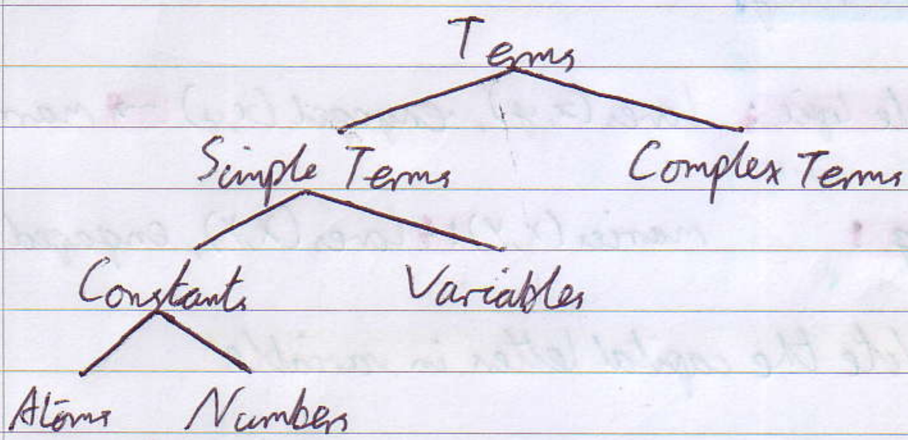
Basics of Prolog

- A logic programming language that implements Logic programming à la ELA.

- It always chooses the left-most query atom and the top-most rule.

- Basic syntax:



- Atoms:
  - A series of letters, numbers and underscores, starting w/ lower case
  - Any sequence of characters in single quotes
  - A sequence of special characters, eg : , ; . :-

- Numbers
  - Floats or integers

- Variables
  - Letters, digits or underscores, starting with upper-case or an underscore.

- Complex Terms
  - An atom functor, followed by arguments in brackets

• Arity and Predicates

- The number of terms in a complex is its arity
- A complex term is called a predicate
- Predicates can exist with different arity and the same functor - they will be treated as different predicates.
- Arity is usually denoted like this:

  myFunctor/2 → myFunctor has arity of 2.

• Rules in Prolog

- Predicate logic: $Loves(x,y), engaged(x,y) \rightarrow marries(x,y)$

- Prolog: `marries(X,Y) :- Loves(X,Y), engaged(X,Y).`

  Note the capital letters in variables

- Negation as failure: `innocent(X) :- \+ guilty(X).`

• Example Prolog Program:

Rules:

```
loves(chuck, sarah).
engaged(chuck, sarah).
marries(X,Y) :- loves(X,Y), engaged(X,Y).
```

Note: full-stops after every rule.

Query:

```
?- marries(X,Y)
```

Reply:

```
X = chuck,
Y = sarah
```

• Unification in Prolog.

- **Definition**: two terms unify if they are the same term or if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal.

- When prolog unifies two terms it performs all the necessary instantiations.

