INTRODUCTION TO PROLOG

PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2018

Dalhousie University

STRUCTURE OF A PROLOG PROGRAM

Where, declaratively, Haskell expresses a computation as a system of functions, a Prolog program describes predicates of objects and relations between objects.

STRUCTURE OF A PROLOG PROGRAM

Where, declaratively, Haskell expresses a computation as a system of functions, a Prolog program describes predicates of objects and relations between objects.

The program consists of

- A set of facts, predicates and relations that are known to hold, and
- A set of rules, predicates and relations that are known to hold if other predicates or relations hold.

STRUCTURE OF A PROLOG PROGRAM

Where, declaratively, Haskell expresses a computation as a system of functions, a Prolog program describes predicates of objects and relations between objects.

The program consists of

- A set of facts, predicates and relations that are known to hold, and
- A set of rules, predicates and relations that are known to hold if other predicates or relations hold.

To run a Prolog program, you pose a query. The program “magically” reports all answers to your query that it can prove using the rules and facts included in the program.

STRUCTURE OF A PROLOG PROGRAM: EXAMPLE

```
% Facts <-- This is a comment
person_height(norbert, (6,0)).
person_height(nelly,   (5,1)).
person_height(luca,    (5,3)).
person_height(mateo,   (3,11)).

% Rules
taller_shorter(X, Y) :-
  person_height(X, (FX, _)), person_height(Y, (FY, _)), FX > FY.
taller_shorter(X, Y) :-
  person_height(X, (FX, IX)), person_height(Y, (FY, IY)),
  FX =:= FY, IX > IY.

% Queries
:- person_height(norbert, (F, I)).   % F = 6, I = 0.
:- taller_shorter(luca, nelly).      % true.
:- taller_shorter(X, nelly).         % X = norbert; X = luca.
```

ATOMS, NUMBERS, AND VARIABLES

Atoms:

- Composed of letters, digits, and underscores
- Start with a lowercase letter
- Examples: `nelly` `person0` `other_Item`

Numbers:

- Integers: `1` `-3451913`
- Floating point: `1.0` `-12.318` `4.89e-3`

Variables:

- Composed of letters, digits, and underscores
- Start with an uppercase letter
- Examples: `Person` `_has_underscore`
- Special variable: _ (wildcard)

TERMS

Simple term:

- Atom, number or variable

Complex term:

- Predicate:
  - ⟨*atom*⟩(⟨*term*⟩[,...])
  - Examples: `taller_shorter(X,Y)` `person_height(norbert,(6,0))`
- Infix relation:
  - ⟨*term*⟩ ⟨*rel*⟩ ⟨*term*⟩
  - Examples: `X = pred(Y, Z)` `Number > 4`
- Tuple:
  - (⟨*term*⟩[,...])
  - Examples: `(6,0)` `(Tail, Head)`
- List:
  - [⟨*term*⟩[,...][|⟨*list*⟩]]
  - Examples: `[]` `[X]` `[_|_]` `[A,B|Rest]`

FACTS AND RULES

Fact:

- States what holds.
- $\langle \mathit{term} \rangle$`.`
- Examples: `loves_teaching(norbert).` `married(norbert,nelly).`

Rule:

- States how to deduce new facts from known facts.
- $\langle \mathit{head} \rangle$ `:-` $\langle \mathit{term}_1 \rangle$`,...`.
- $\langle \mathit{head} \rangle$ holds if $\langle \mathit{term}_1 \rangle, \ldots$ hold simultaneously.
- Example:

```
taller_shorter(X, Y) :-
    person_height(X, (FX, IX)),
    person_height(Y, (FY, IY)), FX =:= FY, IX > IY.
```

FACTS AND RULES

Fact:

- States what holds.
- ⟨*term*⟩`.`
- Examples: `loves_teaching(norbert).` `married(norbert,nelly).`
- Can be read as a rule: ⟨*term*⟩ `:- true.`

Rule:

- States how to deduce new facts from known facts.
- ⟨*head*⟩ `:-` ⟨*term*$_1$⟩`,....`
- ⟨*head*⟩ holds if ⟨*term*$_1$⟩,... hold simultaneously.
- Example:

```
taller_shorter(X, Y) :-
    person_height(X, (FX, IX)),
    person_height(Y, (FY, IY)), FX =:= FY, IX > IY.
```

CONJUNCTION AND DISJUNCTION (1)

The goals of a rule are combined conjunctively:

```
between(X, Smaller, Bigger) :- X > Smaller, X < Bigger.
```

says that `X` is between `Smaller` and `Bigger` if `X > Smaller` and `X < Bigger`.

You should read “`,`” as “and”.

CONJUNCTION AND DISJUNCTION (1)

The goals of a rule are combined conjunctively:

```
between(X, Smaller, Bigger) :- X > Smaller, X < Bigger.
```

says that `X` is between `Smaller` and `Bigger` if `X > Smaller` and `X < Bigger`.

You should read "`,`" as "and".

We express disjunction, the possibility to make a predicate true in different ways, by stating multiple facts or rules for this predicate:

```
elem_list(Elem, [Elem|_]).
elem_list(Elem, [_|Tail]) :- elem_list(Elem, Tail).
```

`Elem` is a member of `List` if

- `Elem` is the head of `List` or
- `Elem` is an element of the tail of `List`.

CONJUNCTION AND DISJUNCTION (2)

There is a shorthand for writing disjunctions:

```
outside(X, Smaller, Bigger) :- X < Smaller; X > Bigger.
```

says that `X` is outside the range (`Smaller`, `Bigger`) if `X < Smaller` or `X > Bigger`.

You should read "`;`" as "or".

CONJUNCTION AND DISJUNCTION (2)

There is a shorthand for writing disjunctions:

```
outside(X, Smaller, Bigger) :- X < Smaller; X > Bigger.
```

says that `X` is outside the range (`Smaller`, `Bigger`) if `X < Smaller` or `X > Bigger`.

You should read “`;`” as “or”.

In fact, this is also how you ask Prolog for more answers:

```
a_number(1).
a_number(2).
```

CONJUNCTION AND DISJUNCTION (2)

There is a shorthand for writing disjunctions:

```
outside(X, Smaller, Bigger) :- X < Smaller; X > Bigger.
```

says that `X` is outside the range (`Smaller`, `Bigger`) if `X < Smaller` or `X > Bigger`.

You should read "`;`" as "or".

In fact, this is also how you ask Prolog for more answers:

```
a_number(1).
a_number(2).

?- a_number(X).
X = 1
```

CONJUNCTION AND DISJUNCTION (2)

There is a shorthand for writing disjunctions:

```
outside(X, Smaller, Bigger) :- X < Smaller; X > Bigger.
```

says that `X` is outside the range (`Smaller`, `Bigger`) if `X < Smaller` or `X > Bigger`.

You should read "`;`" as "or".

In fact, this is also how you ask Prolog for more answers:

```
a_number(1).
a_number(2).

?- a_number(X).
X = 1 ;
X = 2.
```

UNIFICATION (1)

A query term holds if it unifies with a term provable using the rules and facts in the program.

UNIFICATION (1)

A query term holds if it unifies with a term provable using the rules and facts in the program.

Intuitively, two terms unify if the variables on both sides can be replaced with terms to make the two terms the same.

- Every occurrence of a given variable needs to be replaced with the same term.

UNIFICATION (1)

A query term holds if it unifies with a term provable using the rules and facts in the program.

Intuitively, two terms unify if the variables on both sides can be replaced with terms to make the two terms the same.

- Every occurrence of a given variable needs to be replaced with the same term.

Examples: (`=` tests whether two terms unify, `\=` tests whether they don't)

- `X = X` `X = Y` `X = a(Y)` `a(X,y,z) = a(y,X,z)` `a \= b`
  all succeed (individually).

UNIFICATION (1)

A query term holds if it unifies with a term provable using the rules and facts in the program.

Intuitively, two terms unify if the variables on both sides can be replaced with terms to make the two terms the same.

- Every occurrence of a given variable needs to be replaced with the same term.

Examples: (`=` tests whether two terms unify, `\=` tests whether they don't)

- `X = X` `X = Y` `X = a(Y)` `a(X,y,z) = a(y,X,z)` `a \= b`
  all succeed (individually).
- `X = a, X = b` fails because `X = a` forces `X` to equal `a` and then `a \= b`.

UNIFICATION (2)

Formal definition:

UNIFICATION (2)

Formal definition:

- Two identical terms unify.

UNIFICATION (2)

Formal definition:

- Two identical terms unify.
- A variable unifies with any other term.

UNIFICATION (2)

Formal definition:

- Two identical terms unify.
- A variable unifies with any other term.
- If T_1 and T_2 are complex terms, they unify if
  - They have the same functor and arity,
  - Their corresponding arguments unify, and
  - The resulting variable instantiations are compatible.

UNIFICATION (2)

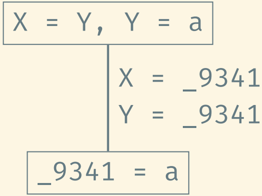
Formal definition:

- Two identical terms unify.
- A variable unifies with any other term.
- If T_1 and T_2 are complex terms, they unify if
    - They have the same functor and arity,
    - Their corresponding arguments unify, and
    - The resulting variable instantiations are compatible.
- If none of the above rules applies to T_1 and T_2, then T_1 and T_2 do not unify.

