Programming Languages

SLD Resolution
Prolog

SLD Resolution

Operational semantics of Prolog

Extra-logical aspects

General resolution

Recall

To determine whether a first-order formula σ is valid:

1. Convert its negation $\neg\sigma$ to clausal form.

   This yields a set $\mathcal{C}$ of clauses such that
   $\neg\sigma$ is satisfiable iff $\mathcal{C}$ is satisfiable.

2. Repeatedly apply the resolution rule:

$$\frac{\{\sigma_1, \ldots, \sigma_p, \ell_1, \ldots, \ell_n\} \quad \{\neg\tau_1, \ldots, \neg\tau_q, \ell'_1, \ldots, \ell'_m\} \quad (p, q > 0) \qquad \mathbf{S} = \mathrm{mgu}(\{\sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{?}{=} \ldots \stackrel{?}{=} \sigma_p \stackrel{?}{=} \tau_1 \stackrel{?}{=} \tau_2 \stackrel{?}{=} \ldots \stackrel{?}{=} \tau_q\})}{\mathbf{S}(\{\ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_m\})}$$

3. If the empty clause is reached, $\neg\sigma$ is unsatisfiable and σ is valid.

4. The method may not terminate.

General resolution

Applying the general resolution method can be very costly.
Each step requires using search and selection criteria:

Search.	Choose two clauses.
Selection.	Choose a subset of literals from each clause.

The number of options is exponential in the problem size.

Furthermore, each step adds a new clause.

Furthermore, unification equations must be solved at each step.

Furthermore, the method requires using *breadth-first search* (BFS).

SLD Resolution

We will see a variant of general resolution, **SLD resolution**.
It is a *tradeoff*: less generality in exchange for greater efficiency.

Less generality

It cannot be applied to arbitrary first-order formulas.
It can only be applied to **Horn clauses**.

Greater efficiency

The search/selection options are reduced.

Horn clauses

Recall that a clause is a set of literals:

$$\{\ell_1, \ldots, \ell_n\}$$

where each literal is a possibly negated atomic formula:

$$\ell ::= \underbrace{\mathbf{P}(t_1, \ldots, t_n)}_{\text{positive literal}} \mid \underbrace{\neg\mathbf{P}(t_1, \ldots, t_n)}_{\text{negative literal}}$$

Definition (Horn clauses)

Clauses are of the following types, depending on the number of positive/negative literals they contain:

	#positive	#negative
goal clause	0	$*$
definition clause	1	$*$
Hornclause	≤ 1	$*$

Horn clauses

Example — Horn clauses

$$\underbrace{\underbrace{\{\mathbf{P}(\mathtt{X})\} \quad \{\mathbf{P}(\mathtt{X}), \neg\mathbf{Q}(\mathtt{Y},\mathtt{X}), \neg\mathbf{R}(\mathtt{Y})\}}_{\textbf{definition clauses}} \quad \underbrace{\{\} \quad \{\neg\mathbf{P}(\mathtt{X}), \neg\mathbf{Q}(\mathtt{Y},\mathtt{X})\}}_{\textbf{goal clauses}}}_{\textbf{Horn clauses}}$$

Observation

There are formulas that cannot be written as Horn clauses.
For example:

$$\mathbf{P} \vee \mathbf{Q}$$

SLD resolution rule

The SLD resolution rule always involves a **definition** clause and a **goal** clause:

$$\frac{\{\mathbf{P}(t_1,\ldots,t_k),\neg\sigma_1,\ldots,\neg\sigma_n\} \quad \{\neg\mathbf{P}(s_1,\ldots,s_k),\neg\tau_1,\ldots,\neg\tau_m\} \qquad \mathbf{S} = \mathrm{mgu}(\{\mathbf{P}(t_1,\ldots,t_k) \stackrel{?}{=} \mathbf{P}(s_1,\ldots,s_k)\})}{\mathbf{S}(\{\neg\sigma_1,\ldots,\neg\sigma_n,\neg\tau_1,\ldots,\neg\tau_m\})}$$

Special case of the general resolution rule.

The selection is binary (one literal from each clause).

The resolvent is a new **goal** clause.

