Programming Paradigms

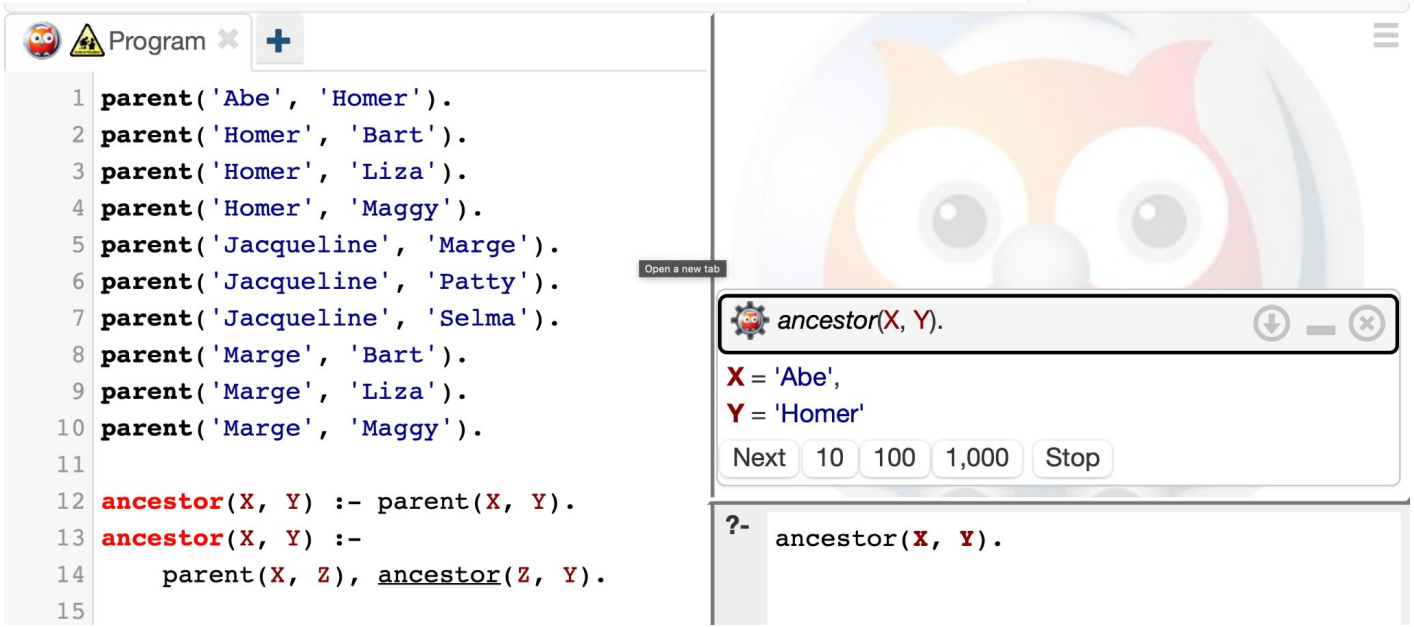
Lecture 10. Introduction to Prolog

Outline

- Recapping Prolog syntax
- Unification
- Proof search
- Recursion

Some Prolog-like programming languages

Development environment



https://swish.swi-prolog.org

Syntax: simple facts

```
person(mia).
person(jack).
person(yolanda).

likes(vincent, mia).
likes(marsellus, mia).
likes(pumpkin, honey_bunny).
likes(honey_bunny, pumpkin).
```

Syntax: clauses and arity

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').
```

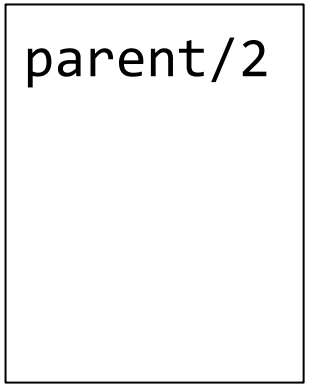
Syntax: clauses and arity

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').
```

parent/2

Syntax: clauses and arity

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

male('Abe').
male('Homer').
male('Bart').
```



Syntax: clauses and arity

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

male('Abe').
male('Homer').
male('Bart').
```

```
parent/2

male/1
```

Syntax: clauses and arity

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

male('Abe').
male('Homer').
male('Bart').

father('Homer') :- male('Homer'), parent('Homer').
```

```
parent/2

male/1
```

Syntax: clauses and arity

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

male('Abe').
male('Homer').
male('Bart').

father('Homer') :- male('Homer'), parent('Homer').
```

```
parent/2

male/1

father/1
```

Syntax: atoms, variables and numbers

1. Atoms — start with lowercase letter, or put in single quotes
   `jack, 'Homer', somethingElse_123`

Syntax: atoms, variables and numbers

1. Atoms — start with lowercase letter, or put in single quotes
   `jack, 'Homer', somethingElse_123`

2. Variables — start with underscore or uppercase letter
   `X, Y, Homer, Parent, Child, _G123, _`

Syntax: atoms, variables and numbers

1. Atoms — start with lowercase letter, or put in single quotes
   `jack, 'Homer', somethingElse_123`

2. Variables — start with underscore or uppercase letter
   `X, Y, Homer, Parent, Child, _G123, _`

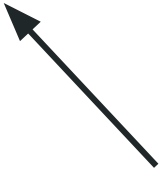
3. Numbers — as usual,
   but we are interested only in integers most of the time
   `12, -5, 0`

Syntax: complex terms

```
functor(term1, term2, ..., termN)
```

Syntax: complex terms

```
functor(term1, term2, ..., termN)
```



Only atom, cannot be a variable!

Syntax: complex terms

```
functor(term1, term2, ..., termN)
```

```
parent(jack, anne)
```

Syntax: complex terms

```
functor(term1, term2, ..., termN)
```

```
parent(jack, anne)
add(1, 2)
```

Syntax: complex terms

```
functor(term1, term2, ..., termN)
```

```
parent(jack, anne)
add(1, 2)
add(1, add(3, 4))
```

Syntax: complex terms

```
functor(term1, term2, ..., termN)
```

```
parent(jack, anne)
add(1, 2)
add(1, add(3, 4))
add(X, add(3, Y))
```

Syntax: complex terms

```
functor(term1, term2, ..., termN)


parent(jack, anne)
add(1, 2)
add(1, add(3, 4))
add(X, add(3, Y))

X(jack)
```

Syntax: complex terms

```
functor(term1, term2, ..., termN)
```

```
parent(jack, anne)
add(1, 2)
add(1, add(3, 4))
add(X, add(3, Y))
```

`X(jack)` — not a valid term

Unification

Consider the three groups of terms:

1. Constants (atoms, numbers) — `hello, 23, 'a book'`
2. Variables — `X, Book, _Example, _`
3. Complex terms — `connected(1, Y), parent('Homer')`

Unification

Consider the three groups of terms:

1. Constants (atoms, numbers) — `hello, 23, 'a book'`
2. Variables — `X, Book, _Example, _`
3. Complex terms — `connected(1, Y), parent('Homer')`

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- hello = hello
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- hello = hello
```

true

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- 42 = 42
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- 42 = 42
```

true

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- 42 = 21+21
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- 42 = 21+21
```

false

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- hello = X
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- hello = X
```

X = hello

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- word(hello) = word(X)
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- word(hello) = word(X)
```

X = hello

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- word(hello) = X
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- word(hello) = X

X = word(hello)
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- parent(X, 'Bart') = parent('Homer', Y)
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- parent(X, 'Bart') = parent('Homer', Y)
```

```
X = 'Homer',
Y = 'Bart'
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- parent(X, 'Bart') = parent('Homer', X)
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- parent(X, 'Bart') = parent('Homer', X)
```

false

Does **not** unify!

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- friend(X, 'Bart') = friend('Bart', X)
```

Unification

Intuitively, two terms unify

1. if they are the same, or
2. if variables in both terms can be instantiated to make the terms the same

```
?- friend(X, 'Bart') = friend('Bart', X)

X = 'Bart'
```

Proof search

```
?- musicalChild(X)
```

```
human('Bart').
human('Liza').

child('Bart').
child('Liza').

playsSax('Liza').

musicalChild(X) :- human(X), child(X), playsSax(X).
```

Proof search

```
?- m(X)
```

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```

Proof search

```
?- m(X)
```

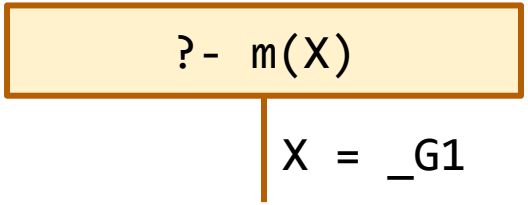
```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```

Proof search

```
?- m(X)
```

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(_G1) :- h(_G1), c(_G1), p(_G1).
```

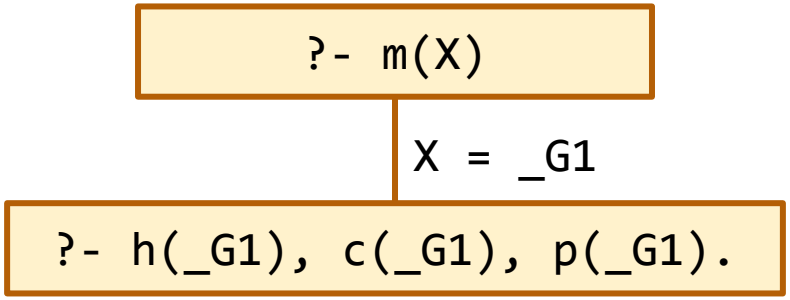
Proof search




```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```

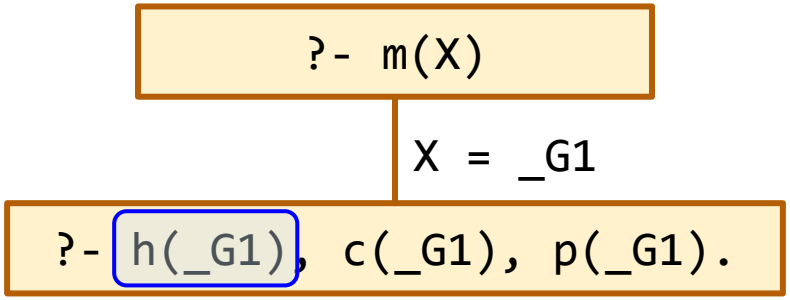
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



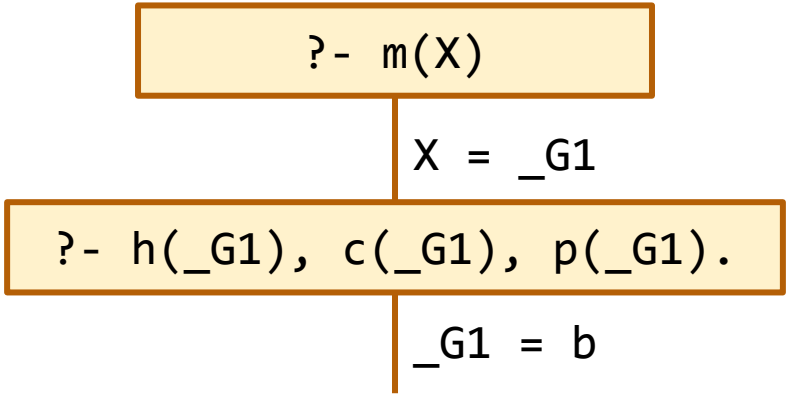
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