UNIFICATION (3)

$\boxed{\texttt{X = Y, Y = a}}$

UNIFICATION (3)

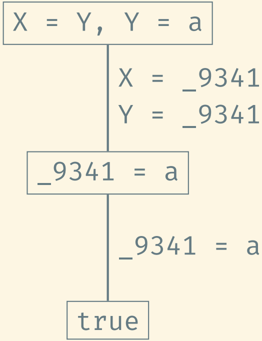


`X = Y, Y = a`

`X = _9341`
`Y = _9341`

`_9341 = a`

UNIFICATION (3)



`X = Y, Y = a`

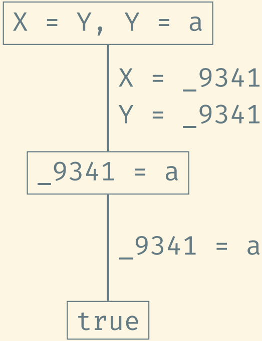
`X = _9341`
`Y = _9341`

`_9341 = a`

`_9341 = a`

`true`

UNIFICATION (3)



`X = Y, Y = a`

`X = _9341`
`Y = _9341`

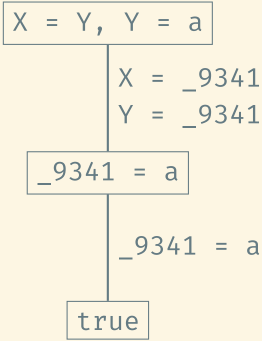
`_9341 = a`

`_9341 = a`

`true`

`X = Y, X = a, Y = b`

UNIFICATION (3)



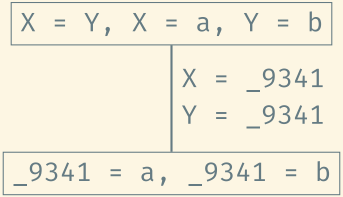
X = Y, Y = a

X = _9341
Y = _9341

_9341 = a

_9341 = a

true

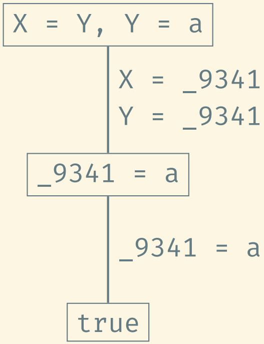


X = Y, X = a, Y = b

X = _9341
Y = _9341

_9341 = a, _9341 = b

UNIFICATION (3)



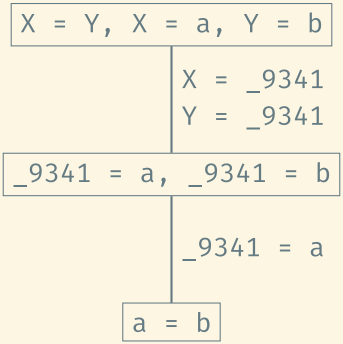
`X = Y, Y = a`

`X = _9341`
`Y = _9341`

`_9341 = a`

`_9341 = a`

`true`



`X = Y, X = a, Y = b`

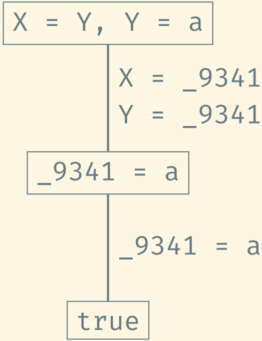
`X = _9341`
`Y = _9341`

`_9341 = a, _9341 = b`

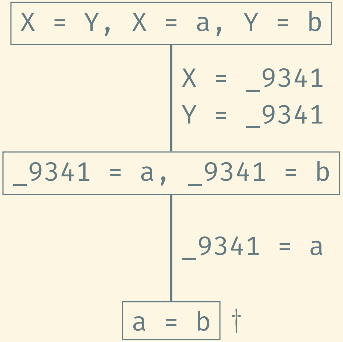
`_9341 = a`

`a = b`

UNIFICATION (3)



```
X = Y, Y = a
X = _9341
Y = _9341
_9341 = a
_9341 = a
true
```



```
X = Y, X = a, Y = b
X = _9341
Y = _9341
_9341 = a, _9341 = b
_9341 = a
a = b †
```

OCCURS CHECK

What about the query `X = f(X)`?

OCCURS CHECK

What about the query `X = f(X)`?

Logically, this should fail because there is no (finite) instantiation of `X` that makes the two sides equal.

OCCURS CHECK

What about the query `X = f(X)`?

Logically, this should fail because there is no (finite) instantiation of `X` that makes the two sides equal.

In Prolog, this query succeeds with the answer `X = f(X)`.

OCCURS CHECK

What about the query `X = f(X)`?

Logically, this should fail because there is no (finite) instantiation of `X` that makes the two sides equal.

In Prolog, this query succeeds with the answer `X = f(X)`.

In the interest of efficiency, Prolog does not check whether a variable occurs in its own replacement.

OCCURS CHECK

What about the query `X = f(X)`?

Logically, this should fail because there is no (finite) instantiation of `X` that makes the two sides equal.

In Prolog, this query succeeds with the answer `X = f(X)`.

In the interest of efficiency, Prolog does not check whether a variable occurs in its own replacement.

If you want to test for unification with occurs check, use `unify_with_occurs_check/2`, so

```
?- unify_with_occurs_check(X, f(X)).
false.
```

BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```

BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

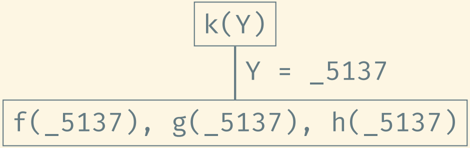
```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```

$\boxed{\texttt{k(Y)}}$

BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

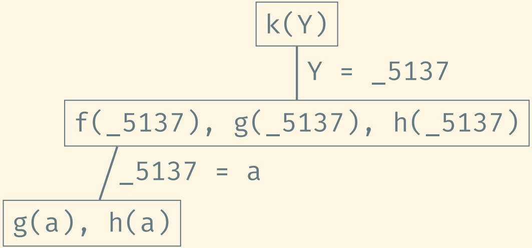
```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

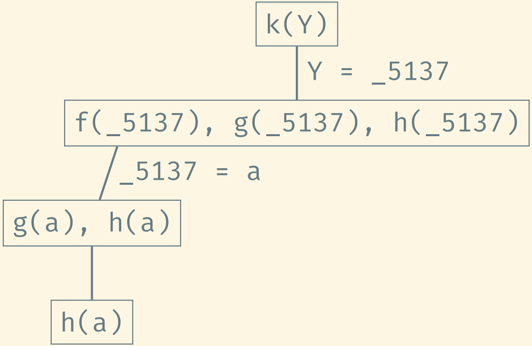
```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

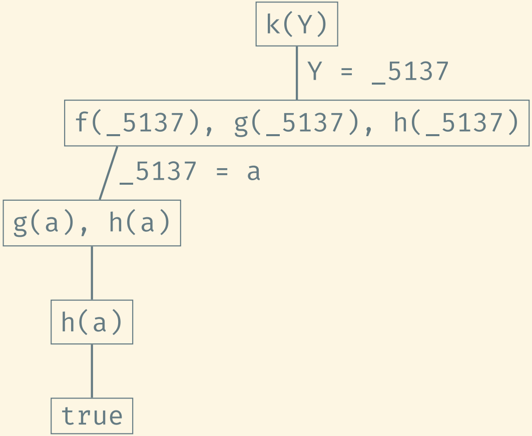
```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



k(Y)

Y = _5137

f(_5137), g(_5137), h(_5137)

_5137 = a

g(a), h(a)

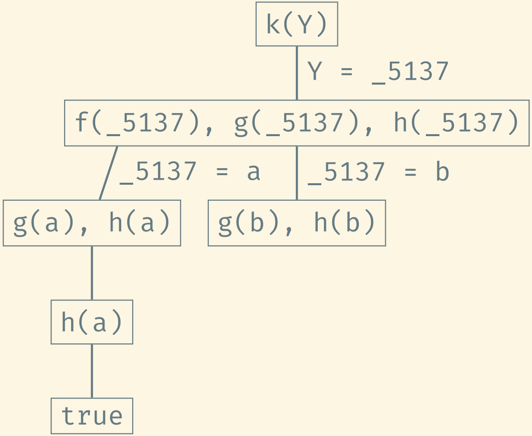
h(a)

true

BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



k(Y)

Y = _5137

f(_5137), g(_5137), h(_5137)

_5137 = a

_5137 = b

g(a), h(a)

g(b), h(b)

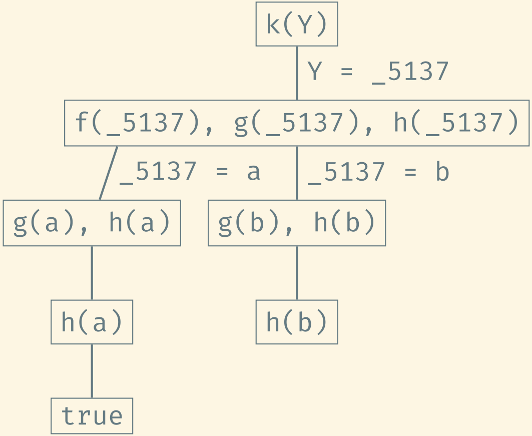
h(a)

true

BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



k(Y)

Y = _5137

f(_5137), g(_5137), h(_5137)

_5137 = a

_5137 = b

g(a), h(a)

g(b), h(b)

h(a)

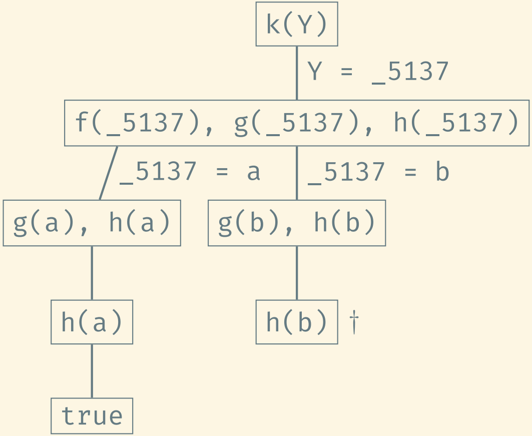
h(b)

true

BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

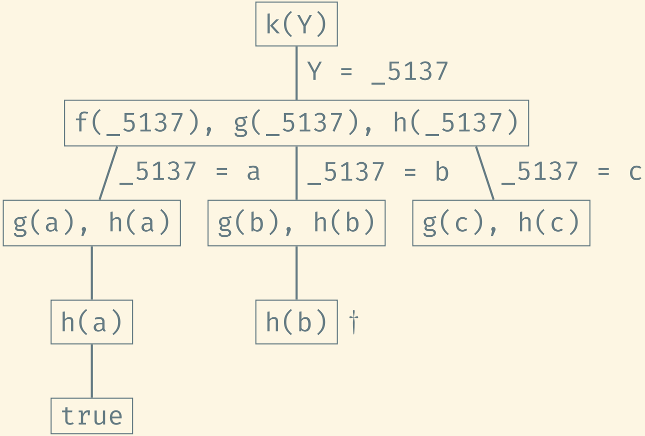
```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

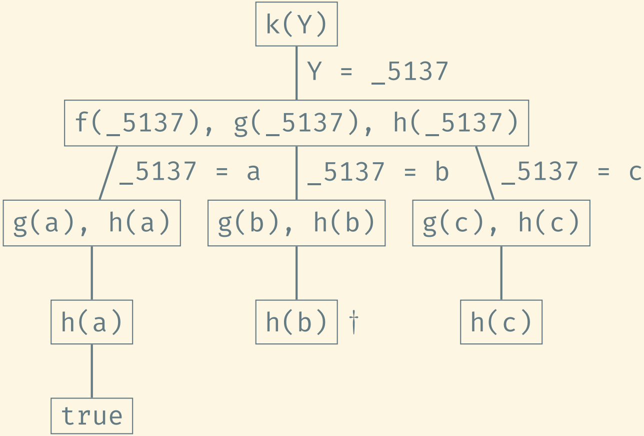
```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

