Comparisons, continued

Some predicates for grading:

```
grade(Score,Grade) :- Score >= 90, Grade = 'A'.
grade(Score,Grade) :- Score >= 80, Score < 90, Grade = 'B'.
grade(Score,Grade) :- Score >= 70, Score < 80, Grade = 'C'.
grade(Score,Grade) :- Score < 70, Grade = 'F'.

print_grade(Score) :- grade(Score,Grade),
    format('~w -> ~w\n', [Score, Grade]).
```

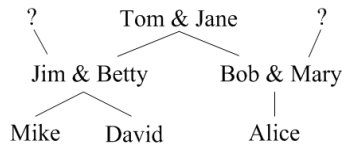
Usage:

```
?- grade(95,G).
G = 'A'
Yes

?- print_grade(80).
80 -> B
Yes

?- print_grade(55).
55 -> F
Yes
```


Another example of rules

Here is a set of facts describing parents and children:

```
male(tom).      parent(tom,betty).
male(jim).      parent(tom,bob).
male(bob).      parent(jane,betty).
male(mike).     parent(jane,bob).
male(david).    parent(jim,mike).
                parent(jim,david).
female(jane).   parent(betty,mike).
female(betty).  parent(betty,david).
female(mary).   parent(bob,alice).
female(alice).  parent(mary,alice).
```



`parent(P,C)` is read as "P is a parent of C".

Problem: Define rules for `father(F,C)` and `grandmother(GM,GC)`.

Another example, continued

```
father(F,C) :- parent(F,C), male(F).
mother(M,C) :- parent(M,C), female(M).

grandmother(GM,GC) :- parent(P,C), mother(GM,P).
```

Who is Bob's father?

For who is Tom the father?

What are all the father/child relationships?

What are all the father/daughter relationships?

What are the grandmother/grandchild relationships?

Problems: Define `sibling(A,B)`, such that "`A` is a sibling of `B`".
Using `sibling`, define `brother(B,A)` such that "`B` is `A`'s brother".

Another example, continued

```
sibling(A,B) :- father(F,A), mother(M,A),
                father(F,B), mother(M,B), A \== B.
```

Queries:

Is Mike a sibling of Alice?

What are the sibling relationships?

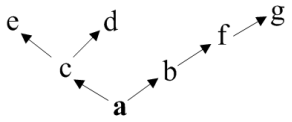
Who is somebody's brother?

Is the following an equivalent definition of `sibling`?

```
sibling2(S1,S2) :- parent(P,S1), parent(P,S2), S1 \== S2.
```

Recursive predicates

Consider an abstract set of parent/child relationships:




```
parent(a,b).  parent(c,d).
parent(a,c).  parent(b,f).
parent(c,e).  parent(f,g).
```

If a predicate contains a goal that refers to itself the predicate is said to be recursive.

```
ancestor(A,X) :- parent(A, X).
ancestor(A,X) :- parent(P, X), ancestor(A,P).
```

"A is an ancestor of X if A is the parent of X or P is the parent of X and A is an ancestor of P."

```
?- ancestor(a,f).              % Is a an ancestor of f?
Yes

?- ancestor(c,b).              % Is c an ancestor of b?
No

?- ancestor(c,Descendant).     % What are the descendants of b?
Descendent = e ;
Descendent = d ;
No
```


Recursion, continued

A recursive rule can be used to perform an iterative computation.

Here is a predicate that prints the integers from 1 through N:

```
printN(0).
printN(N) :- N > 0, M is N - 1, printN(M), writeln(N).
```

Usage:

```
?- printN(3).
1
2
3

Yes
```

Note that we're asking if `printN(3)` can be proven. The side effect of Prolog proving it is that the numbers 1, 2, and 3 are printed.

Is `printN(0).` needed?

Which is better—the above or using `between/3`?

