Logic in Computer Science (ECS666U/7018P/U)

Nikos Tzevelekos

Lecture 11
Introduction to Prolog

Predicate Logic Recap and Entailment

Formulas made up of terms, predicates and logical connectives:

$$t \quad = \quad x \mid f(t,\ldots,t)$$

$$\varphi \quad = \quad P(t,\ldots,t) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \forall x.\varphi \mid \exists x.\varphi$$

Meaning / Semantics:

- Requires the notion of model, i.e. setting for variables and quantifiers, functions and relations
- Validity: $\models \varphi$ holds if $M,E \models \varphi$ holds for all M,E
- *Entailment:* $T \models \varphi$ holds if, for all for all M,E, if $M,E \models T$ then $M,E \models \varphi$

T is a theory, i.e. a set of formulas (axioms)

How to show $T \models \varphi$?

- use a semantic argument (assuming any M,E show $M,E \models T \rightarrow \varphi$)
- use a *formal proof system*

Proof systems

There are several proof systems for predicate logic.

For example, Natural Deduction uses Introduction and Elimination rules:

$$\frac{\varphi_1 \qquad \varphi_2}{\varphi_1 \wedge \varphi_2}\wedge\mathrm{I} \qquad \frac{\varphi_1 \wedge \varphi_2}{\varphi_1}\wedge\mathrm{E} \qquad \frac{\varphi_1 \wedge \varphi_2}{\varphi_2}\wedge\mathrm{E} \qquad \cdots$$

$$\frac{\varphi_1 \rightarrow \varphi_2 \qquad \varphi_1}{\varphi_2}\rightarrow\mathrm{E}\ \text{(modus ponens)} \qquad \frac{\varphi}{\forall x.\varphi}\forall\mathrm{I}\ \text{(if } x \text{ is not free in any axiom)} \qquad \frac{\forall x.\varphi}{\varphi[t/x]}\forall\mathrm{E} \qquad \cdots$$

Provability: $T \vdash \varphi$ holds if there is a proof of φ using T as assumptions

Theorem (Soundness and Completeness): $T \vdash \varphi \Leftrightarrow T \vDash \varphi$

Soundness ($\Longrightarrow$): only true entailments are provable
Completeness ($\Longleftarrow$): all true entailments are provable

Gödel's Completeness Theorem (1929)

Example (MP and ∀E)

For example, here is a classical logical syllogism (going back to Aristotle):

- *all humans are mortal*
- *Socrates is a human*
- *Socrates is mortal*

$$\dfrac{\dfrac{\forall x.\ \mathsf{human}(x) \to \mathsf{mortal}(x)}{\mathsf{human}(\mathsf{Socrates}) \to \mathsf{mortal}(\mathsf{Socrates})}\,\forall\mathrm{E} \qquad \mathsf{human}(\mathsf{Socrates})}{\mathsf{mortal}(\mathsf{Socrates})}\,{\to}\mathrm{E}$$

We showed:

$$\forall x.\ \mathsf{human}(x) \to \mathsf{mortal}(x),\ \mathsf{human}(\mathsf{Socrates}) \vdash \mathsf{mortal}(\mathsf{Socrates})$$

Entailment => not SAT

Observe that $\varphi_1, \varphi_2, \ldots, \varphi_n \vDash \varphi$ holds iff $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n \wedge \neg\varphi$ is not SAT.

For entailment, we can therefore study SAT of predicate logic formulas.

In fact, we focus on formulas in CNF:

- No quantifiers
- Each formula is a conjunction (or set) of *clauses* $c_1 \wedge c_2 \wedge \ldots \wedge c_m$
- Each clause if of the form $\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$ with each φ_i a *literal* (a basic formula $P(t_1,\ldots,t_m)$ or its negation)

Theorem: $T \vDash \varphi$ holds if, and only if, $\forall x_1 . \forall x_2 \ldots . \forall x_n .\text{CNF}^*(T,\neg\varphi)$ is not SAT.

Example transformation to CNF

0. $\forall x.\ \mathsf{human}(x) \rightarrow \mathsf{mortal}(x),\ \mathsf{human}(\mathsf{Socrates}) \models \mathsf{mortal}(\mathsf{Socrates})$

1. $\forall x.\ \mathsf{human}(x) \rightarrow \mathsf{mortal}(x) \land \mathsf{human}(\mathsf{Socrates}) \land \neg \mathsf{mortal}(\mathsf{Socrates})$

2. $\forall x.\ (\ \mathsf{human}(x) \rightarrow \mathsf{mortal}(x) \land \mathsf{human}(\mathsf{Socrates}) \land \neg \mathsf{mortal}(\mathsf{Socrates})\ )$

4. $\mathsf{human}(x) \rightarrow \mathsf{mortal}(x) \land \mathsf{human}(\mathsf{Socrates}) \land \neg \mathsf{mortal}(\mathsf{Socrates})$

5. $(\neg \mathsf{human}(x) \lor \mathsf{mortal}(x)) \land \mathsf{human}(\mathsf{Socrates}) \land \neg \mathsf{mortal}(\mathsf{Socrates})$

i.e. in SAT notation:

$[\ [\neg \mathsf{human}(x),\ \mathsf{mortal}(x)],\ [\mathsf{human}(\mathsf{Socrates})],\ [\neg \mathsf{mortal}(\mathsf{Socrates})]\ ]$

Transformation to CNF

The transformation CNF*($T, \neg\varphi$) is done in several steps:

1. First, observe that $\varphi_1, \varphi_2, \ldots, \varphi_n \models \varphi$ holds iff $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n \wedge \neg\varphi$ is not SAT. This reduces entailment to (not) SAT.
2. We can transform every formula to an equivalent one in prenex normal form: $Q_1 x_1 . Q_2 x_2 \ldots . Q_n x_n . \varphi$

   where each Q_i is a quantifier and φ is quantifier-free.
3. We can remove existential quantifiers using *Skolemisation*.