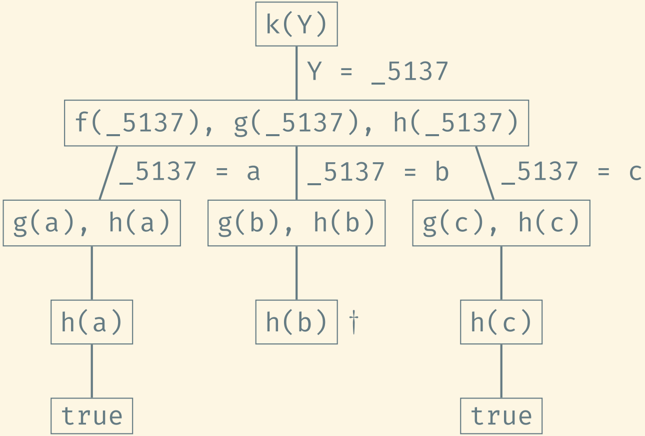
```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



BACKTRACKING

To find the answers to a query, Prolog applies depth-first search with unification. When searching for a fact or rule that unifies with a goal, it searches the database from top to bottom.

```
f(a).
f(b).
f(c).

g(a).
g(b).
g(c).

h(a).
h(c).

k(X) :-
  f(X), g(X), h(X).
```



LISTS

Sequences and collections are represented as lists.

LISTS

Sequences and collections are represented as lists.

Since list elements can themselves be lists, we can use lists to represent complicated data structures such as trees (even though they are often better represented as deeply nested complex terms).

LISTS

Sequences and collections are represented as lists.

Since list elements can themselves be lists, we can use lists to represent complicated data structures such as trees (even though they are often better represented as deeply nested complex terms).

- Empty list: `[]`
- Head and tail: `[a|[b,c,d]] = [a,b,c,d]`
  `[a|[]]      = [a]`
- Multiple heads: `[a,b|[c,d]] = [a,b,c,d]`

CONTROL FLOW CONSTRUCTS

The notion of “control flow” is much weaker in Prolog than even in a functional language because we are (mostly) not concerned with the order in which the Prolog interpreter does things.

CONTROL FLOW CONSTRUCTS

The notion of “control flow” is much weaker in Prolog than even in a functional language because we are (mostly) not concerned with the order in which the Prolog interpreter does things.

What we do need is a way to build up arbitrarily complex relations

CONTROL FLOW CONSTRUCTS

The notion of “control flow” is much weaker in Prolog than even in a functional language because we are (mostly) not concerned with the order in which the Prolog interpreter does things.

What we do need is a way to build up arbitrarily complex relations … inductively

CONTROL FLOW CONSTRUCTS

The notion of “control flow” is much weaker in Prolog than even in a functional language because we are (mostly) not concerned with the order in which the Prolog interpreter does things.

What we do need is a way to build up arbitrarily complex relations … inductively … using recursion.

RECURSION

Summing a list of integers:

```
sum([], 0).
sum([X|Xs], Sum) :-
  sum(Xs, Sum1), Sum is Sum1 + X.
```

RECURSION

Summing a list of integers:

```
sum([], 0).
sum([X|Xs], Sum) :-
  sum(Xs, Sum1), Sum is Sum1 + X.
```

Better:

```
sum([], 0).
sum([X|Xs], Sum) :-
  Sum #= Sum1 + X, sum(Xs, Sum1).
```

MAPPING A PREDICATE OVER A LIST OR LISTS

```
odd(X) :- 1 is X mod 2.

?- maplist(odd,[1,3,5]).
true.
?- maplist(odd,[1,2,3]).
false.
```

MAPPING A PREDICATE OVER A LIST OR LISTS

```
odd(X) :- 1 is X mod 2.

?- maplist(odd,[1,3,5]).
true.
?- maplist(odd,[1,2,3]).
false.

?- maplist(<,[1,3,5],[2,7,8]).
true.
?- maplist(<,[1,3,9],[2,7,8]).
false.
```

MAPPING A PREDICATE OVER A LIST OR LISTS

```
odd(X) :- 1 is X mod 2.

?- maplist(odd,[1,3,5]).
true.
?- maplist(odd,[1,2,3]).
false.

?- maplist(<,[1,3,5],[2,7,8]).
true.
?- maplist(<,[1,3,9],[2,7,8]).
false.

add(X,Y,Sum) :- Sum is X+Y.

?- maplist(add,[1,3,5],[4,8,9],Sums).
Sums = [5,11,14].
```

BUILT-IN PREDICATES

Primitives:

- `true`, `false`
- `fail` (is the same as `false`)

BUILT-IN PREDICATES

Primitives:

- `true`, `false`
- `fail` (is the same as `false`)

Unification:

- `=` (arguments unify), `\=` (arguments do not unify)

BUILT-IN PREDICATES

Primitives:

- `true`, `false`
- `fail` (is the same as `false`)

Unification:

- `=` (arguments unify), `\=` (arguments do not unify)

Arithmetic and numeric comparisons: (Use with caution.)

- `+`, `-`, `*`, `/`, `//`
- `<`, `>`, `>=`, `=<`, `=:=`, `=\=`
- `5 \= 2+3` but `X is 2+3, 5 = X`

BUILT-IN PREDICATES

Primitives:

- `true`, `false`
- `fail` (is the same as `false`)

Unification:

- `=` (arguments unify), `\=` (arguments do not unify)

Arithmetic and numeric comparisons: (Use with caution.)

- `+`, `-`, `*`, `/`, `//`
- `<`, `>`, `>=`, `=<`, `=:=`, `=\=`
- `5 \= 2+3` but `X is 2+3, 5 = X`

Lots more

CONTROL FLOW: GOAL ORDERING (1)

Given the facts

```
f(e).  g(a).  g(b).  g(c).  g(d).  g(e).
```

the following two predicates are logically the same:

```
h1(X) :- f(X), g(X).    h2(X) :- g(X), f(X).
```

CONTROL FLOW: GOAL ORDERING (1)

Given the facts

```
f(e).  g(a).  g(b).  g(c).  g(d).  g(e).
```

the following two predicates are logically the same:

```
h1(X) :- f(X), g(X).    h2(X) :- g(X), f(X).
```

Which one is more efficient?

CONTROL FLOW: GOAL ORDERING (1)

Given the facts

```
f(e).  g(a).  g(b).  g(c).  g(d).  g(e).
```

the following two predicates are logically the same:

```
h1(X) :- f(X), g(X).    h2(X) :- g(X), f(X).
```

Which one is more efficient?

- `h1` instantiates `X = e` and then succeeds because `g(e)` holds.
- `h2` instantiates `X = a`, `X = b`, … and fails on all instantiations except `X = e`.

CONTROL FLOW: GOAL ORDERING (2)

Consider the facts

```
child(anne,bridget).    child(bridget,caroline).
child(caroline,donna).  child(donna,emily).
```

and the following logically equivalent definitions of a descendant relationship:

```
descend1(X,Y)                      descend2(X,Y)
  :- child(X,Z), descend1(Z,Y).      :- descend2(Z,Y), child(X,Z).
descend1(X,Y) :- child(X,Y).       descend2(X,Y) :- child(X,Y).
```

CONTROL FLOW: GOAL ORDERING (2)

Consider the facts

```
child(anne,bridget).    child(bridget,caroline).
child(caroline,donna).  child(donna,emily).
```

and the following logically equivalent definitions of a descendant relationship:

```
descend1(X,Y)                         descend2(X,Y)
  :- child(X,Z), descend1(Z,Y).         :- descend2(Z,Y), child(X,Z).
descend1(X,Y) :- child(X,Y).          descend2(X,Y) :- child(X,Y).
```

Now ask the queries `descend1(anne,bridget)` and `descend2(anne,bridget)`. What happens?

CONTROL FLOW: GOAL ORDERING (2)

Consider the facts

```
child(anne,bridget).    child(bridget,caroline).
child(caroline,donna).  child(donna,emily).
```

and the following logically equivalent definitions of a descendant relationship:

```
descend1(X,Y)                           descend2(X,Y)
  :- child(X,Z), descend1(Z,Y).           :- descend2(Z,Y), child(X,Z).
descend1(X,Y) :- child(X,Y).            descend2(X,Y) :- child(X,Y).
```

Now ask the queries `descend1(anne,bridget)` and `descend2(anne,bridget)`. What happens?

- `descend1(anne,bridget)` succeeds.
- `descend2(anne,bridget)` does not terminate.

CONTROL FLOW: CUT

`!` (read “cut”) is a predicate that always succeeds, but with a side effect:

- It commits Prolog to all choices (unification of variables) that were made since the parent goal was unified with the left-hand side of the rule.
- This includes the choice to use this particular rule.

CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

```
?- p(X).
```

CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

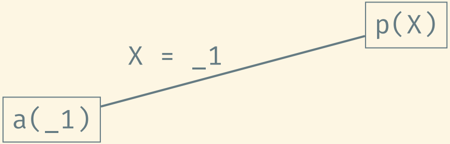
```
?- p(X).
```

`p(X)`

CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

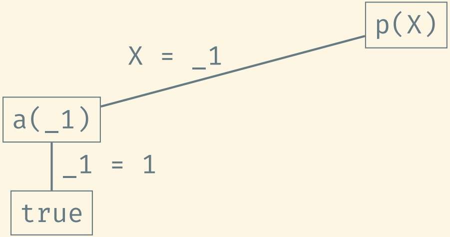
```
?- p(X).
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

```
?- p(X).
```



p(X)

