Introduction to Prolog

Outline

- Elements of Prolog
- Encoding a programming language
- Program transformation techniques

Prolog Terms

- **Atoms:** `prolog`, `cs4212`, `'abc def'`, `'X'`, `+++`, `'a+b'`
- **Integers:** `10`, `-2`
- **Floating point numbers:** `1.0`, `3.14159`, `-2.71e10`
- **Variables:** `X`, `Prolog`, `AnythingCapitalized`
- **Compound terms:**

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```

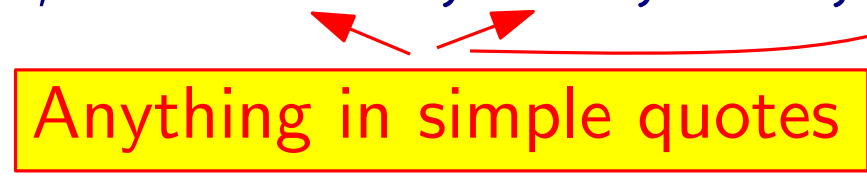
Prolog Terms

- **Atoms:** `prolog`, `cs4212`, `'abc def'`, `'X'`, `+++`, `'a+b'`
- **Integers:** `10`, `-2`
- **Floating point numbers:** `1.0`, `3.14159`, `-2.71e10`
- **Variables:** `X`, `Prolog`, `AnythingCapitalized`
- **Compound terms:**

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```

Prolog Terms

- **Atoms:** `prolog`, `cs4212`, `'abc def'`, `'X'`, `+++`, `'a+b'`



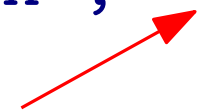
- **Integers:** `10`, `-2`
- **Floating point numbers:** `1.0`, `3.14159`, `-2.71e10`
- **Variables:** `X`, `Prolog`, `AnythingCapitalized`
- **Compound terms:**

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```

Prolog Terms

- **Atoms:** `prolog`, `cs4212`, `'abc def'`, `'X'`, `+++`, `'a+b'`
- **Integers:** `10`, `-2`
- **Floating point numbers:** `1.0`, `3.14159`, `-2.71e10`
- **Variables:** `X`, `Prolog`, `AnythingCapitalized`
- **Compound terms:**

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```



Sequences of special characters

Prolog Terms

- **Atoms:** `prolog`, `cs4212`, `'abc def'`, `'X'`, `+++`, `'a+b'`
- **Integers:** `10`, `-2`
- **Floating point numbers:** `1.0`, `3.14159`, `-2.71e10`

As usual

- **Variables:** `X`, `Prolog`, `AnythingCapitalized`
- **Compound terms:**

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```

Prolog Terms

- **Atoms:** `prolog`, `cs4212`, `'abc def'`, `'X'`, `+++`, `'a+b'`
- **Integers:** `10`, `-2`
- **Floating point numbers:** `1.0`, `3.14159`, `-2.71e10`
- **Variables:** `X`, `Prolog`, `AnythingCapitalized`

Capitalized identifiers are variables.

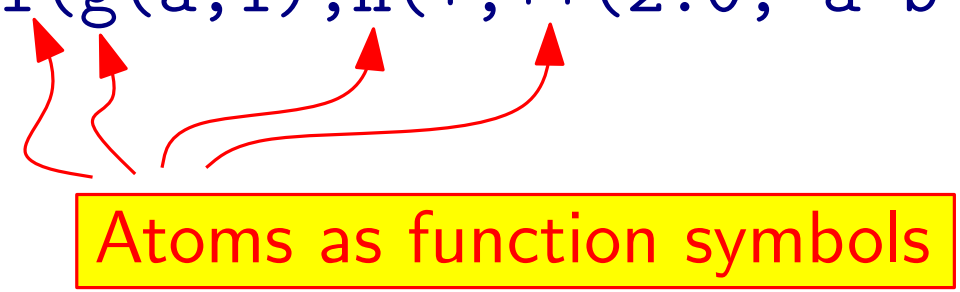
- **Compound terms:**

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```

Prolog Terms

- **Atoms:** `prolog`, `cs4212`, `'abc def'`, `'X'`, `+++`, `'a+b'`
- **Integers:** `10`, `-2`
- **Floating point numbers:** `1.0`, `3.14159`, `-2.71e10`
- **Variables:** `X`, `Prolog`, `AnythingCapitalized`
- **Compound terms:**

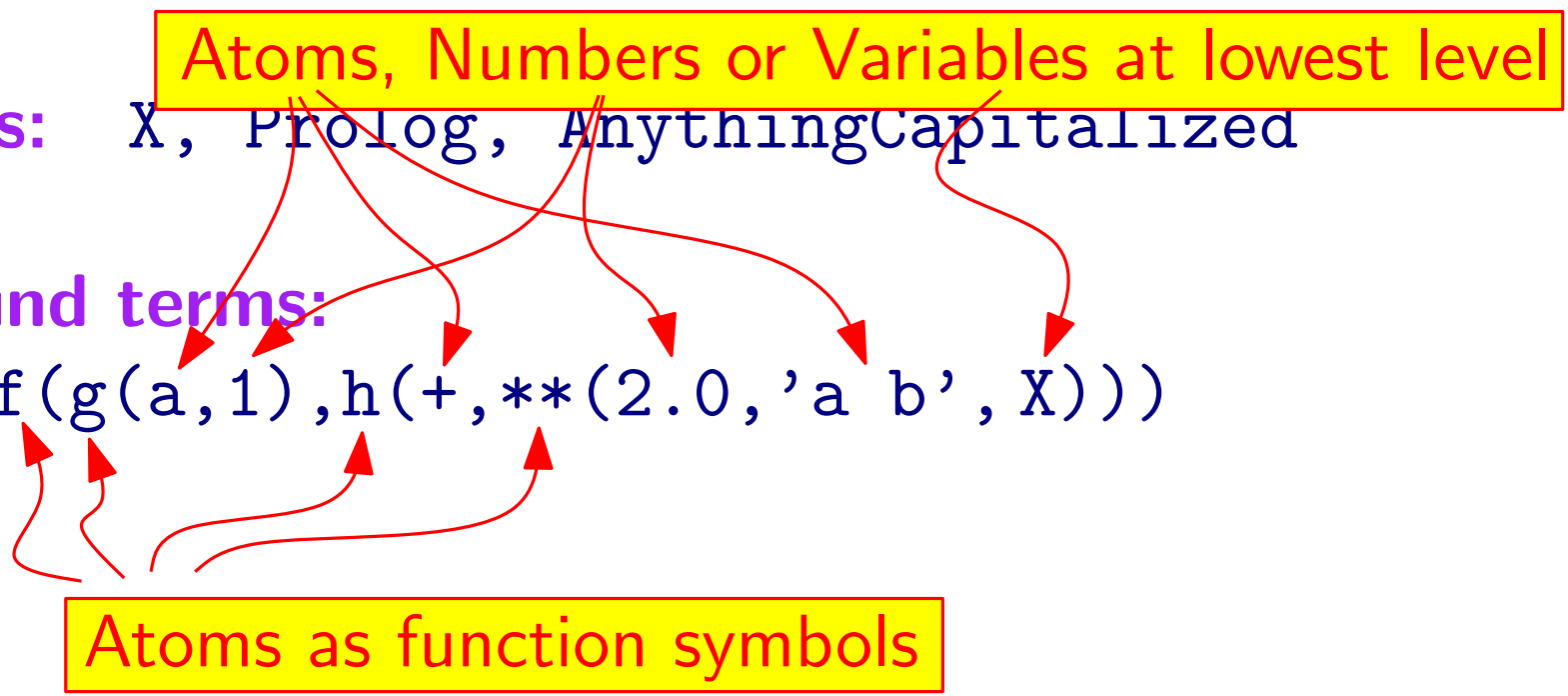
```
f(g(a,1),h(+,**(2.0,'a b',X)))
```



Prolog Terms

- **Atoms:** `prolog`, `cs4212`, `'abc def'`, `'X'`, `+++`, `'a+b'`
- **Integers:** `10`, `-2`
- **Floating point numbers:** `1.0`, `3.14159`, `-2.71e10`
- **Variables:** `X`, `Prolog`, `AnythingCapitalized`
- **Compound terms:**

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```



Terms simulate the syntax of mathematical functional notation.