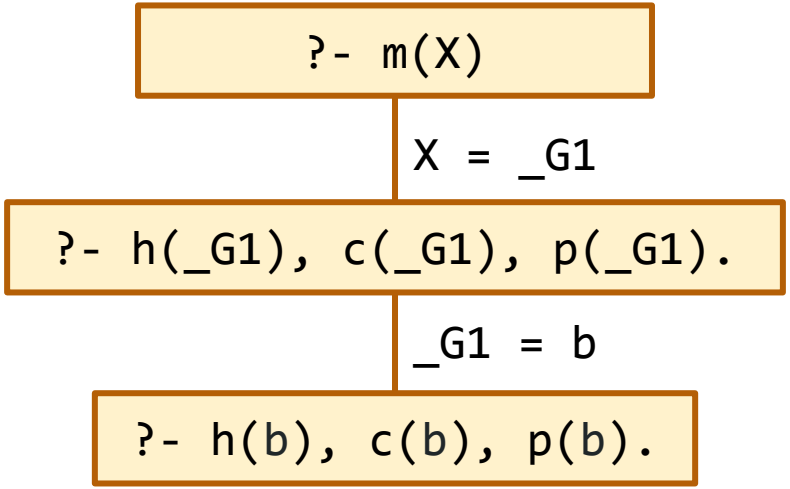
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



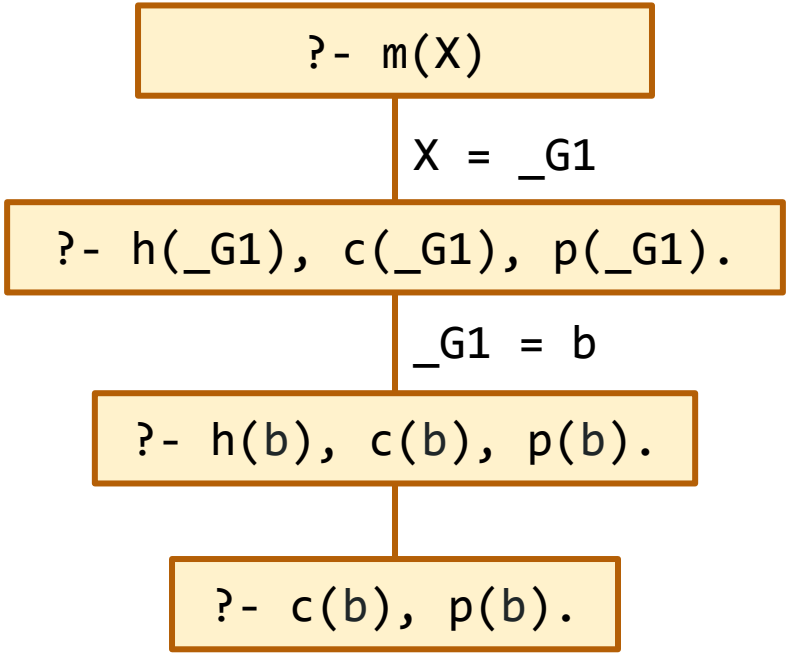
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



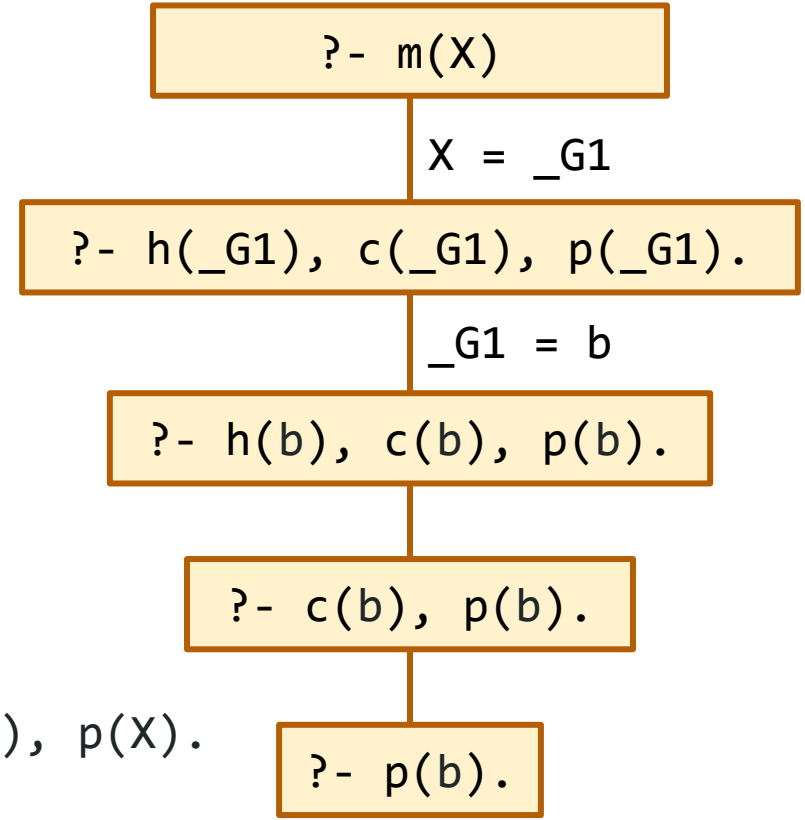
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



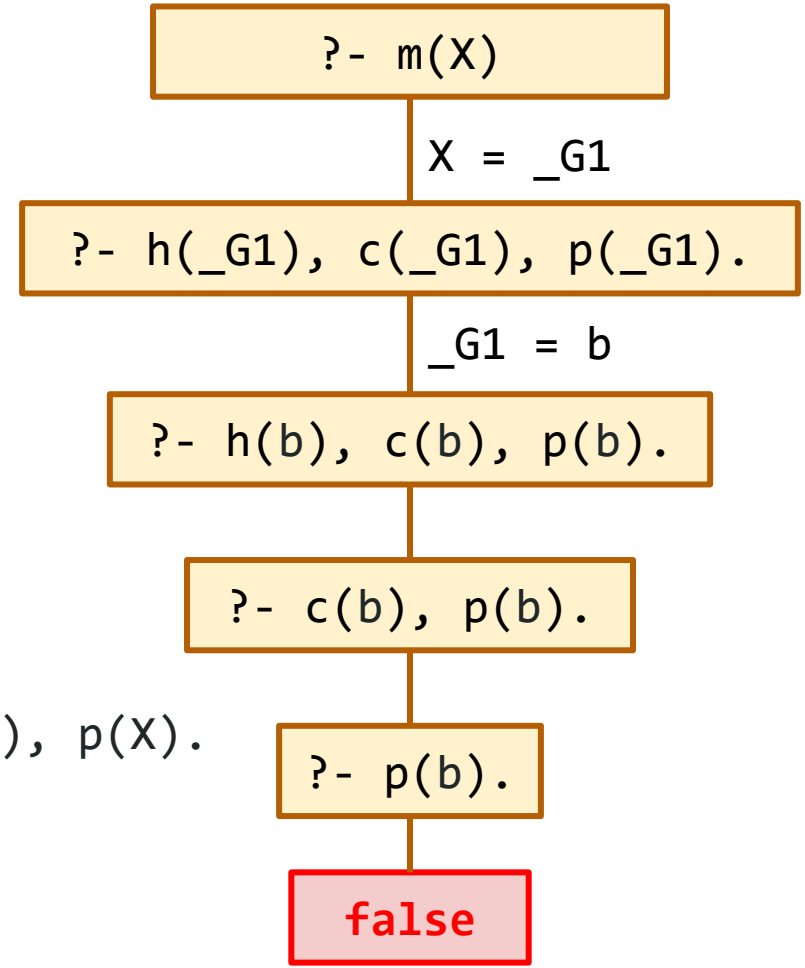
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



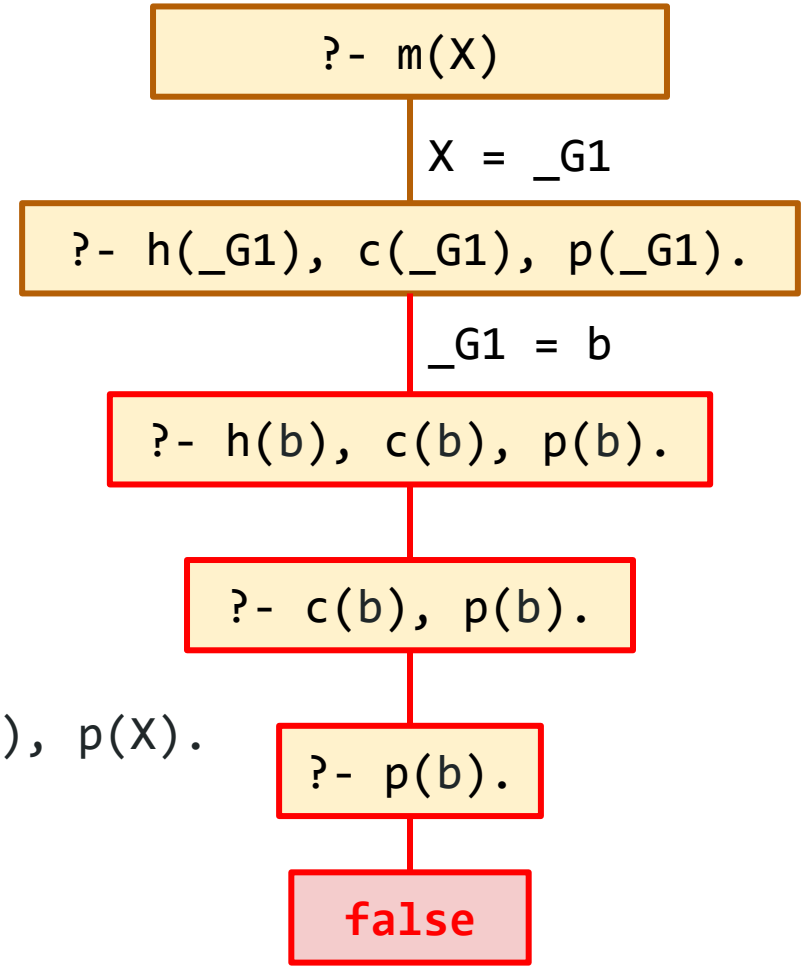
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



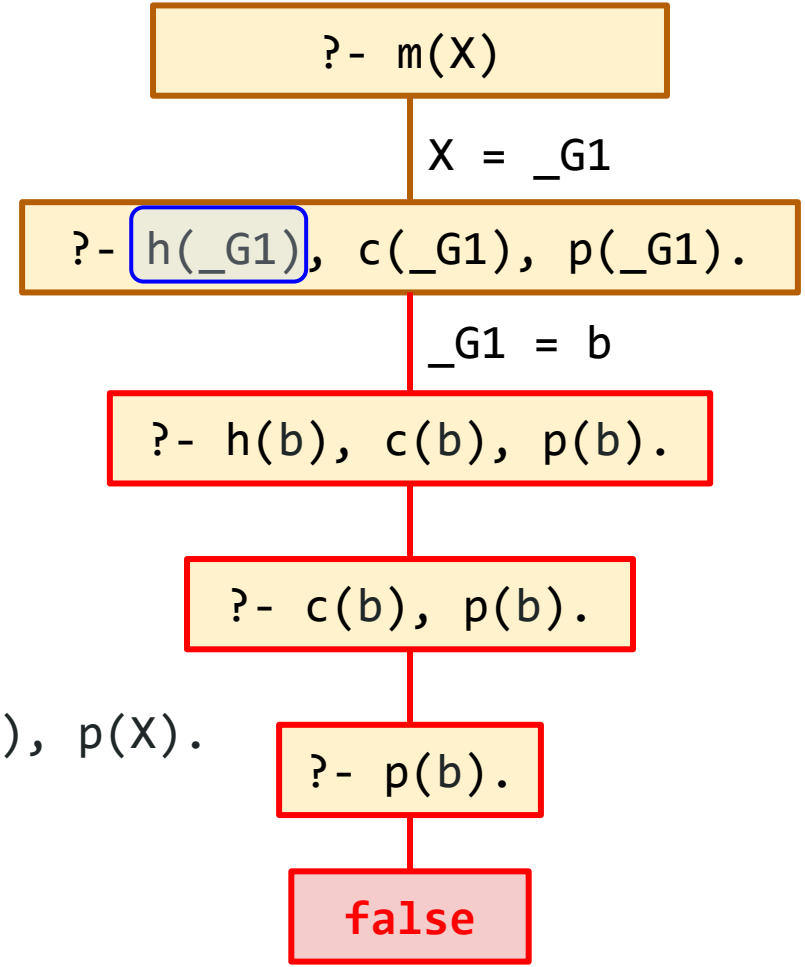
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



