Prolog - next 2 weeks

- Overview
- Prolog and Predicate Calculus
- Substitution and Unification
- Prolog Technique
  - Goals and Subgoals
  - List Processing
  - Recursion
  - Miscellaneous Functions
- Examples
- Conclusion

Prolog

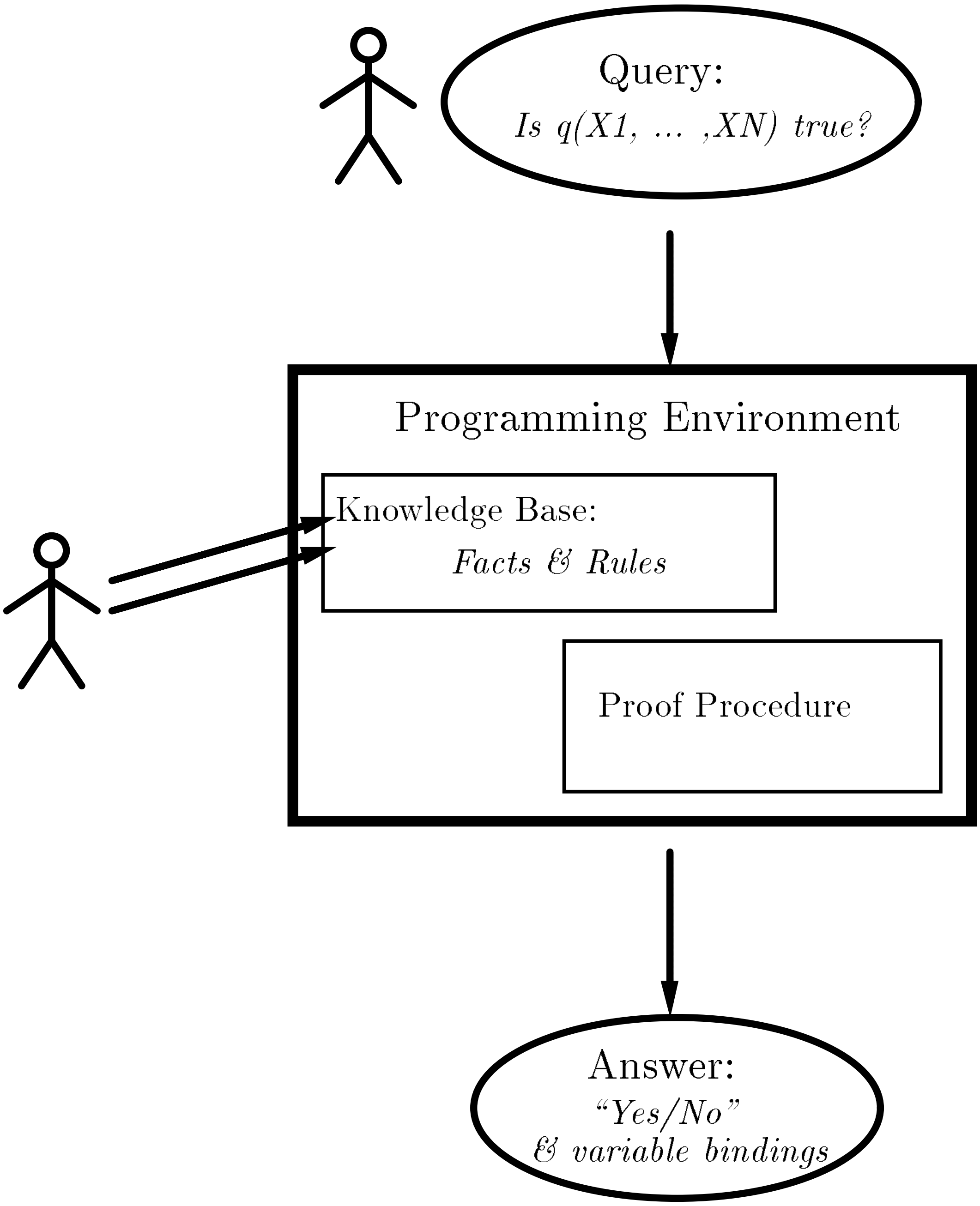
Programming in **Log**ic

- idea emerged in early 1970's;
  most work done at Univ. of Edinburgh

- based on a subset of first-order logic
  - feed it theorems and pose queries, system does the rest

- main uses:
  - originally, mainly an AI language esp. expert systems
  - now finding uses in database systems and even rapid prototyping systems of industrial software

Logic Programming Framework



Logic and Prolog

- Syntax: — Horn clauses of *terms*

- Proof process: *Resolution*

  - *unification* + *substitution*:
    pattern matching between terms + binding unresolved variables as needed.

  - *automatic backtracking*:
    if one attempt fails, try again until all search paths exhausted.

Prolog Terms

BNF:

<term>	::=	<constant> | <variable> | <functor>
<constant>	::=	<atom starting with l.case letter>
<variable>	::=	<atom starting with u.case letter>
<functor>	::=	<constant> [(<term> {, <term> })]

Examples:

`a1  b  fred  married`	→ constants
`X  Fred  Yc3`	→ variables
`found  married(fred)  f(g(Y),X)`	→ functors

DEFN: a *ground term* is a term with no variables.

f(q) g(f(w),w1(b,c)) are ground terms.
f(A) g(f(w),w1(B,c)) are not.

How Prolog Does Proofs

User provides:	Knowledge base (list of Horn clauses) + a query γ.
Prolog finds:	a proof (if one exists) + variable bindings for γ OR report of failure.

Prolog's Syntax

We would like to have full first-order logic $(\wedge, \vee, \neg, \Rightarrow, \forall, \exists)$, but we settle for a subset:

$\longrightarrow$ positive Horn clauses.

$$\begin{aligned}\langle \textit{ Horn clause } \rangle ::= & \langle \textit{ term } \rangle\textbf{.} \\ & \mid \langle \textit{ term } \rangle : - \langle \textit{ term } \rangle \ \{\textbf{,} \langle \textit{ term } \rangle \}\textbf{.}\end{aligned}$$

Examples:

1. `father(john,sue).`
2. `ancestor(odin,X).`
3. `parent(X,Y) :- mother(X,Y).`
4. `grandparent(X,Y) :- parent(X,Z), parent(Z,Y).`

Reading Prolog as Predicate Calculus

","	means	AND / &
": −"	means	IMPLIES / $\Leftarrow$

Variables are assumed to be quantified by $\forall$.