   Starting from any φ, we get an $\exists$-free ψ such that: φ is SAT iff ψ is
4. We are left with formulas of the form $\forall x_1 . \forall x_2 \ldots . \forall x_n . \varphi$, with φ quantifier-free. We can remove the quantifiers and assume all free variables are implicitly quantified.
5. Finally, we transform our quantifier-free formula to CNF.

Transformation to CNF – Skolemisation

Note step 3:

3. We can remove existential quantifiers using *Skolemisation*.

   Starting from any φ, we get an $\exists$-free ψ such that: φ is SAT iff ψ is

This is based on a trick whereby we use new functions instead of $\exists$'s. E.g:

- $\forall x.\ \text{student}(x) \rightarrow \exists y.\ \text{instructor}(y) \land \text{younger}(x,y)$
- $\forall x.\ \exists y.\ \text{student}(x) \rightarrow \text{instructor}(y) \land \text{younger}(x,y)$
- $\forall x.\ \text{student}(x) \rightarrow \text{instructor}(\text{someone}(x)) \land \text{younger}(x,\text{someone}(x))$

where someone is a new function symbol.

The 1st and 3rd formulas above are equi-SAT (one is SAT iff the other one is)*.

Example resolution

We want to show not SAT for:

$$[\ [\neg\text{human}(x), \text{mortal}(x)],\ [\text{human}(\text{Socrates})], [\neg\text{mortal}(\text{Socrates})]\ ]$$

Resolution is a simple proof system based on 1 rule (+ structural rules) that is used in logical programming. In its basic form:

$$\frac{[\varphi_1, \varphi_2, \ldots, \varphi_n, \varphi] \qquad\qquad [\psi_1, \psi_2, \ldots, \psi_m, \neg\varphi]}{[\varphi_1, \varphi_2, \ldots, \varphi_n, \psi_1, \psi_2, \ldots, \psi_m]}$$

Back to our example, proof using resolution:

$$\frac{[\neg\text{human}(x), \text{mortal}(x)] \qquad\qquad [\text{human}(\text{Socrates})]}{???}$$

to proceed, we need to *unify* human(x) with human(Socrates)

Most general unifier (MGU)

Unifying formulas φ and ψ means:

- finding a *substitution* (of terms for variables) that makes the two formulas the same
- the substitution needs to be *as general as possible*

For example, how can we unify:

- $\mathsf{student}(x)$ with $\mathsf{student}(\mathsf{Maz})$
- $\mathsf{younger}(x,y)$ with $\mathsf{younger}(\mathsf{Maz},z)$
- $\mathsf{human}(x)$ with $\mathsf{human}(\mathsf{Socrates})$

Most general unifier (MGU)

Unifying formulas φ and ψ means:

- finding a *substitution* (of terms for variables) that makes the two formulas the same
- the substitution needs to be as general as possible

Formally, a substitution is a map θ from variables to terms:

$$\theta = [\, x_1 \mapsto t_1,\ x_2 \mapsto t_2,\ \ldots\ x_n \mapsto t_n \,]$$

We write $\theta(x_i)$ for each t_i.

Applying a substitution θ to a formula φ means replacing each x with $\theta(x)$ in φ. We write $(\varphi)\theta$ for the resulting formula. Thus, $\theta = \mathrm{MGU}(\varphi, \psi)$ if:

- $(\varphi)\theta = (\psi)\theta$
- for any other unifier θ_1, i.e. such that $(\varphi)\theta_1 = (\psi)\theta_1$, we can get θ_1 "via" θ :

$$\theta_1 = (\theta)\theta_2 \quad \text{for some substitution } \theta_2$$

Full resolution rule

The resolution rule uses a MGU between the resolved formulas:

$$\frac{[\ \varphi_1, \varphi_2, \ldots, \varphi_n, \varphi\ ] \qquad\qquad [\ \psi_1, \psi_2, \ldots, \psi_m, \neg\psi\ ]}{[\ (\varphi_1)\theta, (\varphi_2)\theta, \ldots, (\varphi_n)\theta, (\psi_1)\theta, (\psi_2)\theta, \ldots, (\psi_m)\theta\ ]}\ \mathrm{MGU}(\varphi, \psi) = \theta$$

Note: the empty disjunction [ ] is False.

Previous proof using resolution. We want to show not SAT for:

[[¬human(x), mortal(x)], [human(Socrates)], [¬mortal(Socrates)]]

We derive a contradiction as follows:

$$\frac{\dfrac{[\neg\mathrm{human}(x), \mathrm{mortal}(x)] \qquad [\mathrm{human}(\mathrm{Socrates})]}{[\mathrm{mortal}(\mathrm{Socrates})]}\ \mathrm{MGU}\text{: } [\ x \mapsto \mathrm{Socrates}\ ] \qquad [\neg\mathrm{mortal}(\mathrm{Socrates})]}{[\ ]}$$

Another example

Assuming the following axioms:

1. indian(x) $\wedge$ mild(x) $\rightarrow$ likes(Sam,x)
2. indian(dahl)
3. indian(madras)
4. mild(dahl)
5. spicy(madras)

prove the *goal*: likes(Sam,dahl).