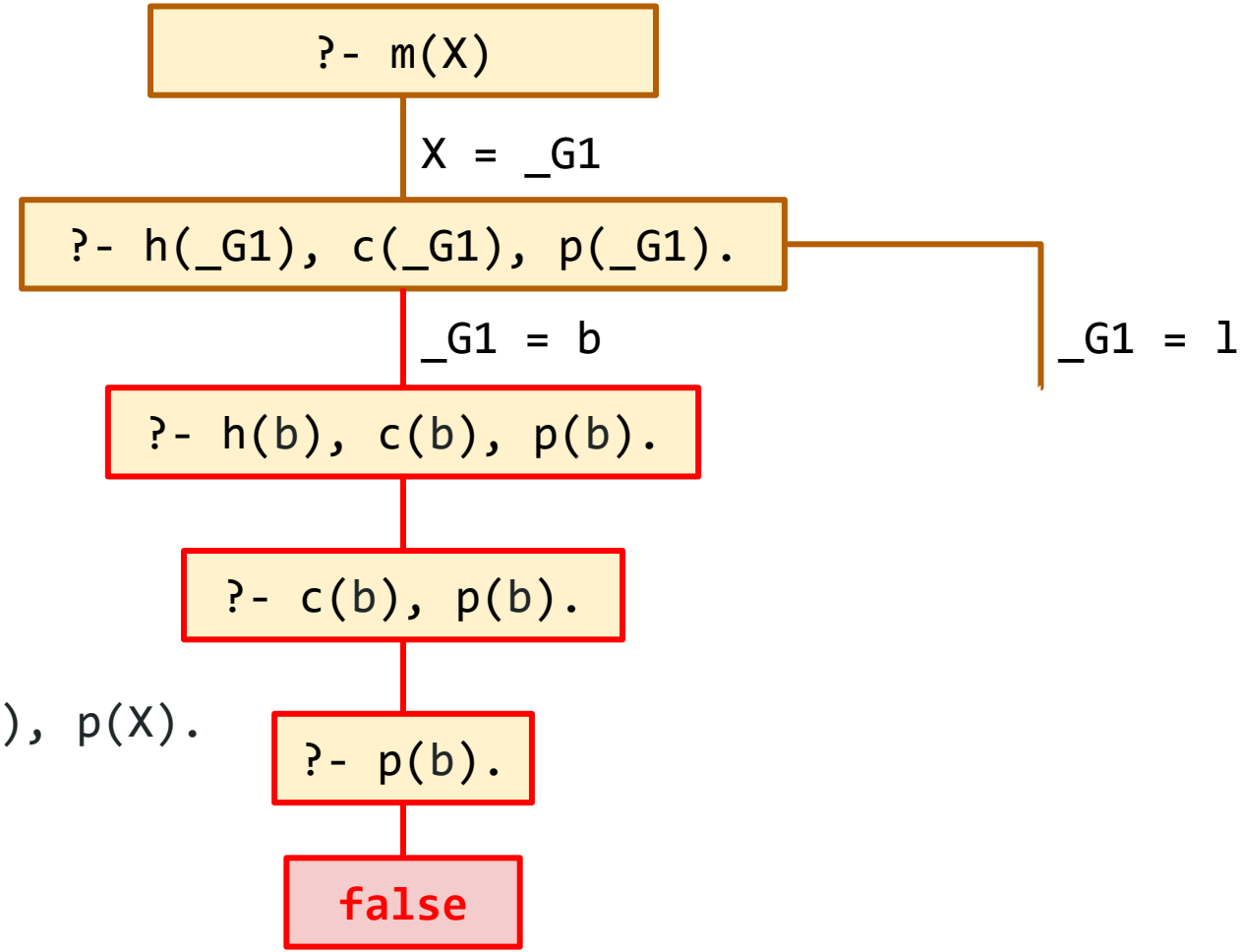
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



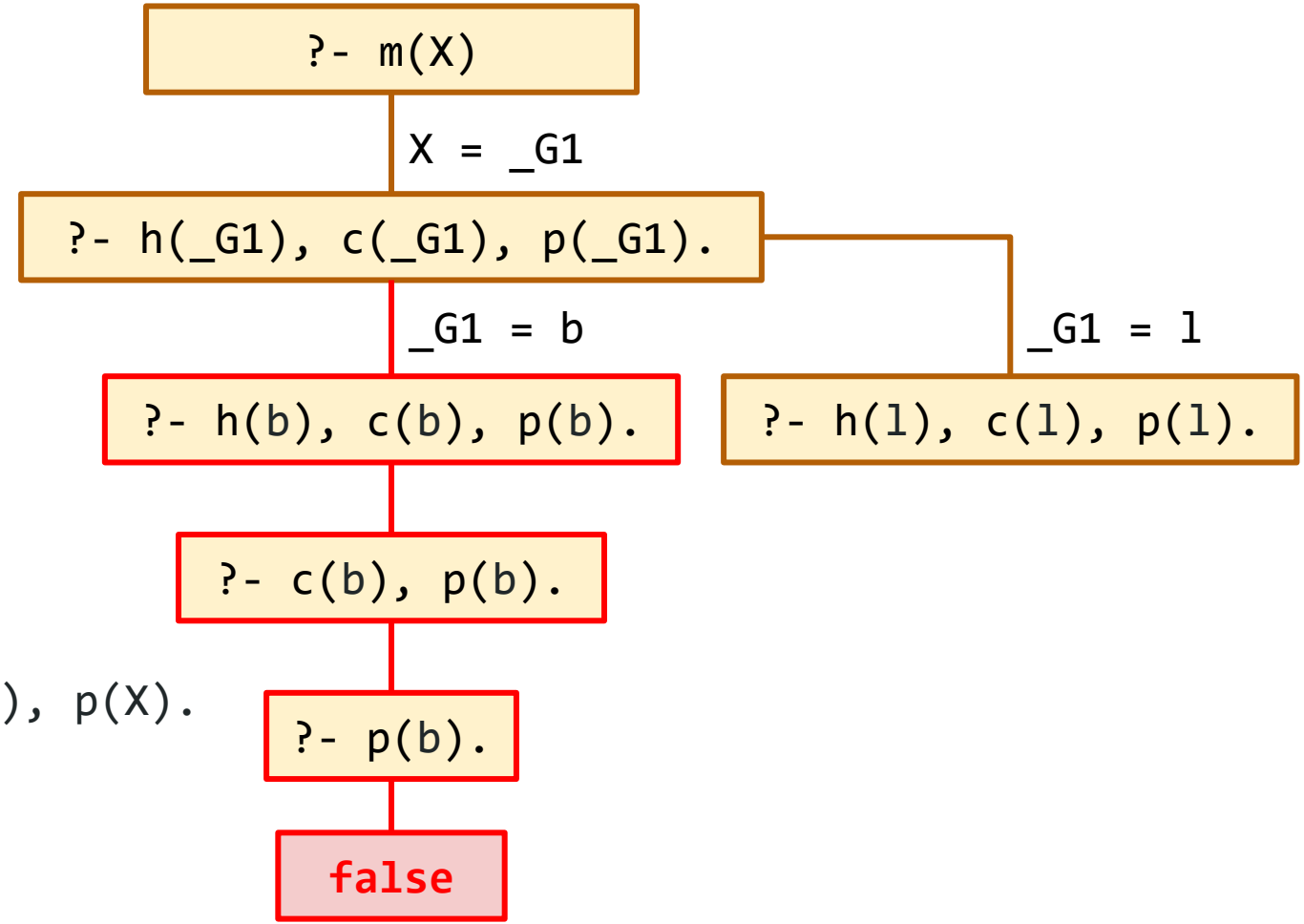
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



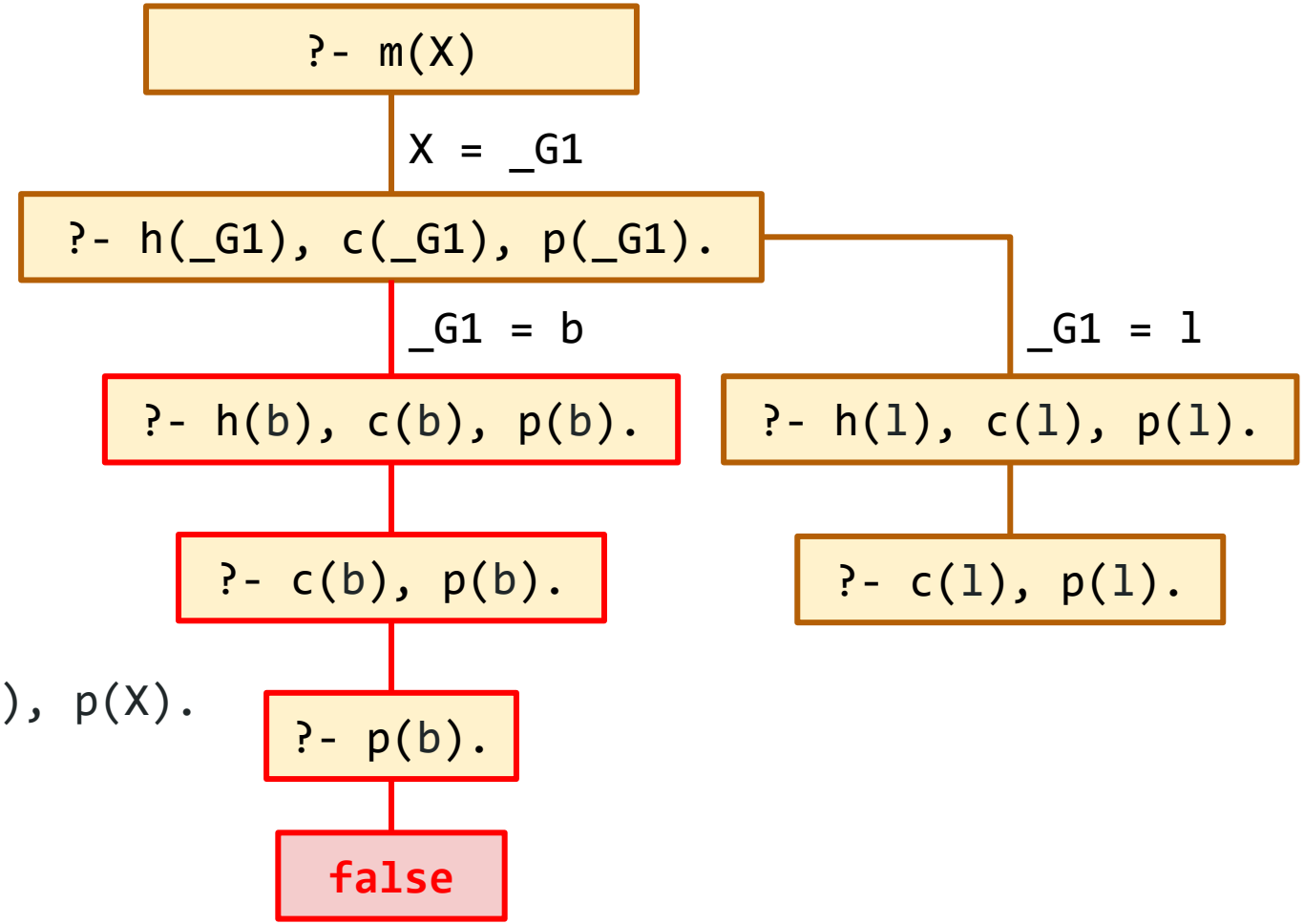
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