Interpretations of examples:

1. `father(john,sue)`

2. $\forall$ X . `ancestor(odin,X)`

3. $\forall$ X, Y . `mother(X,Y)` $\Rightarrow$ `parent(X,Y)`

4. $\forall$ X, Y, Z . (`parent(X,Z)` $\wedge$ `parent(Z,Y)`)
   $\Rightarrow$ `grandparent(X,Y)`

Horn Clauses vs. Predicate Calculus

- `FACT:` any PC formula can be rewritten in <u>clausal form</u> using only $\vee, \wedge$ and $\neg$.

  e.g. `A` $\Rightarrow$ `B` equiv. to $\neg A \vee B$

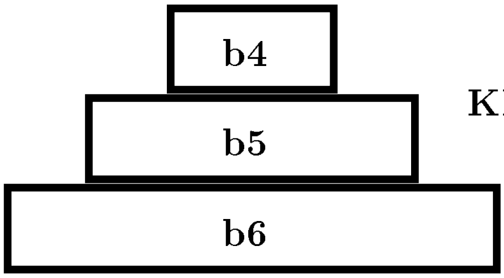
- `FACT:` any clausal form formula can be rewritten into <u>CNF</u> (Conjunctive Normal Form).

  e.g. $(a_1 \vee \neg a_2 \cdots) \wedge (\neg b_1 \vee \neg b_2 \cdots) \wedge \cdots$

- <u>Horn clauses</u> are CNF formulas with exactly one positive literal.

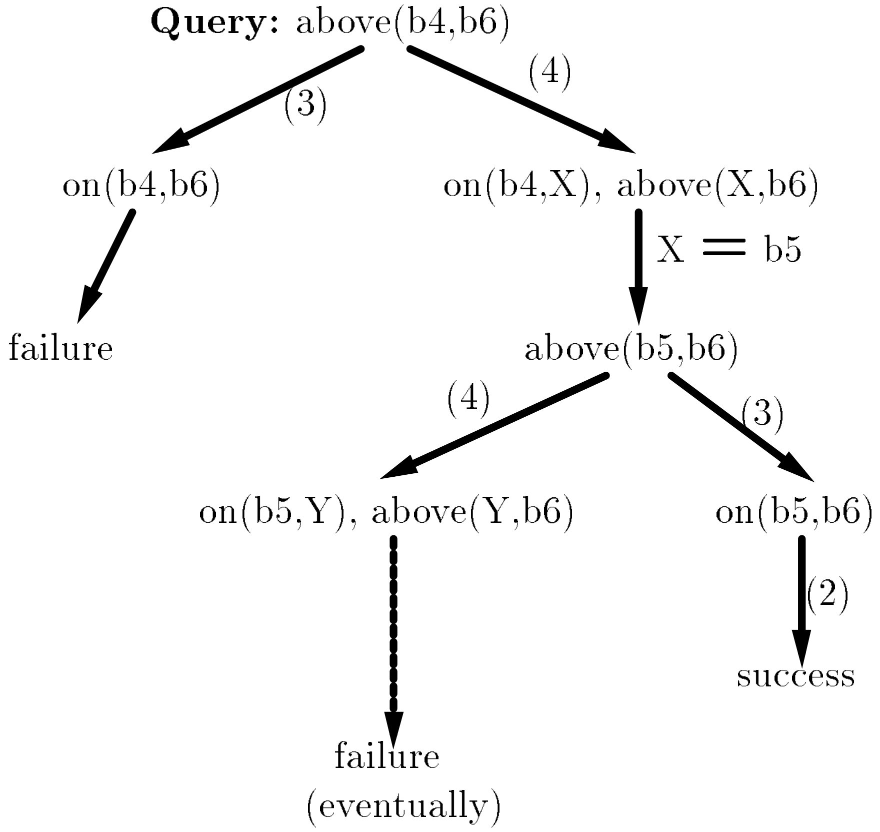
  e.g.
  $$\begin{array}{ll} & \neg p_1 \ \vee \ \neg p_2 \ \vee \ p_3 \ \vee \ \neg p_4 \\ \equiv & p_3 \ \Leftarrow \ (p_1 \ \wedge \ p_2 \ \wedge \ p_4) \\ \equiv & p_3 \ :- \ p_1, \ p_2, \ p_4. \qquad \text{[Prolog]} \end{array}$$

An Example



KB:

(1) on(b4, b5).
(2) on(b5,b6).
(3) above(X,Y) :− on(X,Y).
(4) above(X,Y) :− on(X,Z), above(Z,Y).



Example (cont'd)

Can also have variables in the proof:

$$\begin{tikzcd}
\text{above(b5,W)} \arrow[d, "(3)"] \\
\text{on(b5,W)} \arrow[d, "W = b6"] \\
\text{success}
\end{tikzcd}$$

An Example Prolog Session

```
eddit% cprolog
C-Prolog version 1.4
| ?- [boxaxioms].
[boxaxioms consulted 272 bytes 3.72529 e-0.9 sec]

yes
| ?- above(b4,b6).

yes
| ?- above(b5,W).

W = b6   <---  pressed [Enter] here
yes
| ?- above(X,Y).

X = b4
Y = b5 ; <-- user put ";" and [Enter] here
X = b5
Y = b6 ; <-- and here
X = b4
Y = b6 ; <-- and here

no
| ?- halt.

[ Prolog execution halted ]
eddie%
```

Substitution

A substitution is a set

$$\{v_1 \setminus t_1, \ldots, v_n \setminus t_n\}$$

where	v_is	are distinct variable names
	t_is	are terms which do not use <u>any</u> of the v_js, $j = 1, \ldots, n$.

Positive Examples:

$\{X\setminus a,\ Y\setminus b,\ Z\setminus f(a,b)\}$

$\{X\setminus W,\ Y\setminus f(W,V,a),\ Z\setminus W\}$

Negative Examples:

$\{f(X)\setminus a\}$

$\{X\setminus a,\ X\setminus b\}$

$\{X\setminus f(X)\}$

$\{X\setminus f(Y),\ Y\setminus g(q)\}$

Applying a Substitution

Notation: t is a term
σ is a substitution

then $t\sigma$ is the term resulting from performing the substitution σ on t.