Add the negation of the goal:

6. $\neg$likes(Sam,dahl)

Rewrite 1 in CNF:

1. [$\neg$indian(x), $\neg$mild(x), likes(Sam,x)]

Now, applying resolution to 1 & 2, with MGU [$x \mapsto$ dahl]:

7. [$\neg$mild(dahl), likes(Sam,dahl)]

Next, applying resolution to 7 & 4, with MGU []:

8. [likes(Sam,dahl)]

Finally, applying resolution to 8 & 6, with MGU []:

9. [ ]

Automating provability

Resolution alone is not complete. It also needs a *factoring* rule:

$$\frac{[\varphi_1, \varphi_2, \varphi_3, \ldots, \varphi_n]}{[(\varphi_2)\theta, (\varphi_3)\theta, \ldots, (\varphi_n)\theta]} \quad \mathrm{MGU}(\varphi_1, \varphi_2) = \theta$$

Even with those two rules, automating provability is not possible (recall SAT is undecidable in predicate logic).

However, if $T \vDash \varphi$ holds then there is a proof using resolution.
The trouble is finding it!

Even if we have no guarantee to find the proof (or a contradiction), we can still try to search for it efficiently (state explosion problem).

Horn clauses have a nice resolution strategy and are used in Prolog.

Horn clauses and Prolog

In the rest of the lecture we will focus on a special case of CNF formulas where clauses are ***Horn clauses*** :

$$\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n \quad \textit{with } \textbf{\textit{at most one positive}} \textit{ literal } \varphi_i$$

(a literal is *positive* if it is of the form $P(t_1,\ldots,t_m)$, instead of $\neg P(t_1,\ldots,t_m)$)

Horn clauses can also be written:

$$\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m \rightarrow \psi$$

with each ψ_i a positive literal, and ψ a positive literal or False.

Prolog uses more convenient notation:

- Rules: ψ :- $\psi_1, \psi_2, \ldots, \psi_m$. (for $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m \rightarrow \psi$)
- Facts: ψ. (for True $\rightarrow \psi$)
- Queries: ?- $\psi_1, \psi_2, \ldots, \psi_m$. (for $\psi_1 \wedge \ldots \wedge \psi_m \rightarrow$ False, i.e. $\neg(\psi_1 \wedge \ldots \wedge \psi_m)$)

Horn clauses and Prolog

In the rest of the lecture we will focus on a special case of CNF formulas where clauses are ***Horn clauses*** :

$$\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n \quad \textit{with } \textbf{\textit{at most one positive}} \textit{ literal } \varphi_i$$

(a literal is *positive* if it is of the form $P(t_1,\ldots,t_m)$, instead of $\neg P(t_1,\ldots,t_m)$)

Horn clauses can also be written:

$$\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m \rightarrow \psi$$

with each ψ_i a positive literal, and ψ a positive literal or False.

Prolog uses more convenient notation (pos. literals A, B, etc. for clarity):

• Rules:	A :- B_1, B_2, …, B_m .	(read: A if B_1 and B_2 and … and B_m)
• Facts:	A.	(read: A holds)
• Queries:	?- Q_1, Q_2, …, Q_m .	(read: Q_1 and Q_2 and … and Q_m?)

Logic programming: knowledge representation + computation

Example (likes.pl)

```
indian(dahl).
indian(tandoori).
indian(kurma).

indian(madras).
mild(dahl).
mild(tandoori).
mild(kurma).

italian(pizza).
italian(spaghetti).
italian(lasagne).

likes(sam,Food) :- indian(Food), mild(Food).
likes(sam,Food) :- italian(Food).
likes(sam,chips).
```

```
?- likes(sam,dahl).
True
?- likes(sam,madras).
False
```

Example (likes.pl)

```
indian(dahl).
indian(tandoori).
indian(kurma).

indian(madras).
mild(dahl).
mild(tandoori).
mild(kurma).

italian(pizza).
italian(spaghetti).
italian(lasagne).

likes(sam,Food) :- indian(Food), mild(Food).
likes(sam,Food) :- italian(Food).
likes(sam,chips).
```

```
?- likes(sam,dahl).
True
?- likes(sam,madras).
False
```

Notes:

A Prolog program can be seen as a theory T.

A query ?- *Q* is a request for a proof of the entailment $T \models Q$.

Here, ?- likes(sam,dahl) returned True, so entailment was proven.

Whereas ?- likes(sam,madras) returned False.

This does not mean that Sam does not like madras, but that it is not entailed from the given axioms and, therefore, it cannot be proved either (*closed-world assumption*).

Example (family.pl)

```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(victoria,edward).
parent(albert,edward).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).

?- father(albert,edward).
True
?- father(albert,alice).
False
?- father(X,Y).
X=albert, Y=edward
```

Lowercase-starting identifiers: constants, functions, predicates.

Uppercase-starting (or _foo): variables.