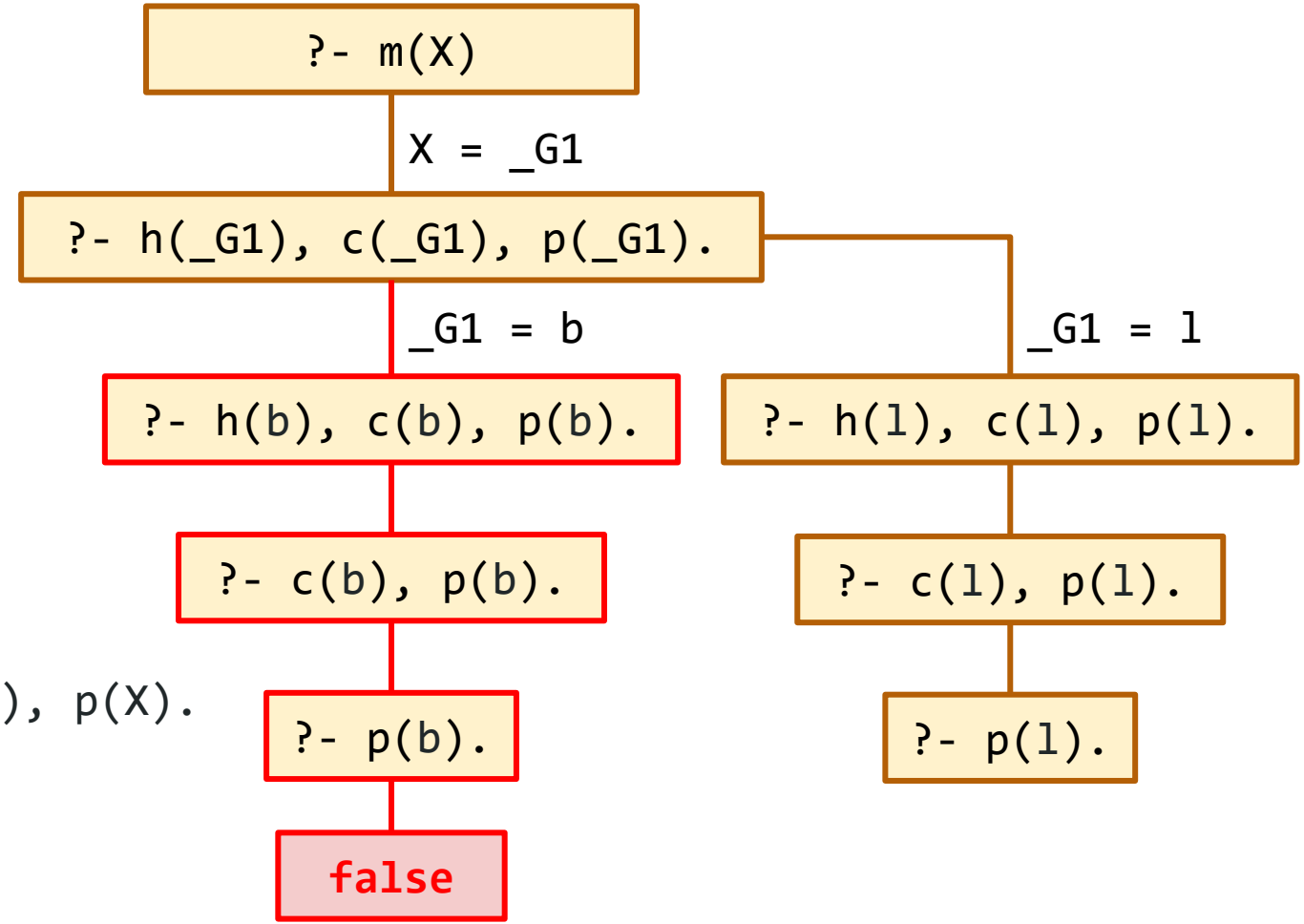
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



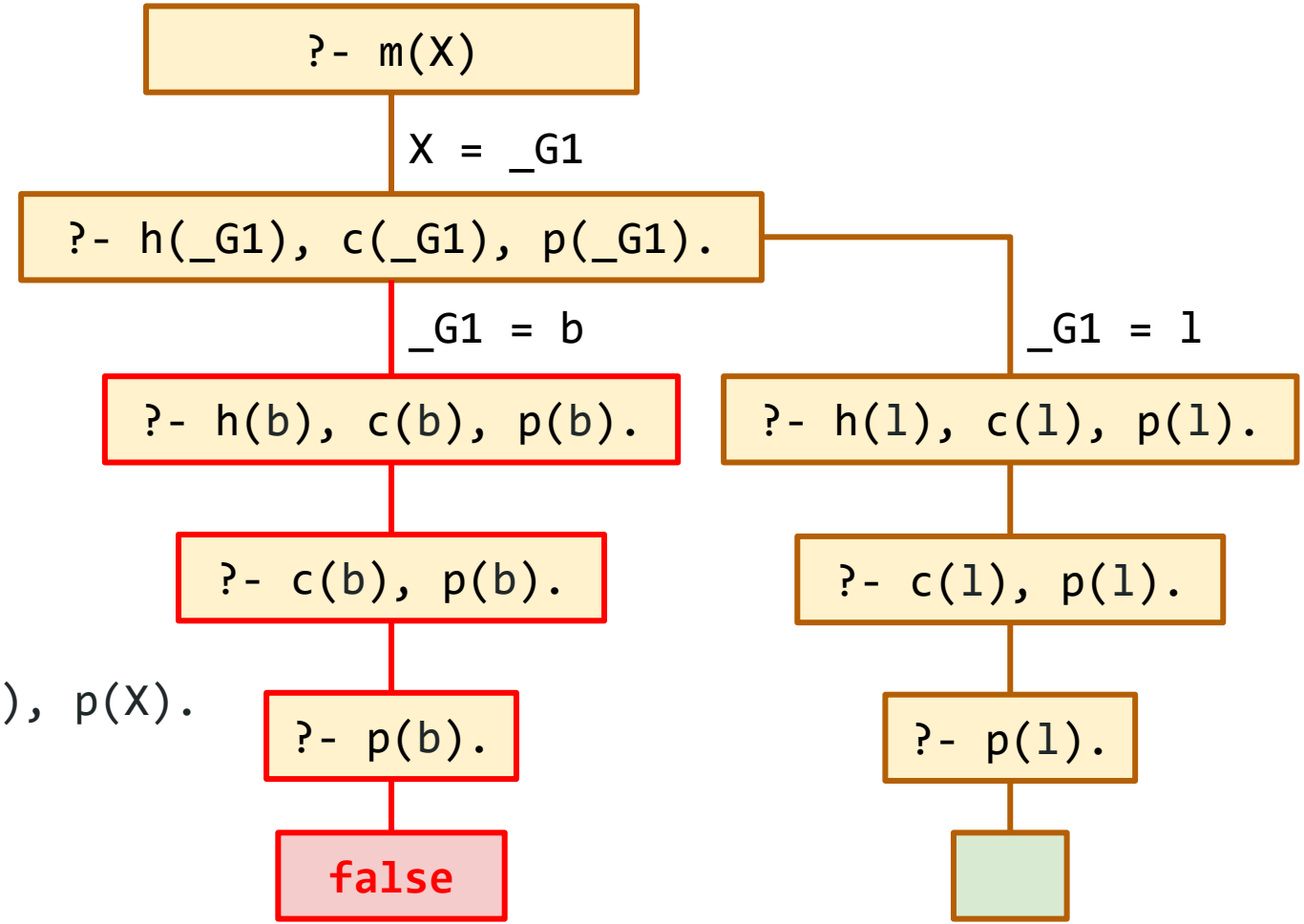
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



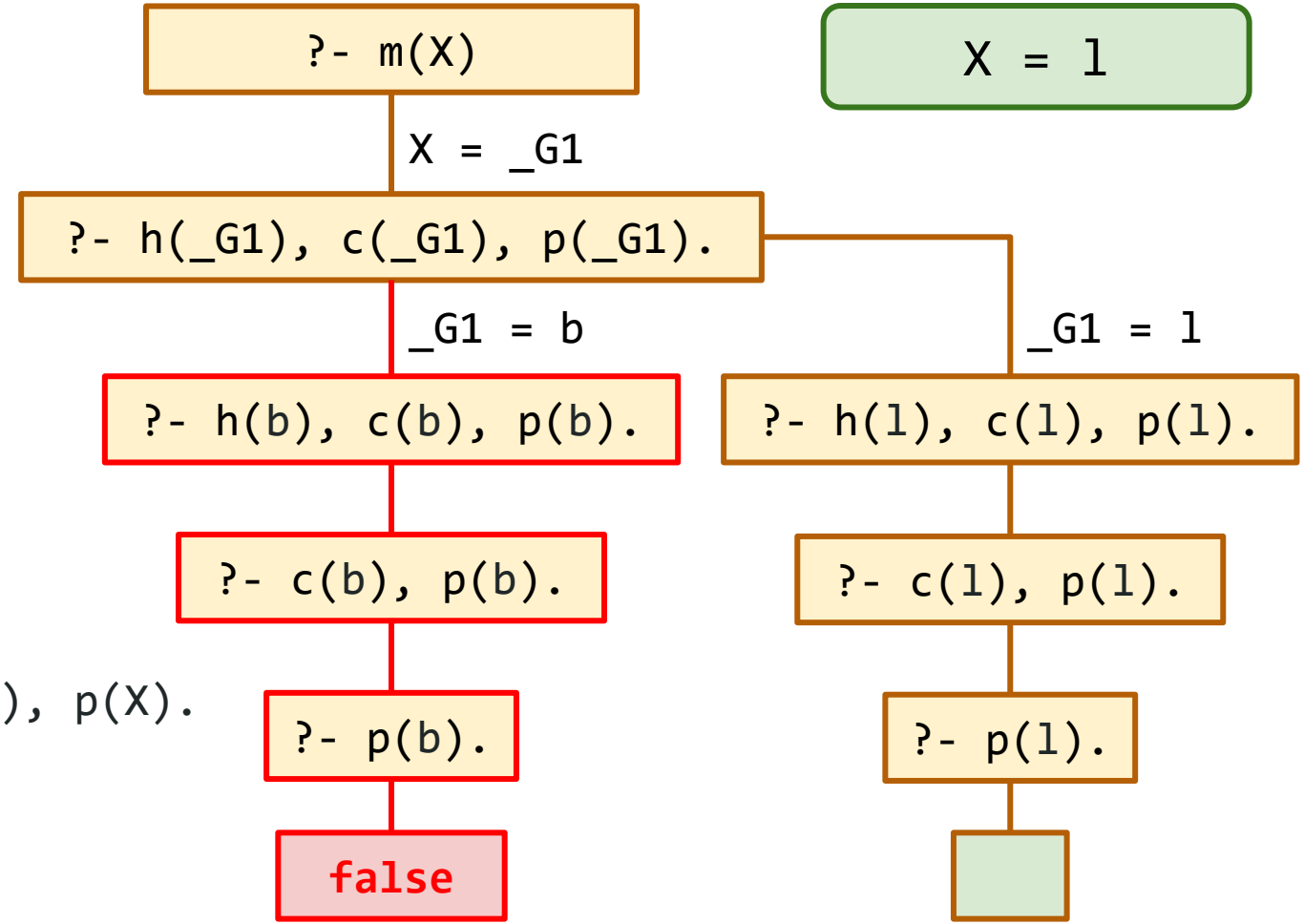
Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



Proof search

```
h(b).
h(l).

c(b).
c(l).

p(l).

m(X) :- h(X), c(X), p(X).
```



Proof search: example 2

```
?- sibling(X, Y)
```

```
parent(a, b).
parent(a, c).

sibling(X, Y) :- parent(P, X), parent(P, Y).
```

Question: Is there a valid answer? Is yes, then how many?