Recursion, continued

A predicate to calculate the sum of the integers from 1 to N:

```
sumN(0,0).
sumN(N,Sum) :-
    N > 0, M is N - 1, sumN(M, Temp), Sum is Temp + N.
```

Usage:

```
?- sumN(4,X).
X = 10

Yes
```

Note that this predicate can't be used to determine N for a given sum:

```
?- sumN(N, 10).
ERROR: >/2: Arguments are not sufficiently instantiated
```

Could we write `sumN` using `between/3`?

Sidebar: A common mistake with arithmetic

Here is the correct definition for `sumN`:

```
sumN(0,0).
sumN(N,Sum) :-
    N > 0, M is N - 1, sumN(M, Temp), Sum is Temp + N.
```

Here is a COMMON MISTAKE:

```
sumN(0,0).
sumN(N,Sum) :-
    N > 0, M is N - 1, sumN(M, Sum), Sum is Sum + N.
```

Unless `N` is zero, `Sum is Sum + N` fails every time!

Remember that `is/2` unifies its left operand with the result of arithmetically evaluating it's right operand. Further remember that unification is neither assignment or comparison.

Recursion, continued

A common example of recursion is a factorial computation:

```
factorial(0,1).

factorial(N,F) :-
   N>0,
   N1 is N-1,
   factorial(N1,F1),
   F is N * F1.
```

The above example comes from
http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/2_2.html

Near the bottom the page has an excellent animation of the computation `factorial(3,X).` TRY IT!

Generating alternatives with recursion

Here is a predicate to test whether a number is odd:

```
odd(N) :- N mod 2 =:= 1.
```

Remember that `=:=` evaluates its operands.

An alternative:

```
odd(1).
odd(N) :- odd(M), N is M + 2.
```

How do the behavior of the two differ?

Generating alternatives, continued

For reference:

```
odd(1).
odd(N) :- odd(M), N is M + 2.
```

Usage:

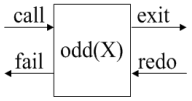
```
?- odd(5).
Yes

?- odd(X).
X = 1 ;
X = 3 ;
X = 5 ;
...
```

How does `odd(X).` work?

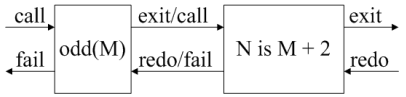
What does `odd(2).` do?

Generating alternatives, continued

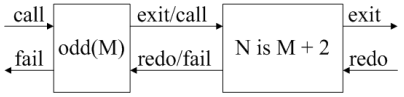
Query: `?- odd(X).`



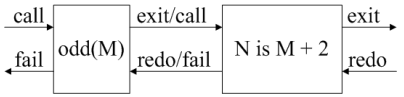

```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



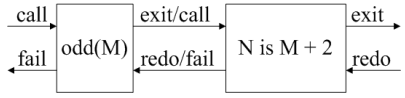

```
odd(1).
odd(N) :- odd(M), N is M + 2.
```




```
odd(1).
odd(N) :- odd(M), N is M + 2.
```




```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



Try: `?- gtrace, odd(X).` (Or `?- trace, odd(X).`)
`?- gtrace, odd(2).`

Generating alternatives, continued

For reference:

```
odd(1).
odd(N) :- odd(M), N is M + 2.
```

With generative predicates like this one a key point to understand is that if an alternative is requested, another activation of the predicate is created.

As a contrast, think about how execution differs with this set of clauses:

```
odd(1).
odd(3).
odd(N) :- odd(M), N is M + 2.
```


Lists

Basics

Built-in list-related predicates

`member/2`

`append/3`

`findall/3`

Low-level list processing

Lists

A Prolog list can be literally specified by enclosing a comma-separated series of terms in square brackets:

```
[1, 2, 3]

[just, a, test, here]

[1, [one], 1.0, [a,[b,['c this']]]]
```

Common mistake: Entering a list literal as a query is taken as a request to consult files.

```
 ?- [abc, 123].
ERROR: source_sink `abc' does not exist
```