SLD Derivations

An SLD derivation starts with $n \geq 0$ **definition** clauses and one **goal** clause:

$$D_1 \quad \ldots \quad D_n \quad G_1$$

At each step:

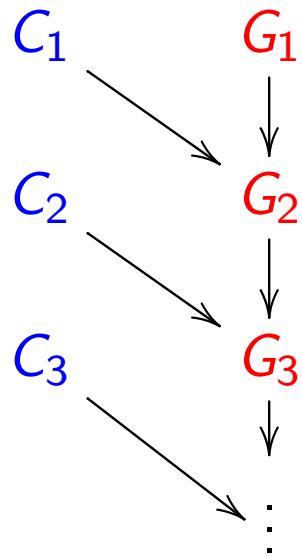
- Choose a definition clause D_j with $1 \leq j \leq n$.
- Apply the SLD resolution rule to D_j and G_i.
- The resolvent is a new goal clause G_{i+1}.

SLD Derivations

That is, given a set of clauses:

$$D_1 \quad \ldots \quad D_n \quad G_1$$

an SLD derivation is of the form:



Each C_i must be one of the original clauses $\{D_1, \ldots, D_n\}$.
The clause G_{i+1} is obtained by applying SLD resolution to C_i and G_i.

SLD Derivations

Observations:

- **Search** is simplified.
  It is limited to choosing C_i as one of the n clauses $D_1, \ldots, D_n$.
  The goal clause G_i is fixed, no alternatives.
- **Selection** is simplified.
  It is limited to choosing one of the negative literals of G_i.
  The definition clause C_i has a single positive literal.

SLD Derivations

Example

Given the hypotheses:

$$\boxed{1} \quad \forall \mathtt{X}.\, \mathtt{a}(\mathtt{0}, \mathtt{X}, \mathtt{X})$$
$$\boxed{2} \quad \forall \mathtt{X}.\, \forall \mathtt{Y}.\, \forall \mathtt{Z}.\, (\mathtt{a}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}) \Rightarrow \mathtt{a}(\mathtt{s}(\mathtt{X}), \mathtt{Y}, \mathtt{s}(\mathtt{Z})))$$

We want to prove:

$$\boxed{3} \quad \exists \mathtt{X}.\, \mathtt{a}(\mathtt{s}(\mathtt{0}), \mathtt{X}, \mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{0}))))$$

That is, we want to prove that $(\boxed{1} \wedge \boxed{2}) \Rightarrow \boxed{3}$ is valid.
It suffices to see that $\neg((\boxed{1} \wedge \boxed{2}) \Rightarrow \boxed{3})$ is unsatisfiable.
That is, it suffices to see that $\boxed{1} \wedge \boxed{2} \wedge \neg \boxed{3}$ is unsatisfiable.

SLD Derivations

Writing $\boxed{1} \wedge \boxed{2} \wedge \neg \boxed{3}$ in clausal form, we have:

$$\begin{array}{ll} \boxed{1} & \{\mathtt{a(0, X, X)}\} \\ \boxed{2} & \{\neg\mathtt{a(X, Y, Z)}, \mathtt{a(s(X), Y, s(Z))}\} \\ \boxed{3} & \{\neg\mathtt{a(s(0), X, s(s(s(0))))}\} \end{array}$$

$\boxed{1}$ and $\boxed{2}$ are **definition** clauses.

$\boxed{3}$ is the **goal** clause.

SLD Derivations

Let's search for an **SLD refutation** (an SLD derivation that reaches $\{\,\}$):

$\boxed{3} = \{\neg \mathtt{a}(\mathtt{s}(0), \mathtt{X}, \mathtt{s}(\mathtt{s}(\mathtt{s}(0))))\}$

- $\boxed{3}$ and $\boxed{2} = \{\neg \mathtt{a}(\mathtt{X}_4, \mathtt{Y}_4, \mathtt{Z}_4), \mathtt{a}(\mathtt{s}(\mathtt{X}_4), \mathtt{Y}_4, \mathtt{s}(\mathtt{Z}_4))\}$

  $\mathbf{S}_4 = \{\mathtt{X}_4 := 0,\ \mathtt{X} := \mathtt{Y}_4,\ \mathtt{Z}_4 := \mathtt{s}(\mathtt{s}(0))\}.$

  $\boxed{4} = \{\neg \mathtt{a}(0, \mathtt{Y}_4, \mathtt{s}(\mathtt{s}(0)))\}.$

- $\boxed{4}$ and $\boxed{1} = \{\mathtt{a}(0, \mathtt{X}_5, \mathtt{X}_5)\}$

  $\mathbf{S}_5 = \{\mathtt{Y}_4 := \mathtt{s}(\mathtt{s}(0)), \mathtt{X}_5 := \mathtt{s}(\mathtt{s}(0))\}$

  $\boxed{5} = \{\,\}$

SLD Derivations

Definition (Answer substitution)

Given an SLD refutation, with steps:

the **answer substitution** is the composition $\mathbf{S}_{n-1} \circ \ldots \circ \mathbf{S}_1$.