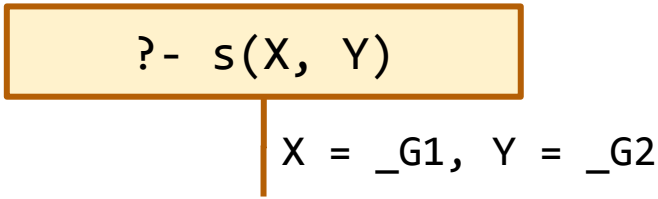
Proof search: example 2

```
?- s(X, Y)
```

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```

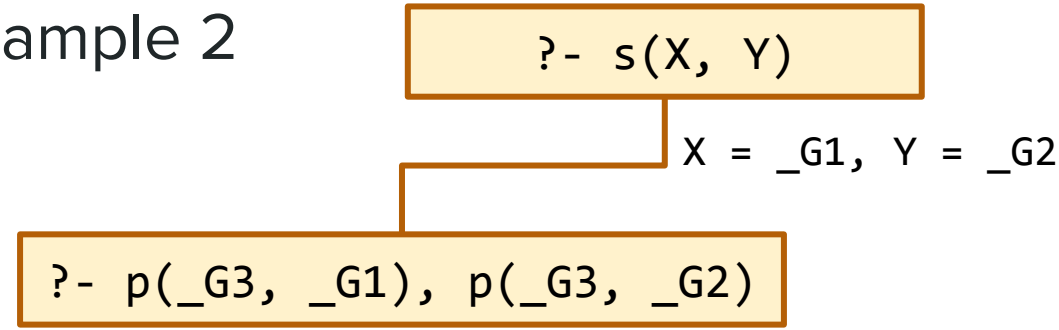
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



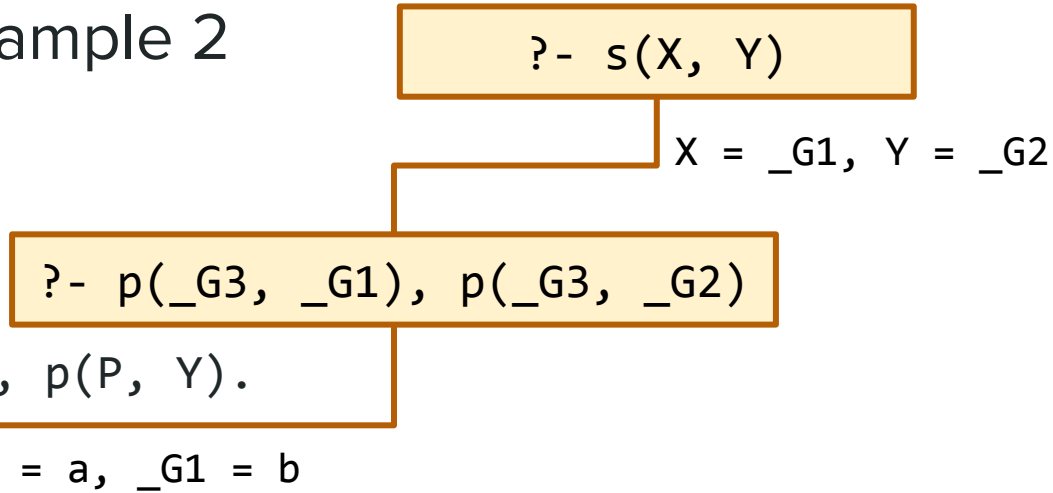
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



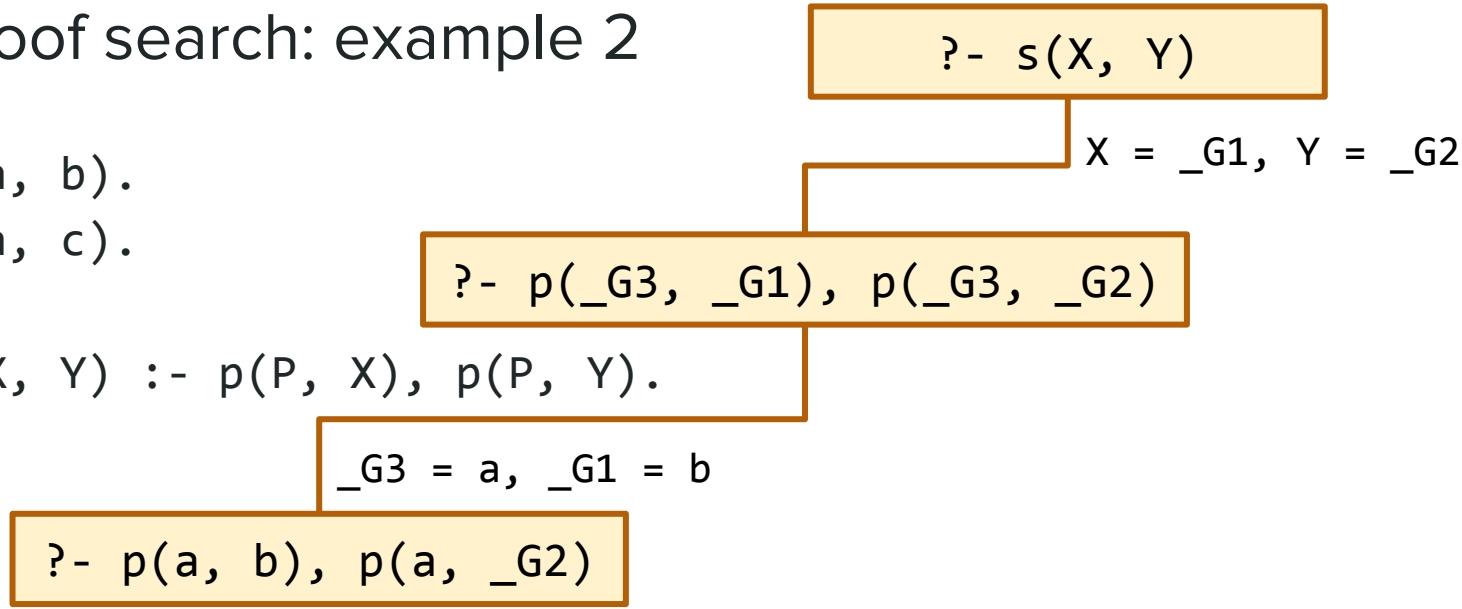
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



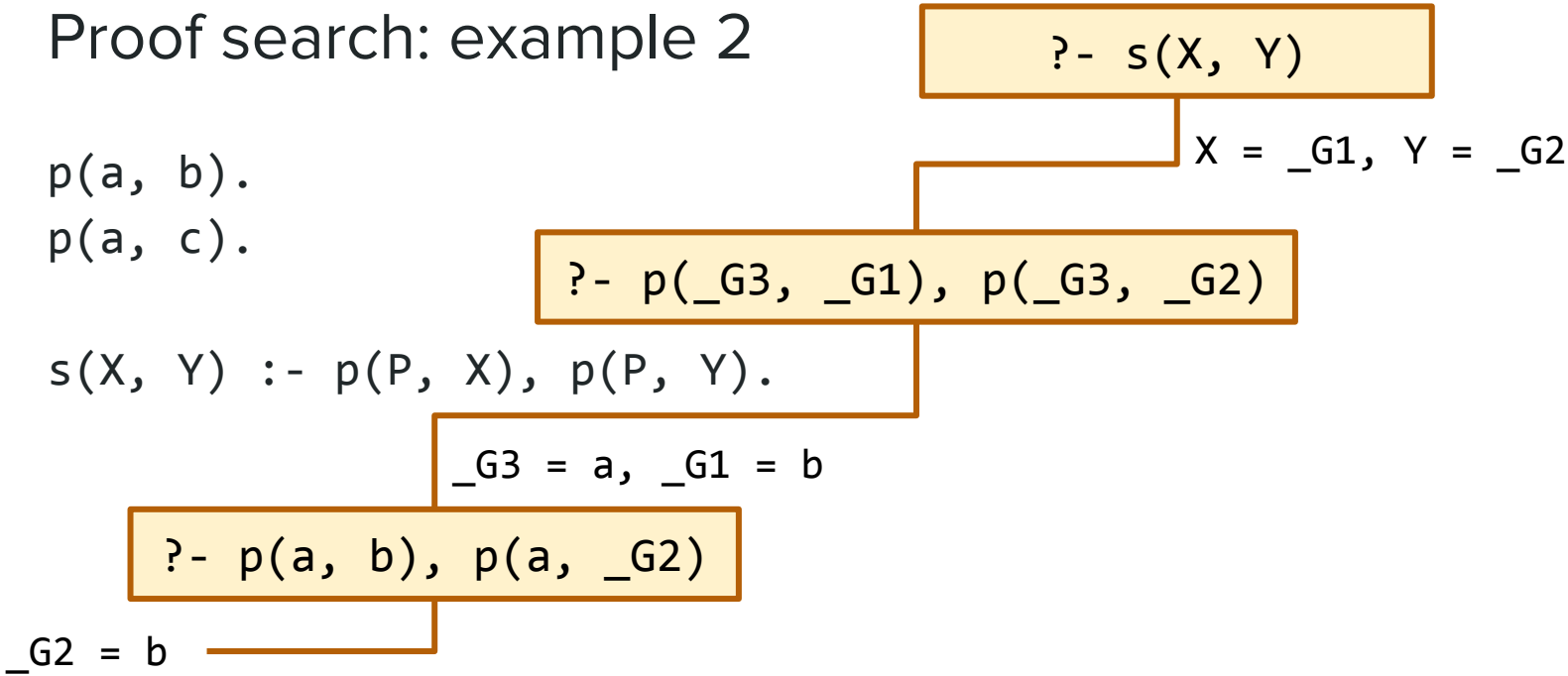
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



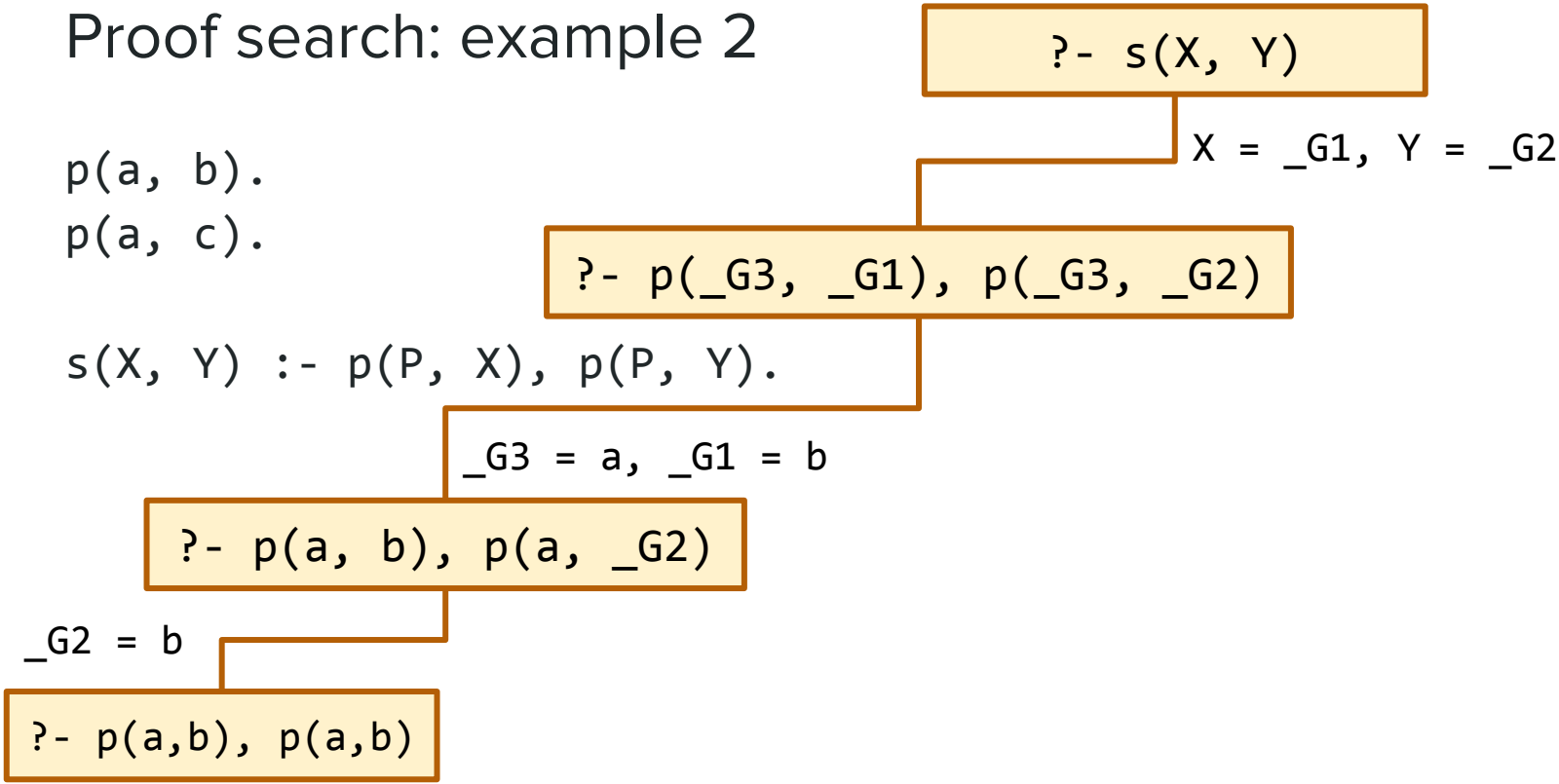
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