Prolog Terms

- **Atoms:** `prolog`, `cs4212`, `'abc def'`, `'X'`, `+++`, `'a+b'`
- **Integers:** `10`, `-2`
- **Floating point numbers:** `1.0`, `3.14159`, `-2.71e10`
- **Variables:** `X`, `Prolog`, `AnythingCapitalized`
- **Compound terms:**

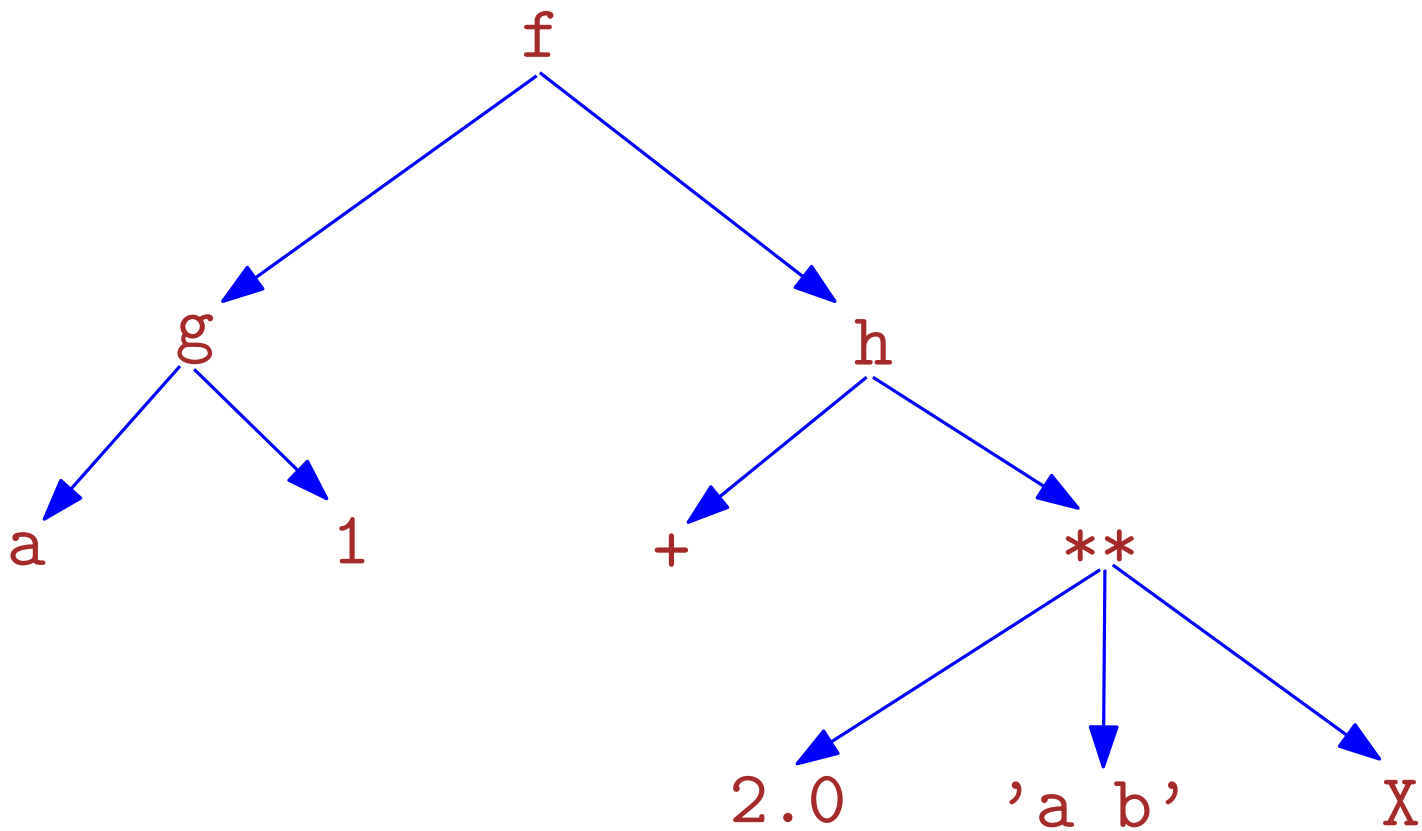
Similar to structures in C

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```

Terms are the way data is encoded in Prolog; they can potentially get very large.