Example — answer substitution

In the previous example, the answer substitution is $\mathbf{S}_5 \circ \mathbf{S}_4$.

The value of $\mathtt{X}$ in the original goal clause $\boxed{3}$ is $\mathtt{s}(\mathtt{s}(0))$.

This says that $\mathtt{X} = \mathtt{s}(\mathtt{s}(0))$ satisfies our original query:

$$\exists \mathtt{X}.\, \mathtt{a}(\mathtt{s}(0), \mathtt{X}, \mathtt{s}(\mathtt{s}(\mathtt{s}(0))))$$

Completeness of the SLD resolution method

The resolution method is complete for Horn clauses.

More precisely, if $D_1, \ldots, D_n$ are **definition** clauses and G is a **goal** clause:

Theorem

If $\{D_1, \ldots, D_n, G\}$ is unsatisfiable, there exists an SLD refutation.

SLD Resolution

Operational semantics of Prolog

Extra-logical aspects

Semantics of Prolog

A Prolog program is a *list* of definition clauses.
A Prolog query is a goal clause.

The notation changes slightly.

Example

Clauses	**Prolog**
$\{\mathtt{a(0,X,X)}\}$	`a(0,X,X).`
$\{\mathtt{a(s(X),Y,s(Z))}, \neg\mathtt{a(X,Y,Z)}\}$	`a(s(X),Y,s(Z)) :- a(X,Y,Z).`
$\{\neg\mathtt{a(s(0),X,s(s(s(0))))}\}$	`?- a(s(0),X,s(s(s(0)))).`

Clauses are *lists*: order and multiplicity are relevant.

Semantics of Prolog

Execution is based on the SLD resolution rule.
Written in Prolog notation:

$$\frac{\texttt{?-}\ \mathrm{p}(t_1, ..., t_k), \sigma_1, \ldots, \sigma_n. \qquad \mathrm{p}(s_1, ..., s_k)\ \texttt{:-}\ \tau_1, \ldots, \tau_m. \qquad \mathbf{S} = \mathrm{mgu}(\mathrm{p}(t_1, ..., t_k) \stackrel{?}{=} \mathrm{p}(s_1, ..., s_k))}{\mathbf{S}(\texttt{?-}\ \tau_1, \ldots, \tau_m, \sigma_1, \ldots, \sigma_n.)}$$

The order of literals in the goal clause is relevant.
Selection criterion: always choose the **first literal** of the clause.