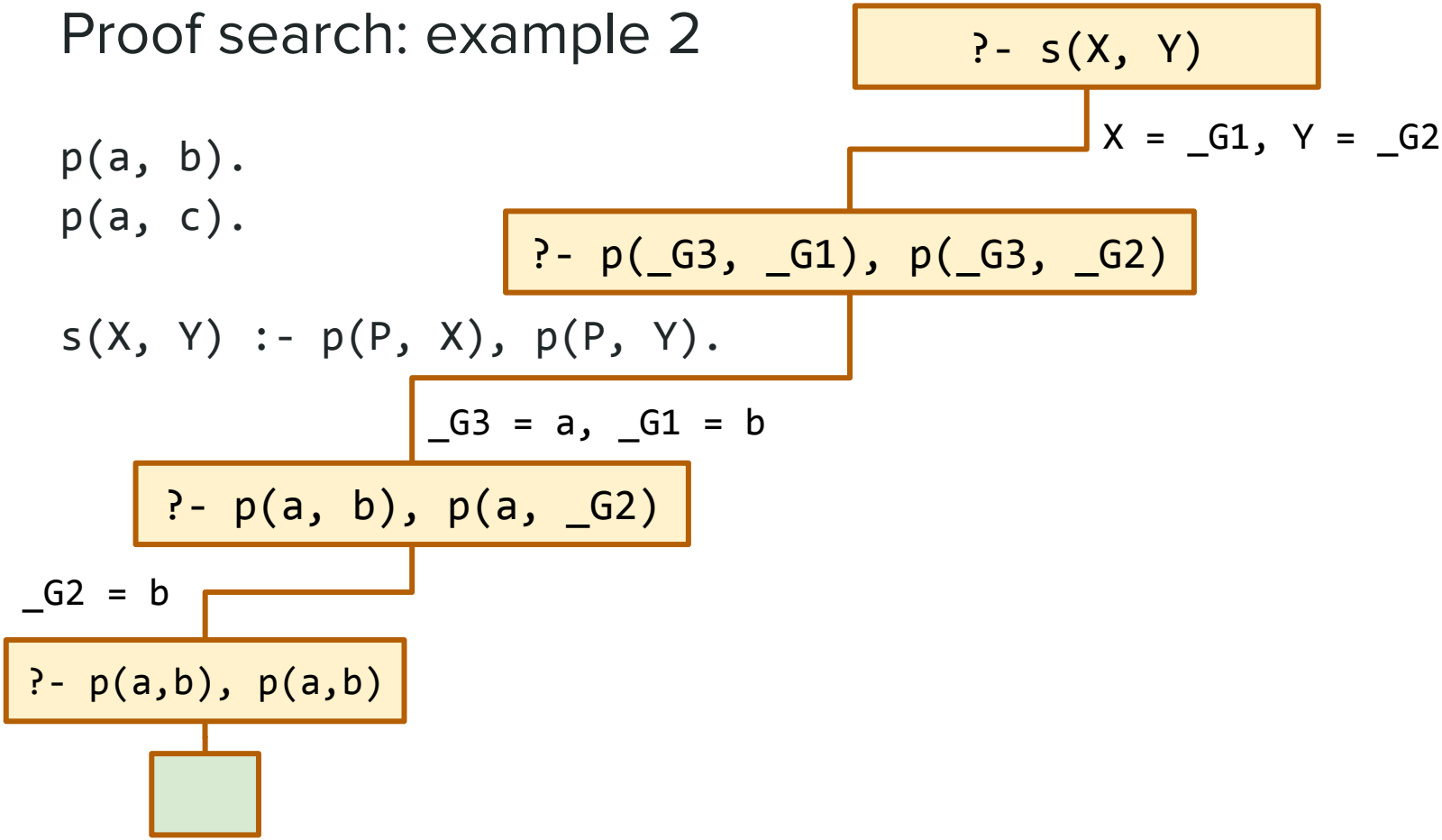
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



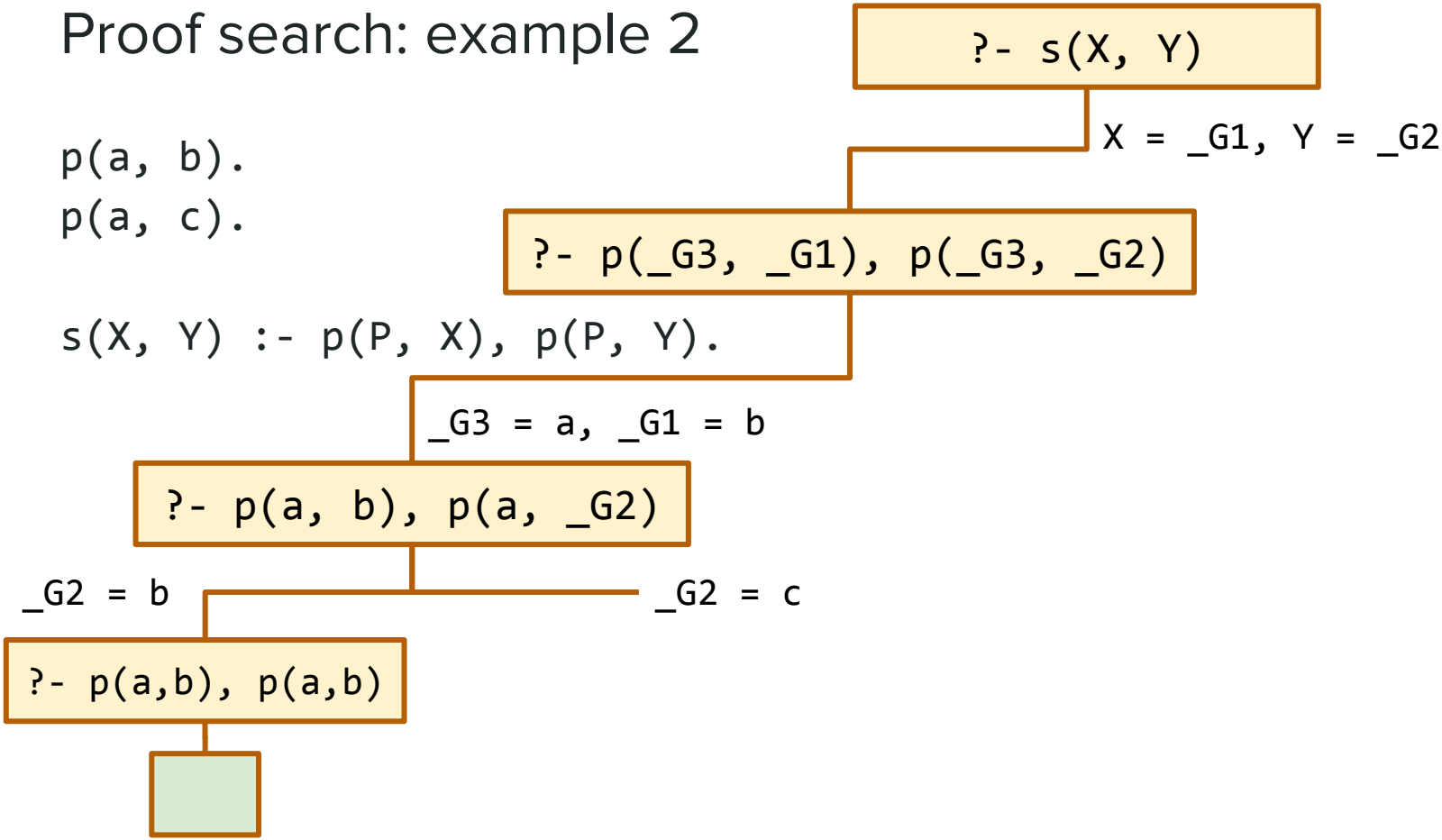
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



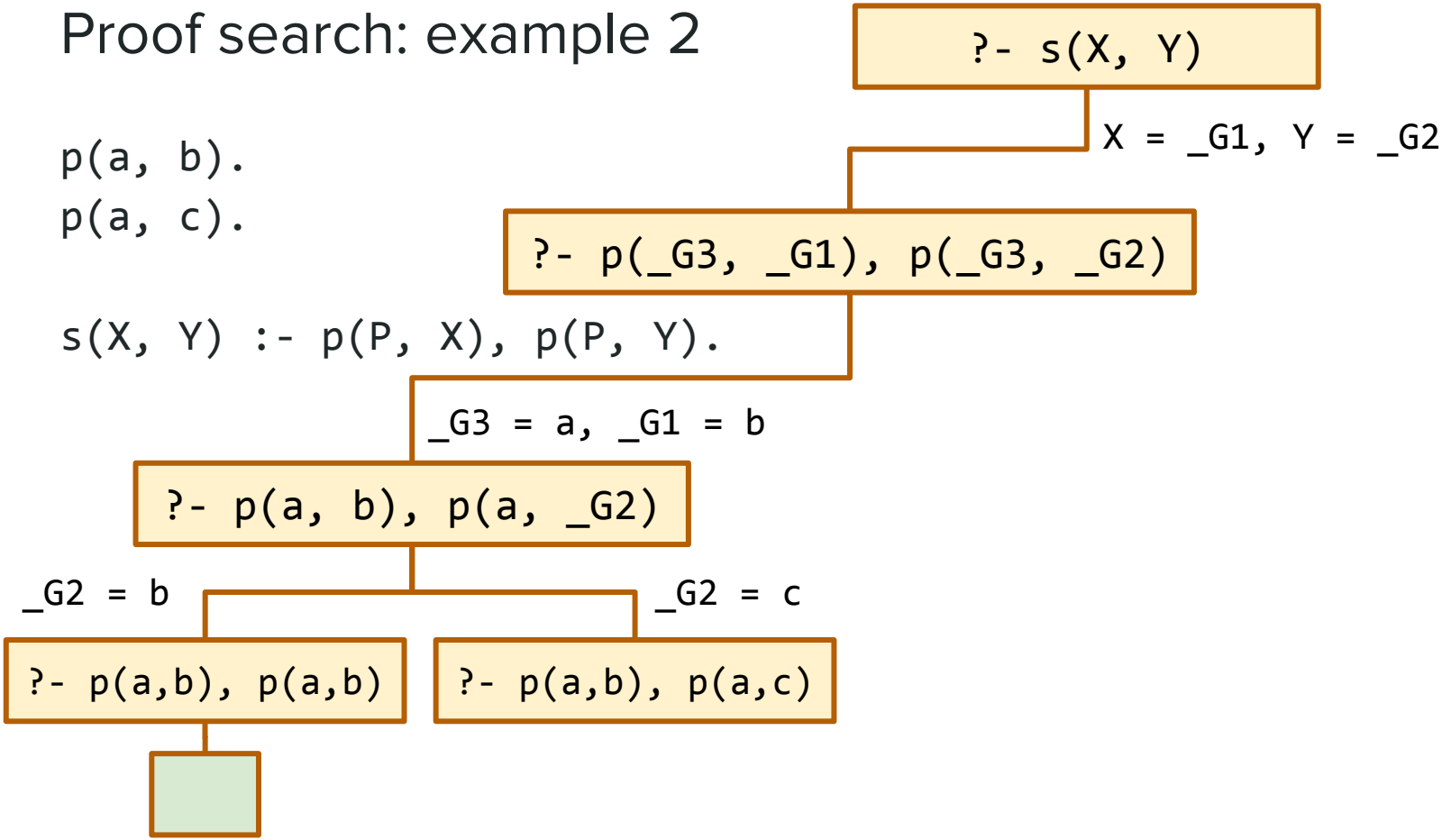
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