X = _1

a(_1)

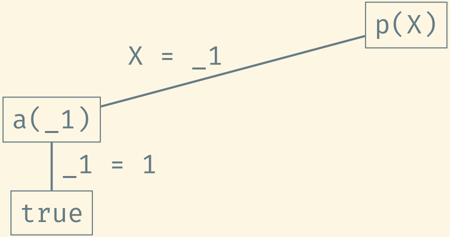
_1 = 1

true

CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

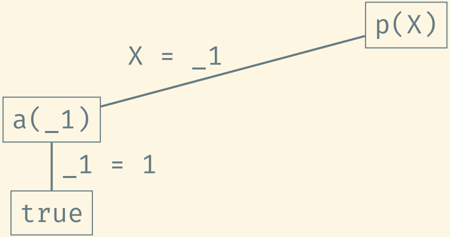
```
?- p(X).
X = 1
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

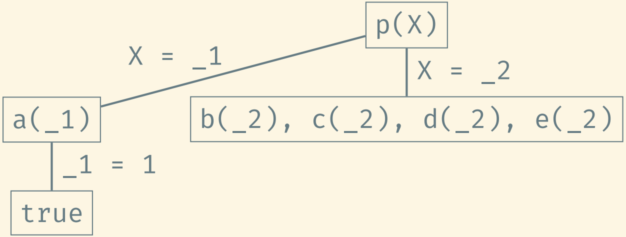
```
?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

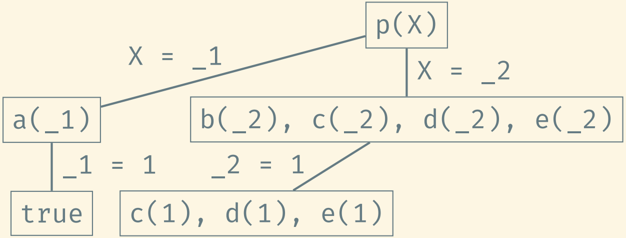
```
?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

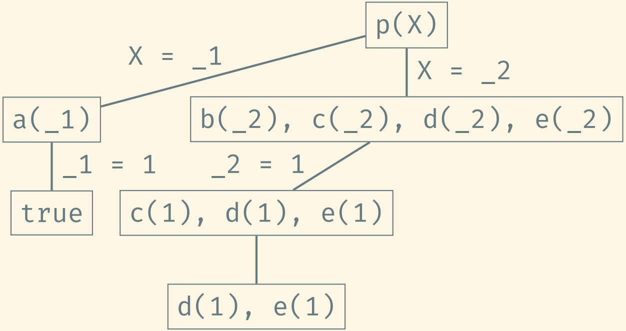
```
?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

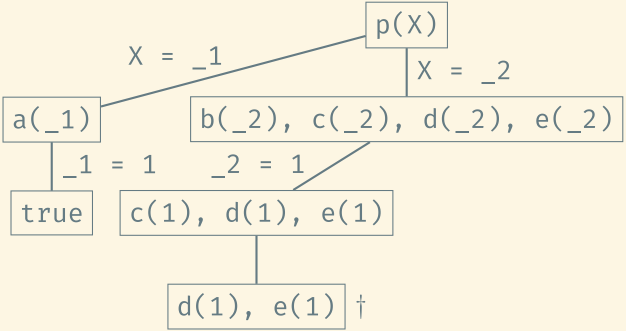
```
?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

```
?- p(X).
X = 1 ;
```



p(X)

X = _1

X = _2

a(_1)

b(_2), c(_2), d(_2), e(_2)

_1 = 1

_2 = 1

true

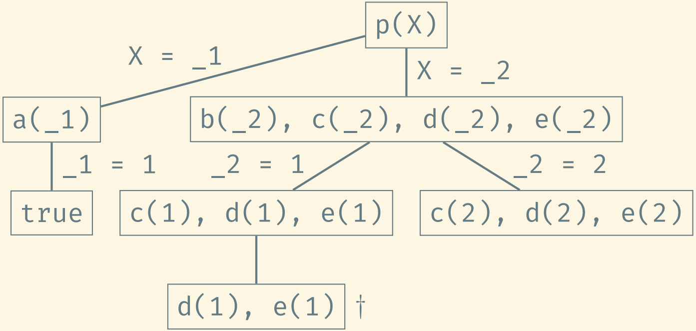
c(1), d(1), e(1)

d(1), e(1) †

CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

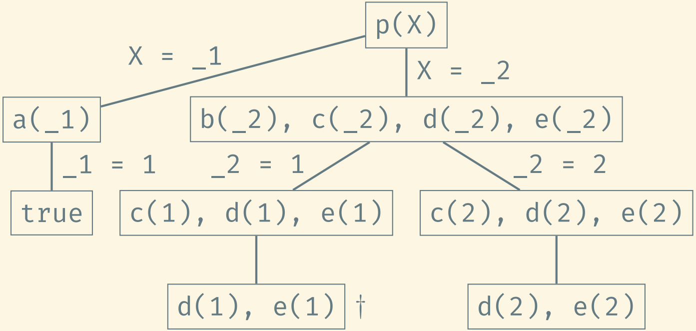
```
?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

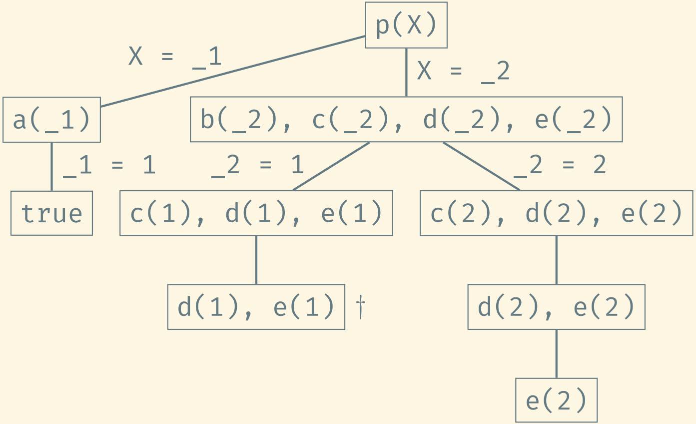
```
?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

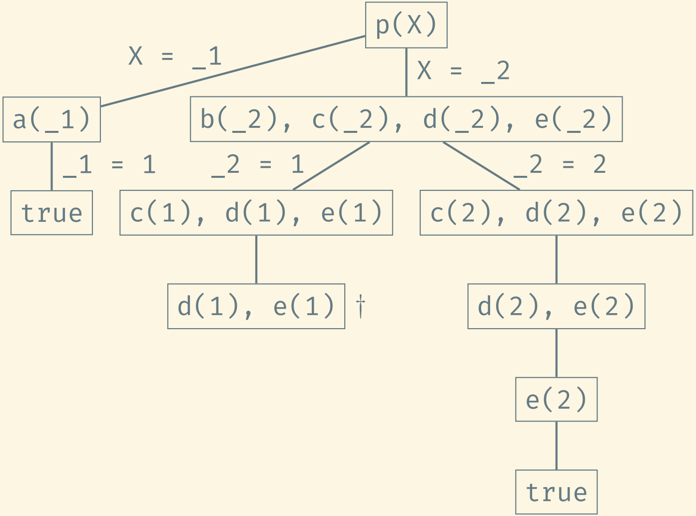
```
?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

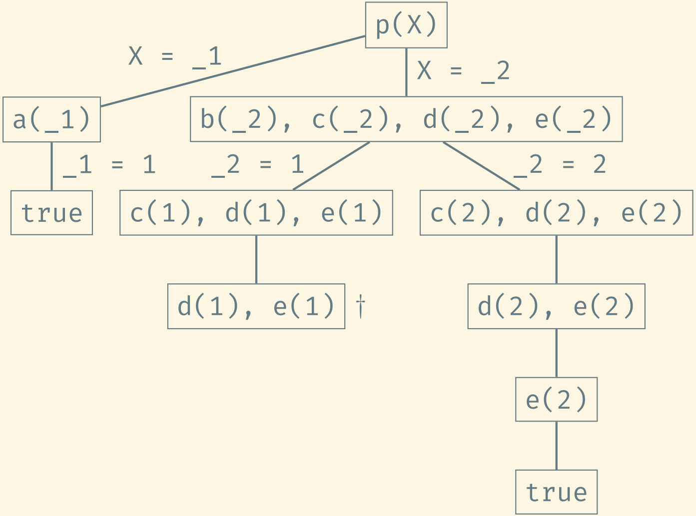
```
?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

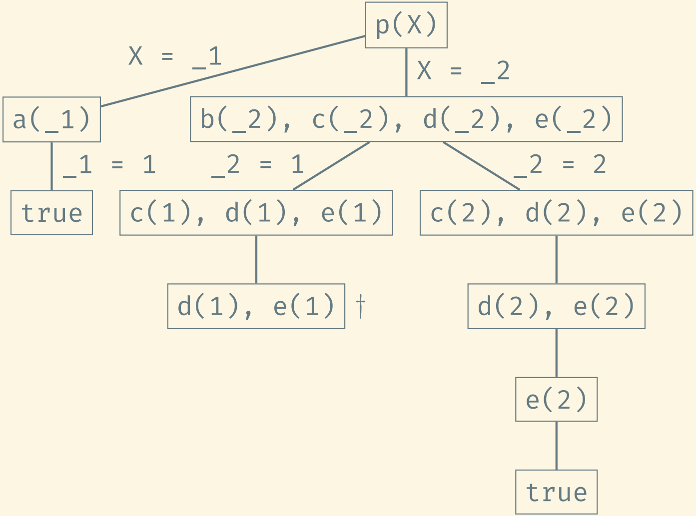
```
?- p(X).
X = 1 ;
X = 2
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