Examples:

$$
\begin{array}{ll}
X\{X\backslash a\} & = \\
f(X)\{X\backslash a\} & =
\end{array}
$$

Let $t = f(a, h(Y, b), X)$

$$
\begin{array}{ll}
t\{X\backslash b\} & = \\
t\{X\backslash b,\ Y\backslash f(Z)\} & = \\
t\{X\backslash Z,\ Y\backslash f(Z, a)\} & = \\
t\{W\backslash Z\} & =
\end{array}
$$

Note:

- σ need not include all variables in t.
- σ can include variables not in t.

Composing Substitutions

We can compose substitutions
(barring variable name clashes).

If σ and θ are substitutions, then so is $\sigma \circ \theta$.

DEFN: For any term t, $t[\sigma \circ \theta] = (t\sigma)\theta$.

For example,

$$
\begin{array}{rl}
 & f(X) \ \ [\{X\backslash Z\} \ \circ \ \{Z\backslash a\}] \\
= & (f(X) \ \{X\backslash Z\}) \ \ \{Z\backslash a\} \\
= & f(Z) \ \ \{Z\backslash a\} \\
= & f(a)
\end{array}
$$

Also,

$$
\begin{array}{lcll}
\{X\backslash a\} & \circ & \{Y\backslash b\} & = \\
\{X\backslash Z\} & \circ & \{Z\backslash a\} & = \\
\{X\backslash a\} & \circ & \{X\backslash b\} & =
\end{array}
$$

Unifiers

DEFN: t_1 and t_2 are *unified* by substitution σ
iff $t_1\sigma = t_2\sigma$.

If so, σ is called a *unifier*,
and t_1 and t_2 are said to be unifiable.

Examples:

t_1	t_2	unifier σ	$t_1\sigma$ $(= t_2\sigma)$
f(a)	f(X)		
f(X,b)	f(a,Y)		
f(b,c)	f(b,c)		
f(b,c)	g(X,c)		
f(a,b)	f(c,d)		
f(X,Y)	f(W,Z)		
f(a,b)	f(X,X)		
f(X,a)	f(Y,Y)		
f(g(U),d)	f(X,U)		
f(X)	f(g(Y))		
f(X)	f(Y)		
f(X)	f(g(X))		
f(X,g(X))	f(Y,Y)		

Multiple Unifiers

Assume $t_1 = f(X)$ and $t_2 = f(Y)$.

σ	$t_1\sigma$ $(= t_2\sigma)$
{ X \ Y}	
{ Y \ X}	
{ Y \ Z, X \ Z}	
{ Y \ a, X \ a}	
{ Y \ g(b,Z), X \ g(b,Z)}	
{ X \ Y, W \ h(q,Z)}	

Q: Is there a "best" unifier?

Wish list:

- no irrelevant constants

  {Y \ X} preferred over {X \ a, Y \ a}

- no irrelevant bindings

  {Y \ X} preferred over {Y \ X, W \ h(q,Z)}

In Search of the "Best" Unifier

Assume σ_1 and σ_2 are both unifiers for some pair of terms.

- Suppose σ_1 has a constant where σ_2 has a variable. Then $\exists$ subst. θ such that $\sigma_2 \circ \theta = \sigma_1$

  e.g. $\sigma_1 = \{X \backslash a, Y \backslash a\}$ $\sigma_2 = \{X \backslash Z, Y \backslash Z\}$

  then $\theta = \{Z \backslash a\}$

  and $\{X \backslash Z, Y \backslash Z\} \circ \{Z \backslash a\} = \{X \backslash a, Y \backslash a\}$

- Suppose σ_1 has an extra binding over σ_2. Then $\exists$ subst. θ such that $\sigma_2 \circ \theta = \sigma_1$

  e.g. $\sigma_1 = \{X \backslash a, W \backslash b\}$ $\sigma_2 = \{X \backslash a\}$

  then $\theta = \{W \backslash b\}$

  and $\{X \backslash a\} \circ \{W \backslash b\} = \{X \backslash a, W \backslash b\}$

Moral:

inferior unifier = better unifier composed with another subst.

“Most General Unifier” (MGU)

DEFN: σ is an mgu for t_1 and t_2 iff
(1) σ unifies both t_1 and t_2, and
(2) $\forall$ subst. μ that unify t_1 and t_2,
$\exists$ subst. θ such that $\sigma \circ \theta = \mu$.

i.e. $t_1\mu = (t_1\sigma)\theta$
and $t_2\mu = (t_2\sigma)\theta$.

Basic idea: An mgu has

- no unnec./irrelevant constants
  (uses constants only when necessary)

- no extraneous bindings

An Example

$$\boxed{t_1 = f(X) \quad t_2 = f(Y)}$$

then $\sigma = \{X \backslash Y\}$ is an mgu.

Why?

Consider $\mu = \{X \backslash a, Y \backslash a\}$, another unifier for t_1 and t_2.

Then $\exists\ \theta = \{Y \backslash a\}$ such that $\sigma \circ \theta = \mu$.

Note also that $\{Y \backslash X\}$ and $\{Y \backslash Z, X \backslash Z\}$ are also mgus for t_1 and t_2.

Another Example

$$t_1 = f(W,\ g(Z),\ Z)$$

$$t_2 = f(X,\ Y,\ h(X))$$

Need
$$\begin{array}{ccc} W & \longleftrightarrow & X \\ g(Z) & \longleftrightarrow & Y \\ Z & \longleftrightarrow & h(X) \end{array}$$

so mgu $\sigma =$

More MGU Examples

t_1	t_2	MGU σ
f(X,a)	f(a,Y)	
f(h(X,a),b)	f(h(g(a,b),Y),b)	
g(a,W,h(X))	g(Y,f(Y,Z),Z)	
f(X,g(X),Z)	f(Z,Y,h(Y))	
f(X,h(b,X))	f(g(P,a),h(b,g(Q,Q)))	

Prolog's Proof Process

`DEFN:` A goal is either a sequence of terms or the special goal `success`.

e.g. `G` $=$ `p(X,Y), q(Y)`
`G` $=$ `above(b4,b5), on(b4,b5), success`

`DEFN:` $G_1, \ldots, G_n$ is a top-down proof of G_1 (given a set of true Horn clauses/axioms) iff

1. $G_n =$ `success`

2. G_i is a subgoal of G_{i-1}, $i = 1, \ldots, n$ according to either (I) or (II) below.

Subgoals

Given goal $\mathtt{G} = g_1, \ldots, g_m$ (g_js are terms)

(I) If there is an <u>atomic</u> axiom t where t and g_j have mgu σ, then

$$g_1\sigma, \ldots, g_{j-1}\sigma, g_{j+1}\sigma, \ldots, g_m\sigma$$

is a subgoal of $\mathtt{G}$.

