Unification

Two atomic formulae, A and B, with distinct variables, unify iff they can be made identical by replacing their variables by other terms.

- `loves(bob,Y)` unifies with `loves(bob,sue)` by replacing `Y` by `sue`.
- `loves(bob,Y)` unifies with `loves(X,santa)` by replacing `Y` by `santa` and `X` by `bob`. Both formulas become

  `loves(bob,santa)`

Formally, we used a *substitution* {Y \ santa, X \ bob} which is called a *unifier* of `loves(bob,Y)` and `loves(X,santa)`.

- Note that `loves(bob,X)` does *not* unify with `loves(tony,Y)`: no substitution for `X,Y` can make the two formulae syntactically equal.

More examples

`a(X,X)` unifies with `a(b,b)` and `a(c,c)` but *not* with `a(b,c)` because each occurrence of `X` must be replaced by the `same` term.

In `love(sue,tony)`, `sue` and `tony` are *constant symbols* (very simple data structures). We can construct more complex data structures using *function terms*, e.g.

`owns(john, car(blue,corvette))`

Database:

```
owns(john, car(red,corvette))
owns(john, cat(black,siamese,sylverser))
owns(elvis,copyright(song,"jailhouse rock"))
owns(tolstoy,copyright(book,"war and peace"))
owns(elvis,car(red,cadillac))
```


Yet More Examples

Queries:

- "retrieve everything that john owns"

  `owns(john,X)`

  answers: `X = car(red,corvette)`
  `X = cat(black,siamese,sylverser)`

- "retrieve the color and make of john's car"

  `owns(john,car(Color,Model))`

  answer: `Color = red`
  `Make = corvette`

- "who owns the copyright on the song "jailhouse rock"?"

  `owns(Who,copyright(song,"jailhouse rock"))`

  answers: `Who = elvis`

Examples (Cont'd)

Note that variables are *existentially* quantified:

$\exists$`Make, owns(Who,car(red,Make))`

answer: `Who = john`
`Who = elvis`

`Make` is *bound* to the quantifier, whereas `Who` is free. These are included in answers to queries.

- "Who owns a copyright?"

  $\exists$`X,Y owns(Who,copyright(X,Y))`

  answers: `Who = elvis`
  `Who = tolstoy`

In Prolog, existentially quantified variable is denoted by `_`:

`owns(Who, _)` means $\exists$`X, owns(Who,X)`

`owns(Who, copyright(_,_))` means
$\exists$`X,Y owns(Who,copyright(X,Y))`

Examples (Cont'd)

`owns(Who,car(red,Model))` unifies with `owns(elvis, car(red,cadillac))` using substitution

{Who \ elvis, Model \ cadillac}.

`owns(Who,car(red,Model))` also unifies with `owns(john, car(red,corvette))` using substitution

{Who \ john, Model \ corvette}.

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:

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

Negative Examples:

$\{f(X)\backslash a\}$
$\{X\backslash a,\ X\backslash b\}$
$\{X\backslash f(X)\}$
$\{X\backslash f(Y),\ Y\backslash g(q)\}$

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:

$X\{X\backslash a\} \quad =$
$f(X)\{X\backslash a\} \quad =$

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

$t\{X\backslash b\} \quad =$
$t\{X\backslash b,\ Y\backslash f(Z)\} \quad =$
$t\{X\backslash Z,\ Y\backslash f(Z,a)\} \quad =$
$t\{W\backslash Z\} \quad =$

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,

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

Also,

$\{X\backslash a\} \circ \{Y\backslash b\} \quad =$
$\{X\backslash Z\} \circ \{Z\backslash a\} \quad =$
$\{X\backslash a\} \circ \{X\backslash b\} \quad =$

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 unifyable.

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

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

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

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

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 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.

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.

$\langle$ *Horn clause* $\rangle$ ::= $\langle$ *term* $\rangle$.
  | $\langle$ *term* $\rangle$: $-$ $\langle$ *term* $\rangle$ {, $\langle$ *term* $\rangle$ }.