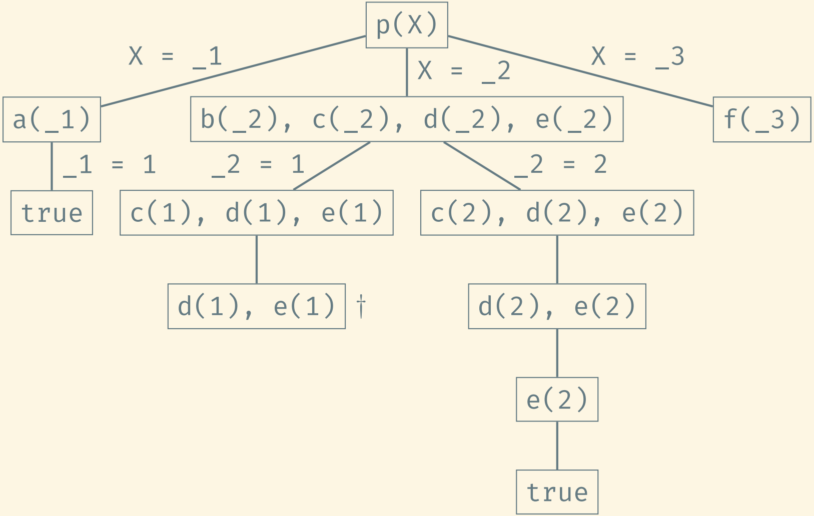
```
?- p(X).
X = 1 ;
X = 2 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

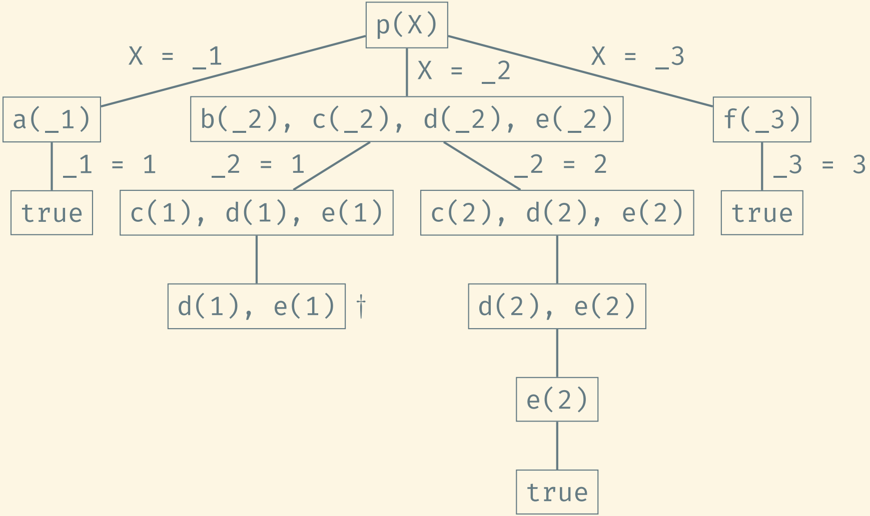
```
?- p(X).
X = 1 ;
X = 2 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

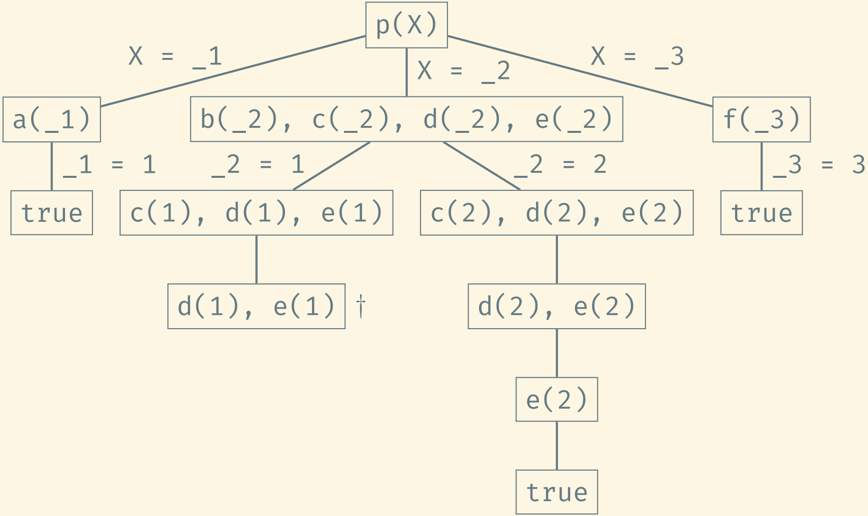
```
?- p(X).
X = 1 ;
X = 2 ;
```



CUT: FIRST EXAMPLE (1)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).    p(X) :- b(X), c(X), d(X), e(X).    p(X) :- f(X).
```

```
?- p(X).
X = 1 ;
X = 2 ;
X = 3.
```



CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).
```

CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
```

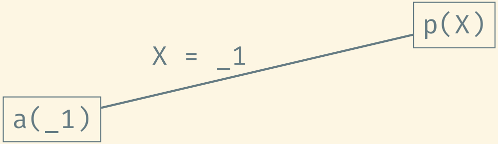
CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
```

p(X)

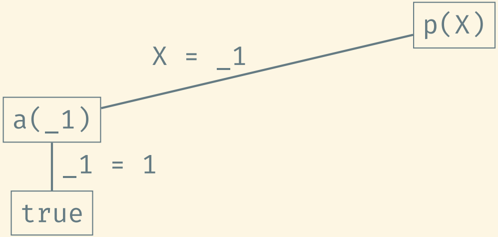
CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
```



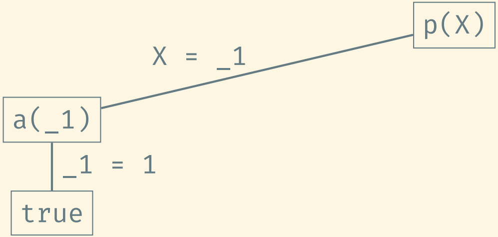
CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
```



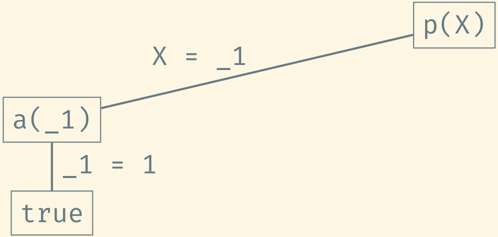
CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
X = 1
```



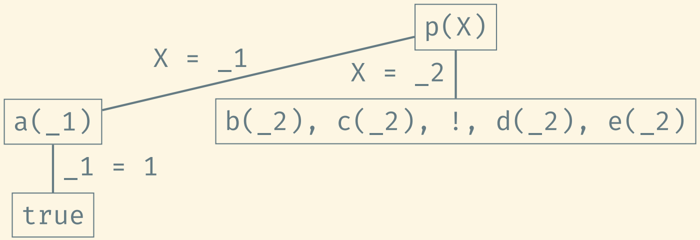
CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (2)

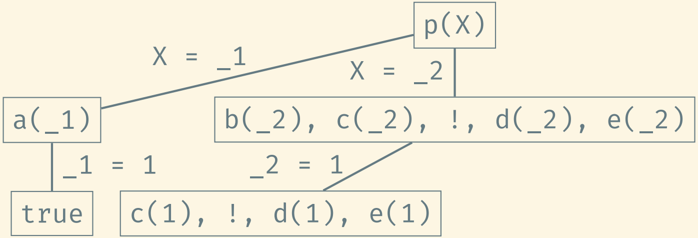
```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (2)

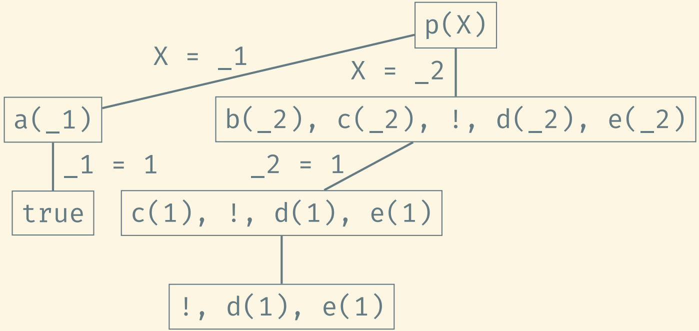
```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).
```

```
?- p(X).
X = 1 ;
```



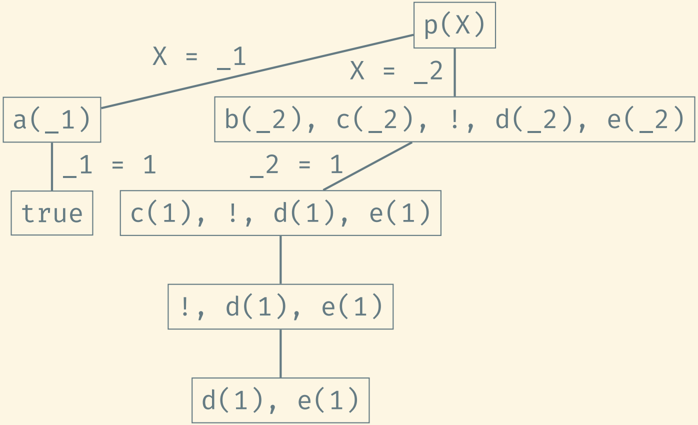
CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (2)