(II) If there is a rule axiom t :– $t_1, \ldots, t_k$. where t and g_j have mgu σ, then

$$t_1\sigma, \ldots, t_k\sigma, g_1\sigma, \ldots, g_{j-1}\sigma, g_{j+1}\sigma, \ldots, g_m\sigma$$

is a subgoal of $\mathtt{G}$.

Restriction: variables of axioms must be distinct from variables of subgoals. Must rename if necessary.

This is a lot simpler than it sounds.

Example

KB:

(1)	`on(b4,b5).`
(2)	`on(b5,b6).`
(3)	`above(X,Y) :- on(X,Y).`
(4)	`above(X,Y) :- on(X,Z), above(Z,Y).`
(5)	`above(c1,c2).`

Goal: `above(A,B).`

Possible subgoals:

- `on(A,B)` $\sigma = \{X \backslash A, Y \backslash B\}$
  using rule II, axiom (3)

- `on(A,Z),above(Z,B)` $\sigma = \{X \backslash A, Y \backslash B\}$
  using rule II, axiom (4)

- `success` $\sigma = \{A \backslash c1, B \backslash c2\}$
  using rule I, axiom (5)

Possible subgoals of

`on(A,`Z_1`), above(`Z_1`,B)`:

- `above(b5,B)` $\sigma = \{A\backslash b4, Z_1\backslash b5\}$
  using rule I, axiom (1)

- `above(b6,B)` $\sigma = \{A\backslash b5, Z_1\backslash b6\}$
  using rule I, axiom (2)

- `on(A,c1)` $\sigma = \{Z_1\backslash c1, B\backslash c2\}$
  using rule I, axiom (5)

- `on(`Z_1`,B), on(A,`Z_1`)` $\sigma = \{X\backslash Z_1, Y\backslash B\}$
  using rule II, axiom (3)

- `on(`Z_1`,Z), above(Z,B), on(A,`Z_1`)`
  $\sigma = \{X\backslash Z_1, Y\backslash B\}$
  using rule II, axiom (4)

Lists in Prolog

special constant symbol	[ ]	$\approx$	`nil`
special function symbol	$\bullet$	$\approx$	`cons`

Lisp list: `(a b c)`

Prolog list: $\bullet$`(a,` $\bullet$`(b,` $\bullet$`(c, []))`

$\approx$ `(cons 'a (cons 'b (cons 'c nil)))`

Prolog shorthand:

1. $[t_1, \ldots, t_n] \equiv (\texttt{list}\ 't_1 \cdots 't_n)$

2. $[t_1, \ldots, t_n \mid \texttt{L}] \equiv (\texttt{cons}\ 't_1\ \ (\texttt{cons}\ 't_2\ (\cdots$
$(\texttt{cons}\ 't_n\ \texttt{L})\ \cdots)$

Thus `[a,b,c]` $\equiv$ `[a | [b,c]]`
$\equiv$ `[a,b | [c]]`
$\equiv$ `[a,b,c | []]`

List Membership

Prolog uses predicates, not functions. Want to define a two-place predicate

`member(X,Y)`

that holds iff `X` is a member of list `Y`.

Solution:

```
(1) member(X,[X|Y]).
(2) member(X,[Z|Y]) :- member(X,Y).
```

Interpretation:

1. `X` is a member of any list of which it is the first element.
2. If `X` is a member of list `Y`, then `X` is a member of any list that has `Y` as its `cdr`.

An Example

```
| ?- trace, member(c,[a,b,c,d]).
   (2) 1 Call: member(c,[a,b,c,d]) ?
   (3) 2 Call: member(c,[b,c,d]) ?
   (4) 3 Call: member(c,[c,d]) ?
   (4) 3 Exit: member(c,[c,d])
   (3) 2 Exit: member(c,[b,c,d])
   (2) 1 Exit: member(c,[a,b,c,d])
yes
| ?- trace, member(x,[a,b,c]).
   (2) 1 Call: member(x,[a,b,c]) ?
   (3) 2 Call: member(x,[b,c]) ?
   (4) 3 Call: member(x,[c]) ?
   (5) 4 Call: member(x,[]) ?
   (5) 4 Fail: member(x,[])
   (4) 3 Fail: member(x,[c])
   (3) 2 Fail: member(x,[b,c])
   (2) 1 Fail: member(x,[a,b,c])
no
```

Inductive Reasoning

- Identify Base Cases

  Handled directly as facts

  Simple solution, e.g null

- Identify Decomposition

  Reduce General Case to simpler sub cases

- Identify Composition

  Compose solution of Subcases to solution of original general case

Example - member

- Desired, `member( X , Y )` succeeds if X is a member of list Y

- Base case:
  X is at head of list Y
  ```
  member( X , [X,Y] ).
  ```

- Induction case:
  X is member of rest of Y except head
  ```
  member( X, [Z,Y] ) :- member( X, Y).
  ```

- Order of axioms matters.

Example - Append

- Desired: `append(X,Y,Z)`

  Z is result of appending X onto Y

- Base case:

  Append empty list onto Y

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

- Decomposition:

  Process head of X

  Process rest of X

  Reduce length of X

- Composition:

  Add to head of Z

- Solution:

  `append([E|X],Y,[E|Z]):-append(X,Y,Z).`

An Example

append([a,b],[c,d],V)

$$\Big\downarrow (2) \qquad E_1 = a \quad X_1 = [b] \qquad Y_1 = [c, d] V = [a|Z_1]$$

append([b],[c,d],Z_1)

$$\Big\downarrow (2) \qquad E_2 = b \quad X_2 = [] \qquad Y_2 = [c, d] Z_1 = [b|Z_2]$$

append([ ],[c,d],Z_2)

$$\Big\downarrow (1) \qquad Y_2 = [c, d] \qquad Z_2 = [c, d]$$

success

`V` $=$ `[a` $|Z_1$`]` $=$ `[a | [b` $|Z_2$`]]`

$=$ `[a | [b | [c,d]]]` $=$ `[a,b,c,d]`

More Examples

append([a,b],[c],[a,b,c,d])

(2) $E_1 = a \quad X_1 = [b]$

$Y_1 = [c] \quad Z_1 = [b, c, d]$

append([b],[c],[b,c,d])

(2) $E_2 = b \quad X_2 = []$

$Y_2 = [c] \quad Z_2 = [c, d]$

append([ ],[c],[c,d])

No possible unification.

failure

More Examples

$$\begin{tikzcd}
\text{append([a,b],W,[a,b,c])} \arrow[d, "(2)"', "\begin{array}{l} E_1 = a \quad X_1 = [b] \\ W = Y_1 \;\; Z_1 = [a,b,c] \end{array}"] \\
\text{append([ ],}Y_1\text{,[b,c])} \arrow[d, "(1)"', "Y_1 = [c]"] \\
\text{success}
\end{tikzcd}$$

$$\Rightarrow W = Y_1 = [c, d]$$

Similarly,

`append(D,[c,d],[a,b,c,d])`

succeeds with `D` $=$ `[a,b]`.