- **Precise Examples:**

1. If T_1 and T_2 are constants...
   - they unify if T_1 and T_2 are the same atom or number

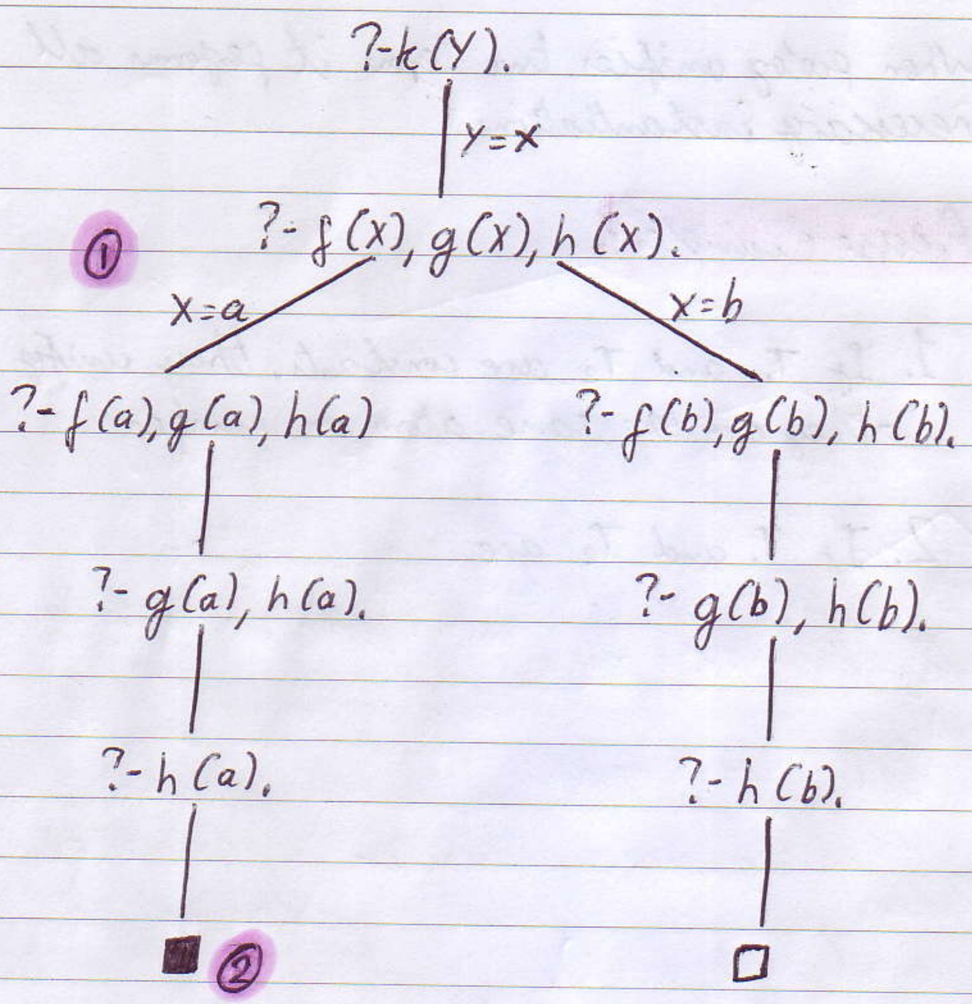
2. If T_1 is a variable and T_2 is any kind of term...
   - T_1 and T_2 unify
   - T_1 is instantiated to T_2

3. If T_1 and T_2 are both variables...
   - they are instantiated to each other

4. If T_1 and T_2 are complex terms, they unify if...
   - they have the same functor and arity
   - all their corresponding arguments unify
   - the variable instantiations are compatible.

Search Trees

Rules: f(a). f(b).
g(a). g(b).
h(b).
k(X) :- f(X), g(X), h(X).