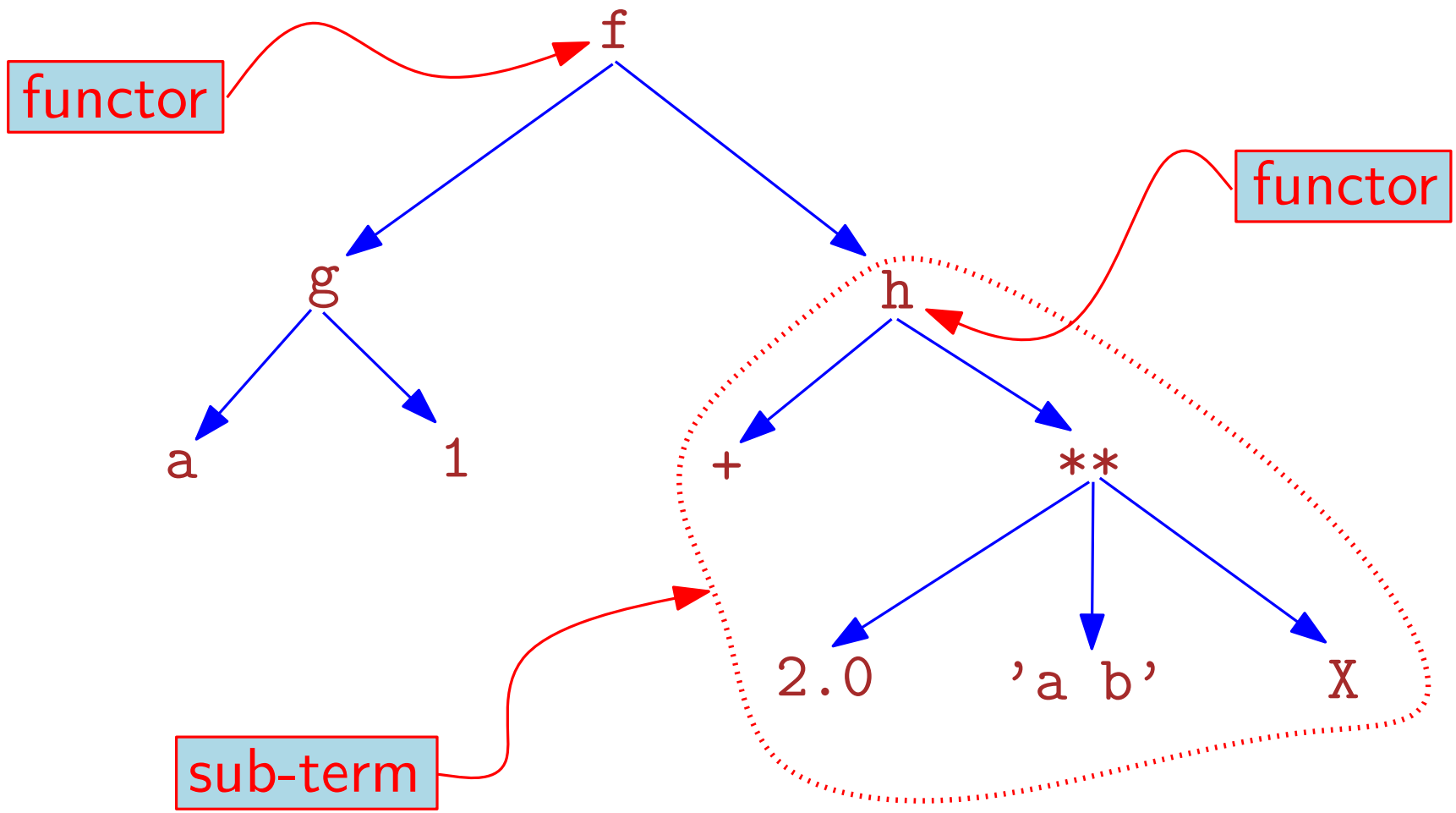
Tree Representation of Terms

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```



Tree Representation of Terms

```
f(g(a,1),h(+,**(2.0,'a b',X)))
```



Prolog Queries

- Interpreter presents prompt

  `?-`

- User may type in a *query* (also called *goal*)

  → succeeds or fails

- Simplest query: *unification request*

  → equality of two terms

Unification Requests

```
?- a = a.
```

```
?- a = b.
```

```
?- f(a,b) = f(a,b).
```

```
?- f(a,c) = f(a,b).
```

```
?- f(a,b) = g(a,b).
```

```
?- f(a,b) = f(a).
```

arity: number of arguments of a function

```
?- f(a) = f.
```

an atom is a function with arity 0.

Unification Requests

```
?- a = a.
```
success

```
?- a = b.
```

```
?- f(a,b) = f(a,b).
```

```
?- f(a,c) = f(a,b).
```

```
?- f(a,b) = g(a,b).
```

```
?- f(a,b) = f(a).
```
arity: number of arguments of a function

```
?- f(a) = f.
```
an atom is a function with arity 0.

Unification Requests

```
?- a = a.
```
success

```
?- a = b.
```
failure

```
?- f(a,b) = f(a,b).
```

```
?- f(a,c) = f(a,b).
```

```
?- f(a,b) = g(a,b).
```

```
?- f(a,b) = f(a).
```
arity: number of arguments of a function

```
?- f(a) = f.
```
an atom is a function with arity 0.

Unification Requests

```
?- a = a.
```
success

```
?- a = b.
```
failure

```
?- f(a,b) = f(a,b).
```
success

```
?- f(a,c) = f(a,b).
```

```
?- f(a,b) = g(a,b).
```

```
?- f(a,b) = f(a).
```
arity: number of arguments of a function

```
?- f(a) = f.
```
an atom is a function with arity 0.

Unification Requests

```
?- a = a.
```
success

```
?- a = b.
```
failure

```
?- f(a,b) = f(a,b).
```
success

```
?- f(a,c) = f(a,b).
```
failure

```
?- f(a,b) = g(a,b).
```

```
?- f(a,b) = f(a).
```
arity: number of arguments of a function

```
?- f(a) = f.
```
an atom is a function with arity 0.

Unification Requests

```
?- a = a.
```
success

```
?- a = b.
```
failure

```
?- f(a,b) = f(a,b).
```
success

```
?- f(a,c) = f(a,b).
```
failure

```
?- f(a,b) = g(a,b).
```
failure

```
?- f(a,b) = f(a).
```
arity: number of arguments of a function

```
?- f(a) = f.
```
an atom is a function with arity 0.

Unification Requests

```
?- a = a.
```
success

```
?- a = b.
```
failure

```
?- f(a,b) = f(a,b).
```
success

```
?- f(a,c) = f(a,b).
```
failure

```
?- f(a,b) = g(a,b).
```
failure

```
?- f(a,b) = f(a).
```
failure

arity: number of arguments of a function

```
?- f(a) = f.
```
an atom is a function with arity 0.

must be the same for success

Unification Requests

```
?- a = a.
```
success

```
?- a = b.
```
failure

```
?- f(a,b) = f(a,b).
```
success

```
?- f(a,c) = f(a,b).
```
failure

```
?- f(a,b) = g(a,b).
```
failure

```
?- f(a,b) = f(a).
```
failure

arity: number of arguments of a function

```
?- f(a) = f.
```
failure

an atom is a function with arity 0.

must be the same for success

Unification as Computation

- Unification requests may contain variables.
- The system computes values for variables sot that terms become unifiable

  $\rightarrow$Not always possible

- Once a variable is bound to a value, that value does not change

  $\rightarrow$Single assignment variables

Unification of Terms with Variables

```
?- f(X,b) = f(a,Y).
```

succeeds with `X=a` and `Y=b`.

```
?- f(X,X) = f(a,b).
```

fails.

```
?- f(X,X) = f(a,a).
```

succeeds with `X=a`.

```
?- f(g(a,X),X) = f(Y,b).
```

succeeds with `Y=g(a,b)` and `X=b`.

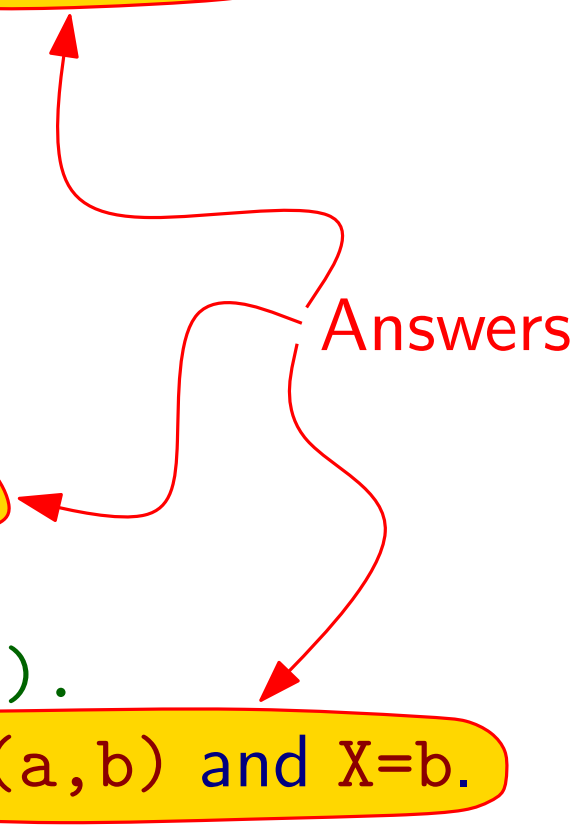
Unification of Terms with Variables

```
?- f(X,b) = f(a,Y).
```
succeeds with `X=a and Y=b.`

```
?- f(X,X) = f(a,b).
```
fails.

```
?- f(X,X) = f(a,a).
```
succeeds with `X=a.`

```
?- f(g(a,X),X) = f(Y,b).
```
succeeds with `Y=g(a,b) and X=b.`