Example - Reverse a List

- Desired `rev( L , M )`
  where the list M is the result of reversing the order of the list L

- Solution without helping functions

- Base case:
  Reverse empty list
  `rev( [], [] ).`

- Induction case:
  Append head of L to tail of output
  `rev([H,T],M) :- rev(T,Z),append(Z,[H],M).`

Using Helping Functions

- One common technique is to design *helping functions* that do the real work of generating the desired results.

- Use extra parameters to helping functions to hold intermediate values or partial results.

- Use extra parameters to preserve original input for subesquent processing or second pass.

- Example:

```
doit(X,Y) :- realDoIt(X,Y,Y,[]).
```

Example - Reverse a List

- Use helping function to hold partial result

- Base case:

  Start with empty output list

  `rev2(L1,L2) :- revzap(L1,[],L2).`

- Induction case:

  Accumulate results so far in 2nd arg of revzap

  ```
  revzap([],L,L).
  revzap([X,L],L2,L3):-revzap(L,[X,L2],L3).
  ```

Equality in Lists

The predicate for equality of lists is trivial:

```
equal(X,X).
```

Unification does all the work!

Note that equal(t_1, t_2) succeeds if t_1 and t_2 are merely unifiable:

`equal([a,b,c],[a|X])` $\leadsto$ success + `X = [b,c]`

List Membership

Recall:

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

We can rewrite this as:

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

"_" is a special Prolog term that will unify with <u>anything</u>.

We use "_" instead of a more normal variable name to emphasize that we don't care what it unifies with.

Each occurrence may unify to something different!

Negation as Failure

Consider the intersection predicate for lists:

```
(1) intersection([ ],X,[ ]).
(2) intersection([X|L],Y,[X|Z]) :- member(X,Y),
        intersection(L,Y,Z).
(3) intersection([X|L],Y,Z) :- nonmember(X,Y),
        intersection(L,Y,Z).
```

Problem: How to define `nonmember`.

$\Rightarrow$ need negation.

```
nonmember(X,Y) :- not(member(X,Y)).
```

`not` is a special function symbol:

$$\text{Rule :}\quad \text{Goal } \texttt{not(P)} \begin{cases} \text{succeeds} & \text{if } \texttt{P} \text{ fails} \\ \text{fails} & \text{if } \texttt{P} \text{ succeeds} \end{cases}$$

Example: `append(X,Y,[a]), not(X=[ ])`

will succeed with `X = [a], Y = [ ]`.

An Extended Example

We will represent the natural numbers as Prolog terms:

constant	0	
unary function symbol	`s`	$\approx$ `successor`

0	represents	0
`s(0)`	...	1
`s(s(0))`	...	2
`s(s(s(0)))`	...	3
	etc.	

[Prolog <u>does</u> have numbers, but we ignore them for now.]

Addition Predicate

Want to define `sum`:

`sum(X,Y,Z)` holds iff $X + Y = Z$.

Solution:

$$\texttt{sum(0,Y,Y).} \qquad (1)$$

$$\texttt{sum(s(X),Y,s(Z)) :- sum(X,Y,Z)} \qquad (2)$$

An Example

$$\mathrm{sum(s(s(0)),s(s(s(0))),Total)}$$

$$(2)\ \Big\downarrow \quad \begin{aligned} Total &= s(Z) \\ X &= s(0) \\ Y &= s(s(s(0))) \end{aligned}$$

$$\mathrm{sum(s(0),s(s(s(0))),Z)}$$

$$(2)\ \Big\downarrow \quad \begin{aligned} Z &= s(Z_1) \\ X_1 &= 0 \\ Y_1 &= s(s(s(0))) \end{aligned}$$

$$\mathrm{sum(0,s(s(s(0))),}Z_1\mathrm{)}$$

$$(1)\ \Big\downarrow \quad Z_1 = s(s(s(0)))$$

$$\mathrm{success}$$

$$\Rightarrow Total = s(s(s(s(s(0)))))$$

Another Example

Notation: $\overline{n} \equiv \underbrace{s(s(\cdots s(}_{n \text{ times}} 0) \cdots)$

$$
\begin{array}{c}
\text{sum(s(s(0)),s(s(s(0))),s(s(s(s(0)))))} \\
\Big\downarrow (2) \quad X = \overline{1},\ Y = \overline{3},\ Z = \overline{3} \\
\text{sum(s(0),s(s(s(0))),s(s(s(0))))} \\
\Big\downarrow (2) \quad X_1 = 0,\ Y_1 = \overline{3},\ Z_1 = \overline{2} \\
\text{sum(0,s(s(s(0))),s(s(0)))} \\
\Big\downarrow \quad \text{No unification possible.} \\
\text{failure}
\end{array}
$$

$\Rightarrow$ `sum(`$\overline{2}, \overline{3}, \overline{4}$`)` fails

Addition as Subtraction

$$\begin{array}{ll}
\mathrm{sum}(\overline{1}, W, \overline{3}) & \\
\quad\Big\downarrow (2) & X = 0,\; Y = W,\; Z = \overline{2} \\
\mathrm{sum}(\overline{0}, Y, \overline{2}) & \\
\quad\Big\downarrow (1) & Y = \overline{2} \\
\text{success} & \Rightarrow \mathtt{W} = \overline{2}
\end{array}$$