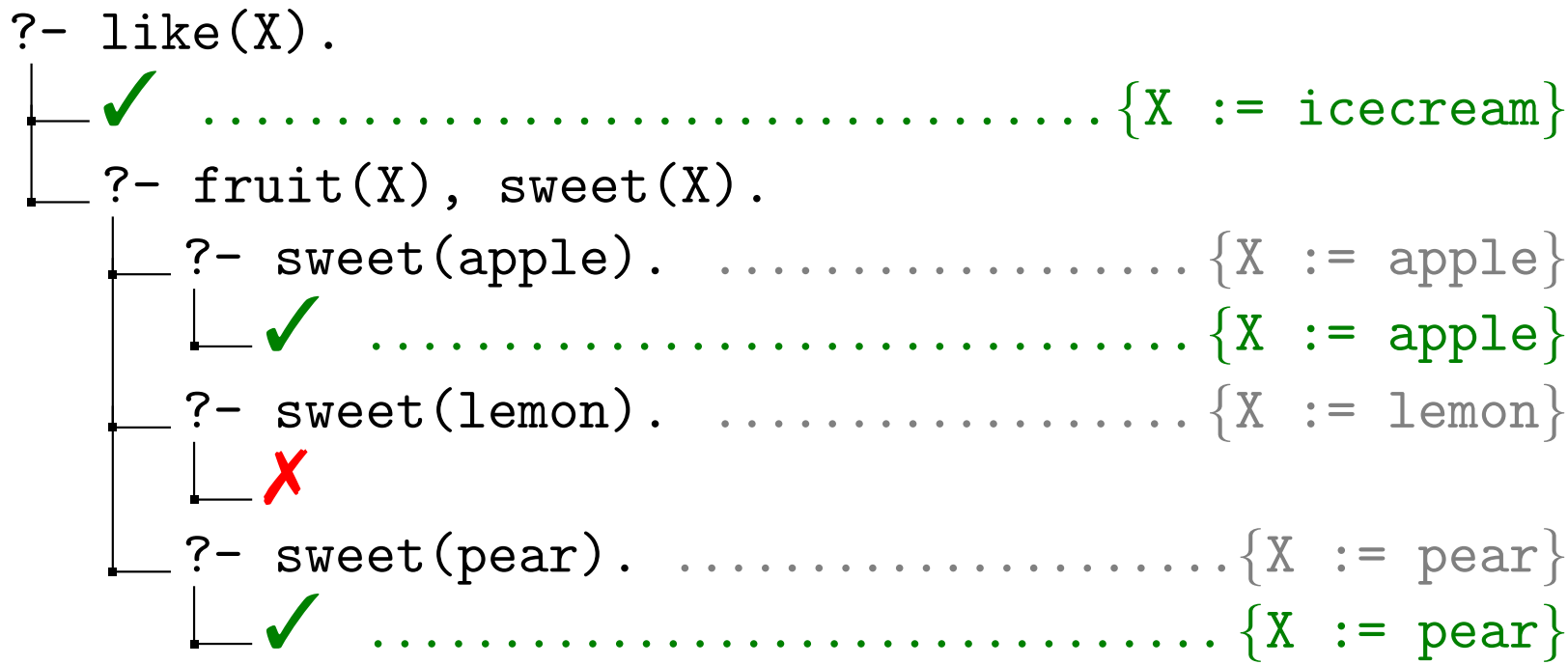
Semantics of Prolog

Prolog successively searches for all refutations using DFS.

Search criterion: rules are used in order of appearance.

Example — refutation tree

```
fruit(apple).        sweet(apple).
fruit(lemon).        sweet(pear).
fruit(pear).
like(icecream).      like(X) :- fruit(X), sweet(X).
```



Semantics of Prolog

Depth-first search (DFS) is **incomplete**.
It can cause Prolog to never find possible refutations.

Example — incompleteness of DFS

```
isWonderful(X) :- isWonderful(suc(X)).
isWonderful(zero).
```

```
?- isWonderful(zero).
 |_ ?- isWonderful(suc(zero)).
    |_ ?- isWonderful(suc(suc(zero))).
       |_ ?- isWonderful(suc(suc(suc(zero)))).
          |_ ...
```

The order of rules becomes relevant.

Tradeoff: greater efficiency at the cost of less declarativeness.
Breadth-first search (BFS) is complete but very costly.

Semantics of Prolog

When unifying, Prolog **does not** use the occurs-check rule.
For example, `X` unifies with `f(X)`. This is **incorrect**.
It can cause Prolog to find an incorrect "refutation".

Example — incorrect refutation due to omitted occurs check

```
isSuccessor(X, suc(X)).
```

```
?- isSuccessor(Y, Y).
└─✓ ..........................{Y := X, X := suc(X)}
```

Tradeoff: greater efficiency at the cost of logical incorrectness.
In many contexts, the occurs-check rule is unnecessary.
The burden of proving correctness falls on the programmers.

Example: list concatenation

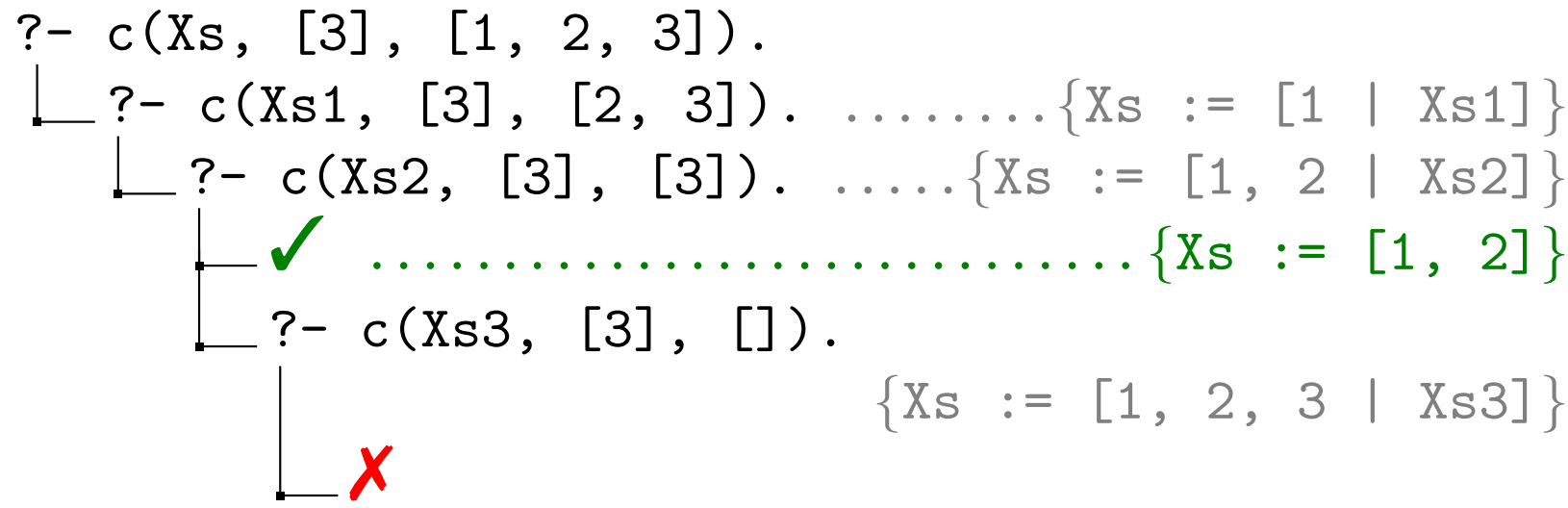
```
c([], Ys, Ys).
c([X | Xs], Ys, [X | Zs]) :- c(Xs, Ys, Zs).
```

```
?- c([1, 2], [3, 4], Zs).
└─?- c([2], [3, 4], Zs1). .....{Zs := [1 | Zs1]}
   └─?- c([], [3, 4], Zs2). ..{Zs := [1, 2 | Zs2]}
      └─✓ .....................{Zs := [1, 2, 3, 4]}
```

```
?- c([1, 2], Ys, [1, 2, 3, 4]).
└─?- c([2], Ys, [2, 3, 4]).
   └─?- c([], Ys, [3, 4]).
      └─✓ ..........................{Ys := [3, 4]}
```