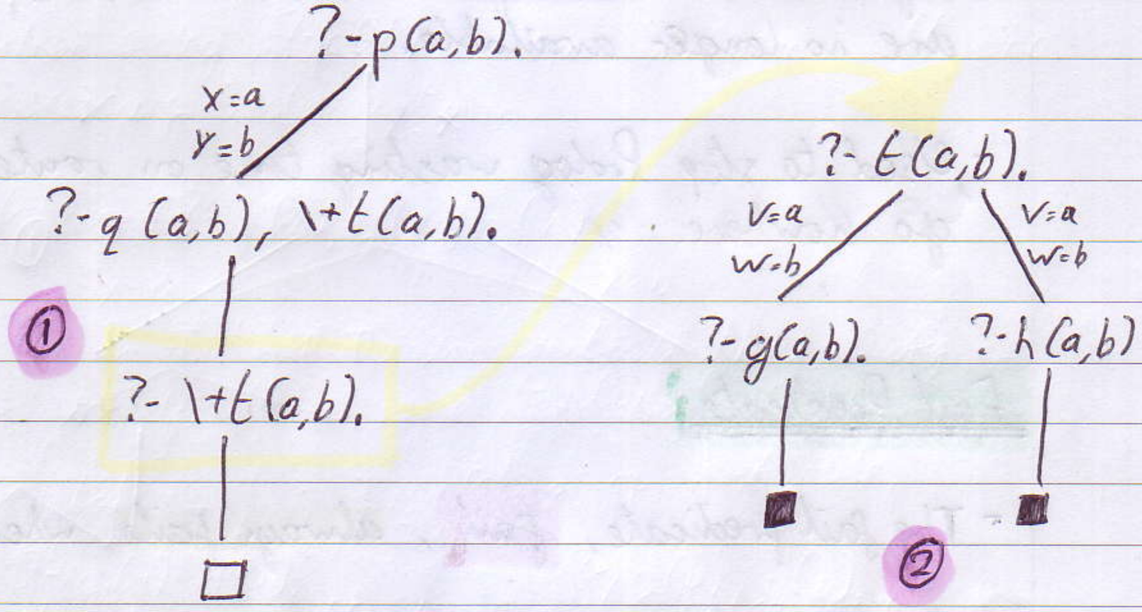


1. Match left-most term to matching top-most rule.

2. Failure, so backtrack to last choice point.

• Search Tree With Negation as Failure

```
p(a,b):- q(X,Y), \+t(X,Y).
t(V,W):- g(V,W).
t(V,W):- h(V,W).
q(a,b).
```



1. Encountered \+, so check w/ a sub-tree

2. All computations of t(a,b) fail, so \+t(a,b) succeeds.

• Finding Alternatives

- Prolog can be forced to find alternatives by pressing semi-colon when a result is returned.

The Cut

- The cut predicate, !, always succeeds and commits prolog to any choices made above the cut.

- When the cut is encountered, all "choice-points" before it become "fixed", so that unexplored options are no longer available.

- Used to stop Prolog wasting time on routes that go nowhere.

Fail Predicate

- The fail predicate, fail, always fails when reached.

- Used in negation as failure:

```
?- \+ Term.
```

↓ creates

```
① \+ Term :- Term, !, fail.
② \+ Term.
```

1. Matches ① first.
2. If "Term" , "!" succeeds and "fail" causes a fail
3. If "Term" fails, ② is tried next and succeeds.

Dynamic Variables

:- dynamic (f) => "f is a thing, but not true for anything"

Recursion

Base Case

Eg. descends (X, Y) :- child (X, Y).
descends (X, Y) :- child (X, Z), descends (Z, Y).

Recursive Case

Lists

- Denoted by [a, b, c, d]
- Can contain any type of Prolog term
- [] => Empty list
- Head = first item
  Tail = the rest of the list; itself a list
- The | operator splits the head from a tail

[X, Y | Z] = [a, b, c, d, e, f].

↳ Yes
X = a
Y = b
Z = [c, d, e, f]

_ can denote an anonymous variable.

• Membership

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

or

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

Built in to Prolog.

IMPORTANT!

Arithmetic

Number is Expression

- Uses integers and real numbers.

Arith.	Prolog
2+3=5	?- 5 is 2+3.
3-5 = -2	?- -2 is 3-5
1 is the remainder of 7/2	?- 1 is mod(7,2).

true

- The is/2 predicate forces Prolog to use arithmetic

• Defining Predicates

```
addThreeAndDouble(X, Y) :- Y is (X+3) * 2.
```

```
?- aTAD(1, Y).
Y = 8
```

```
?- aTAD(X, 10).
X = 2
```

Fine.

Won't Work.

- Restrictions

  - Free to use variables on the right side of is, but when Prolog actually carries out the evaluation they must be instantiated with a variable-free term, which must be an arithmetic expression.

