שאלה 1:

a. Similarities and differences:

	Predicate Symbols	**Individual Constant Symbols**
Functor Symbols	Both denote a relationship, but functor symbols can be nested, where predicate symbols cannot.	Individual constant symbols denote a specific entity, where functor symbols denote relations.
Individual Constant Symbols	Individual constant symbols denote a specific entity, where predicate symbols denote relations. Some are built in like `true`.	

b. RLP queries are not always finite. Consider this example:

```
parent(x,y).
parent(x,y).
sibling(X,Y):-parent(Z,X),parent(Z,Y).

familyMembers(X):-parent(Z,X).
familyMembers(X):-parent(Z,X), familyMember(Z).
familyMembers(X):-sibling(Z,X).
familyMembers(X):-sibling(Z,X), familyMember(Z).
```

Notice that familyMembers(X) will find the family members of X, but will also look into X's family members to find their family members, which in turn include X again, and the query will run forever, even for a small family - leading to an infinite number of answers.

c. We'll look at RLP as a context free language, and the letters in the language will be nodes in the RLP proof tree (meaning, each time a query is performed, a new node is created as a son to the node where the query was performed). We'll recall the pumping lemma for context free language from Automata course. It follows from the pumping lemma that exists $p \geq 1$ so that p is an upper limit to the size of a subsection vwx of a word uvwxy in a CFG so that uv^nwx^ny is also in the CFG - so in proof-tree terms, there is an infinite loop where some node will perform a query x that produces a node that will perform query x again and again.
Therefore we can predict that upper limit (or limit the upper limit by a constant) and we could deduce that if a node in a proof tree is of depth p+1 or more, we are in an infinite loop and can be cut off.

d. It couldn't, since functors included in LP could have a variable "size" meaning more queries could be performed in the subcontext of the functor, and those could have a variable upper limit of queries because of nested functors.

e. **Finite with success:**

```
parent(a,b).
?-parent(X,b)
X=a;
fail.
```

Infinite with success: `see` **b.**

Infinite with failure:

```
edge(a,b).
edge(b,a).
edge(c,d).
edge(d,c).
path(X,Y):-path(X,Z),path(Z,Y).
path(X,Y):-edge(X,Y).
?- path(a,d) ????
```

A path doesn't exists between a and d; however the path calls will be self-recursive since both sides will be stuck in loops. So the query is infinite and will never succeed.

f. No, because if X is found in Xs, Prolog will backtrack again to member_check, which will fail since X was found before. Then it will change to the other option for not_member, which is true for all variables, and the query will return true for an X that is a member of Xs.

שאלה 2:

סעיף 1:

<if-statement> -> (<body>) '->' <body> ';' <body> '.'

הערה: התייחסנו למקרה כאילו ניתן להוסיף תנאים מורכבים, ולא רק נוסחאות אטומיות לתנאי ה-if.

סעיף 2:

<if-statement>:

components:

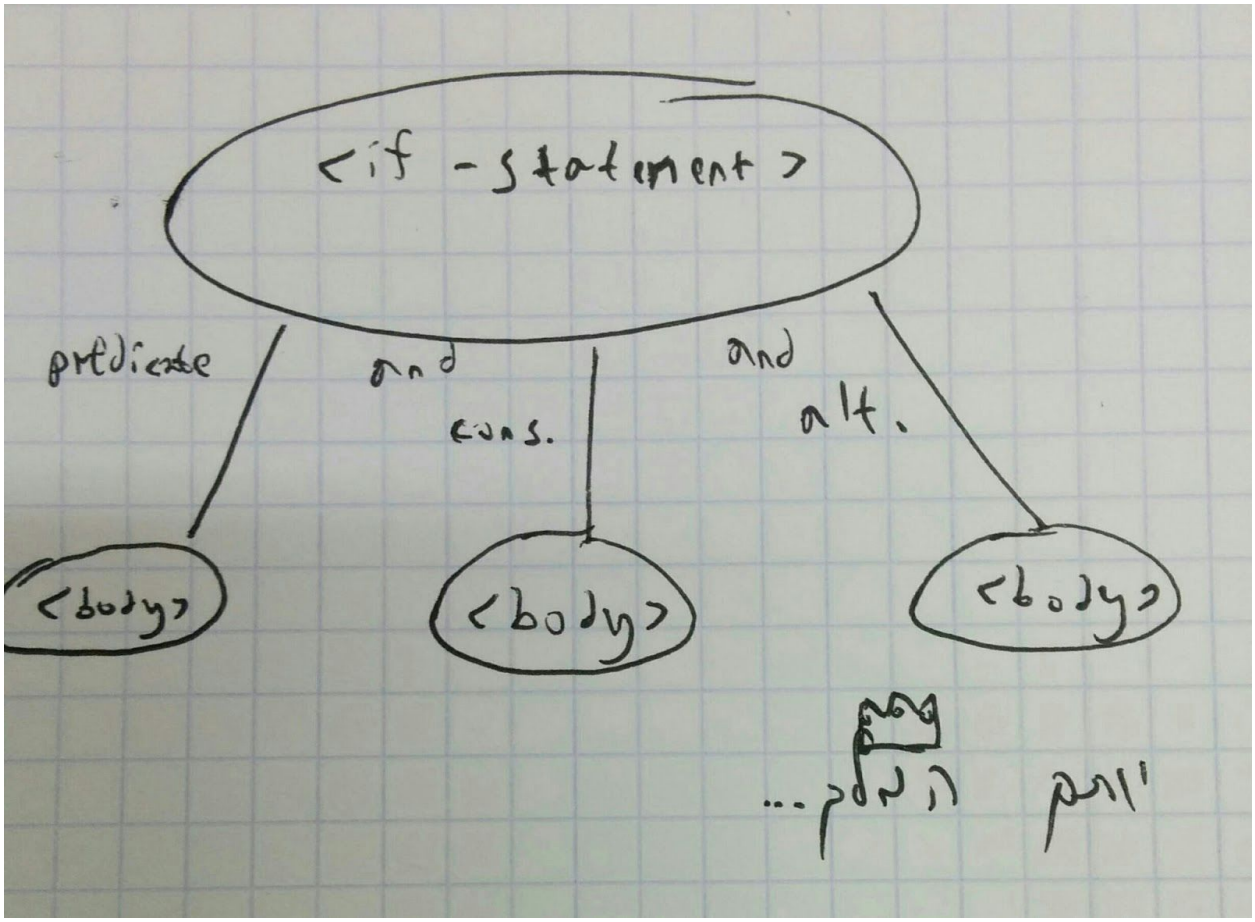
predicate: <body>

consequence: <body>

alternative: <body>

סעיף 3:

העץ:



סעיף 4:

השינויים שיש לבצע הם רק בתחביר הקונקרטי, (להחליף את ה-'->' ב-then, ולהוסיף if ו-else במקומות הנכונים), משום שהביטוי, לאחר פרסור אינו משנה את משמעותו.

שאלה 4:

סעיף ד':
כן. אנו יכולים ע"י פרוצדורה לכפות על עץ ההוכחה לעבוד בצורה של BFS, כפי שהראינו בשיעור וכך להיות בטוחים שתוחזר מילה חוקית ע"י מספר סופי של צעדים, או לדעת שמעומק מסוים (שניתן לדעת ע"י חישוב קבוע הניפוח לשפה שהגדרנו) הגענו ללולאה ולעצור.

שאלה 5:

a.

1. $unify[m(p(A,\ p(d(0),\ word3),\ X)), m(p(d(B),\ p(B,\ word3),\ C))]\ =>$

1. {}: m(p(A, p(d(0), word3), X)),
   m(p(d(B), p(B, word3), C))
2. {A=d(B)} m(p(d(B), p(d(0), word3), X)),
   m(p(d(B), p(B, word3), C))
3. {A=d(B), B=d(0)} m(p(d(d(0)), p(d(0), word3), X)),
   m(p(d(d(0)), p(d(0), word3), C))
4. {A=d(B), B=d(0), X=C} m(p(d(d(0)), p(d(0), word3), C)),
   m(p(d(d(0)), p(d(0), word3), C))

=> Success.