Example (family.pl)

```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(victoria,edward).
parent(albert,edward).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
```

```
?- father(albert,edward).
True

?- father(albert,alice).
False

?- father(X,Y).
X=albert, Y=edward
```

Lowercase-starting identifiers: constants, functions, predicates.

Uppercase-starting (or _foo): variables.

Proof search:

- starting goal father(albert,edward)
- find the first line in the program whose head is unifiable with the goal
  - father(X,Y) :- parent(X,Y), male(X)

    works with X = albert, Y = edward

    then, our new goal becomes parent(albert,edward), male(albert)
- find the first line whose head is unifiable with parent(albert,edward)
  - parent(albert,edward) works

    then, our new goal is male(albert)
- find the first line whose head is unifiable with male(albert)
  - male(albert) works

    Success. Return True.

Example (family.pl)

```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(victoria,edward).
parent(albert,edward).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).


?- father(albert,edward).
True
?- father(albert,alice).
False
?- father(X,Y).
X=albert, Y=edward
```

Proof search:

- starting goal father(albert,alice)
- find the first line in the program whose head is unifiable with the goal
  - father(X,Y) :- parent(X,Y), male(X)

    works with X = albert, Y = alice

    then, our new goal becomes parent(albert,alice), male(albert)
- find the first line whose head is unifiable with parent(albert,alice)
  - nothing works: Fail. Return False.

Example (family.pl)

```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(victoria,edward).
parent(albert,edward).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).


?- father(albert,edward).
True
?- father(albert,alice).
False
?- father(X,Y).
X=albert, Y=edward
```

Proof search:

- starting goal father(Z,W)
- find the first line in the program whose head is unifiable with the goal
  - father(X,Y) :- parent(X,Y), male(X)

    works with Z = X, W = Y

    then, our new goal becomes parent(Z,W), male(Z)
- find the first line whose head is unifiable with parent(Z,W)
  - parent(victoria,edward) works with Z = victoria, W = edward

    then, our new goal is male(victoria)
- find the first line whose head is unifiable with male(victoria)
  - nothing works: Fail. Backtrack.

Example (family.pl)

```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(victoria,edward).
parent(albert,edward).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).


?- father(albert,edward).
True
?- father(albert,alice).
False
?- father(X,Y).
X=albert, Y=edward
```

<u>Proof search:</u>

- starting goal father(Z,W)
- find the first line in the program whose head is unifiable with the goal
  - father(X,Y) :- parent(X,Y), male(X)

    works with Z = X, W = Y

    then, our new goal becomes parent(Z,W), male(Z)
- find the <u>next line</u> whose head is unifiable with parent(Z,W)
  - parent(albert,edward) works with Z = albert, W = edward

    then, our new goal is male(albert)
- find the first line whose head is unifiable with male(albert)
  - male(albert) works: Success.
    Return Z = albert, W = edward

Proof search

- at each step we keep a set of goals $Q_1, \ldots, Q_m$ that need proof, starting from the given query
- for each line in the program (search top to bottom):
  - if it is a fact A unifiable with Q_1 then solve $(Q_2)\theta, \ldots, (Q_m)\theta$ (if $m>1$);

    if empty goal ($m=1$): Success. Return True or satisfying substitution.
  - if it is a rule A :- $B_1, \ldots, B_n$
    with A unifiable with Q_i then solve $(Q_2)\theta, \ldots, (Q_m)\theta, (B_1)\theta, \ldots, (B_n)\theta$.
- if no line is unifiable with Q_1 as above: Fail.

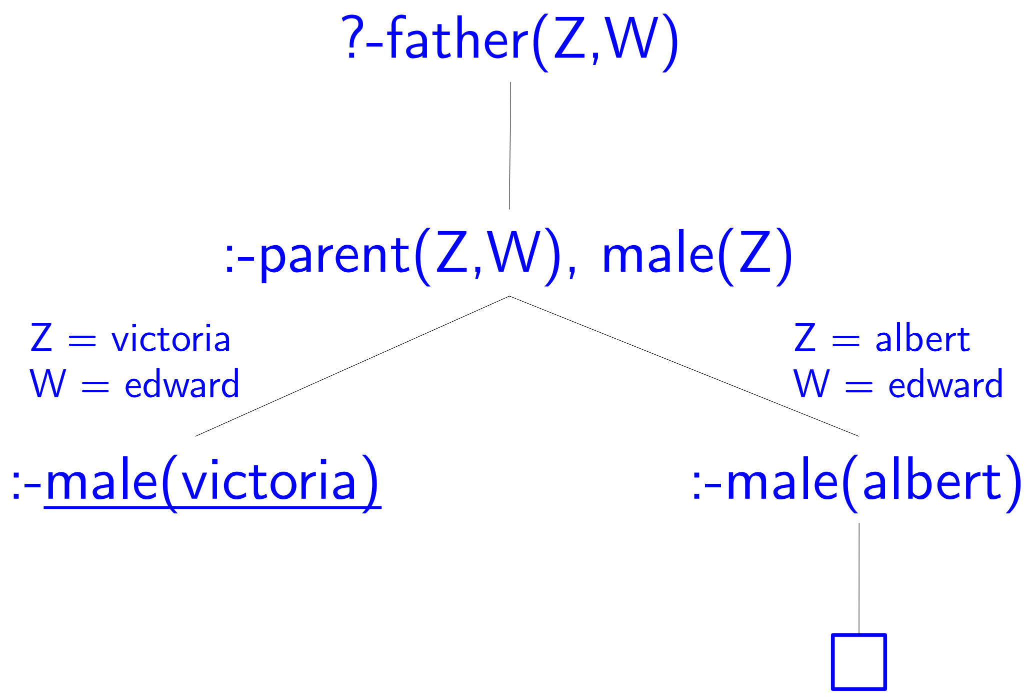
  Failure leads to backtracking to find more matchings that could work. If no more matchings possible then return False.