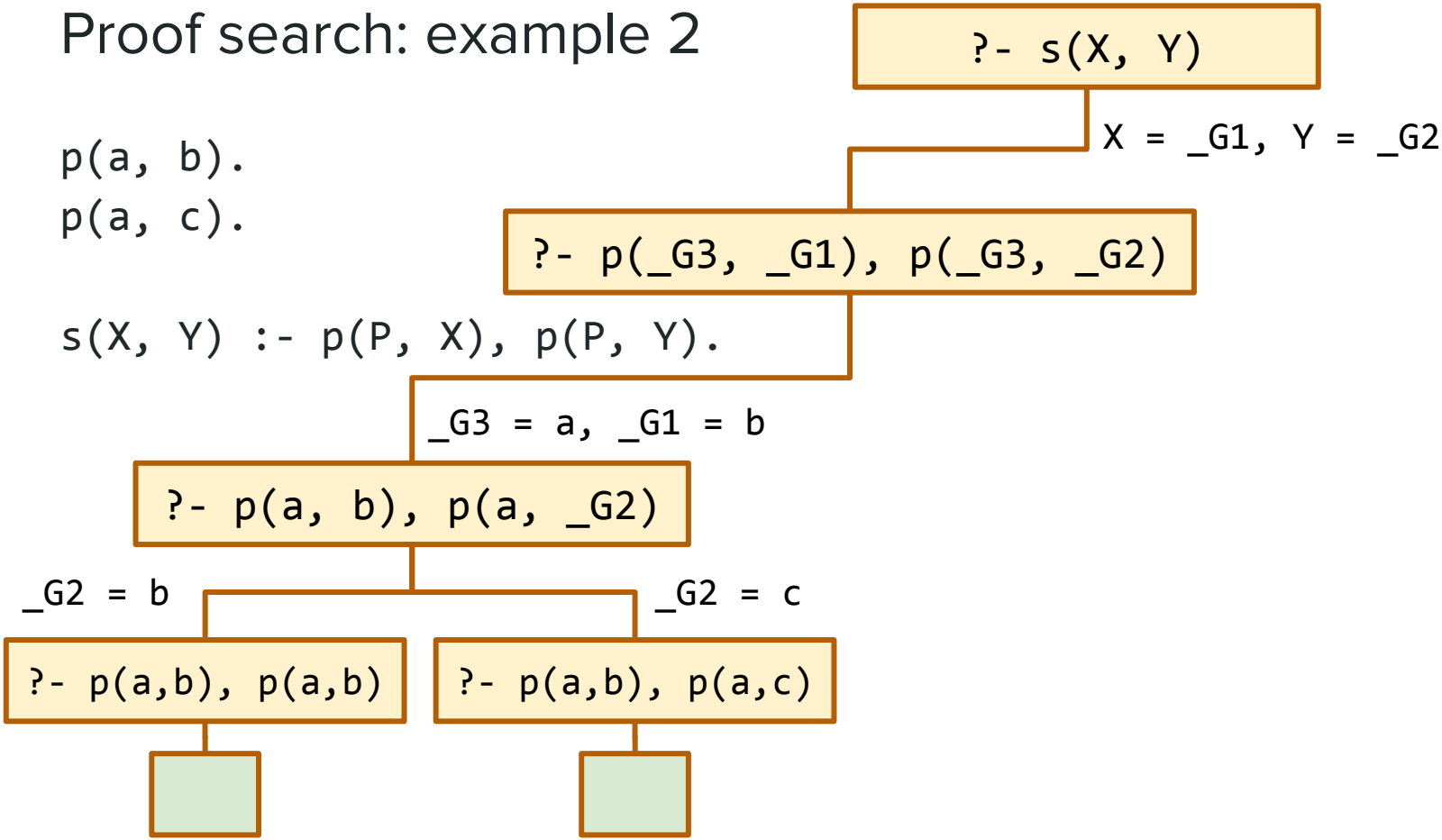
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



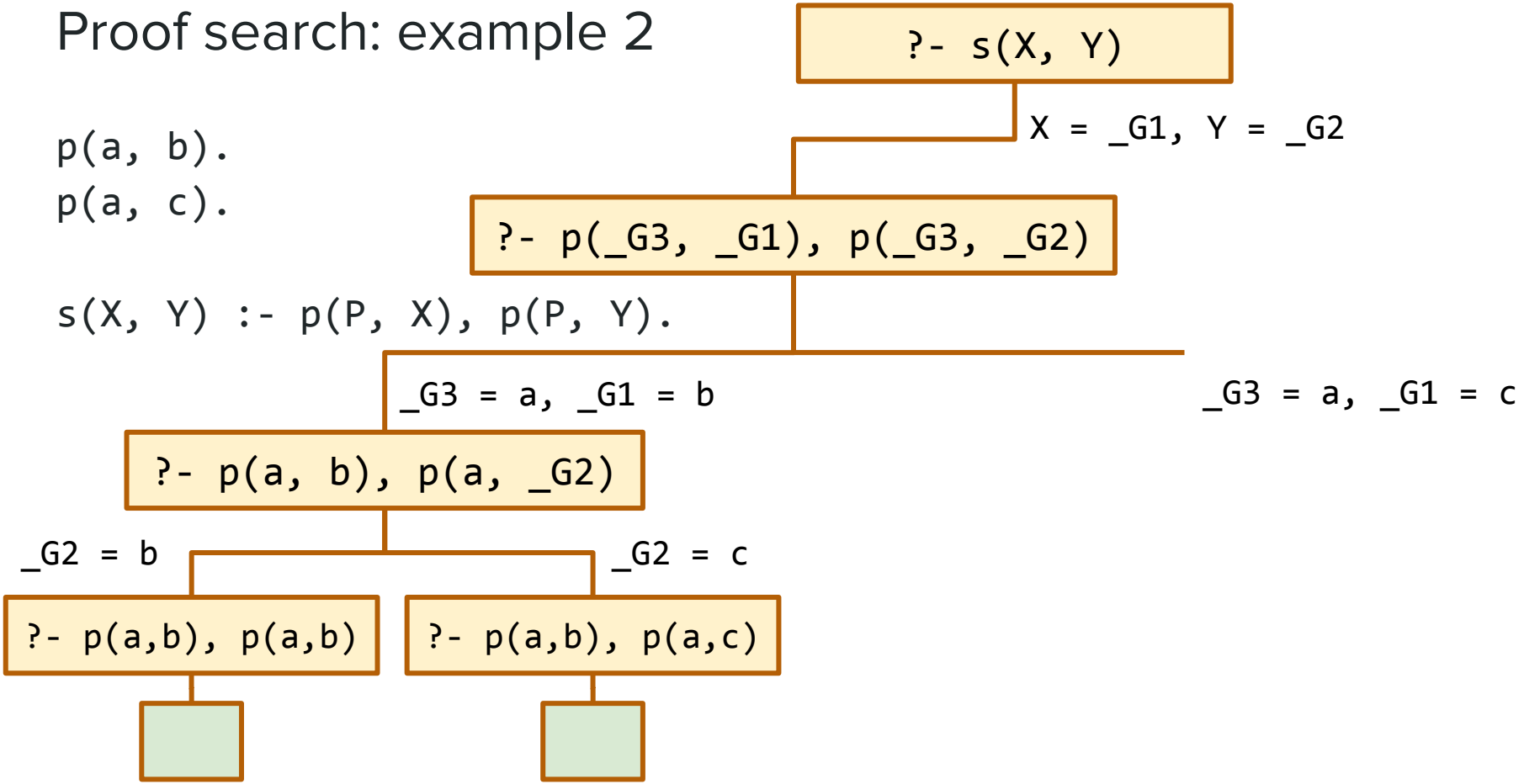
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



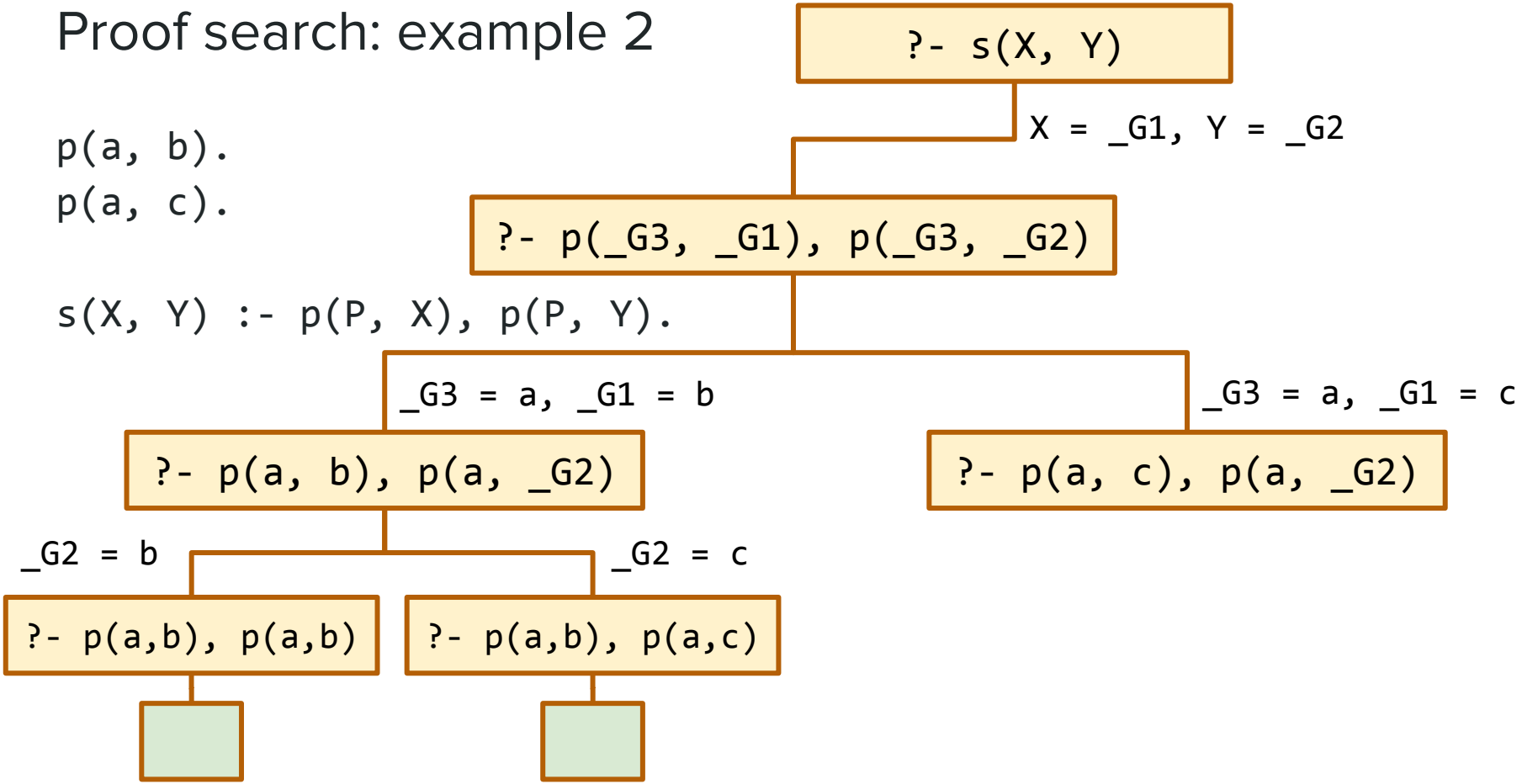
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