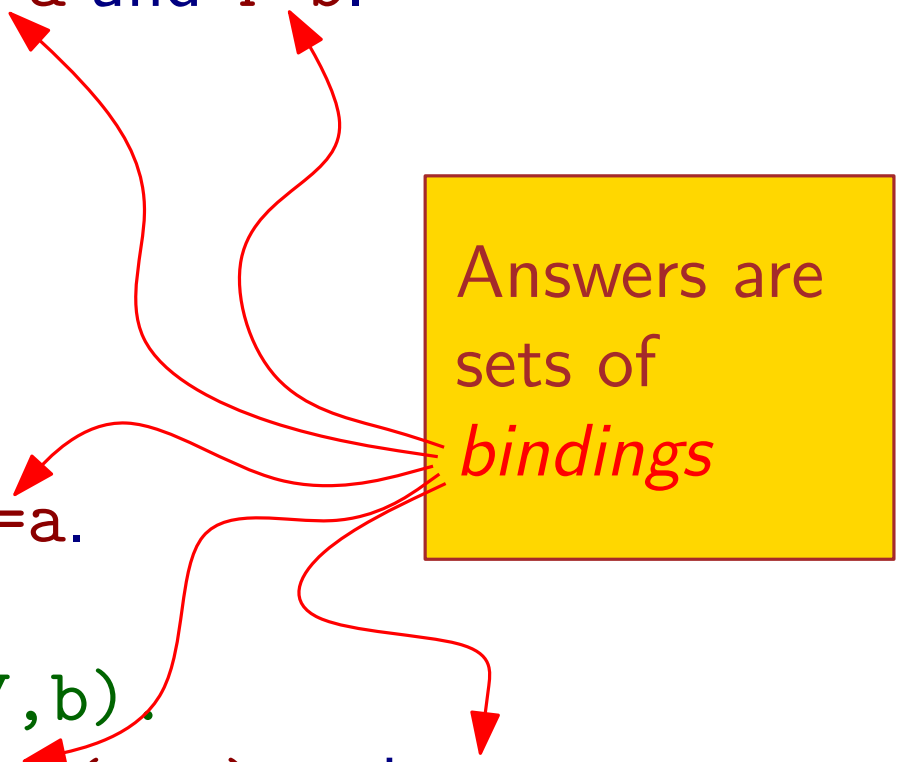
Unification of Terms with Variables

```
?- f(X,b) = f(a,Y).
```
succeeds with X=a and Y=b.

```
?- f(X,X) = f(a,b).
```
fails.

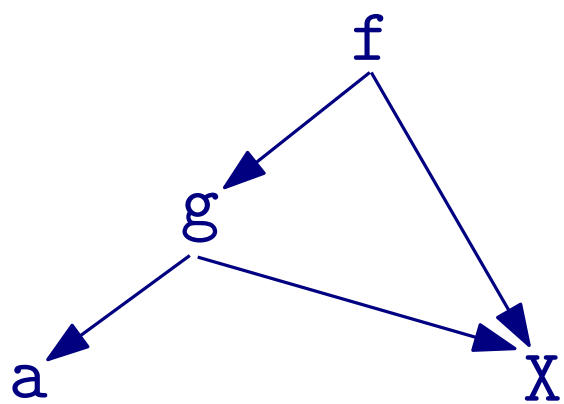
```
?- f(X,X) = f(a,a).
```
succeeds with X=a.

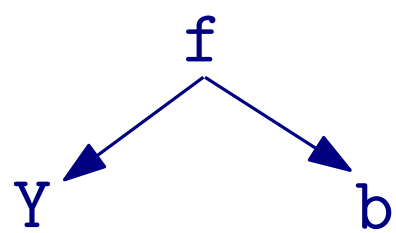
```
?- f(g(a,X),X) = f(Y,b).
```
succeeds with Y=g(a,b) and X=b.



Tree Representation of Unification

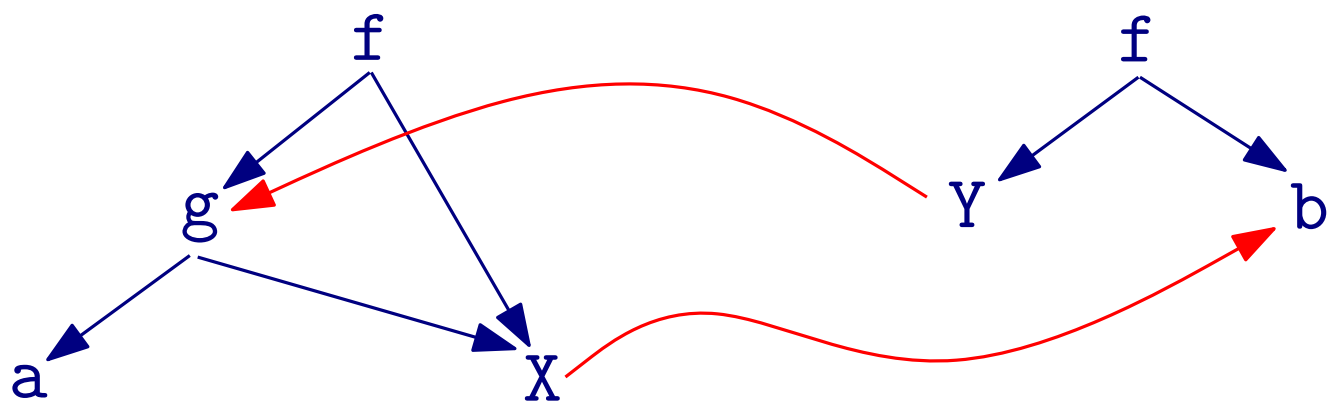
```
?- f(g(a,X),X) = f(Y,b).
```





Tree Representation of Unification

```
?- f(g(a,X),X) = f(Y,b).
```



Multiple Unifications

```
?- X = a, Y = b.
```

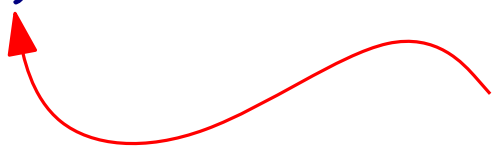
```
?- X=f(Y,Y), Y=a.
```

```
?- f(X,a) = f(b,Y), g(Y,b)=g(a,X).
```

```
?- f(X1,X2,X3) = f(g(X2,X2),g(X3,X3),g(a,b)).
```

Multiple Unifications

```
?- X = a, Y = b.
```



Conjunction

If one unification request fails, the whole query fails.

```
?- X=f(Y,Y), Y=a.
```

```
?- f(X,a) = f(b,Y), g(Y,b)=g(a,X).
```

```
?- f(X1,X2,X3) = f(g(X2,X2),g(X3,X3),g(a,b)).
```

Unification Algorithm (no vars)

(0) Initial unification request: $\Sigma_1 = \Pi_1, \Sigma_2 = \Pi_2, \ldots$

(1) If $functor(\Sigma_1) \neq functor(\Pi_1)$ or $arity(\Sigma_1) \neq arity(\Pi_1)$ stop with failure.

(2) If $arity(\Sigma_1) = 0$, remove $\Sigma_1 = \Pi_1$ from current unification request and go to last step.

(3) Denote by $\Sigma_{11}, \Sigma_{12}, \ldots, \Sigma_{1k}$ the arguments of Σ_1, and by $\Pi_{11}, \Pi_{12}, \ldots, \Pi_{1k}$ the arguments of Π_1.