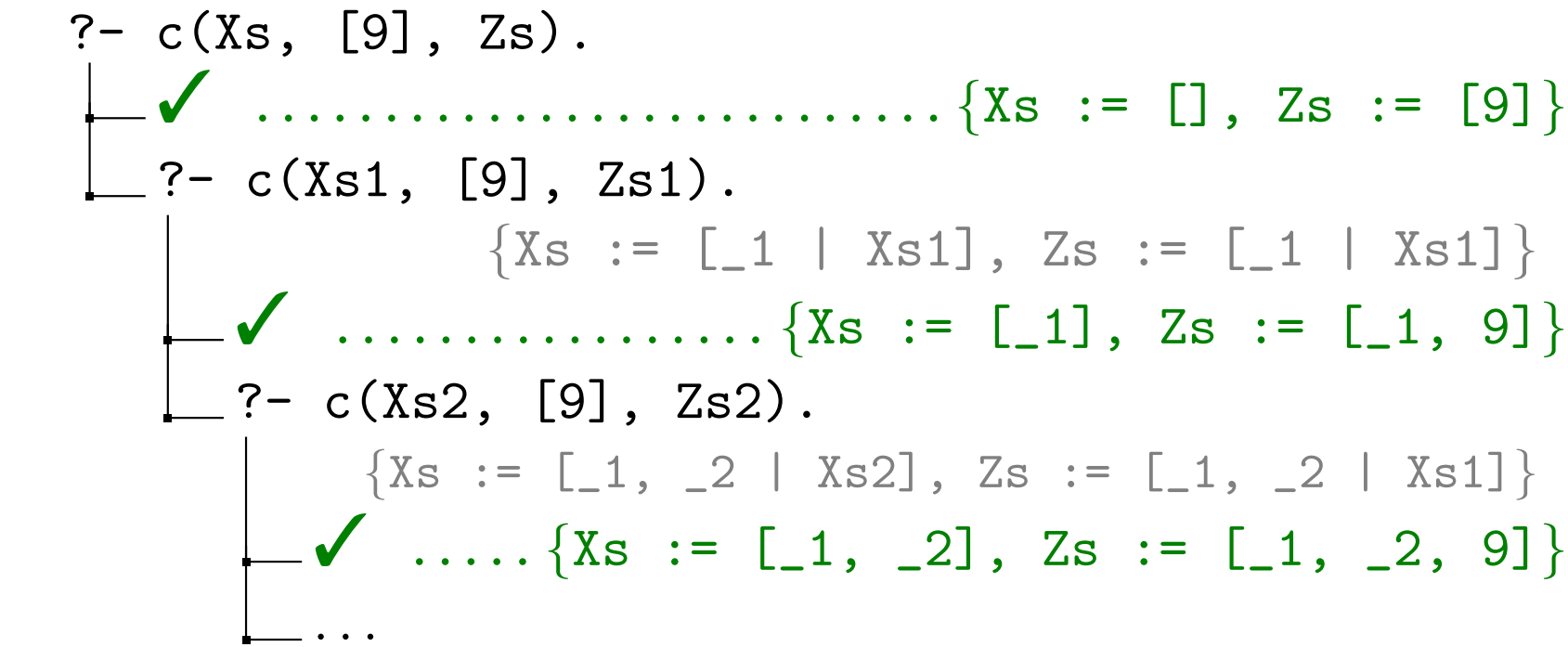
Example: list concatenation

```
c([], Ys, Ys).
c([X | Xs], Ys, [X | Zs]) :- c(Xs, Ys, Zs).
```