Note: In practice, Prolog does not apply the substitutions but carry them along (like an *environment*).

SLD trees

Recall that in Prolog program rules are scanned top to bottom. Moreover, at each step, the remaining goals are examined left to right.

Looking at the full tree on the right, our search strategy is depth-first-search.

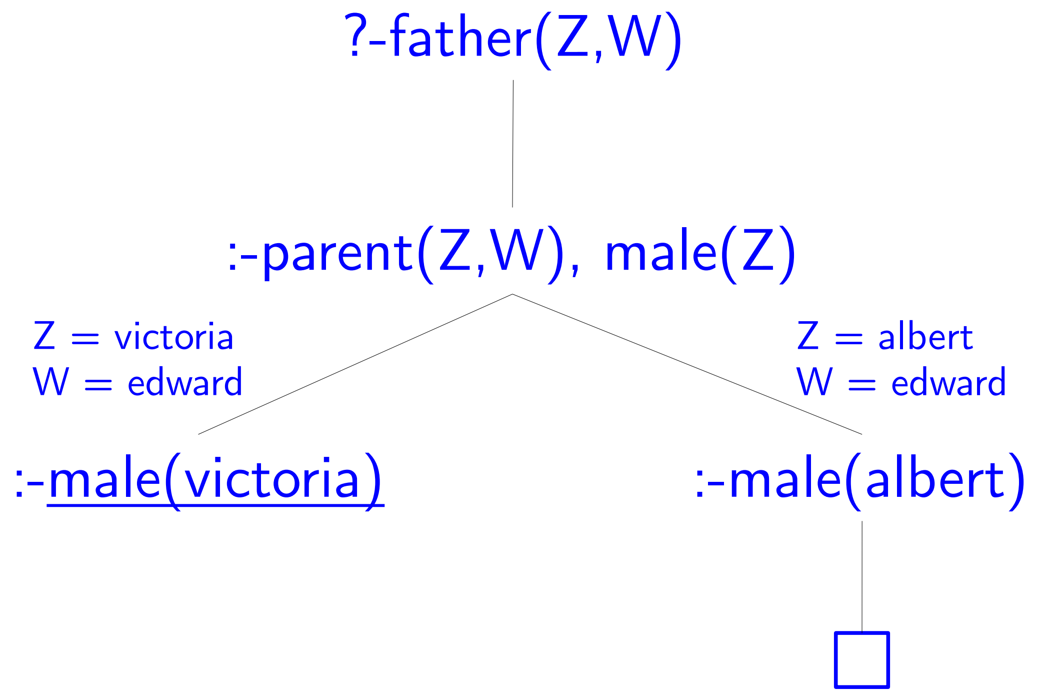


```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(victoria,edward).
parent(albert,edward).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
```

SLD trees

Recall that in Prolog program rules are scanned top to bottom. Moreover, at each step, the remaining goals are examined left to right.

Looking at the full tree on the right, our search strategy is depth-first-search.



```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(victoria,edward).
parent(albert,edward).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
```

These trees correspond to *SLD resolution*:

- S : at each node, we *select* a goal (leftmost) and a rule/fact to apply resolution with
- L : each path from the root to a success leaf is a *linear* proof
- D : *definite* rules used (one positive literal)

SLD trees

Recall that in Prolog program rules are scanned top to bottom. Moreover, at each step, the remaining goals are examined left to right.

Looking at the full tree on the right, our search strategy is depth-first-search.

$$\begin{tikzcd} ?\text{-father(albert,alice)} \arrow[d, dash] \\ :\text{-}\underline{\text{parent(albert,alice), male(albert)}} \end{tikzcd}$$

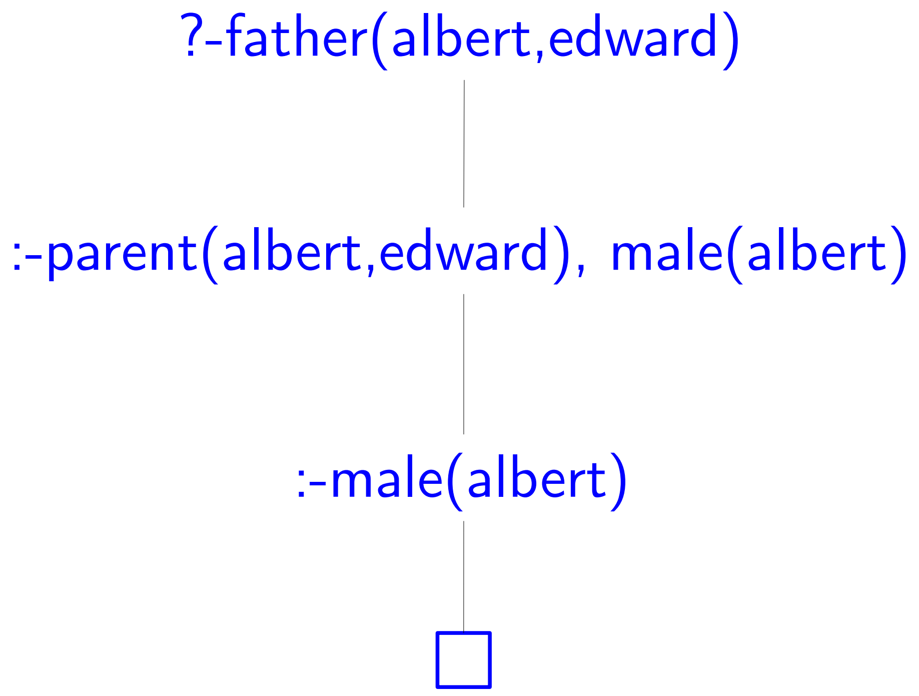
```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(victoria,edward).
parent(albert,edward).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
```