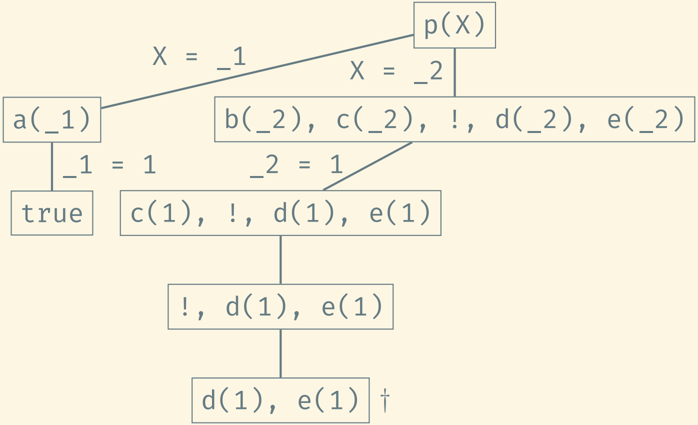
```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (2)

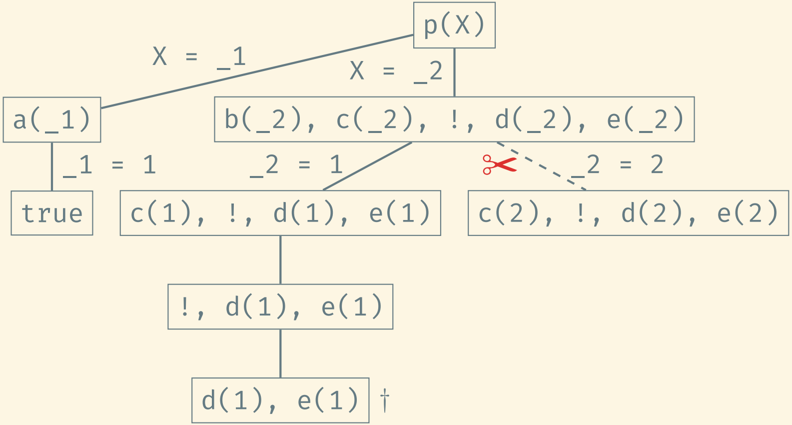
```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).
```

```
?- p(X).
X = 1 ;
```



CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).

?- p(X).
X = 1 ;
```



p(X)

X = _1

X = _2

a(_1)

b(_2), c(_2), !, d(_2), e(_2)

_1 = 1

_2 = 1

✂

_2 = 2

true

c(1), !, d(1), e(1)

c(2), !, d(2), e(2)

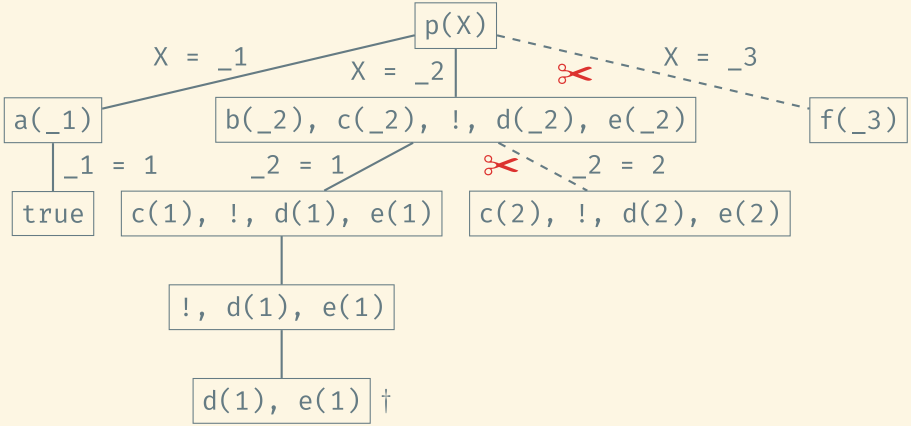
!, d(1), e(1)

d(1), e(1) †

CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).
```

```
?- p(X).
X = 1 ;
```



p(X)

X = _1

X = _2

✂ X = _3

a(_1)

b(_2), c(_2), !, d(_2), e(_2)

f(_3)

_1 = 1

_2 = 1

✂ _2 = 2

true

c(1), !, d(1), e(1)

c(2), !, d(2), e(2)

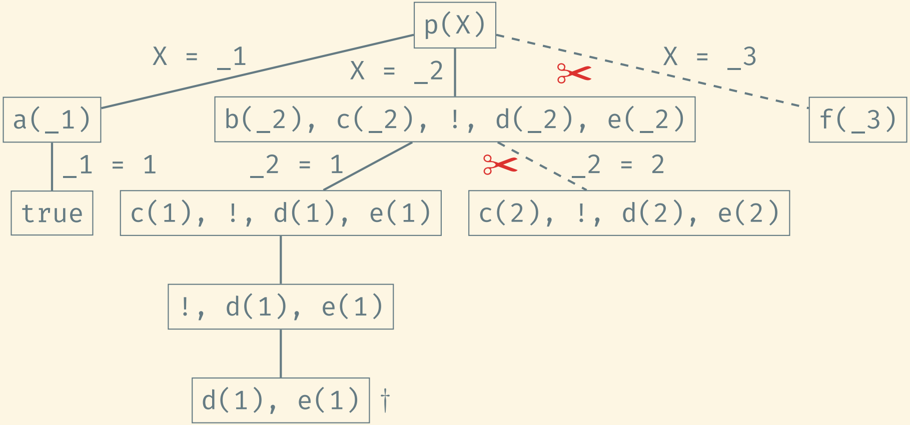
!, d(1), e(1)

d(1), e(1) †

CUT: FIRST EXAMPLE (2)

```
a(1).    b(1). b(2).    c(1). c(2).    d(2).    e(2).    f(3).
p(X) :- a(X).   p(X) :- b(X), c(X), !, d(X), e(X).   p(X) :- f(X).
```

```
?- p(X).
X = 1 ;
false.
```



CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
q(4,7).

a(1).  a(2).
b(4).  b(5).
```

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
q(4,7).

a(1).  a(2).
b(4).  b(5).

?- p(X,Y).
```

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
q(4,7).

a(1).  a(2).
b(4).  b(5).

?- p(X,Y).
```

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
q(4,7).

a(1).  a(2).
b(4).  b(5).

?- p(X,Y).
```

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
q(4,7).

a(1).  a(2).
b(4).  b(5).

?- p(X,Y).
```

CUT: SECOND EXAMPLE

`p(X,Y) :- q(X,Y).`
`p(3,6).`

`q(X,Y) :- a(X), !, b(Y).`
~~`q(4,7).`~~

`a(1).` ~~`a(2).`~~
`b(4).` `b(5).`

`?- p(X,Y).`

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
~~q(4,7).~~

a(1).  ~~a(2).~~
b(4).  b(5).

?- p(X,Y).
```

CUT: SECOND EXAMPLE

`p(X,Y) :- q(X,Y).`
`p(3,6).`

`q(X,Y) :- a(X), !, b(Y).`
~~`q(4,7).`~~

`a(1).` ~~`a(2).`~~
`b(4).` `b(5).`

```
?- p(X,Y).
X = 1, Y = 4
```

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
~~q(4,7).~~

a(1).  ~~a(2).~~
b(4).  b(5).

?- p(X,Y).
X = 1, Y = 4 ;
```

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
~~q(4,7).~~

a(1).  ~~a(2).~~
b(4).  b(5).

?- p(X,Y).
X = 1, Y = 4 ;
```

CUT: SECOND EXAMPLE

`p(X,Y) :- q(X,Y).`
`p(3,6).`

`q(X,Y) :- a(X), !, b(Y).`
~~`q(4,7).`~~

`a(1).` ~~`a(2).`~~
`b(4).` `b(5).`

```
?- p(X,Y).
X = 1, Y = 4 ;
X = 1, Y = 5
```

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
~~q(4,7)~~.

a(1).  ~~a(2)~~.
b(4).  b(5).

?- p(X,Y).
X = 1, Y = 4 ;
X = 1, Y = 5 ;
```

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
q(4,7).

a(1).  a(2).
b(4).  b(5).

?- p(X,Y).
X = 1, Y = 4 ;
X = 1, Y = 5 ;
```

CUT: SECOND EXAMPLE

```
p(X,Y) :- q(X,Y).
p(3,6).

q(X,Y) :- a(X), !, b(Y).
q(4,7).

a(1).  a(2).
b(4).  b(5).

?- p(X,Y).
X = 1, Y = 4 ;
X = 1, Y = 5 ;
X = 3, Y = 6.
```

CUT: THIRD EXAMPLE (1)

A predicate to compute the maximum:

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

CUT: THIRD EXAMPLE (1)

A predicate to compute the maximum:

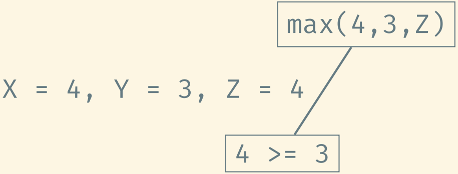
```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```

`max(4,3,Z)`

CUT: THIRD EXAMPLE (1)

A predicate to compute the maximum:

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



`max(4,3,Z)`

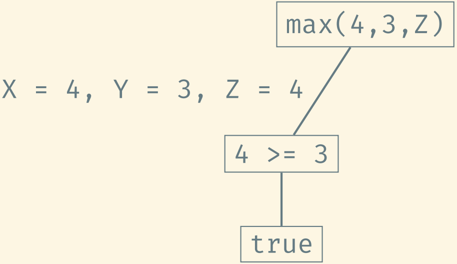
`X = 4, Y = 3, Z = 4`

`4 >= 3`

CUT: THIRD EXAMPLE (1)

A predicate to compute the maximum:

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



`max(4,3,Z)`

`X = 4, Y = 3, Z = 4`

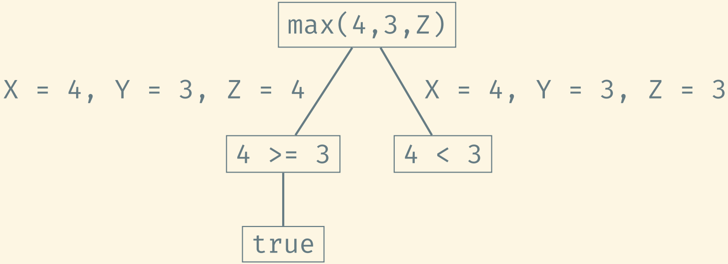
`4 >= 3`

`true`

CUT: THIRD EXAMPLE (1)

A predicate to compute the maximum:

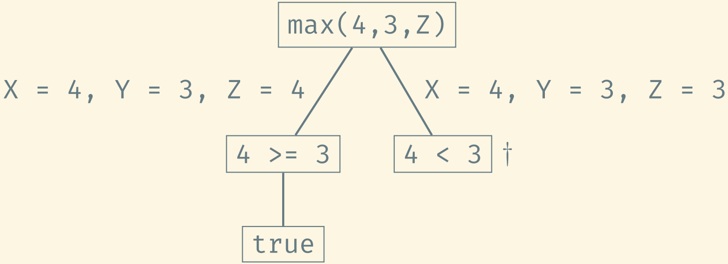
```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```



CUT: THIRD EXAMPLE (1)

A predicate to compute the maximum:

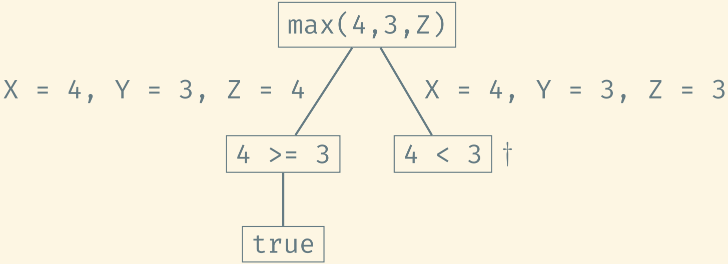
```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```



CUT: THIRD EXAMPLE (1)

A predicate to compute the maximum:

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



This is correct but inefficient.