Example: list concatenation

```
c([], Ys, Ys).
c([X | Xs], Ys, [X | Zs]) :- c(Xs, Ys, Zs).
```



SLD Resolution

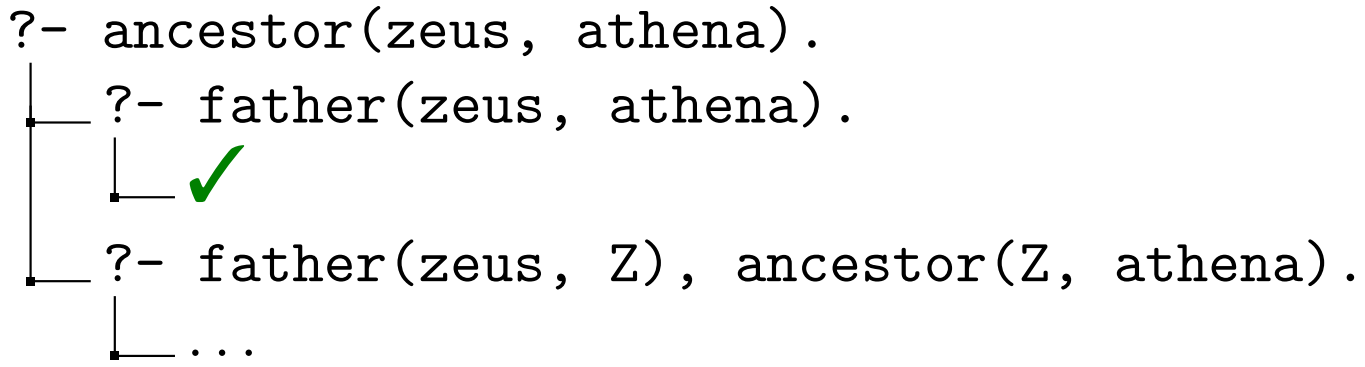
Operational semantics of Prolog

Extra-logical aspects

Cut operator

Consider the following program:

```
father(zeus, athena). % ...knowledge base...
ancestor(X, Y) :- father(X, Y).
ancestor(X, Y) :- father(X, Z), ancestor(Z, Y).
```



We would like a way to prune the search tree.

Cut operator

We rewrite the program by adding a *cut* ("!"):

```
father(zeus, athena). % ...knowledge base...
ancestor(X, Y) :- father(X, Y), !.
ancestor(X, Y) :- father(X, Z), ancestor(Z, Y).
```

The operator ! does not have a declarative/logical interpretation.

It is an extra-logical operator.

Its behavior is explained from an operational point of view.

The cut operator indicates that, if it is reached, alternatives to the rule in which it appears should not be explored.