One way to "see" a list is to `write` it; another way is to unify it with a variable:

```
?- write([1,2,3,4]).
[1, 2, 3, 4]

?- X = [a,1,b,2].
X = [a, 1, b, 2]
```

Lists, continued

Unifications can involve lists:

```
?- [1,2,3] = [X,Y,Z].
X = 1
Y = 2
Z = 3

?- [X,Y] = [1,[2,[3,4]]].
X = 1
Y = [2, [3, 4]]

?- [X,Y] = [1].
No

?- Z = [X,Y,X], X = 1, Y = [2,3].
Z = [1, [2, 3], 1]
X = 1
Y = [2, 3]
```

Note the similarity to ML patterns.

Lists, continued

What is produced by the following queries?

```
?- [A, B] = [X, A], X = a.

?- [A, B, C] = [C, C, A].
```

Write a predicate `empty(L)` that succeeds iff `L` is an empty list. BE SURE it succeeds only on lists and no other types.

Write a predicate `f(X)` that succeeds iff `X` is a list with one or three elements or `X` is an odd number.

Lists, continued

Answers:

```
?- [A, B] = [X, A], X = a.
A = a
B = a
X = a

?- [A,B,C] = [C,C,A].
A = _G296
B = _G296
C = _G296

empty([]).

f([_]).
f([_,_,_]).
f(N) :- number(N), N mod 2 =:= 1.
```


Built-in list-related predicates

Section A.1 in the SWI-Prolog manual (page 270 in the PDF) describes a number of predicates for list manipulation. Here are a few of them:

`length(?List, ?Len)` can be used to get the length of a list OR make a list of variables:

```
?- length([10,20,30],Len).
Len = 3

?- length(L,3).
L = [_G319, _G322, _G325]
```

Speculate: What is the result of `?- length(L,N).`?

`reverse(?List, ?Reversed)` unifies `List` with `Reversed`, a reverse of itself.

```
?- reverse([1,2,3], R).
R = [3, 2, 1]

?- reverse([1,2,3], [1,2,3]).
No
```

Speculate: What is the result of `?- reverse(X,Y).`?

List predicates, continued

`numlist(+Low, +High, -List)` unifies `List` with the integers from `Low` to `High`, inclusive:

```
?- numlist(5,10,L).
L = [5, 6, 7, 8, 9, 10]
```

`sumlist(+List, -Sum)` unifies `Sum` with the sum of the values in `List`, which must all be numbers or structures that can be evaluated with `is/2`.

```
?- numlist(1,5,L), sumlist(L,Sum).
L = [1, 2, 3, 4, 5]
Sum = 15

?- sumlist([1+2, 3*4, 5-6/7],X).
X = 19.1429
```


List predicates, continued

`atom_chars(?Atom, ?Charlist)` resembles `implode` and `explode` in ML:

```
?- atom_chars('abc',L).
L = [a, b, c]


?- atom_chars(A, [a, b, c]).
A = abc
```


List predicates, continued

`msort(+List, -Sorted)` unifies `List` with `Sorted`, a sorted copy of `List`:

```
?- msort([3,1,7], L).
L = [1, 3, 7]
```

If `List` is heterogeneous, elements are sorted in "standard order":

```
?- msort([xyz, 5, [1,2], abc, 1, 5, x(a)], Sorted).

Sorted = [1, 5, 5, abc, xyz, x(a), [1, 2]]
```

`sort/2` is like `msort/2` but also removes duplicates.

The `member` predicate

`member(?Elem, ?List)` succeeds when `Elem` can be unified with a member of `List`.

For one thing, `member` can be used to check for membership:

```
?- member(30, [10, twenty, 30]).
Yes
```

It can also be used to generate the members of a list:

```
?- member(X, [10, twenty, 30]).
X = 10 ;
X = twenty ;
X = 30 ;
No
```

Problem: Write a predicate `has_vowel(+Atom)` that succeeds iff `Atom` has a vowel.

```
?- has_vowel('ack').
Yes

?- has_vowel('pfft').
No
```

member, continued

Solution:

```
has_vowel(Atom) :-

    atom_chars(Atom,Chars),

    member(Char,Chars),

    member(Char,[a,e,i,o,u]).
```

How does it work?