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)`

A Prolog Session

```
eddie% xsb
 XSB Version 1.7.2 (7/10/97)
 [Solaris, optimal mode]
| ? - [user].  % enter consultation mode
 [Compiling user]
loves(tom,X) :- famous(X),handsome(X).
famous(marilyn-monroe).
famous(cindy-crawford).
handsome(cindy-crawford).
handsome(marilyn-monroe).
<CTRL-D>
 [user compiled, cpu time used: 0.0890 seconds]
 [user loaded]
 yes
| ? - loves(tom, Who).
 Who = marilyn-monroe <ret>
 yes % there are more answers
| ? -loves(tom, Who).
 Who = marilyn-monroe ; <ret>
 Who = cindy-crawford ; <ret>
 no % there are no more answers
| ? - loves(tom,marilyn-monroe).
 yes  % the answer is yes
| ? - loves(tom,marsha-chechik).
 no   % oh well...
| ? - consult('file.pl');
   % add contents of file.pl to the Prolog KB
| ? - halt. % exit Prolog
End XSB (cputime 0.23 secs, elapsetime 475.89 secs)
eddie%
```


Negation in Prolog

• Prolog uses negation as failure, i.e. if something cannot be proved true then it is assumed to be false.
• This is NOT logical negation but easy to implement.
• Negation may appear only in queries and in rule bodies.

Eg. a $\leftarrow$ b $\wedge$ $\sim$c
is written as
  `a :- b, not c.`
meaning "infer a if b can be inferred and c *cannot be inferred.*

Example

```
loves(bill,X) :- handsome(X), not loves(tom,X).
```

(Bill loves anyone handsome unless Tom loves her.)

```
loves(tom,X) :- famous(X), not dead(X).
```

(Tom loves anyone who is famous, unless she is dead.)

```
famous(martha-stewart).
famous(cindy-crawford).
famous(marilyn-monroe).
dead(marilyn-monroe).
handsome(cindy-crawford).
handsome(marilyn-monroe).
handsome(marsha-chechik).
| ?- loves(tom,X).
  X = cindy-crawford;
  X = martha-stewart;
| ?- loves(bill,X).
  X = marilyn-monroe;
  X = marsha-chechik;
  no
```


Safety

Consider (*) `hates(tom,X) :- not loves(tom,X).`

Why is this NOT what we want?

(1) the answer is infinite, i.e., for any person `p` not mentioned in the database, we cannot infer `loves(tom,p)`, so we can infer `hates(tom,p)`. (*) is therefore said to be unsafe.

(2) rule (*) does not require `X` to be a person. Since we cannot infer

```
loves(tom,hammer)
loves(tom,verbs)
loves(tom,green)
loves(tom,abc)
```

so, we can infer that tom hates all these things.

Safety (Cont'd)

To avoid these problems, rules with negation should be guarded:

```
hates(jane,X) :- person(X), handsome(X),
     not loves(tom,X).
```

(tom hates every handsome person whom he does not love.)

Here `person` and `handsome` are ground literals, binding `X` to some values in the database.

A Note on Quantification

$\forall$ X [happy(X) $\leftarrow$ $\forall$ Y loves(Y,X)]

(A person, X, is happy if everyone loves him. Not Horn.)

$\forall$ X [happy(X) $\leftarrow$ $\exists$ Y loves(Y,X)]

(A person, X, is happy if someone, Y, loves him.)

Equivalent to the following Horn rule:

(1) $\forall$X $\forall$Y [happy(X) $\leftarrow$ loves(Y,X)]

Why? (Left as an exercise)

Examples:

loves(bill,marsha) $\Rightarrow$ happy(marsha) {X\marsha, Y\bill}

loves(bill,cindy) $\Rightarrow$ happy(cindy) {X\cindy, Y\bill}

loves(tom,cindy) $\Rightarrow$ happy(cindy) {X\cindy, Y\tom}

So, in Horn logic, existential quantification can appear in the premise of a rule.

Three main ideas in Prolog

- Horn rules (done)
- Unification (done)
- Top-down Inference (next)

Reasoning

• Forward - generate everything you can from facts and rules and use that to answer queries
  - inefficient (generates much more than necessary)
  - might not terminate!