Cut operator

Semantics of the cut operator

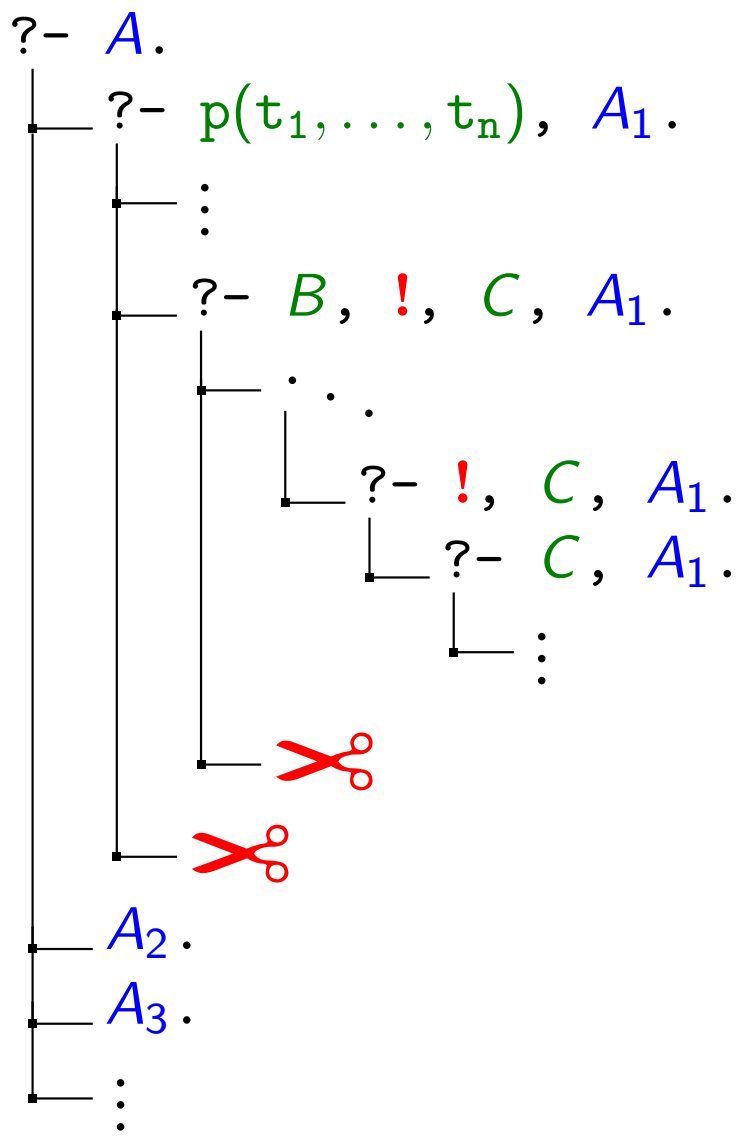
The predicate **!** succeeds immediately.

When *backtracking* occurs:

- Backtrack to the point where the rule containing the cut was chosen.
- Discard all alternative choices.
- Continue searching backward.

Cut operator

Graphically:



Cut operator

Example — "benign" cuts (green cuts)

In some cases, ! does not alter the semantics of the program.
It can be used to build equivalent, more efficient programs.

```
add(N, zero, N) :- !.
add(zero, N, N).
add(suc(N), M, suc(P)) :- add(N, M, P).

?- add(suc(suc(suc(suc(...)))), zero, P).
```

Example — "dangerous" cuts (red cuts)

In other cases, the semantics can be altered.

```
maximum(A, B, A) :- A >= B, !.
maximum(A, B, B).
```

```
?- maximum(2, 1, C).
>> C = 2
```

```
?- maximum(2, 1, 1).
>> true.
```

Negation as failure

Definition (negation operator)

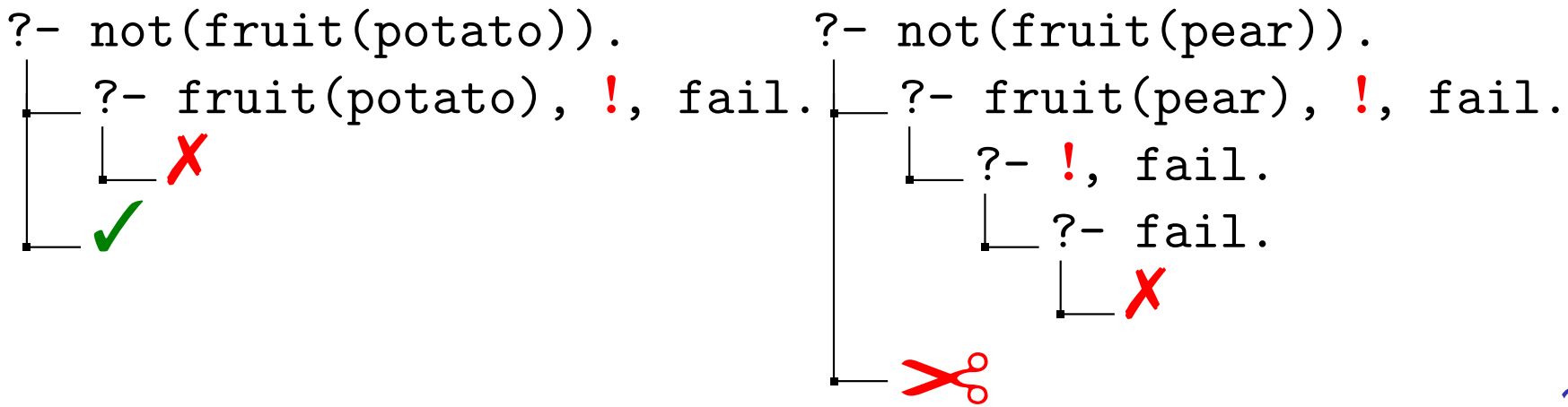
If `fail` is a predicate that always fails, negation can be defined in Prolog as follows:

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

Observation. `not(P)` succeeds if and only if `P` fails.