(4) Set the new unification request to
$$\Sigma_{11} = \Pi_{11}, \Sigma_{12} = \Pi_{12}, \ldots, \Sigma_{1k} = \Pi_{1k}, \Sigma_2 = \Pi_2, \ldots$$

(5) If current unification request is not empty, go to the first step, otherwise terminate with success.

Unification Algorithm (no vars)

(0) Initial unification request: $\Sigma_1 = \Pi_1, \Sigma_2 = \Pi_2, \ldots$

(1) If $functor(\Sigma_1) \neq functor(\Pi_1)$ or $arity(\Sigma_1) \neq arity(\Pi_1)$ stop with failure.

(2) I
r

(3) D
I

Remember, terms can potentially get very large, so an algorithm is in fact required.

(4) Set the new unification request to
$$\Sigma_{11} = \Pi_{11}, \Sigma_{12} = \Pi_{12}, \ldots, \Sigma_{1k} = \Pi_{1k}, \Sigma_2 = \Pi_2, \ldots$$

(5) If current unification request is not empty, go to the first step, otherwise terminate with success.

Example

- `f(a,g(b,c)) = f(a,g(b,c)), h(x,y)=h(x,y).`
- `a=a, g(b,c)=g(b,c), h(x,y)=h(x,y).`
- `g(b,c)=g(b,c), h(x,y)=h(x,y).`
- `b=b, c=c, h(x,y)=h(x,y).`
- `c=c, h(x,y)=h(x,y).`
- `h(x,y)=h(x,y).`
- `x=x, y=y`
- `y=y`
- empty: success

Unification Algorithm for Terms with Vars

(0) Initial unification request: $\Sigma_1 = \Pi_1, \Sigma_2 = \Pi_2, \ldots$; answer is empty

(1) If Σ_1 is a variable, add $\Sigma_1 = \Pi_1$ to the answer, and replace Σ_1 by Π_1 everywhere in the query and rhs of bindings in answer. Remove $\Sigma_1 = \Pi_1$ from current unification request, and go to last step.

(2) If Π_1 is a variable, add $\Pi_1 = \Sigma_1$ to the answer, and replace Π_1 by Σ_1 everywhere in the query and rhs of bindings in answer. Remove $\Sigma_1 = \Pi_1$ from current unification request, and go to last step.

(3) If $functor(\Sigma_1) \neq functor(\Pi_1)$ or $arity(\Sigma_1) \neq arity(\Pi_1)$ stop with failure.

(4) If $arity(\Sigma_1) = 0$, remove $\Sigma_1 = \Pi_1$ from current unification request and go to last step.

(5) Denote by $\Sigma_{11}, \Sigma_{12}, \ldots, \Sigma_{1k}$ the arguments of Σ_1, and by $\Pi_{11}, \Pi_{12}, \ldots, \Pi_{1k}$ the arguments of Π_1.

(6) Set the new unification request to
$$\Sigma_{11} = \Pi_{11}, \Sigma_{12} = \Pi_{12}, \ldots, \Sigma_{1k} = \Pi_{1k}, \Sigma_2 = \Pi_2, \ldots$$

(7) If current unification request is not empty, go to the first step, otherwise terminate with success.

Example

- `f(X1,g(X3,X3),X3)=f(g(X2,X2),X2,g(a,b))`
  answer: empty

- `X1=g(X2,X2),g(X3,X3)=X2,X3=g(a,b)`
  answer: empty

- `g(X3,X3)=X2,X3=g(a,b)`
  answer: {X1=g(X2,X2)}

- `X3=g(a,b)`
  answer: {X1=g(g(X3,X3),g(X3,X3)),X2=g(X3,X3)}

- empty
  answer: {X1=g(g(g(a,b),g(a,b)),g(g(a,b),g(a,b))),
  X2=g(g(a,b),g(a,b)), X3=g(a,b)}

Example

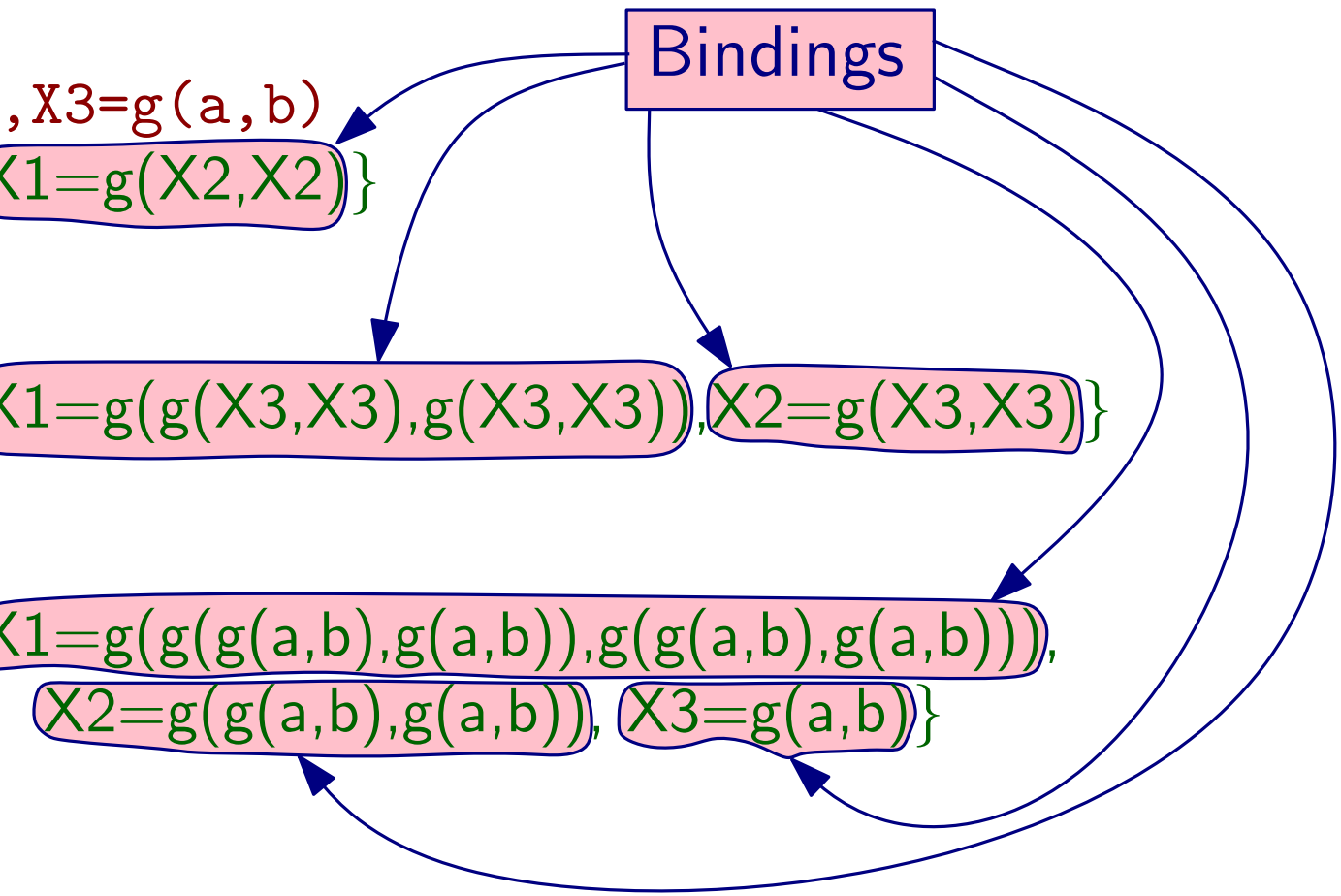
- `f(X1,g(X3,X3),X3)=f(g(X2,X2),X2,g(a,b))`
  answer: empty

- `X1=g(X2,X2),g(X3,X3)=X2,X3=g(a,b)`
  answer: empty

- `g(X3,X3)=X2,X3=g(a,b)`
  answer: {X1=g(X2,X2)}

- `X3=g(a,b)`
  answer: {X1=g(g(X3,X3),g(X3,X3)), X2=g(X3,X3)}

- empty
  answer: {X1=g(g(g(a,b),g(a,b)),g(g(a,b),g(a,b))),
  X2=g(g(a,b),g(a,b)), X3=g(a,b)}



Facts

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Facts

Type the following into file `example.pl`

Predicate symbol

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Facts

Ground terms (no vars, but functors allowed).

