(X) & p(2) & q(2))  
horn( r(X) & p(2) & q(2))   
r(X) & p(2) & q(2) // this is horn in pure form from our assertion we know either p(X) or q(X) hold  
// since we know p(X) or q(X) hold, and we are saying p(2) & q(2) that mean X = 2  
Since X = 2, that mean r(X) = r(2)  
  
**To derive r(2) from {p(2) or q(2) p(X) ==> r(X), q(X) ==> r(X)} we call prove((p(2) or q(2)) & p(X) => (r(X) & q(X)) ⇒ r(X))**Here is the call stack:  
prove((p(2) or q(2)) & p(X) => (r(X) & q(X)) ⇒ r(X))  
horn((p(2) or q(2)) & p(X) => (r(X) & q(X)) ⇒ r(X))  
((p(2) or q(2)) & p(X) => (r(X) & q(X)) ⇒ r(X))   
cprove(((p(2) or q(2)) & p(X)) **=>** ((r(X) & q(X)) ⇒ r(X)))  
// asserting that (p(2) or q(2)) & p(X)) holds we will prove right side  
prove( ((r(X) & q(X)) ⇒ r(X)))  
horn( ((r(X) & q(X)) ⇒ r(X)))  
((r(X) & q(X)) ⇒ r(X))   
cprove( ((r(X) & q(X)) **⇒** r(X))  
// asserting that r(X) and q(X) hold we prove right side  
prove(r(X))  
horn(r(X))  
r(X)   
NOW Prolog will unify what what we know about X so far:  
We know  
(p(2) or q(2)) & p(X) is true. Having OR means one of p(2) call it P0 or q(2) call it Q0 is true, hold on that thought.  
Next we also know r(X), q(X) are also true. Now from before q(2) can be true, hence X is 2 here, and every X will unify with 2, making r(X), p(X) and q(X) unify with r(2) and q(2).   
We also need to show in case q(2) didn’t hold, that mean p(2) held, per first & we have, that mean X will unify with 2, again all X will unify with 2 making a complete sentence.

ii) cprove(p(X) ⇒ q(X)), here we will try to prove left side then prove right side, but to prove right side we need to prove left side which will require proving right side.   
  
Label can be used to store all the things I want to watch out for, if I see same label being proven few times with exact same call stack I am probably stuck in a loop, so I terminate.   
No you can’t just catch only genuine loops, there will be cases where the program terminate too soon stopping a complex predicate of being evaluated, so it misclassify a genuine program as a semi decidable one.

b)   
STRIPS, is a simple version of planning, where I have 3 main component:  
1- The world, where my world is a finite set of facts only, and my world follows the closed world assumption that state (only I can influence my world, anything that I don’t know is false and anything I know is true)   
2- Goal states: list of facts that I want to bring into my world.   
3- Actions: how will they change my world? How and when can I do them?

To grasp a block:  
My world is: there is a block, the block is on the table, there is nothing on top of the block, there is a hand, the hand is empty, the hand is over the table.   
Goal: hand isn’t empty, block is over the table, there is nothing on the table,   
Action: Lower hand, close hand, raise hand  
  
Backward chaining planning:  
I have set of goals:  
First - choose an unsatisfied goal: By choosing member from the goal list that isn’t true in my world list.   
Then - I find an action to make the goal true  
Last check if my preconditions to bring the action to true are true, if they are perform the action. If they’re not then they will be added to my goal list, and repeat.

ii)

The problem of subgoal interactions is when the goal you are trying to reach will undermine another goal you have already performed.   
E.g. moving a block from A to B, if you have already moved another block to B and now you’re trying to move the stuff away from B despite you are the one who moved the block you are undermining your own actions.  
  
Protected goals are a list of goals that we have performed and want to keep in our world, when a new action is being performed we check that it won’t undermine any of the protected goals if it will, we will stop the action and find another way to do it, otherwise we will perform it then add the goal to set of protected goals.  
  
iii) ramification problem, aka frame problem, is that most planning algorithms assume a closed world assumption; that nothing change unless we change it.   
Inference engine that observe our world and update the information in it all the time will allow the backwards chaining planner to still function correctly.