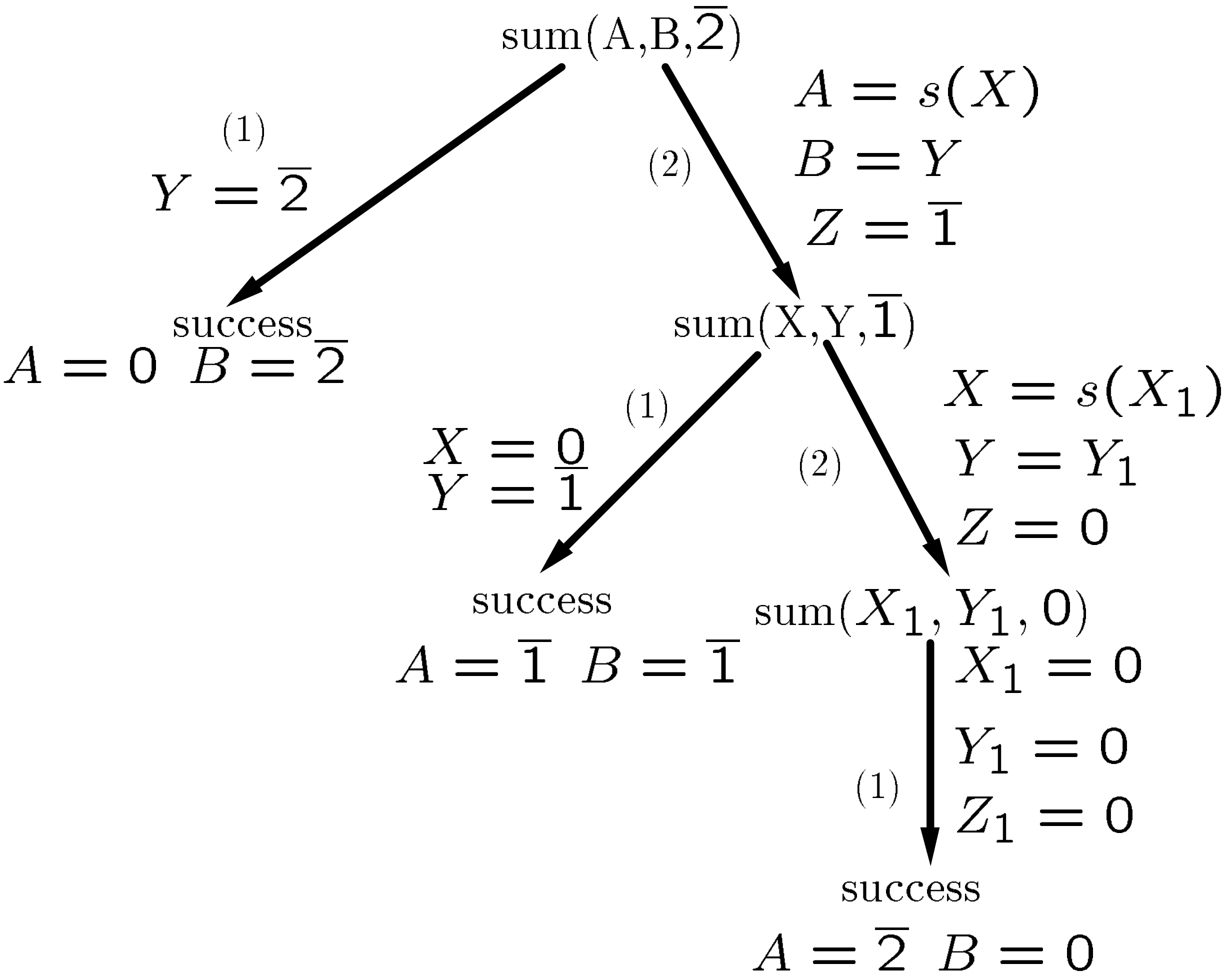
Similarly, $\mathtt{sum(D,}\overline{3},\overline{8}) \rightsquigarrow \mathtt{D} = \overline{5}$

$\Rightarrow$ can define subtraction via one axiom:

```
diff(X,Y,Z) :- sum(Y,Z,X).
```

`diff(X,Y,Z)` holds iff $X - Y = Z$.

Creative Use of Variables



Similarly, `sum(A,B,`$\overline{5}$`)` succeeds with

$A = 0$ $\qquad B = \overline{5}$
$A = \overline{1}$ $\qquad B = \overline{4}$
$A = \overline{2}$ $\qquad B = \overline{3}$

etc.

Assignment and Arithmetic

`X is Y`	Assign Y to X
`X + Y`	addition
`X - Y`	subtraction
`X * Y`	multiplication
`X / Y`	division
`X mod Y`	remainder of `X / Y`

Arithmetic Compare

Arithmetic Comparisons apply to *numbers* only.
Use arithmetic comparisons as goals in terms.

`X = Y`	X and Y the same number
`X \= Y`	X and Y different numbers
`X < Y`	X is less than Y
`X > Y`	X is greater than Y
`X =< Y`	X is less than or equal to Y
`X >= Y`	X is greater than or equal to Y

Term Comparisons in Prolg

- Term comparisons in Prolog are used to compare and order *terms*. Comparisons do not cause variables to be instantiated.

- Comparison predicates succeeed or fail. No side effects, substitutions or error conditions.

- Don't use predicates described below when you want arithmetic comparison or unification.

Comparison Predicates

`T1 == T2`	Literally identical terms
`T1 \== T2`	Not literally identical
`T1 @< T2`	T1 before T2 in standard order
`T1 @> T2`	T1 after T2 in standard order
`T1 @<= T2`	T1 not after T2 in standard order
`T1 @>= T2`	T1 not before T2 in standard order

compare predicate

compare(*Op* , *T1*, *T2*)

Succeeds if the result of comparing *T1* and *T2* is *Op*

Op is one of =, <, >

Special Tool: The Cut Symbol “!”

In general,

```
P :- Q, R, S.
P :- not(Q), T, U.
```

May have to try `Q` twice.

Solution: The cut symbol “!”

```
P :- Q, !, R, S.        (1)
P :- T, U.              (2)
```

Behaviour:

- if `Q` succeeds, then `P` succeeds only if `R` and `S` succeed. `T` and `U` will never be attempted even if `R` or `S` cannot be satisfied.
- if `Q` fails, only then will `R` and `S` be attempted.

More on the Cut

- "!" always succeeds; it has the side-effect of committing Prolog to the current rule and the current set of choices for variable values. Effectively, the cut creates a <u>fence</u> that the system is not allowed to back up beyond.

- *e.g.* `A :- B, C, D, !, E, F, G.`
  Prolog can backtrack as much as its likes in attempting to satisfy `B`, `C` and `D`, but once `D` has been satisfied, it may <u>not</u> backtrack beyond the cut. However, it <u>may</u> backtrack over `E`, `F` and `G` as much as it likes in attempting to satisfy them.