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Facts

Typical extension of Prolog programs

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Facts act like a database, and define a relation. They do not contain variables.

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Facts

Prolog interactive prompt.

Typed by user

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Queries:

```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(james,X).
false
```

Facts act like a database, and define a relation. They do not contain variables.

Facts

Multiple answers for one query

Type the following into file `example.pl`

```
parent(john,george).
parent(john,mary).
parent(george,adam).
parent(george,beth).
parent(adam,james).
```

Answers (interpreter output)

Queries:

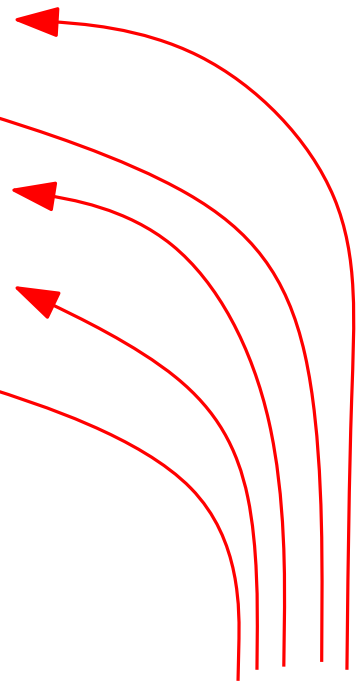
```
?- consult(example).

?- parent(george,adam).
true.

?- parent(john,X).
X = george ;
X = mary

?- parent(
false
```

Typed by user to move on to next answer

Facts act like a database, and define a relation. They do not contain variables.

Queries

```
?- parent(X,Y).
 ◇ X = john   Y = george ;
 ◇ X = john   Y = mary ;
 ◇ X = george   Y= adam   ;
 ◇ X = george   Y = beth ;
 ◇ X = adam   Y= james ;
 ◇ false
```



Typed by user as request to continue search for more solutions

Rules

Add to example.pl

```
ancestor(X,Y) :- parent(X,Y).

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Rules

Read as "if"

Add to example.pl

```
ancestor(X,Y) :- parent(X,Y).

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Conjunction

Rules

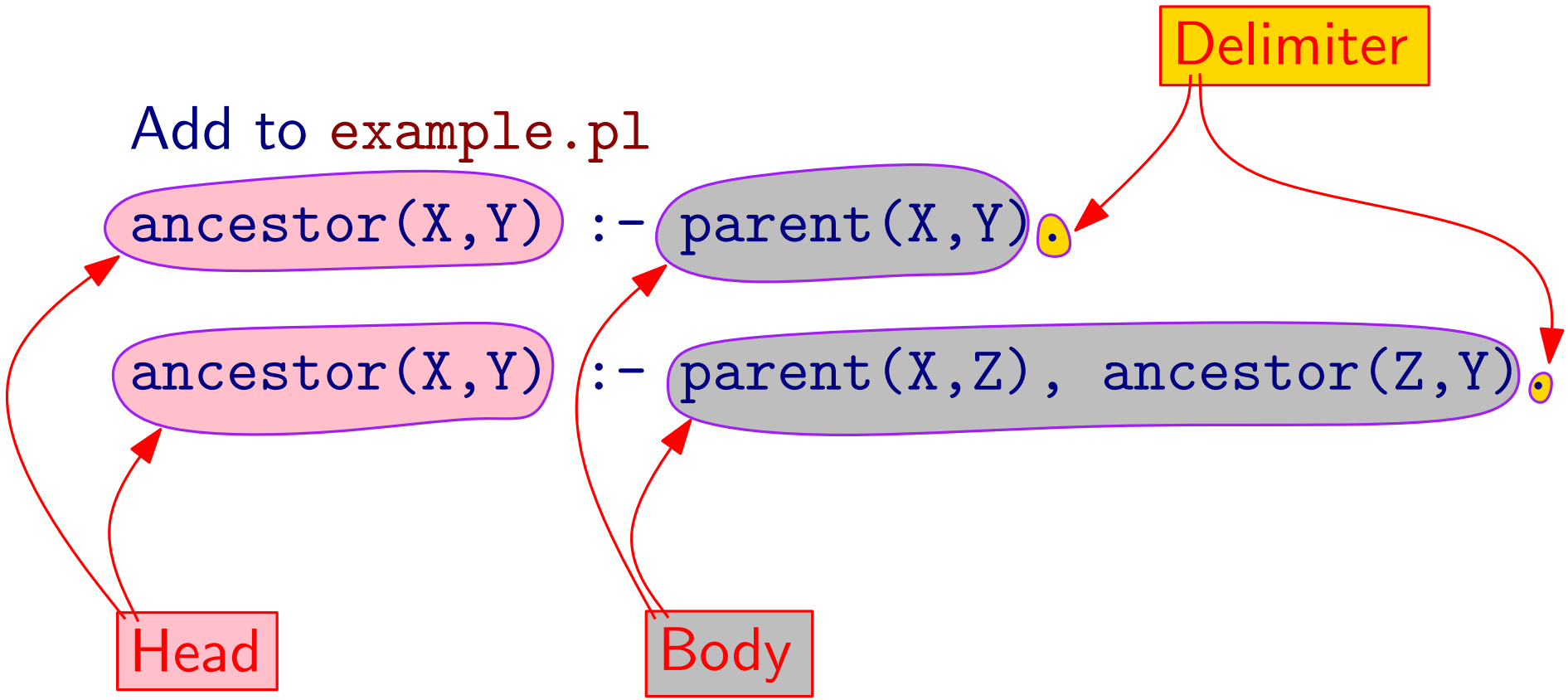
Add to `example.pl`

```
ancestor(X,Y) :- parent(X,Y).
```

```
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

The ancestor of `X` is `Y` **if** there exists a `Z` such that the parent of `X` is `Z`, **and** the ancestor of `Z` is `Y`.

Rules


Delimiter
Add to example.pl
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
Head
Body

More Complex Queries

```
?- ancestor(john,X).

X = george ;

X = mary ;

X = adam ;

X = beth ;

X = james ;

false.
```

Multiple atoms and unification requests can be placed in one query.

```
?- parent(john,X),
   X=Y,parent(Y,adam).