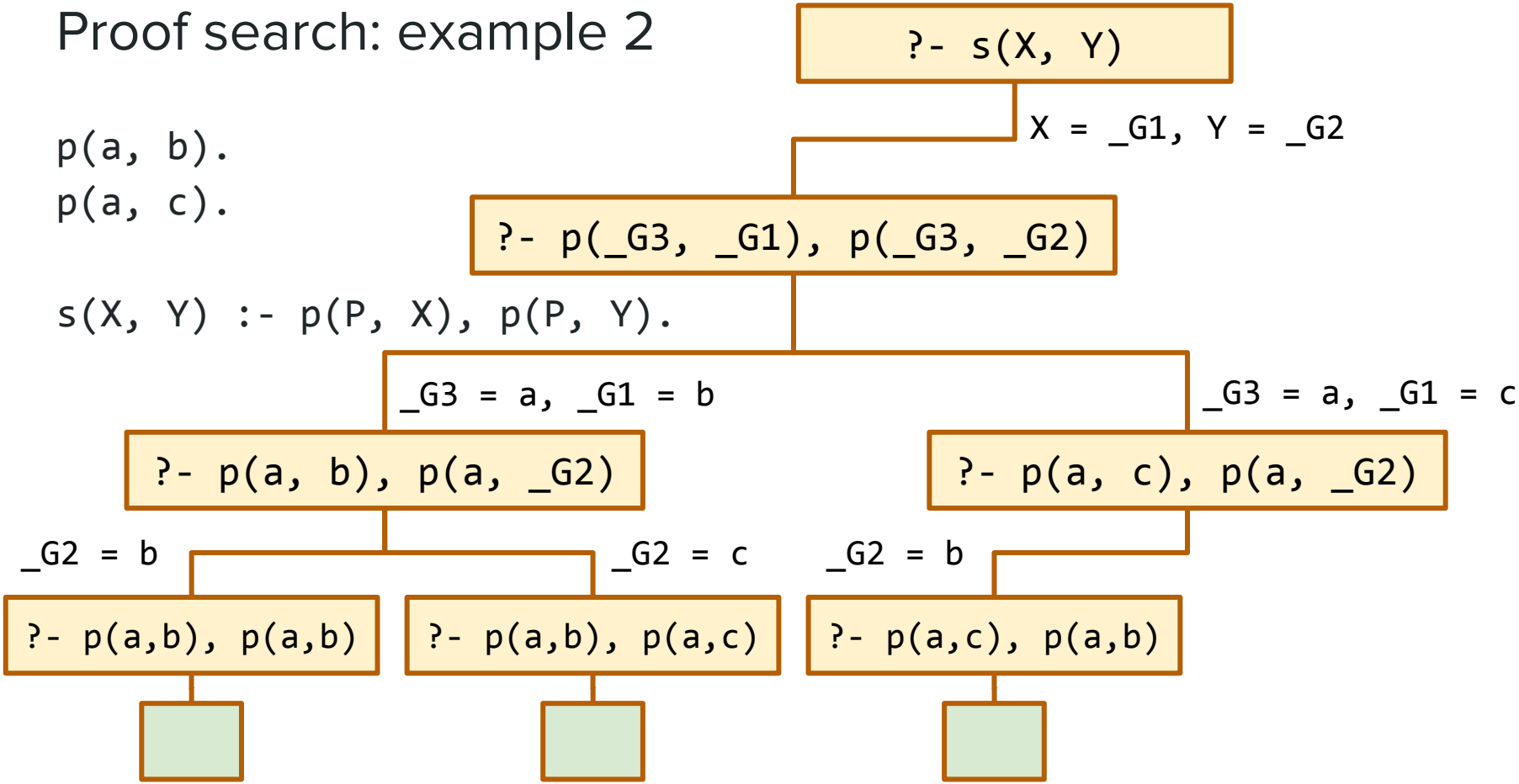
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



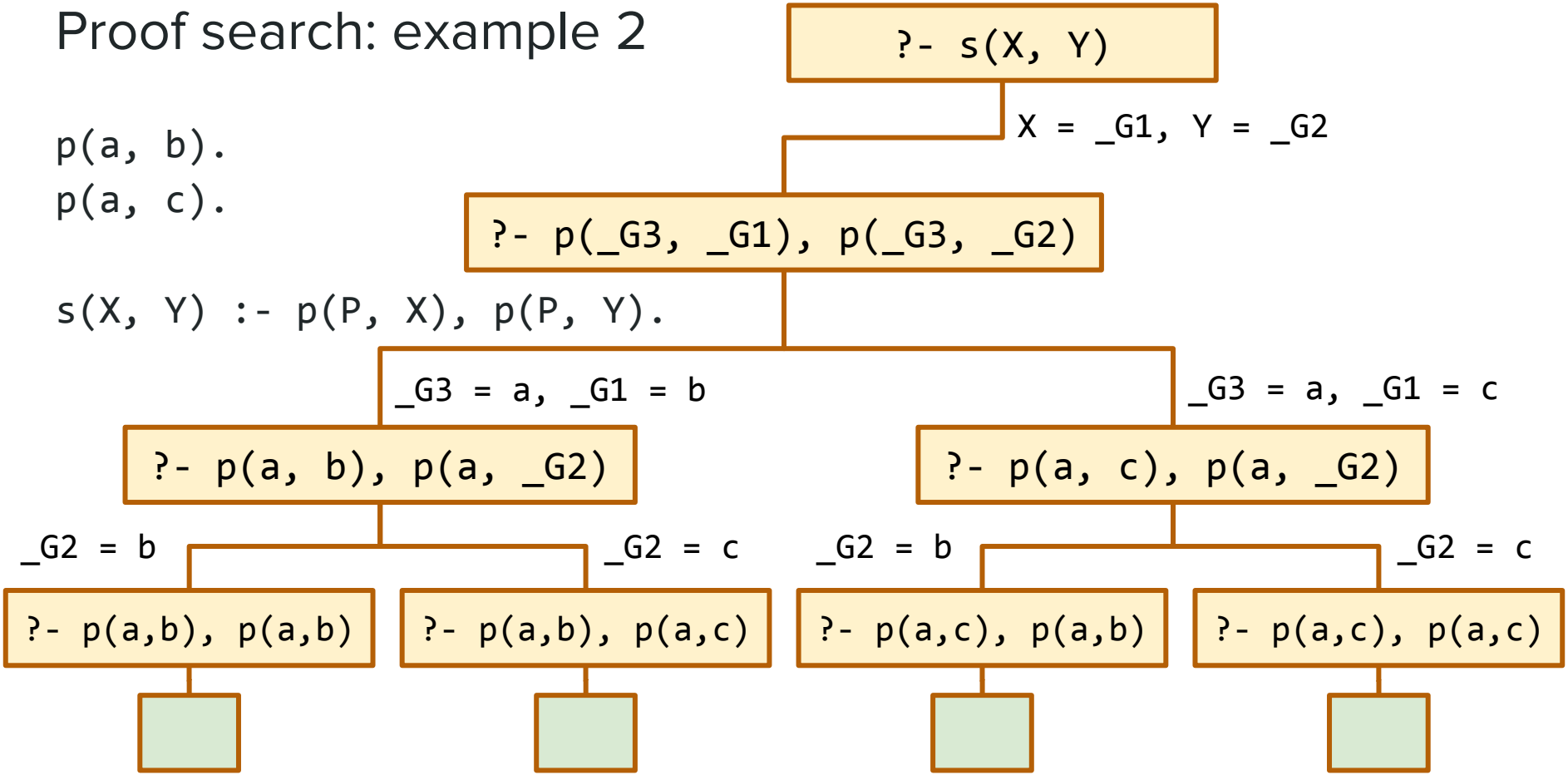
Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



Proof search: example 2

```
p(a, b).
p(a, c).

s(X, Y) :- p(P, X), p(P, Y).
```



Recursion in Prolog: ancestors

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').
```

Recursion in Prolog: ancestors

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

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

Recursion in Prolog: ancestors

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

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

Recursion in Prolog: ancestors

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

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

Recursion in Prolog: ancestors

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

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

Recursion in Prolog: ancestors

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

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

```
?- ancestor(X, Y)
```

Question: Is there a valid answer? Is yes, then how many?

Recursion in Prolog: ancestors

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

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

```
?- ancestor(X, Y)
```

Question: Is there a valid answer? Is yes, then how many?

Recursion in Prolog: ancestors

```
parent('Abe', 'Homer').
parent('Homer', 'Bart').
parent('Homer', 'Liza').
parent('Homer', 'Maggy').

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

```
?- ancestor(X, Y)
```

Question: Is there a valid answer? Is yes, then how many?

Recursion in Prolog: unary numbers

```
unary(z).
unary(s(N)) :- unary(N).
```

Recursion in Prolog: unary numbers

```
unary(z).
unary(s(N)) :- unary(N).


?- unary(s(s(z))).
```

Recursion in Prolog: unary numbers

```
unary(z).
unary(s(N)) :- unary(N).


?- unary(s(s(z))).
true
```

Recursion in Prolog: unary numbers

```
unary(z).
unary(s(N)) :- unary(N).


?- unary(s(s(z))).
true

?- unary(s(s(s))).
```

Recursion in Prolog: unary numbers

```
unary(z).
unary(s(N)) :- unary(N).


?- unary(s(s(z))).
true

?- unary(s(s(s))).
false
```

Recursion in Prolog: unary numbers

```
unary(z).
unary(s(N)) :- unary(N).
```

```
?- unary(s(s(z))).
true
```

```
?- unary(s(s(s))).
false
```

```
?- unary(N)
```

Question: Is there a valid answer? Is yes, then how many?

Recursion in Prolog: adding unary numbers

```
% add(X, Y, X+Y)
```

Recursion in Prolog: adding unary numbers

```
% add(X, Y, X+Y)
add(z, Y, Y).
```

Recursion in Prolog: adding unary numbers

```
% add(X, Y, X+Y)
add(z, Y, Y).
add(s(X), Y, s(R)) :- add(X, Y, R).
```

Recursion in Prolog: adding unary numbers

```
% add(X, Y, X+Y)
add(z, Y, Y).
add(s(X), Y, s(R)) :- add(X, Y, R).

?- add(s(s(z)), s(z), R).
```

Recursion in Prolog: adding unary numbers

```
% add(X, Y, X+Y)
add(z, Y, Y).
add(s(X), Y, s(R)) :- add(X, Y, R).

?- add(s(s(z)), s(z), R).
R = s(s(s(z))
```

Recursion in Prolog: adding unary numbers

```
% add(X, Y, X+Y)
add(z, Y, Y).
add(s(X), Y, s(R)) :- add(X, Y, R).

?- add(s(s(z)), s(z), R).
R = s(s(s(z))

?- add(s(s(z)), Y, s(s(s(z)))).
```

Recursion in Prolog: adding unary numbers

```
% add(X, Y, X+Y)
add(z, Y, Y).
add(s(X), Y, s(R)) :- add(X, Y, R).

?- add(s(s(z)), s(z), R).
R = s(s(s(z))

?- add(s(s(z)), Y, s(s(s(z)))).
Y = s(z)
```

Recursion in Prolog: adding unary numbers

```
% add(X, Y, X+Y)
add(z, Y, Y).
add(s(X), Y, s(R)) :- add(X, Y, R).

?- add(s(s(z)), s(z), R).
R = s(s(s(z))

?- add(s(s(z)), Y, s(s(s(z)))).
Y = s(z)
```

```
?- add(X, Y, s(s(s(z))))
```

Question: Is there a valid answer? Is yes, then how many?

Homework (self-study)

1. Read **Chapters 4 and 5** of Learn Prolog Now!
   http://www.let.rug.nl/bos/lpn/lpnpage.php?pagetype=html&pageid=lpn-htmlch1
2. Work through the **exercises** from both chapters, using SWISH
   https://swish.swi-prolog.org

What was the most unclear part of the lecture for you?

See Moodle

References

1. Learn Prolog Now!