- effectively, the “!” indicates that if you get this far, then you’ve made the only correct choice, and you succeed or fail with that choice. You may not back up in this rule, or use any other rule for the same goal.

- Thus,

```
P :- Q, !, R, S.
P :- T, U.
```

behaves like `P:-` **if** `Q` **then** `R, S` **else** `T, U`

Examples

1. The cut is often used with the special term `fail` (that always fails) to indicate that a goal cannot be satisfied.
   e.g. can define `not` ourselves:

   ```
   not(G) :- G, !, fail.
   not(G).
   ```

   If `G` succeeds, the commit to failure; otherwise succeed.

2. ```
   member(X,[X|_]) :- !.
   member(X,[_|L]) :- member(X,L).
   ```

   if the first clause succeeds, then on failure (later on), we know we don't have to examine the rest of the list. List membership is decided once and for all if we satisfy the first rule.
   ```

Examples

3. *"Everyone has two parents, except Adam and Eve who have none."*

   First attempt:

```
numParents(adam,0) :- !.     (1)
numParents(eve,0) :- !.      (2)
numParents(X,2).             (3)

| ?- numParents(eve,X).
X = 0 ;
no

| ?- numParents(fred,X).
X = 2 ;
no

| ?- numParents(adam,2).
yes
```

   What happened? The last query unified with (3), but not (2) so the cut was not encountered.

Better Solution:

```
numParents(adam,N) :- !, N = 0.     (1)
numParents(eve,N) :- !, N = 0.      (2)
numParents(X,2).                    (3)
```

Note that the order is important: cannot interchange (1) and (3).

Moral: be careful how you use cuts.

```
| ?- numParents(X,0).
X = adam ;
no
```

Exercise: Rewrite axioms so that it also returns `X = eve`.

Assert, Retract and Abolish

These are special Prolog predicates with side-effect — they allow clauses to be added and deleted from the current KB.

- `assert(C)`, `asserta(C)`, `assertz(C)`
  the clause `C` is added somewhere / at the beginning / at the end of the KB.

- `retract C`
  the first clause in the KB that unifies with `C` is deleted.

- `abolish(C,N)`
  remove all instances of clause `C` with arity `N`. `abolish` is often used to wipe out all instances of a temporary variable.

Examples

```
(1)  team(george).
(2)  team(bill).
(3)  team(ted).
(4)  team(bob,doug).
(5)  team(wayne,garth).
(6)  team(tom,dick,harry).
(7)  team(X,Y,Z) :- player(X),player(Y),player(Z).
```

- `abolish(team,2)` wipes out (4), (5).
  `abolish(team,1)` wipes out (1), (2), (3).

[Assuming all axioms are still there:]

- `retract(team(_)).` wipes out (1) on the first call, (2) on the second and (3) on the third.

- `retract(team(_,garth)).` wipes out (5).

- `retract((team(_,_,Z) :- Y))` wipes out (6) on the first call and returns `Z` = `harry` and `Y` = `true`.

  On the second call it wipes out (7) and returns `Z` = `_23` and `Y` = `player(_12), player(_6), player(_9)`.

Another Example

```
mar(fred,judy).
mar(fred,sue).
mar(fred,polly).
mar(dale,sharon).
mar(dale,judy).
mar(javier,june).
big(X) :- mar(X,Y), mar(X,Z), not(Y=Z).
big(X) :- mar(Y,X), mar(Z,X), not(Y=Z).

| ?- big(X).
X = fred ; X = fred ; X = fred ;
X = fred ; X = fred ; X = fred ;
X = dale ; X = dale ;
X = judy ; X = judy ;
no
```

Note that `big(fred)` can be satisfied in 6 ways!

So we introduce:

```
bigamist(X) :- big(X), not(injail(X)),
      assert(injail(X)).
```

```
| ?- bigamist(X).
  X = fred ; X = dale ; X = judy ;
no

| ?- bigamist(X).
no
```

Clause

clause (LHS , RHS)

---

clause searches the Prolog Knowledge Base for a clause of the form:

LHS :- RHS

Relation symbol of LHS must be a constant, but it can contain other variables (which may get instantiated).

Clause Example

```
knows(john,mary).
knows(ellen,ted).
knows(ted,Y) :- knows(Y,mary).

| ?- clause( knows(A,B), RHS ).
A = john
B = mary
RHS = true ;

A = ellen
B = ted
RHS = true ;

A = ted
B = _1
RHS = knows(_1,mary) ;

no
```

Executing Data

Term =.. [*Functor* | *Arglist*]

The =.. operator (pronounced "univ") transforms a list into a *functor*

Term is any prolog term.
Functor is its functor, and
Arglist is its argument list

=.. fails if operands are not properly instantiated

=.. Example

```
knows(john,mary).
knows(ellen,ted).
knows(ted,Y) :- knows(Y,mary).

| ?- X =.. [knows, john, mary] .
X = knows(john,mary) ;

no

| ?- X =.. [knows, john, Y]
X = knows(john, _1)
Y = _1 ;

no

| ?- X =.. [knows, john, Y], call(X).
X = knows(john, _1)
Y = mary;

no
```

Improving Efficiency by Helping Functions

- Compute Fibonacci numbers

  Fib(1) = 1

  Fib(2) = 1

  Fib(N) = Fib(N-1) + Fib(N-2)

- Naive Algorithm using Definition

```
fib(1,1).
fib(2,1).
fib(N,F) :- N > 2,
    N1 is N - 1, fib( N1, F1 ),
    N2 is N - 2, fib( N2, F2 ),
    F is F1 + F2 .
```

Naive Fibonnaci Analysis

- Naive solution wastes effort by repeating parts of the computation

Use `assert` to Save Effort

- When each Fibonnaci Number is computed add it as a fact to the knowledge base.

```
fib( 1, 1 ).
fib( 2, 1 ).

fib( N, F ) :-
    N > 2 ,
    N1 is N - 1 , fib2( N1, F1 ),
    N2 is N - 2 , fib2( N2, F2 ),
    F is F1 + F2,
    asserta( fib( N, F ) ).
```

Using `assert` to Save Effort

- For a call of fib(N,F) all Fibonnaci numbers upto N will become facts in the knowledge base.

- The second recursive call in the definition will always succeed imediately because its results has been asserted in the knowledge base by the first recursive call.

