Seth Denney

1.) Consider the following grammer:

expression -> a | b | (list)
list -> list ; expression | expression

Define in Prolog a predicate parse(L) which will return true iff L is a list of tokens representing a legal expression. If semicolon gives you a problem, make it a character constant. Rewrite the grammar if the left recursion gives you a problem.

parse(['(','(','a',';','b',')',';','a',')'])
yes

```
parse(X) :- step(X,0,1,0).

ab(X) :- X == 'a'; X == 'b'.

step([X],1,_,_) :- X == ')'.
step([X],_,_,_) :- ab(X).
step([H|T],NP1,SC,AB) :- NP1 >= 0, H == '(', NP2 is NP1 + 1, length(T,L), L > NP2,
                         step(T,NP2,1,0).
step([H|T],NP1,SC,AB) :- AB == 0, ab(H),
                         step(T,NP1,0,1).
step([H|T],NP1,SC,AB) :- SC == 0, H == ';', length(T,L), L > NP1,
                         step(T,NP1,1,0).
step([H|T],NP1,SC,AB) :- NP1 > 0, H == ')', NP2 is NP1 - 1, length(T,L), L > NP2,
                         step(T,NP2,0,1).
```

2.) Enter the following program into a file and then get the interpreter to consult the file. Explain what happens and why.

```
bachelor(X) :- not(married(X)), male(X).
married(john).
male(john).
male(bill).
```

After consulting the file, pose the following queries to the interpreter.

```
bachelor(X).
```

Now reverse the order of the definition of a bachelor to read

```
bachelor(X) :- male(X), not (married(X).
```

reconsult the file, and then pose the query bachelor(X). to the interpreter.

Initially, Prolog searches for all facts in the database regarding the marriage status of people. It will only find that john IS married, which, being the only available fact on marriage, will force an evaluation of FALSE for not(married(X)). Once the order is reversed, however, Prolog searches for all facts in the database regarding the gender of people. It will find that both john and bill are male. It then checks its knowledge about marriage status to ensure that one of them is not married, and bill is returned as the only valid match.

3.) Write Prolog clauses that will enable the return of the last element of a list.

last(X,[2,3,4]).
X = 4

last(X,[X]).
last(X,[H|T]) :- last(X,T).

4.) Define predicates to obtain the third element in a list.

third(X,[2,3,4]).
X = 4

third(X,[A,B,X|T]).

5.) Write a predicate vector_reference(L,N,V) that when given a list L and an index N will unify V with the Nth value of the list. (starting with 1). Depending on your implementation, the third query might work for you.

vector_reference([2,3,4], 3, V).
V = 4
vector_reference([2, [2,3],'hello'),2,V).
V = [2,3]
?? vector_reference([2,3,4], N, 3).
?? N = 2

vector_reference([X|T],1,X).
vector_reference([H|T],N,V) :- N2 is N - 1, vector_reference(T,N2,V).

6.) Define predicates so that addup_list([2,3,4],K) will respond K=9.

addup_list(L,X) :- sum(L,X).

sum([X],X).
sum([H|T],X) :- sum(T,Y), X is H + Y.

7.) Given the two Prolog rules:
succ(X,Y) :-number(X), Y is X+1.
succ(X,Y) :-number(Y), X is Y-1.

Given successor functions we can define addition

add(x,y) = x iff y == 0
add(succ(x),y)) = succ(add(x,y))
add(x, succ(y)) = succ(add(x,y))

Convert these axioms to Prolog rules to implement a predicate add(X,Y,Z) that says that Z is the sum of X and Y. Your answer must not involve any build in arithmetic, only the above succ predicate defined above.

Your answer should also support a symmetry of addition and correctly give responses to:

```
add(X,0,3).
add(1,X,4).
add(2,3,Y).
add(X,Y,4).     ; following three should respond with some true unification for X and Y
add(1,X,Y).     ; such as X=0, Y=1 for this query
add(X,Y,Z).     ; or X=0 , Y = 0,  Z = 0 for this query.
```

```
add(0,Y,Y) :- number(Y).
add(X,0,X) :- number(X).
add(X,Y,Z) :- number(X), number(Y), succ(S1,X), succ(Y,S2), add(S1,S2,Z).
add(X,Y,Z) :- number(Y), number(Z), succ(S1,Y), succ(S2,Z), add(X,S1,S2).
add(X,Y,Z) :- number(X), number(Z), succ(S1,X), succ(S2,Z), add(S1,Y,S2).

succ(X,Y) :- number(X), Y is X + 1 ;
             number(Y), X is Y - 1 .
```