SLD trees

Recall that in Prolog program rules are scanned top to bottom. Moreover, at each step, the remaining goals are examined left to right.

Looking at the full tree on the right, our search strategy is depth-first-search.

```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(victoria,edward).
parent(albert,edward).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
```



SLD tree example

Let us consider this program.

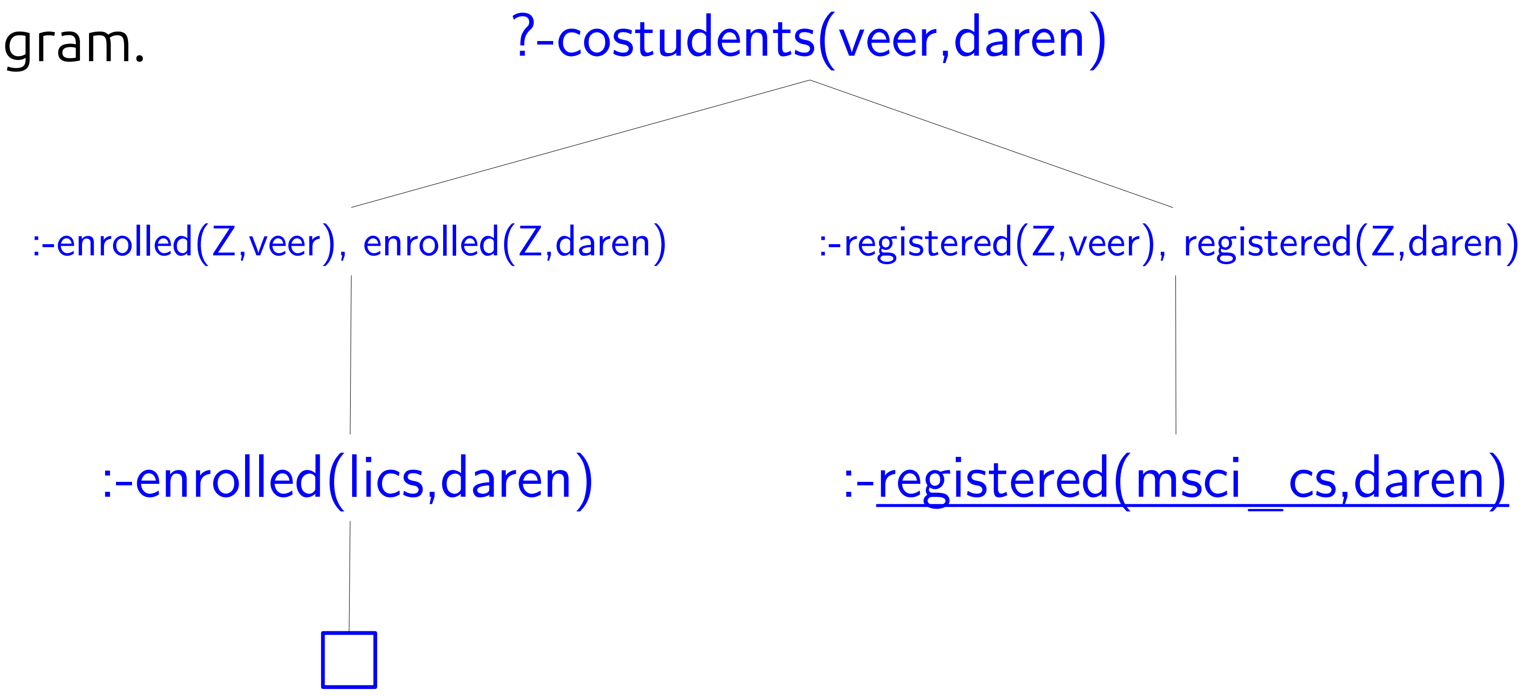
?-costudents(veer,daren)

```
student(maz).
student(daren).
student(veer).
student(tom).
student(sara).
registered(msc cs,maz).
registered(msc cs,daren).
registered(msci cs,tom).
registered(msci cs,veer).
registered(bsc cs,sara).
enrolled(lics,daren).
enrolled(lics,maz).
enrolled(lics,veer).
costudents(X,Y) :- enrolled(Z,X), enrolled(Z,Y).
costudents(X,Y) :- registered(Z,X), registered(Z,Y).
```

SLD tree example

Let us consider this program.

```
student(maz).
student(daren).
student(veer).
student(tom).
student(sara).
registered(msc cs,maz).
registered(msc cs,daren).
registered(msci cs,tom).
registered(msci cs,veer).
registered(bsc cs,sara).
enrolled(lics,daren).
enrolled(lics,maz).
enrolled(lics,veer).
costudents(X,Y) :- enrolled(Z,X), enrolled(Z,Y).
costudents(X,Y) :- registered(Z,X), registered(Z,Y).
```


?-costudents(veer,daren)
:-enrolled(Z,veer), enrolled(Z,daren)
:-registered(Z,veer), registered(Z,daren)
:-enrolled(lics,daren)
:-registered(msci_cs,daren)
□

Depth-first-search, incompleteness

SLD resolution is **complete** (AKA refutation complete): if there is a proof, it will correspond to a root-to-success-leaf in the SLD-tree.