The append predicate

The SWI manual states that `append(?List1, ?List2, ?List3)` succeeds when `List3` unifies with the concatenation of `List1` and `List2`.

```
?- append([1,2],[3,4,5],R).
R = [1, 2, 3, 4, 5]
```

`append` is not limited to concatenation. Note what happens when we supply `List3`, but not `List1` and `List2`:

```
?- append(L1, L2, [1,2,3]).
L1 = []
L2 = [1, 2, 3] ;

L1 = [1]
L2 = [2, 3] ;

L1 = [1, 2]
L2 = [3] ;

L1 = [1, 2, 3]
L2 = [] ;
No
```


append, continued

At hand:

```
?- append(L1, L2, [1,2,3]).
L1 = []
L2 = [1, 2, 3] ;

L1 = [1]
L2 = [2, 3] ;
...
```

Think of `append` as demanding a relationship between the three lists: `List3` must consist of the elements of `List1` followed by the elements of `List2`. If `List1` and `List2` are instantiated, `List3` must be their concatenation. If only `List3` is instantiated then `List1` and `List2` represent (in turn) all possible the ways to split `List3`.

What else can we do with `append`?

append, continued

Here are some more computations that can be expressed with `append`:

```
starts_with(L, Prefix) :- append(Prefix,_,L).

take(L, N, Result) :- append(Result, _, L), length(Result,N).

drop(L, N, Result) :-
                append(Dropped, Result, L), length(Dropped, N).
```

What does the following query do?

```
?- append(A,B,X), append(X,C,[1,2,3]).
```

append, continued

Here is a predicate that generates successive N-long chunks of a list:

```
chunk(L,N,Chunk) :-
    length(Chunk,N),
    append(Chunk,_,L).

chunk(L,N,Chunk) :-
    length(Junk, N),
    append(Junk,Rest,L),
    chunk(Rest,N,Chunk).
```

Usage:

```
?- chunk([1,2,3,4,5],2,Chunk).

Chunk = [1, 2] ;

Chunk = [3, 4] ;

No
```

"Leftovers" are discarded.

The findall predicate

`findall` can be used to create a list of values that satisfy a goal. Here is a simple example:

```
?- findall(F, food(F), Foods).
F = _G350
Foods = [apple, broccoli, carrot, lettuce, rice]
```

`Foods` is in alphabetical order because the food facts happen to be in alphabetical order.

The first argument of `findall` is a *template*. It is not limited to being a single variable. It might be a structure. The second argument can be a series of goals joined with conjunction.

```
?- findall(F-C, (food(F),color(F,C)), FoodAndColors).
F = _G488
C = _G489
FoodAndColors = [apple-red, broccoli-green, carrot-orange,
    lettuce-green, rice-white]
```

Remember that `-/2` is a functor that can be placed between its terms. `findall(-(F,C),...` produces the same result as the above.

findall, continued

The following query creates a list structures representing foods and the lengths of their names.

```
?- findall(Len-F, (food(F),atom_length(F,Len)), LFs).
Len = _G497
F = _G498
LFs = [5-apple, 8-broccoli, 6-carrot, 7-lettuce, 4-rice]
```

Two predicates that are related to `findall` are `bagof/3` and `setof/3` but they differ more than may be grasped at first glance. Section 6.3 in the text has a discussion of them; if they're needed in an assignment we'll discuss them more.

`keysort(+List, -Sorted)` sorts a list of structures with the functor `'-'` based on their first term:

```
 ?- findall(Len-F, (food(F),atom_length(F,Len)), LFs),
    keysort(LFs,Sorted), writeln(Sorted), fail.
[4-rice, 5-apple, 6-carrot, 7-lettuce, 8-broccoli]
No
```