CUT: THIRD EXAMPLE (2)

A more efficient implementation:

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

CUT: THIRD EXAMPLE (2)

A more efficient implementation:

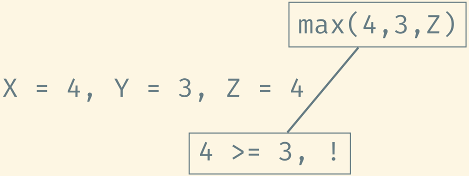
```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.
```

$\boxed{\texttt{max(4,3,Z)}}$

CUT: THIRD EXAMPLE (2)

A more efficient implementation:

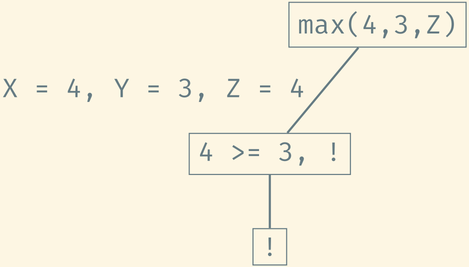
```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.
```



CUT: THIRD EXAMPLE (2)

A more efficient implementation:

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



`max(4,3,Z)`

`X = 4, Y = 3, Z = 4`

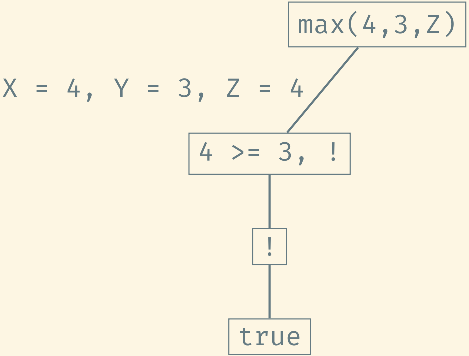
`4 >= 3, !`

`!`

CUT: THIRD EXAMPLE (2)

A more efficient implementation:

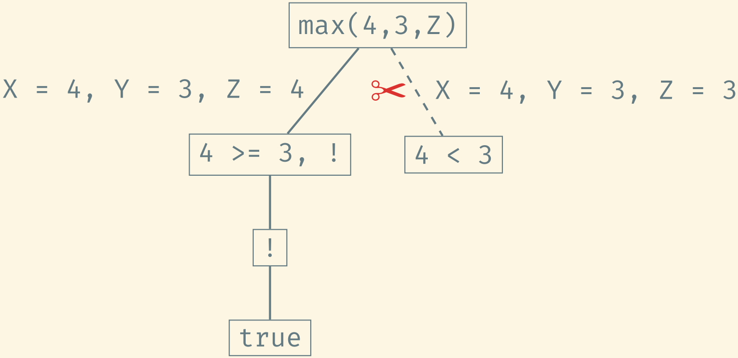
```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.
```



CUT: THIRD EXAMPLE (2)

A more efficient implementation:

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



`max(4,3,Z)`

`X = 4, Y = 3, Z = 4`

✂ `X = 4, Y = 3, Z = 3`

`4 >= 3, !`

`4 < 3`

`!`

`true`

CUT: THIRD EXAMPLE (3)

Even more efficient?

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

CUT: THIRD EXAMPLE (3)

Even more efficient?

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

```
?- max(4,3,Z).
Z = 4.
```

CUT: THIRD EXAMPLE (3)

Even more efficient?

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

```
?- max(4,3,Z).
Z = 4.
```

```
?- max(3,5,Z).
Z = 5.
```

CUT: THIRD EXAMPLE (3)

Even more efficient?

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

```
?- max(4,3,Z).
Z = 4.

?- max(3,5,Z).
Z = 5.

?- max(3,4,3).
false.
```

CUT: THIRD EXAMPLE (3)

Even more efficient?

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

?- max(4,3,Z).
Z = 4.

?- max(3,5,Z).
Z = 5.

?- max(3,4,3).
false.

?- max(4,3,3).
true. % <-- incorrect
```

CUT: THIRD EXAMPLE (4)

Avoiding the second comparison correctly:

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

CUT: THIRD EXAMPLE (4)

Avoiding the second comparison correctly:

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

```
?- max(4,3,Z).
Z = 4.
```

CUT: THIRD EXAMPLE (4)

Avoiding the second comparison correctly:

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

?- max(4,3,Z).
Z = 4.

?- max(3,5,Z).
Z = 5.
```

CUT: THIRD EXAMPLE (4)

Avoiding the second comparison correctly:

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

```
?- max(4,3,Z).
Z = 4.

?- max(3,5,Z).
Z = 5.

?- max(3,4,3).
false.
```

CUT: THIRD EXAMPLE (4)

Avoiding the second comparison correctly:

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

?- max(4,3,Z).
Z = 4.

?- max(3,5,Z).
Z = 5.

?- max(3,4,3).
false.

?- max(4,3,3).
false.
```

NEGATION: CUT AND FAIL

In general, Prolog has no notion of a predicate not being true! It can only decide whether it can prove the predicate using the information in the database.

NEGATION: CUT AND FAIL

In general, Prolog has no notion of a predicate not being true! It can only decide whether it can prove the predicate using the information in the database.

This is called “negation as failure”.

NEGATION: CUT AND FAIL

In general, Prolog has no notion of a predicate not being true! It can only decide whether it can prove the predicate using the information in the database.

This is called “negation as failure”.

It is useful to be able to ask the question: “Are you unable to prove this predicate?”

NEGATION: CUT AND FAIL

In general, Prolog has no notion of a predicate not being true! It can only decide whether it can prove the predicate using the information in the database.

This is called “negation as failure”.

It is useful to be able to ask the question: “Are you unable to prove this predicate?” (Is this predicate false?)

NEGATION: CUT AND FAIL

In general, Prolog has no notion of a predicate not being true! It can only decide whether it can prove the predicate using the information in the database.

This is called “negation as failure”.

It is useful to be able to ask the question: “Are you unable to prove this predicate?” (Is this predicate false?)

Solution:

```
neg(P) :- P, !, fail.
neg(_).
```

NEGATION: CUT AND FAIL

In general, Prolog has no notion of a predicate not being true! It can only decide whether it can prove the predicate using the information in the database.

This is called “negation as failure”.

It is useful to be able to ask the question: “Are you unable to prove this predicate?” (Is this predicate false?)

Solution:

```
neg(P) :- P, !, fail.
neg(_).
```

Example:

```
?- neg(true).
false.
```

NEGATION: CUT AND FAIL

In general, Prolog has no notion of a predicate not being true! It can only decide whether it can prove the predicate using the information in the database.

This is called “negation as failure”.

It is useful to be able to ask the question: “Are you unable to prove this predicate?” (Is this predicate false?)

Solution:

```
neg(P) :- P, !, fail.
neg(_).
```

Example:

```
?- neg(true).
false.
?- neg(false).
true.
```

NEGATION: CUT AND FAIL

In general, Prolog has no notion of a predicate not being true! It can only decide whether it can prove the predicate using the information in the database.

This is called “negation as failure”.

It is useful to be able to ask the question: “Are you unable to prove this predicate?” (Is this predicate false?)

Solution:

```
neg(P) :- P, !, fail.
neg(_).
```

Example:

```
?- neg(true).
false.
?- neg(false).
true.
```

Prolog has a built-in function `\+` that does exactly what `neg` does. Thus, these two queries become `\+ true.` and `\+ false.`

CONTROL FLOW: ONCE

Sometimes, we know that a predicate can match only once or we never need more than one solution.

In these cases, we would like to prevent Prolog from searching for additional solutions, in the interest of efficiency.

CONTROL FLOW: ONCE

Sometimes, we know that a predicate can match only once or we never need more than one solution.

In these cases, we would like to prevent Prolog from searching for additional solutions, in the interest of efficiency.

`once(P)`

- Fails if `P` fails.
- Succeeds if `P` succeeds but finds only one solution.

CONTROL FLOW: ONCE

Sometimes, we know that a predicate can match only once or we never need more than one solution.

In these cases, we would like to prevent Prolog from searching for additional solutions, in the interest of efficiency.

`once(P)`

- Fails if `P` fails.
- Succeeds if `P` succeeds but finds only one solution.

```
a(1). a(2).
?- a(X).
X = 1 ;
X = 2.
```

CONTROL FLOW: ONCE

Sometimes, we know that a predicate can match only once or we never need more than one solution.

In these cases, we would like to prevent Prolog from searching for additional solutions, in the interest of efficiency.

`once(P)`

- Fails if `P` fails.
- Succeeds if `P` succeeds but finds only one solution.

```
a(1). a(2).
?- a(X).
X = 1 ;
X = 2.
```

```
a(1). a(2).
?- once(a(X)).
X = 1.
```

CONTROL FLOW: ONCE

Sometimes, we know that a predicate can match only once or we never need more than one solution.

In these cases, we would like to prevent Prolog from searching for additional solutions, in the interest of efficiency.

`once(P)`

- Fails if `P` fails.
- Succeeds if `P` succeeds but finds only one solution.

```
a(1). a(2).
?- a(X).
X = 1 ;
X = 2.
```

```
a(1). a(2).
?- once(a(X)).
X = 1.
```

Implementation: `once(P) :- call(P), !.`

CONTROL FLOW: -> (1)

Prolog has an if-then construct:

`If -> Then` behaves the same as `once(If), Then`.

Example:

```
a(1). a(2). b(1,3). b(1,4). b(2,5). b(2,6).
p(Y) :- a(X) -> b(X,Y).