• Backward - start from the query, attempting only those inferences that are relevant to answering the query, e.g.
Let a query `A` and a rule `A` $\leftarrow$ `B` be given. To prove `A`, Prolog tries to prove `B`.

Bottom-up Inference

Given a fact, `B` and a rule, `A` $\leftarrow$ `B`, infer A, e.g.

```
A <- B        (1)
B <- C        (2)
C             (3)
```

infer A

rule (1)

infer B

rule (2)

infer C

rule (3)

start

So, A is proved

Top-Down Inference

Given a query, `A`, and a rule, `A` $\leftarrow$ `B`, try to prove `B`, e.g.

```
A <- B        (1)
B <- C        (2)
C             (3)
```

goal A

rule (1)

goal B

rule (2)

goal C

rule (3)

success

So, A is proved

Notes on Inference

- Prolog uses top-down inference, although some other logic programming systems use bottom-up inference.
- Each has its own advantages and disadvantages:
  - Bottom-up may generate a huge number of new facts
  - Top-down may fail
- Top-down and bottom-up are <u>equivalent</u>. So, the choice is an implementation detail

Top-Down vs Bottom-up

Bottom-up inference derives <u>many</u> more facts.

Example 1:

```
p(X,Y,Z) <- q(X),q(Y),q(Z).
q(a1).
q(a).
...
q(an).
```

Bottom-up inference derives n^3 facts of the form p($\mathtt{a}_i$,$\mathtt{a}_j$,$\mathtt{a}_k$):

```
p(a1, a1, a1)
p(a1, a1, a2)
p(a1, a2, a3)
...
```


Top-down vs bottom-up (Cont'd)

Bottom-up inference derives <u>infinitely</u> many facts.

Example 2:

```
p(f(x)) <- p(x)

p(a)
p(f(a))
p(f(f(a)))
p(f(f(f(a))))
...
```

In contast, top-down inference derives only the facts requested by the user, e.g.

```
who does jane love?
what is john's telephone number?
```


Top-down vs bottom-up (Cont'd)

But... top-down inference may fail.
Example 3:

```
A <- B   (1)
B <- C   (2)
A        (3)
```

goal A

↓ rule (1)

goal B

↓ rule (2)

goal C

↓ rule (3)

fail
(no rules infer C)

So, A is not proved

Dealing with Multiple Rules and Premises

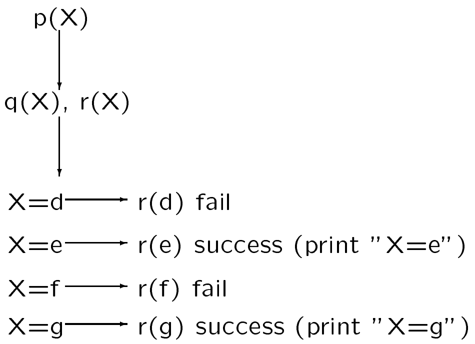
Example:

```
E <- B
E <- C
E <- D
E <- B /\ C /\ D
```

In top-down inference,
- rules with multiple premises give rise to an inference <u>tree</u>
- multiple rules that infer the same fact give rise to <u>backtracking</u>.

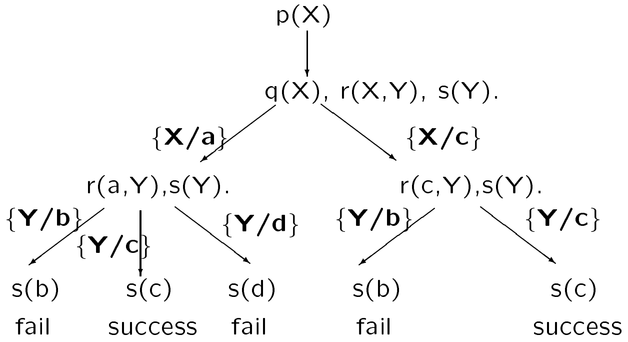
Inference Trees - Example 1

```
p(X) := q(X), r(X).
q(d)     q(f)     r(e)
q(e)     q(g)     r(g)
p(X) % query "find X s.t. p(X) is true"
```



Inference Trees - Example 2

```
p(X) := q(X), r(X,Y), s(Y).
q(a)     r(a,b)     r(a,d)   r(c,c)
q(c)     r(a,c)     r(c,b)   s(c).
Query p(X) %"find X s.t. p(X) is true"
```



Hints on Analyzing Output

We can follow execution of Prolog with `write` statements, e.g.