However, Prolog searches the tree using depth-first search (for efficiency!) and may fail to find a proof even if there is one.

This can happen when the tree is infinite due to recursion. For example:

```
?-father(albert,edward)
```

```
male(albert).
male(edward).
female(alice).
female(victoria).
child(X,Y,Z) :- child(X,Z,Y).
child(edward,victoria,albert).
father(X,Y) :- child(Y,X,Z), male(X).
mother(X,Y) :- child(Y,X,Z), female(X).
```

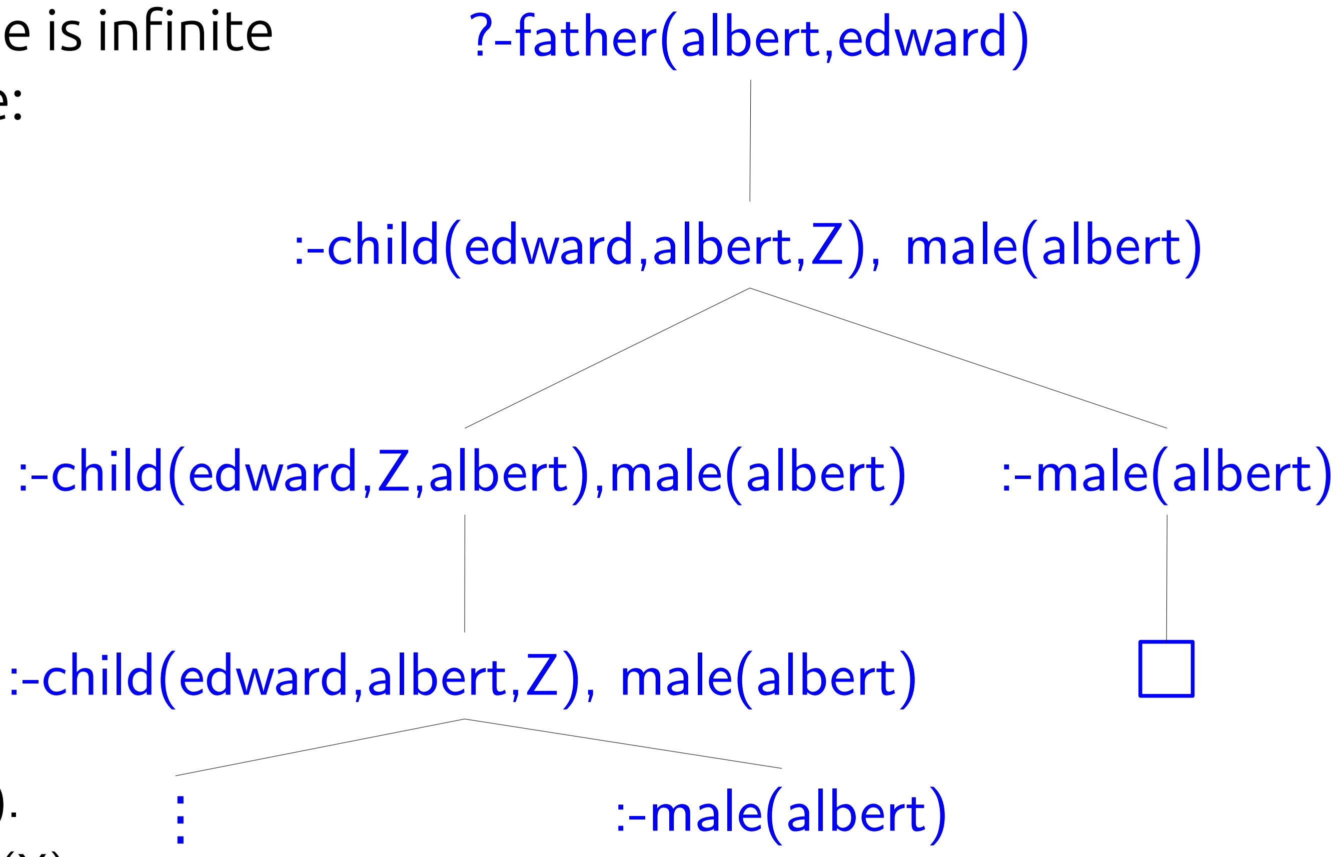
Depth-first-search, incompleteness

SLD resolution is **complete** (AKA refutation complete): if there is a proof, it will correspond to a root-to-success-leaf in the SLD-tree.

However, Prolog searches the tree using depth-first search (for efficiency!) and may fail to find a proof even if there is one.