?- p(Y).
Y = 3;
Y = 4.
```

CONTROL FLOW: -> (2)

There's also a version that acts like if-then-else: `If -> Then; Else`.

It acts as if implemented as

```
If -> Then; Else :- If, !, Then.
If -> Then; Else :- !, Else.
```

Example:

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

COLLECTING ALL ANSWERS (1)

Backtracking produces the different solutions to a query one at a time.

Sometimes, we may want to collect all solutions.

COLLECTING ALL ANSWERS (1)

Backtracking produces the different solutions to a query one at a time. Sometimes, we may want to collect all solutions.

Finding all solutions:

```
a(1,4).  a(1,3).  a(2,4).  a(2,3).

?- findall((X,Y), a(X,Y), List).
List = [(1,4), (1,3), (2,4), (2,3)].
```

COLLECTING ALL ANSWERS (1)

Backtracking produces the different solutions to a query one at a time. Sometimes, we may want to collect all solutions.

Finding all solutions:

```
a(1,4).  a(1,3).  a(2,4).  a(2,3).

?- findall((X,Y), a(X,Y), List).
List = [(1,4), (1,3), (2,4), (2,3)].

?- findall(Y, a(X,Y), List).
List = [4, 3, 4, 3].
```

COLLECTING ALL ANSWERS (2)

Grouping solutions:

```
a(1,4).  a(1,3).  a(2,4).  a(2,3).

?- bagof(Y, a(X,Y), List).
X = 1, List = [4, 3] ;
X = 2, List = [4, 3].
```

COLLECTING ALL ANSWERS (2)

Grouping solutions:

```
a(1,4).  a(1,3).  a(2,4).  a(2,3).

?- bagof(Y, a(X,Y), List).
X = 1, List = [4, 3] ;
X = 2, List = [4, 3].

?- bagof(Y, X^a(X,Y), List).
List = [4, 3, 4, 3].
```

COLLECTING ALL ANSWERS (3)

Grouping solutions, sorted, without duplicates:

```
a(1,4).  a(1,3).  a(2,4).  a(2,3).
```

```
?- setof(Y, a(X,Y), List).
X = 1, List = [3, 4] ;
X = 2, List = [3, 4].
```

COLLECTING ALL ANSWERS (3)

Grouping solutions, sorted, without duplicates:

```
a(1,4).  a(1,3).  a(2,4).  a(2,3).

?- setof(Y, a(X,Y), List).
X = 1, List = [3, 4] ;
X = 2, List = [3, 4].

?- setof(Y, X^a(X,Y), List).
List = [3, 4].
```

FASTER REASONING: CONSTRAINT PROGRAMMING OVER INTEGER DOMAINS (1)

Add this line to the beginning of your Prolog program or to your `.swiplrc` file to enable constraint programming over integer domains:

```
:- use_module(library(clpfd)).
```

What does it do?

FASTER REASONING: CONSTRAINT PROGRAMMING OVER INTEGER DOMAINS (1)

Add this line to the beginning of your Prolog program or to your `.swiplrc` file to enable constraint programming over integer domains:

```
:- use_module(library(clpfd)).
```

What does it do?

Standard arithmetic:

```
?- X is 4+3.
X = 7.
```

FASTER REASONING: CONSTRAINT PROGRAMMING OVER INTEGER DOMAINS (1)

Add this line to the beginning of your Prolog program or to your `.swiplrc` file to enable constraint programming over integer domains:

```
:- use_module(library(clpfd)).
```

What does it do?

Standard arithmetic:

```
?- X is 4+3.
X = 7.
```

Constraints:

```
?- X #= 4+3.
X = 7.
```

FASTER REASONING: CONSTRAINT PROGRAMMING OVER INTEGER DOMAINS (1)

Add this line to the beginning of your Prolog program or to your `.swiplrc` file to enable constraint programming over integer domains:

```
:- use_module(library(clpfd)).
```

What does it do?

Standard arithmetic:

```
?- X is 4+3.
X = 7.
```

```
?- 7 is X+3.
ERROR: Arguments are not
sufficiently instantiated
```

Constraints:

```
?- X #= 4+3.
X = 7.
```

FASTER REASONING: CONSTRAINT PROGRAMMING OVER INTEGER DOMAINS (1)

Add this line to the beginning of your Prolog program or to your `.swiplrc` file to enable constraint programming over integer domains:

```
:- use_module(library(clpfd)).
```

What does it do?

Standard arithmetic:

```
?- X is 4+3.
X = 7.

?- 7 is X+3.
ERROR: Arguments are not
sufficiently instantiated
```

Constraints:

```
?- X #= 4+3.
X = 7.

?- 7 #= X+3.
X = 4.
```

FASTER REASONING: CONSTRAINT PROGRAMMING OVER INTEGER DOMAINS (2)

Standard comparisons:

```
?- 4 > 3.
true.
```

Constraints:

```
?- 4 #> 3.
true.
```

FASTER REASONING: CONSTRAINT PROGRAMMING OVER INTEGER DOMAINS (2)

Standard comparisons:

```
?- 4 > 3.
true.

?- X > 3.
ERROR: Arguments are not
sufficiently instantiated
```

Constraints:

```
?- 4 #> 3.
true.
```

FASTER REASONING: CONSTRAINT PROGRAMMING OVER INTEGER DOMAINS (2)

Standard comparisons:

```
?- 4 > 3.
true.

?- X > 3.
ERROR: Arguments are not
sufficiently instantiated
```

Constraints:

```
?- 4 #> 3.
true.

?- X #> 3.
X in 3..sup.
```

REPORTING SOLUTIONS

Sometimes, a solution satisfying all the constraints is reported directly:

```
?- X #> 3, X #< 5.
X = 4.
```

REPORTING SOLUTIONS

Sometimes, a solution satisfying all the constraints is reported directly:

```
?- X #> 3, X #< 5.
X = 4.
```

Usually, you need to use `label` to generate a solution/solutions:

```
?- X #> 3, X #< 6.
X in 4..5.
```

REPORTING SOLUTIONS

Sometimes, a solution satisfying all the constraints is reported directly:

```
?- X #> 3, X #< 5.
X = 4.
```

Usually, you need to use `label` to generate a solution/solutions:

```
?- X #> 3, X #< 6.
X in 4..5.
```

```
?- X #> 3, X #< 6, label([X]).
X = 4 ;
X = 5.
```

DOMAIN CONSTRAINTS

The most basic constraint specifies the range of values a variable or a list of variables can take:

```
SudokuCell in 1..9.
```

```
ListOfAllSudokuCells ins 1..9.
```

EQUALITY AND INEQUALITY CONSTRAINTS

```
X #= Y+Z.

W*X #> Y+Z.

X #>= 3.

...
```

ALL-DIFFERENT CONSTRAINTS (1)

We can use

`all_different([X,Y,Z])` or `all_distinct([X,Y,Z])`

to ensure `X`, `Y`, and `Z` are distinct.

ALL-DIFFERENT CONSTRAINTS (1)

We can use

`all_different([X,Y,Z])` or `all_distinct([X,Y,Z])`

to ensure `X`, `Y`, and `Z` are distinct.

This works for any arbitrary list.

ALL-DIFFERENT CONSTRAINTS (1)

We can use

`all_different([X,Y,Z])` or `all_distinct([X,Y,Z])`

to ensure `X`, `Y`, and `Z` are distinct.

This works for any arbitrary list.

Usually, **`all_distinct`** is the better choice.

- **`all_distinct`** propagates more strongly.
- **`all_different`** is (in the short term) more efficient.

ALL-DIFFERENT CONSTRAINTS (2)

```
?- maplist(in, Vs, [1\/3..4, 1..2\/4, 1..2\/4, 1..3, 1..3, 1..6]),
   all_distinct(Vs).
false.
```

ALL-DIFFERENT CONSTRAINTS (2)

```
?- maplist(in, Vs, [1\/3..4, 1..2\/4, 1..2\/4, 1..3, 1..3, 1..6]),
   all_distinct(Vs).
false.
```

```
?- maplist(in, Vs, [1\/3..4, 1..2\/4, 1..2\/4, 1..3, 1..3, 1..6]),
   all_different(Vs).
_896 in 1\/3..4,
all_different([_896, _902, _908, _914, _920, _926]),
_902 in 1..2\/4,
_908 in 1..2\/4,
_914 in 1..3,
_920 in 1..3,
_926 in 1..6.
```

THE DIFFERENCE BETWEEN CONSTRAINT PROGRAMMING AND BACKTRACKING

The standard solution search in Prolog employs backtracking. The order in which different variable assignments are tried depends entirely on the structure of the predicates we specify and may require “imperative” tuning to achieve decent performance.

THE DIFFERENCE BETWEEN CONSTRAINT PROGRAMMING AND BACKTRACKING

The standard solution search in Prolog employs backtracking. The order in which different variable assignments are tried depends entirely on the structure of the predicates we specify and may require “imperative” tuning to achieve decent performance.

`clpfd` employs some low-level wizardry to ensure variables are fixed the moment existing constraints and other variable assignments leave us with only one possible value. This in turn may force other variables to have only one possible value left, so they are fixed in turn, and so on.

THE DIFFERENCE BETWEEN CONSTRAINT PROGRAMMING AND BACKTRACKING

The standard solution search in Prolog employs backtracking. The order in which different variable assignments are tried depends entirely on the structure of the predicates we specify and may require “imperative” tuning to achieve decent performance.

`clpfd` employs some low-level wizardry to ensure variables are fixed the moment existing constraints and other variable assignments leave us with only one possible value. This in turn may force other variables to have only one possible value left, so they are fixed in turn, and so on.

This is called constraint propagation and is at the heart of efficient constraint solvers. Depending on the problem, it can be orders of magnitude faster than simple backtracking.

DETERMINISTIC CLAUSE GRAMMARS

… are Prolog’s means to parse input.

DETERMINISTIC CLAUSE GRAMMARS

… are Prolog’s means to parse input.

We need to talk about them, but not before we introduce context-free grammars in the context of syntatic analysis.