Use Helping Function for Efficiency

- Avoid recursive computation by computing Fibonnaci numbers in the forward direction.

```
fib( N, F ) :-
   forwardfib( 2, N, 1, 1, F ).
forwardfib( M, N, F1, F2, F2 ) :-
   M >= N .
forwardfib( M, N, F1, F2, F ) :-
   M < N ,
   NextM is M + 1 ,
   NextF2 is F1 + F2 ,
   forwardfib( NextM, N, F2, NextF2, F ).
```

A Bigger Example: Sorting

Declarative approach:

```
sort(X,Y) :- permute(X,Y), ordered(Y).
```

i.e. `Y` is the sorted version of `X` if `Y` is a permutation of `X` and `Y` is ordered.

`permute` and `ordered` are not difficult to define, but `ordered` will depend of the type list element.

However, this is a terrible way to sort.

A list with n elements has $n!$ permutations, and each of them will be tried until one is found to be ordered.

Sorting Lists

- Common Operation for Organizing Data

- Could use builtin list sorting primitives in Prolog. We'll look at how to do it directly.

- Assume that there is an ordering relation:
  `gt(X,Y)`
  for whatever data type we want to sort.
  Also assume non-empty lists.

- Look at several alternatives:
  bubble sort
  insertion sort
  quick sort

Basic Bubble Sort

Algorithm: swap adjacent elements in the list until all items are in order

```
bubblesort( List, Sorted ) :-
    swap( List, List1 ),
    bubblesort( List1, Sorted ).

bubblesort( Sorted, Sorted ).

swap( [] , [] ).
swap( [ X, Y | Rest ], [ Y, X | Rest ]) :-
    gt( X, Y ).
swap( [ Z | Rest ] , [ Z | Rest1 ] ) :-
    swap( Rest, Rest1 ).
```

Insertion Sort

- To sort a non-empty list `L = [X|T]`
  - Sort the tail T of L
  - Insert the head X into the sorted tail such that the resulting list is sorted

```
insertsort( [], [] ).
insertsort([ X | Tail ], Sorted ) :-
    insertsort( Tail, SortedTail ),
    insert( X, SortedTail, Sorted ).
insert( X, [ Y, Sorted ], [ Y | Sorted1 ] ) :-
    gt( X, Y ) , ! ,
    insert( X, Sorted, Sorted1 ).
insert( X, Sorted, [ X | Sorted ] ).
```

Quicksort

To sort a non-empty list L:

- Select some element X from L as a *pivot*

- Split the rest of L into two lists (Small and Big) such that all elements of L greater than X are in Big and all the rest are in Small

- Sort Small to obtain SortedSmall
  Sort Big to obtain SortedBig

- The whole sorted list is the concatenation of
  SortedSmall and `[ X | SortedBig ]`

Quicksort Algorithm

```
quicksort([] , [] ).
quicksort([X | Tail ], Sorted) :-
    split(X, Tail, Small, Big),
    quicksort(Small, SortedSmall),
    quicksort(Big, SortedBig),
    conc(SortedSmall,[ X | SortedBig],Sorted).

split(X, [], [], []).
split(X, [ Y | Tail ], [ Y | Small ], Big):-
    gt( X, Y ), ! ,
    split( X, Tail, Small, Big ).
split(X, [ Y | Tail ], Small, [ Y | Big ]) :-
    split( X, Tail, Small, Big ).

conc( [], L, L ).
conc( [ X | L1 ], L2, [ X | L3 ] ):-
    conc( L1, L2, L3 ).
```

More Efficient List Concatenation

- Naive list concatenation:

```
conc( [], L, L ).
conc( [ X | L1 ], L2, [ X | L3 ] ):-
    conc( L1, L2, L3 ).
```

- This is very inefficient when first list is long.
  Recurs down first list until it is empty.
  Could do better if we could efficiently find end of first list.

- Change representation of lists to represent a list as a pair of lists.
  A *Difference List* represents a list L and a difference of two lists L1 and L2, written as L1-L2

Difference List Example

```
Let   L = [ a, b, c ]

Then L can be represented as L1-L2 using any of

      L1 = [ a, b, c, d, e ]
      L2 = [ d, e ]

      L1 = [ a, b, c ]
      L2 = [ ]

      L1 = [ a, b, c, d, e | T ]
      L2 = [ d, e | T ]

Where T is any list.
Note that L2 is the end of the list L.
```

Concatentation of Difference Lists

Difference list concatenation:

```
concat( A1-Z1, Z1-Z2, A1-Z2 ).
```

Example:

Let

```
    M = [ a, b, c ]
    M = [ a, b, c | T1 ] - T1
    N = [ d, e ]
    N = [ d, e | T2 ] - T2
```

Then

```
   concat( M, N )
```

is

```
   concat([a, b, c | T1]-T1,[d, e, T2]-T2,L)
```

Resulting in

```
   T1 = [ d, e, | T2 ]
   L  = [ a, b, c, d, e | T2 ] - T2
```

More Efficient Quicksort Using Difference Lists

```
quicksort( List, Sorted ) :-
   quicksort2(List,Sorted-[]).

quicksort2([],Z-Z).

quicksort2([ X | Tail], A1-Z2) :-
    split(X, Tail, Small, Big ),
    quicksort2(Small, A1-[X | A2]),
    quicksort2(Big, A2-Z2 ).
```

Prolog vs. “Standard” Procedures

1. Success vs. failure

```
p(X) :- <body 1>.
p(X) :- <body 2>.
```

Procedural: Try <body 1>.

If that fails, try<body 2>.

2. Automatic backtracking

```
q(a,b).
q(a,c).
q(a,d).
r(c).
r(d).
p(X) :- q(X,Y), r(Y).
```

Call p(a). Assign $Y = b$ then fails.

Reassign $Y = c$ then succeeds.

Can tell Prolog to find another proof.

Reassign $Y = d$ then succeeds.

Try again $\leadsto$ failure.

3. <u>Non-determinism</u> (in principle)

   We can execute a goal in more than one way.

```
p(X) :- <body 1>.
p(X) :- <body 2>.
```

   Which body will be executed?

   (Prolog's use of DFS makes this deterministic.)

4. <u>Variable use</u>

   - variables can be given values <u>before</u> proc. call or <u>by</u> proc. call
   - variables need not have or be given values at all, but can simply be <u>constrained</u> with respect to other variables via unification.
   - can <u>defer decisions</u> about particular values that variables must have as long as possible $\longrightarrow$ major strength of Prolog.