Example — negation as failure

```
fruit(pear).
```

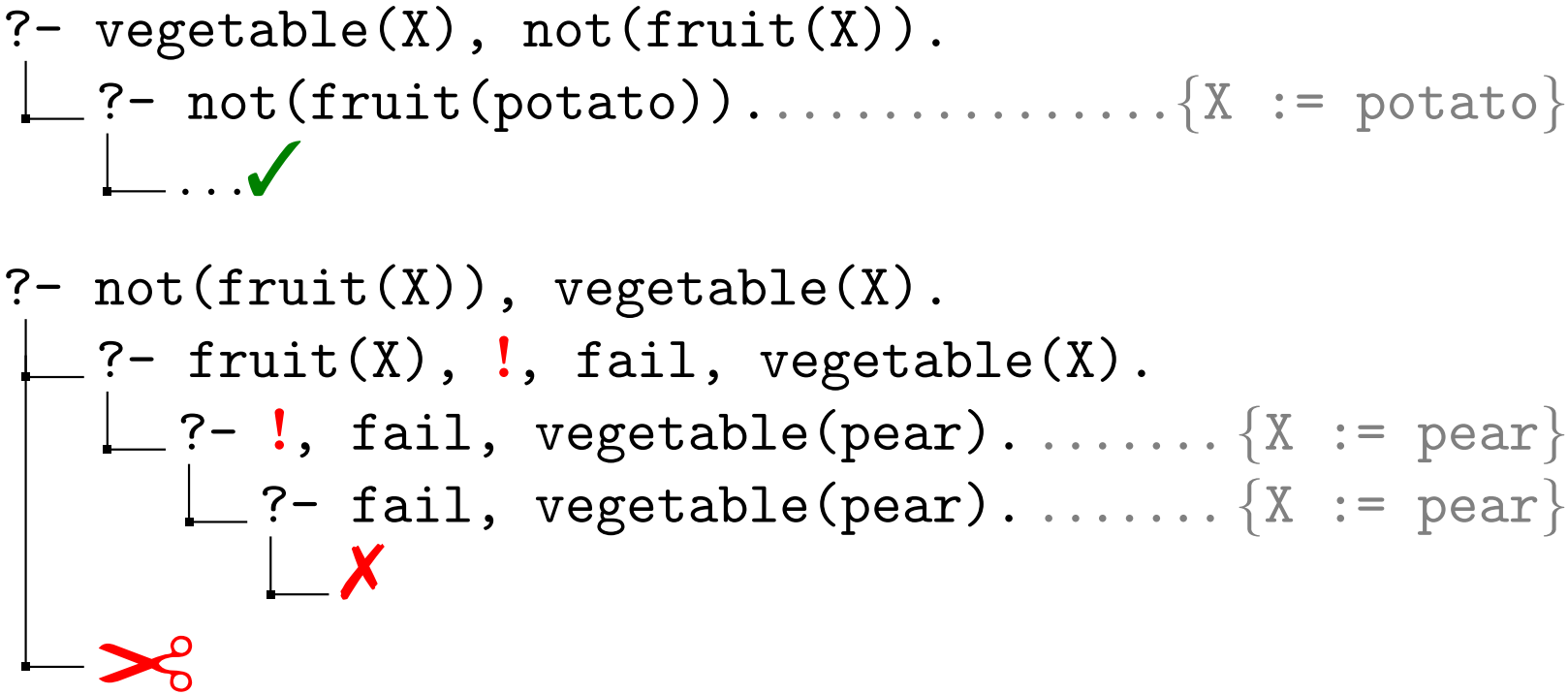


Negation as failure

Negation as failure does not coincide with logical negation.

Example — negation as failure ≠ logical negation

```
fruit(pear). vegetable(potato).
```



The order of literals in the query becomes relevant.

This goes against declarativeness.

Example

1. Define `append(L1, L2, L3)`.
2. Define `occursAtLeastOnce(X, L)` using `append/3`.
3. Define `occursAtLeastTwice(X, L)` using `append/3`.
4. Define `occursExactlyOnce(X, L)` by combining the previous ones.

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?

Recommended reading

Chapters 2 and 3 of the book by Nilsson and Małuciński.
Ulf Nilsson and Jan Małuzyński. *Logic, Programming and Prolog* (2ED).
John Wiley & Sons Ltd., 1995.