X = george,
Y = george ;

false
```

More Complex Queries

```
?- ancestor(X,Y).
X = john,
Y = george ;
X = john,
Y = mary ;
X = george,
Y = adam ;
X = george,
Y = beth ;
X = adam,
Y = james ;
X = john,
Y = adam ;
X = john,
Y = beth ;
X = john,
Y = james ;
X = george,
Y = james ;
false.
```

Resolution

- *Resolution:* the process of answering a query.
- Introduced via demo (watch video recording)
- Important concept: *variable renaming*.
  - ⇒ all variables in a rule are replaced by completely new variables.

    Original: `ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).`

    1^{st} renaming: `ancestor(X1,Y1) :- parent(X1,Z1), ancestor(Z1,Y1).`

    2^{nd} renaming: `ancestor(X2,Y2) :- parent(X2,Z2), ancestor(Z2,Y2).`

    ...

Resolution Algorithm

(0) Assume a query $A_1, A_2, \ldots, A_n$

(1) Pick of rule from the program and rename its variables. Assume the renamed rule has the form:

$$H \ :- \ B_1, B_2, \ldots, B_k$$

$k = 0?$ $\Rightarrow$ $B_1, B_2, \ldots, B_k$ are misssing

(2) New goal:

$$(H = A_1), B_1, B_2, \ldots, B_k, A_2, \ldots, A_n$$

(3) Variable bindings may be produced by the unification request $(H = A_1)$.
$\Rightarrow$ add them to answer, replace bound variables by their values all over the goal.

(4) Continue from step 1 till query is empty, and then return the answer.

Resolution Algorithm

(0) Assume a query $A_1, A_2, \ldots, A_n$

(1) Pick of rule from the program and rename its variables. Assume the renamed rule has the form:

$$H \ :- \ B_1, B_2, \ldots, B_k$$

$k = 0?$ $\Rightarrow$ $B_1, B_2, \ldots, B_k$ are misssing

(2) New goal:

$$(H = A_1), B_1, B_2, \ldots, B_k, A_2, \ldots, A_n$$

(3) Variable bindings may be produced by the unification request $(H = A_1)$.
$\Rightarrow$ add them to answer, replace bound variables by their values all over the goal.

(4) Continue from step 1 till query is empty, and then return the answer.

Backtracking will be used for exhaustive search

Resolution Demo

What have we learned?

- Current state of computation

$$\langle query, partial\ answer, stack\ of\ choice\ points \rangle$$

- First atom in the query:
  - unification request:
    - succeeds: augment partial answer and move on
    - fail: backtracking
      - inspect top of stack for next rule to try
      - if no more rules to try, pop choice point and repeat, till a "next rule" is detected.
  - other atom
    - create choice point, set first rule in program as "next rule to try"
    - create new query, with unification request at the top
    - repeat till stack of choice points is empty
  - empty: print current answer, and backtrack

The Cut

- Syntax: ! (exclamation mark)
- Alters the search for a solution.
- Makes programs more efficient.
- Removes choice points from the stack.
  - when a cut appears in the current goal, the current stack pointer is attached to the cut.
  - when the cut is executed, the top of the stack is moved right beneath the one attached to the cut.

Impure Prolog

- `atom(X)`: succeeds if, at the time of the evaluation, `X` is bound to an atom
- `var(X)`: succeeds if, at the time of the evaluation, `X` is a free variable (i.e. it does not have a binding yet)
- `integer(X)`: succeeds if X is bound to an integer.
- `Term =..  List` : decomposes a term into its components:

```
f(a,b) =.. List       ====>  List = [f,a,b]
f(g(a,b),c) =.. List  ====>  List = [f,g(a,b),c]
```

- `write(X)` : output the binding of `X`.

Prolog Operators

- Infix and functional notations are equivalent

```
?- +(a,b) = a+b.
true.

?- X = +(a,b).
X = a + b.
```

- Expression pattern matching

```
?- X+Y = a+b+c.
X = a+b
Y = c

?- X+Y = a+b*c.
X = a
Y = b*c

?- (a+b)*(X-Y) = Z*(3-4).
Z = a+b
X = 3
Y = 4
```

The op Declaration

```
?- X = 1 a 3. % error
?- op(100,yfx,a). % declare a as operator
?- X = 1 a 3. % succes
?- a(b,c) = b a c. % succes
?- a(a,a) = a a a. % success
?- +(+,+) = + + + . % success, note the spaces
?- X a Y = 1 a 2 a 3. % X = 1 a 2, Y= 3
?- X a Y = 1 a (2 a 3). % X = 1, Y = 2 a 3
?- X a Y = 1 a 2 a 3 a 4. % X = 1 a 2 a 3, Y = 4
?- X + Y a Z = 1 + 2 a 3. % X=1, Y=2, Z=3
?- 1+2 a 3 = +(1,a(2,3)). % succes
```

Example: `while` language

```
?- op(950,fx,while).
?- op(949,xfx,do).
?- X = (while x>0 do {x=x-1 ; y = y+x } ).
```

succeeds

```
?- (while B do S) = (while x>0 do {x=x-1 ; y = y+x } ).
```

succeeds with `B = x>0` and

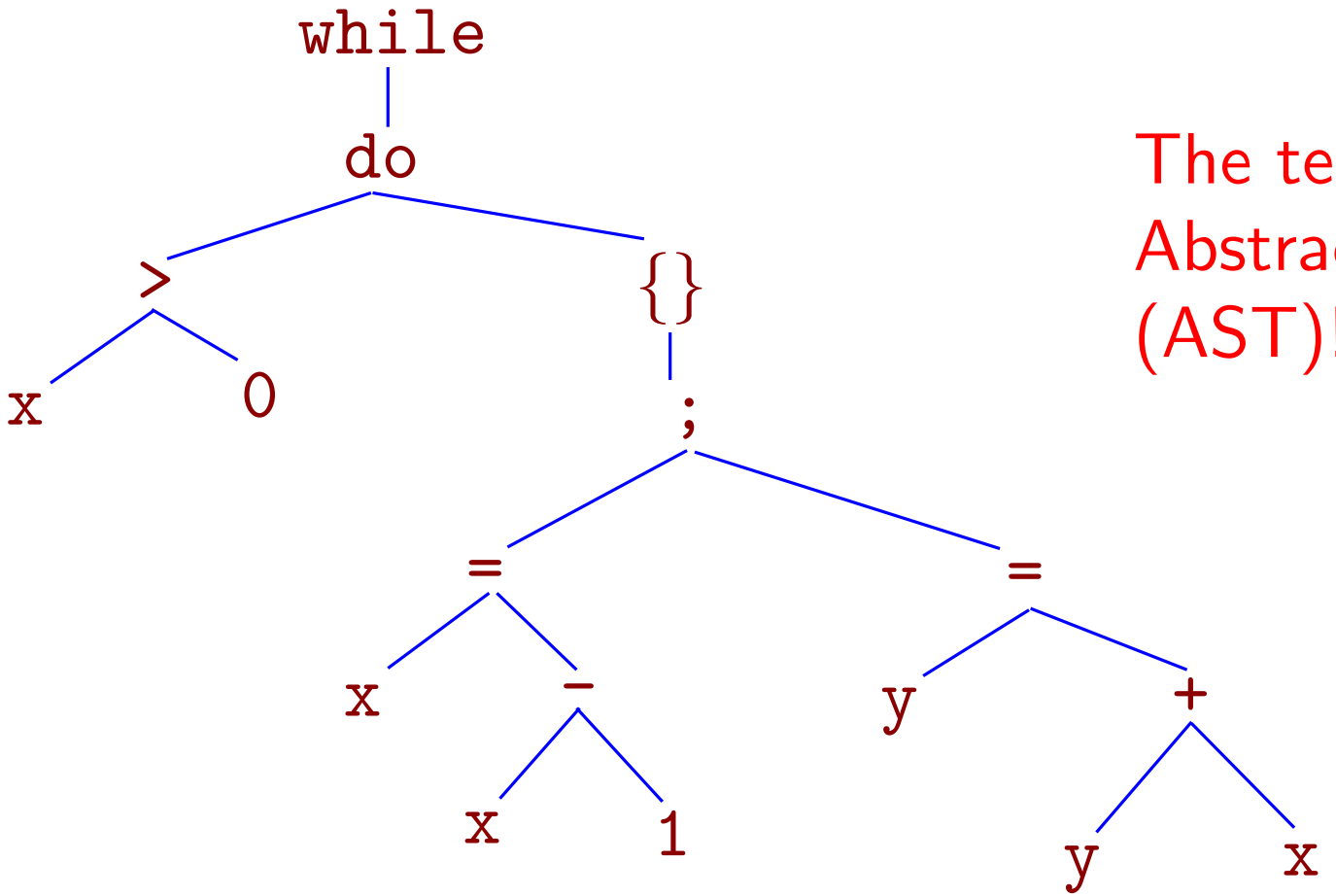
`S = { x=x-1 ; y=y+x}`