2. $unify[m(p(A,\ p(d(0),\ p),\ X)),\ m(p(d(B),\ p(B,\ word3),\ C))]\ =>$

1. {}: m(p(A, p(d(0), p), X)),
   m(p(d(B), p(B, word3), C))
2. {A=d(B)} m(p(d(B), p(d(0), p), X)),
   m(p(d(B), p(B, word3), C))
3. {A=d(B), B=d(0)} m(p(d(d(0)), p(d(0), p), X)),
   m(p(d(d(0)), p(d(0), word3), C))
4. {A=d(B), B=d(0), p=word3} m(p(d(d(0)), p(d(0), word3), X)),
   m(p(d(d(0)), p(d(0), word3), C))
5. {A=d(B), B=d(0), p=word3, X=C} m(p(d(d(0)), p(d(0), word3), C)),
   m(p(d(d(0)), p(d(0), word3), C))

=> Success.

3. $unify[m(p(A,\ p(d(A),\ word3),\ X)),\ m(p(d(B),\ p(B,\ word3),\ C))]\ =>$

1. {}: m(p(A, p(d(A), word3), X)),
   m(p(d(B), p(B, word3), C))
2. {A=d(B)} m(p(d(B), p(d(d(B)), word3), X)),
   m(p(d(B), p(B, word3), C))

=> Next disagreement set will be B=d(d(B)) => Failure.

4. $unify[m(p(A,\ p(d(0),\ word3),\ X)),\ m(p(d(B),\ p(B,\ word3),\ word3))]$ =>

1. {}: m(p(A, p(d(0), word3), X)),
   m(p(d(B), p(B, word3), word3))
2. {A=d(B)} m(p(d(B), p(d(0), word3), X)),
   m(p(d(B), p(B, word3), word3))
3. {A=d(B), B=d(0)} m(p(d(d(0)), p(d(0), word3), X)),
   m(p(d(d(0)), p(d(0), word3), word3))
4. {A=d(B), B=d(0), p=word3} m(p(d(d(0)), p(d(0), word3), word3)),
   m(p(d(d(0)), p(d(0), word3), word3))

=> Success.

b. 1. The cut is necessary - otherwise the substitution will continue unnecessarily, and might return a wrong answer. For example, The input: `((x),[[x,6]],A)` will return multiple answers.

שאלה 6

סעיף א':

?- u_p([1|Y], [1,1|X], [1|1|1]).

3

$X_1 = [1|Y]$

$A_1 = 1$

$Y_1 = [1|X]$

$Z_1 = [1|1]$

u_p([1|Y], [1,X], [1,1,1])

3

$X_2 = [1,Y]$

$A_2 = 1$

$Y_2 = X$

$Z_2 = [1,1]$

u_p([1,Y], X, [1,1])

1

$Y = X_3 = [1,1]$

$X = []$

u_n([1,1])

$A_4 = 1$

u_n(1).

$A = []$

u_n([]).

true!

3

$X_3 = [1,Y]$

$Y_3 = []$

$Z_3 = 1$

$A_3 = 1$

u_p([1,Y], [], [1]).

1 $Y = []$

u_p(1, [], 1).

$A = 0$

u_n(1).

1

$A = 0$

u_n([]).

true!

סעיף ב':
תשובה ראשונה:

X=[1], Y=[ ]

תשובה שניה:

X=[ ], Y=[1]

סעיף ג':
מדובר בעץ הצלחה, שכן קיבלנו לפחות השמה אחת למשתנים שנותנת true.

סעיף ד':
מדובר בעץ סופי, שכן כל ענף מסתיים בהצלחה או כישלון תוך ירידה לעומק סופי.

שאלה 7:

b. We'll note that this changes the proof tree randomly, however for both versions the same answers will be returned. So it follows that the result will be the same but in a different order. This could lead to a different outputs, as consider the normal (#1) and a specific permutation (#2) of the following code:

```
c(X) :- b(X, Y), a(Y).
a(3).
a(X) :- a(X).
b(0, 5).
b(3, 3).
?- c(X).
```

For the first version, we'll enter an infinite loop. For a specific permutation, namely ordering the goals by the number of variables, we'll get at least a first answer.