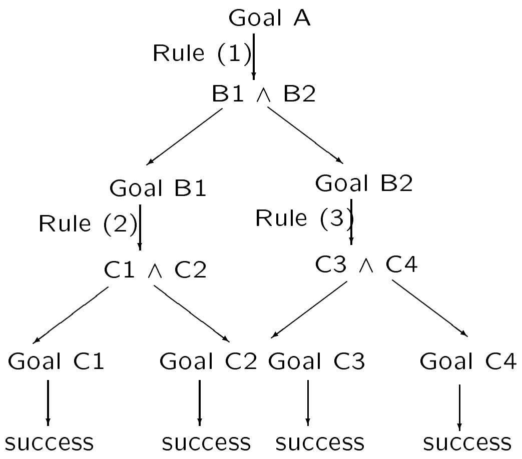
```
a(X) :- b(X), write(X), c(X).
b(a)    b(c)  b(e)
c(a)    b(d)
b(b)    c(d)
Query: a(x).
```

Then Prolog prints

```
a
X = a
bcd
X = d
e
no
```

Inference Trees - Example 3

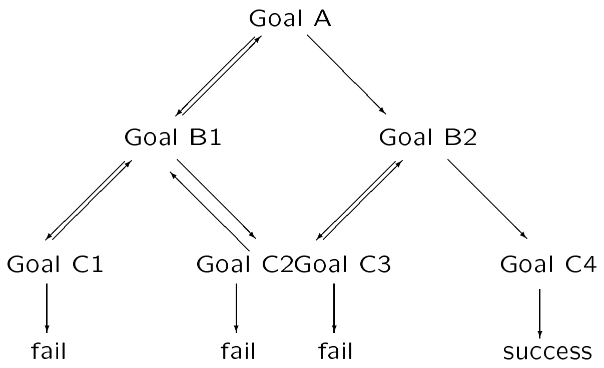
```
(1)  A  <- B1 /\ B2  (4) C1
(2)  B1 <- C1 /\ C2  (5) C2
(3)  B2 <- C3 /\ C3  (6) C3
(7)  C4          A  % query
```



So, goal A is proved. (all paths must succeed)

Inference Trees - Example 4

```
(1)  A  <- B1     (4)  B1 <- C2
(2)  A  <- B2     (5)  B2 <- C3
(3)  B1 <- C1     (6)  B2 <- C4
(7)  C4            A  % query
```



So, goal A is proved. (only one path must succeed)

Recursion in Prolog

If a predicate symbol occurs <u>both</u> in the head and body of a rule, then the rule is *recursive*, e.g.

`a(X) :- b(X,Y),a(Y).`,

where we want to prove `a(X)`.

The predicate `a` acts like a <u>recursive subroutine</u> and is called <u>recursively defined predicate</u>.

Recursion might be indirect, involving several rules:

```
a(X) :- b(X,Y), c(Y).
c(Y) :- d(Y,Z), a(Z).
```

Thus, to prove `a(x)`

Prolog tries to prove `c(Y)` (and `b(X,Y)`)
so it tries to prove `a(Z)` (and `d(Y,Z)`)
i.e., to prove `a(X)`, Prolog tries to prove `a(Z)`.

Predicates `a` and `c` are said to be <u>mutually recursive</u>.

Recursion (Cont'd)

Recursion can be non-linear:

`a(X) :- b(X,Y),a(Y),c(Y,Z),a(Z).`

So, to prove `a(x)`, Prolog tries to prove both `a(Y)` and `a(Z)`.

This generates a <u>recursive proof tree</u>.