More on Lists

- Length of a List

Length of an empty list: 0
Length of a non-empty list: 1 + length of the tail

Prolog:

```
len([], 0).
len([H|T], Len) :- len(T, TLen),
                   Len is TLen + 1.
```

Eg. ?- len([a, b, c], Len).   ∴ Len = 3

no   ?- len([b, c], Len1), Len is Len1 + 1   ($Len = 3$)

no   ?- len([c], Len2), Len1 is Len2+1   ($Len1 = 2$)

no   ?- len([], Len3), Len2 is Len3+1   ($Len2 = 1$)

Len3 = 0

0, 1, 2

- len/2 is easy to understand and relatively efficient, but an alternative exists using accumulators:

• Accumulators

- The accumulator acclen/3 has three arguments:
    - The list
    - The length of the list as an integer
    - An accumulator, keeping track of intermediate values for the length

- acclen/3 method:
    - Initial acc. value is 0
    - Add 1 to the acc. each time we recursively take the head of the list
    - At the end, the length is in the acc.

```
acclen([], Acc, Length):-
        Length = Acc.

acclen([H|T], OldAcc, Length):-
        NewAcc is OldAcc + 1,
        acclen(T, NewAcc, Length).
```

Call with: acclen([a,b,c], 0, Len).

Must init. acc. to 0.

Using a wrapper predicate : length(List, Length) :-
    acclen(List, 0, Length).

- Tail Recursion

- Why is acclen/3 better than len/2 ? Tail Recursion.

- In tail recursion, results are fully calculated by the bottom of the search tree (the base case)
- In non-tail recursion, there are still goals on the stack when we reach the base case.

More on Arithmetic

- Comparing Integers

Arithmetic	Prolog
$x < y$	`x < y`
$x \leq y$	`x =< y`
$x = y$	`x =:= y`
$x \neq y$	`x =\= y`
$x \geq y$	`x >= y`
$x > y$	`x > y`

- These force the left and right sides to be evaluated

- An accumulative "max-finder":

```
accMax([H|T], A, Max):-
    H>A, accMax(T, H, Max).

accMax([H|T], A, Max):-
    H=<A, accMax(T, A, Max).

accMax([], A, A).
```

With a wrapper:

```
max([H,T], Max):-
accMax(T, H, Max).
```

• Potential Improvement w/ The Cut:

```
max(X, Y, Y) :- X =< Y, !.
max(X, Y, X) :- X > Y.
```

- If X =< Y succeeds, the cut commits us to this choice and the second clause of max/3 is not considered.
- If X =< Y fails, Prolog continues to the second clause.

Even More Lists

• Append

- append(L1, L2, L3) is true if L3 is the list produced by joining L1 and L2

```
append([], L, L).

append([H|L1], L2, [H|L3]) :-
        append(L1, L2, L3).
```

- Base case: appending any list to [] returns the same list.
- Recursion: when joining a non-empty list [H|L1] with L2, the result is a list with the head and the result of joining L1 and L2.

?- append([a,b], [c,d], L3).

■

?- append([b], [c,d], L3.1).

L3 = [a|L3.1]

■

?- append([], [c,d], L3.2).

L3.1 = [b, L3.2]

□ L3.2 = [c,d]

• Uses of append/3

- Splitting:

?- append(X, Y, [a,b,c,d]).

X = []   Y = [a,b,c,d,e]
X = [a]   Y = [b,c,d,e]
....

- Prefix:

```
prefix(P,L) :- append(P,_,L).
```

?- prefix([a,b], [a,b,c,d]).
true.

?- prefix([a,b], [a,c,e]).
false.

- Suffix:

```
suffix(S,L):- append(_, S, L).
```

```
?- suffix([a], [b, a]).
true

?- suffix([b], [a,b,c]).
false
```

- Sublist:

```
sublist(Sub, List):-
    suffix(Suffix, List),
    prefix(Sub, Suffix)
```

- Reversing a List

```
basicRev([], []).

basicRev([H|T], R):-
    basicRev(T, RT),
    append(RT, [H], R).
```

- Base case: reversal of an empty list is the empty list

- Recursive: if we reverse [H|T], we get the list obtained by reversing T and appending it to [H].

- Very Inefficient!

- Reversing a List w/ an Accumulator

- The acc. will be a list that starts empty
- Take the head of the input list and add it to the head of the acc.
- At the end the acc will contain the complete, reversed list.

```
accRev([], L, L).

accRev([H|T], Acc, Rev):-
        accRev(T, [H|Acc], Rev).

reverse(L1, L2):- accRev(L1, [], L2).
```

wrapper.