```
?- (while B do S) = while(do(B,S)).
```

succeeds

Tree Representation

- `while x > 0 do { x = x-1 ; y = y+x }`
- `while(do(>(x,0),{}(;(=(x,-(x,1)),=(y,+(y,x))))))`



The term is it's own Abstract Syntax Tree (AST)!!!

How `op` Works?

- `?- op(Precedence,Associativity,Symbol).`
- *Precedence:* 1-1200, lower value binds more tightly
- *Associativity:*
  - `yfx` : binary with left associativity
  - `xfy` : binary with right associativity
  - `xfx` : binary with no associativity
  - `fx` : unary prefix, non-associative
  - `fy` : unary prefix, associative
  - `xf` : unary postfix, non-associative
  - `xy` : unary postfix, associative
- *Symbol:* the new operator, can be any atom.

Predefined Operators

```
?- help(op).
```

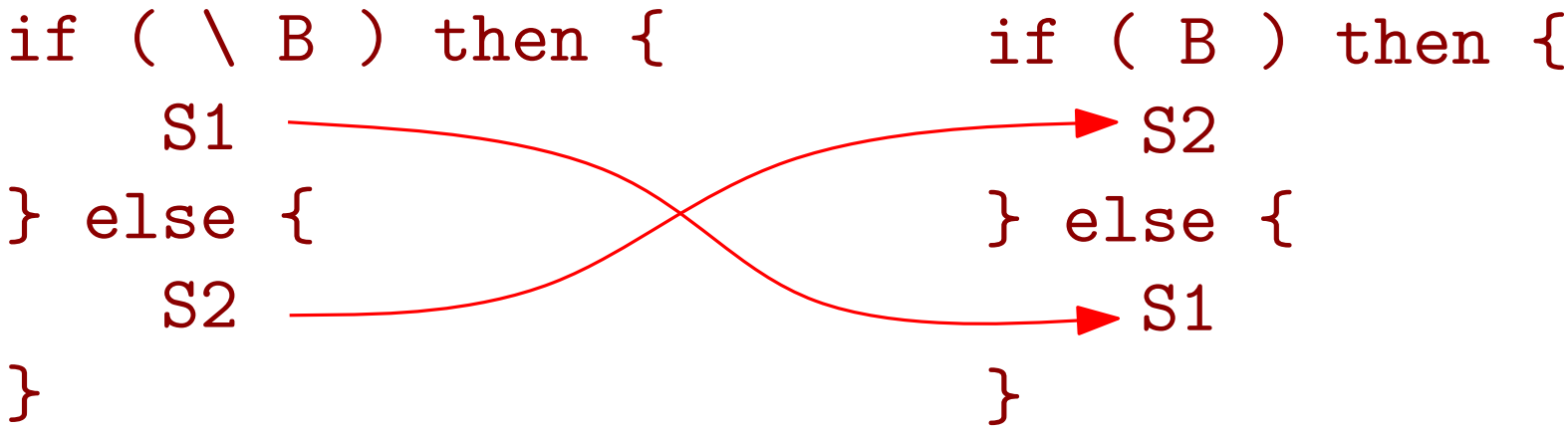
```
---------------------------------------------------------------
| 1200 |xfx  |-->, :-                                         |
| 1200 | fx  |:-, ?-                                          |
| 1150 | fx  |dynamic,    discontiguous,     initialization,  |
|      |     |meta_predicate,  module_transparent, multifile,|
|      |     |thread_local, volatile                          |
| 1100 |xfy  |;, |                                            |
| 1050 |xfy  |->, op*->                                       |
| 1000 |xfy  |,                                               |
|  900 | fy  |\+                                              |

|  900 | fx  |~                                               |
|  700 |xfx  |<, =, =.., =@=,  =:=, =<, ==, =\=,  >, >=, @<, |
|      |     |@=<, @>, @>=, \=, \==, is                       |
|  600 |xfy  |:                                               |
|  500 | yfx |+, -, /\, \/, xor                               |
|  500 | fx  |?                                               |
|  400 | yfx |*, /, //, rdiv, <<, >>, mod, rem                |
|  200 |xfx  |**                                              |
|  200 |xfy  |^                                               |
|__200_|_fy__|+,_-,_\_________________________________________|_
```

A Simple Programming Language

```
?- op(1099,yf,;).
?- op(960,fx,if).
?- op(959,xfx,then).
?- op(958,xfx,else).
?- op(960,fx,while).
?- op(959,xfx,do).
?- op(960,fx,switch).
?- op(959,xfx,of).
?- op(970,xfx,::).
```

```
?- Code = (
          a = 1 ;
          switch a of {
          0:: { x = 1 ; z = x+1 ;} ;
          2:: { x = 2 ;
                x = x - 1 ;
                z = x << 3 ; } ;
          default:: {
                x = 10 ;
                y = 5 ;
                z = 0 ;
                while ( y > 0 ) do {
                    z = z + x ;
                    y = y - 1 ;
                } ; } ;
          };
        ), compileHL(Code,Tac).
```

Program Manipulation



Actually, S1 and S2 must be transforemd as well.

```
transform ( if ( \ B ) then { S1 } else { S2 },
            if ( B ) then { S2t } else { S1t } ) :- !,
                 transform(S1,S1t), transform(S2,S2t).
transform(S,S).
```

Lists

- Lists are widely used in Prolog
- The list of 3 elements a, b, c:
  - `[a,b,c]`
  - syntactic sugar for `.(a, .(b, .(c,[])))`
  - alternative 'cons' writing: `[a|[b|[c|[]]]]`
- `[H|T] = .(H,L)` $\Rightarrow$ the list with head `H` and tail `T`.
- Lists are just regular terms
  - any other functor could be used to encode sequences

Demo on list predicates (watch video)

Arithmetic

```
19 ?- X is 2+3*4.
X = 14.

20 ?- X is sin(3.14).
X = 0.00159265.

21 ?- 5 is 2+3.
true.

22 ?- 5 is X+3.
ERROR: is/2: Arguments are not sufficiently instantiated

factorial(0,1).
factorial(N,X) :- N>0, N1 is N-1,factorial(N1,X1),X is X1*N.

?- factorial(5,X).
X = 120;
false
```

Factorial with Cut

```
factorial(0,1) :- !.
factorial(N,X) :- N1 is N-1, factorial(N1,X1), X is X1*N.
```

Conclusion

- Unification and resolution make an appropriate computation mechanism for symbolic manipulation.
- Imperative programs can be easily encoded as Prolog terms.
- Program transformations can be easily implemented as Prolog rules.