Clause Tree

- Clause tree
  - Shows (recursive) evaluation of all clauses
  - Shows value (instance) of variable for each clause
  - Clause tree is true if all leaves are true

- Factorial example

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

```
factorial(3,6)
  3>0  2 is 3-1  factorial(2,2)  6 is 3*2
  2>0  1 is 2-1  factorial(1,1)  2 is 2*1
  1>0  0 is 1-1  factorial(0,1)  1 is 1*1
    true
```
Tracing

- **trace** lets you step through a goal’s execution
  - **notrace** turns it off

```prolog
?- trace.
true.

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

```
[trace]  ?- my_last(X, [1,2,3]).
  Call: (6) my_last(_G2148, [1, 2, 3]) ? creep
  Call: (7) my_last(_G2148, [2, 3]) ? creep
  Call: (8) my_last(_G2148, [3]) ? creep
  Exit: (8) my_last(3, [3]) ? creep
  Exit: (7) my_last(3, [2, 3]) ? creep
  Exit: (6) my_last(3, [1, 2, 3]) ? creep
  X = 3
```
Goal Execution – Backtracking

- Clauses are tried in order
- If clause fails, try next clause, if available

Example

\[
\begin{align*}
\text{jedi(luke).} & & A=\text{luke}, \\
\text{jedi(yoda).} & & B=\text{vader}; \\
\text{sith(vader).} & & A=\text{luke}, \\
\text{sith(maul).} & & B=\text{maul}; \\
\text{fight}(X,Y) :- \text{jedi}(X), \text{sith}(Y). & & A=\text{yoda}, \\
& & B=\text{vader}; \\
& & A=\text{yoda}, \\
& & B=\text{maul}.
\end{align*}
\]
Prolog (Search / Proof / Execution) Tree

?- fight(A,B).

A=X, B=Y

?- jedi(X), sith(Y).

X=luke

?- jedi(luke), sith(Y).

Y=vader

?- sith(vader).

Y=maul

?- sith(maul).

X=yoda

?- jedi(yoda), sith(Y).

Y=vader

?- sith(vader).

Y=maul

?- sith(maul).
Lists In Prolog

- \([a, b, 1, 'hi', [X, 2]]\)
- But really represented as compound terms
  - \([\ ]\) is an atom
  - \([a, b, c]\) is represented as \(.(a, .(b, .(c, []))))\)
- Matching over lists
  
  ```prolog
  ?- [X, 1, Z] = [a, _, 17]
  X = a,
  Z = 17.
  ```
List Deconstruction

- Syntactically similar to Ocaml: \([H|T]\) like \(h::t\)
  
  ```prolog
  ?- [Head | Tail] = [a,b,c].
  Head = a,
  Tail = [b, c].
  
  ?- [1,2,3,4] = [ _, X | _].
  X = 2
  ```

- This is sufficient for defining complex predicates

- Let’s define \(\text{concat}(L1, L2, C)\)
  
  ```prolog
  ?- concat([a,b,c], [d,e,f], X).
  X = [a,b,c,d,e,f].
  ```
Example: Concatenating Lists

To program this, we define the “rules” of concatenation

- If L1 is empty, then C = L2
  \[ \text{concat( [ ], L2, L2 )}. \]
- Prepending a new element to L1 prepends it to C, so long as C is the concatenation of L1 with some L2
  \[ \text{concat( [E | L1], L2, [E | C] ) :- \text{concat(L1, L2, C)}.} \]

... and we’re done
Why Is The Return Value An Argument?

- Now we can ask what inputs lead to an output

```prolog
?- concat(X, Y, [a,b,c]).

X = [ ],
Y = [a, b, c] ;
X = [a],
Y = [b, c] ;
X = [a, b],
Y = [c] ;
X = [a, b, c],
Y = [ ] ;
```

User types ; to request additional answers
More Syntax: Built-in Predicates

- Equality (a.k.a. unification)
  \[ X = Y \quad f(1,X,2) = f(Y,3,\_\_) \]

- fail and true

- “Consulting” (loading) programs
  ```
  ?- consult('file.pl')
  [file.pl]
  ```

- Output/Input
  ```
  ?- write('Hello world'), nl
  ?- read(X).
  ```

- (Dynamic) type checking
  ```
  ?- atom(elephant)
  ?- atom(Elephant)
  ```

- help