This can happen when the tree is infinite due to recursion. For example:

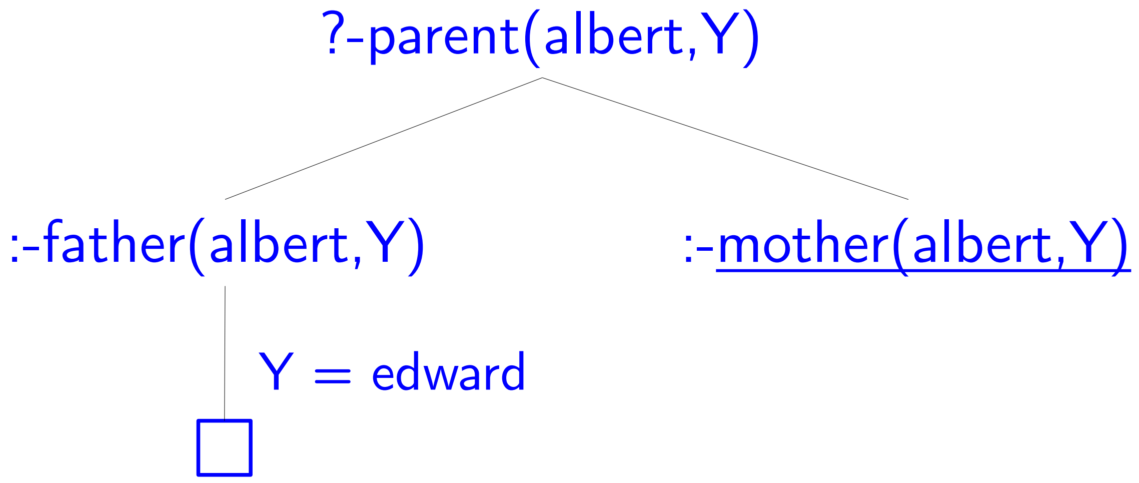
```
male(albert).
male(edward).
female(alice).
female(victoria).
child(X,Y,Z) :- child(X,Z,Y).
child(edward,victoria,albert).
father(X,Y) :- child(Y,X,Z), male(X).
mother(X,Y) :- child(Y,X,Z), female(X).
```


?-father(albert,edward)
:-child(edward,albert,Z), male(albert)
:-child(edward,Z,albert),male(albert)
:-male(albert)
:-child(edward,albert,Z), male(albert)
□
⋮
:-male(albert)

Pruning the tree search: Cut

Sometimes, backtracking when a solution has been found is not useful..

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(albert,edward).
mother(victoria,edward).
```

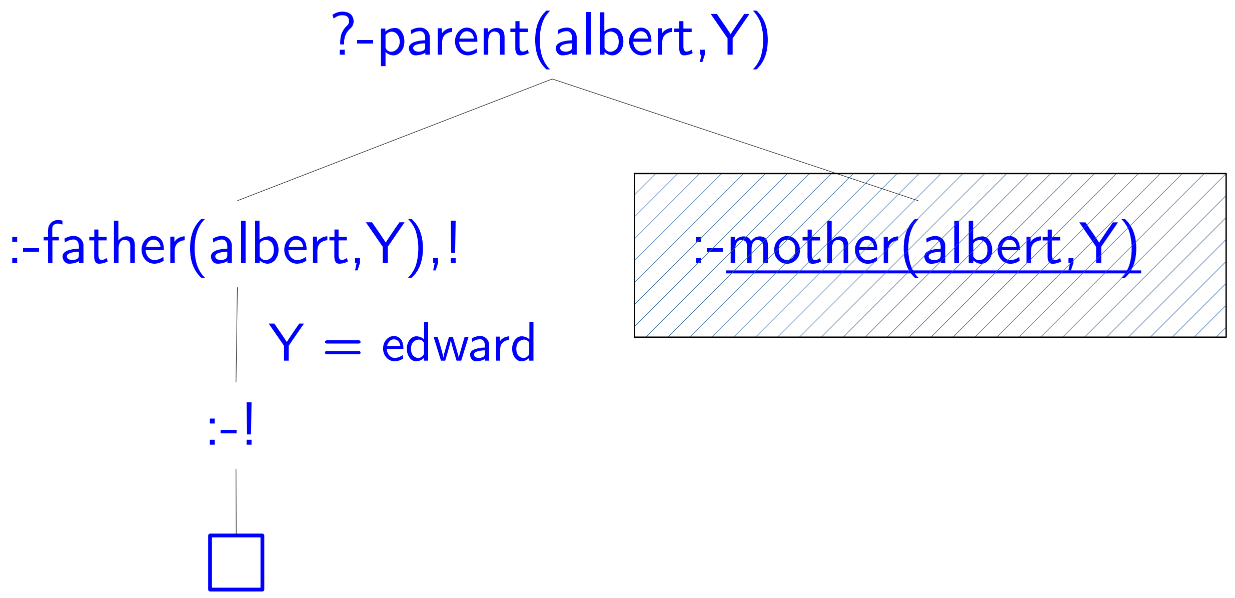


If the father(albert,Y) branch succeeds there is no point in examining mother(albert,Y), as these branches are intended as mutually exclusive.

Pruning the tree search: Cut

Sometimes, backtracking when a solution has been found is not useful..

```
parent(X,Y) :- father(X,Y),!.
parent(X,Y) :- mother(X,Y).
father(albert,edward).
mother(victoria,edward).
```



If the father(albert,Y) branch succeeds there is no point in examining mother(albert,Y), as these branches are intended as mutually exclusive.

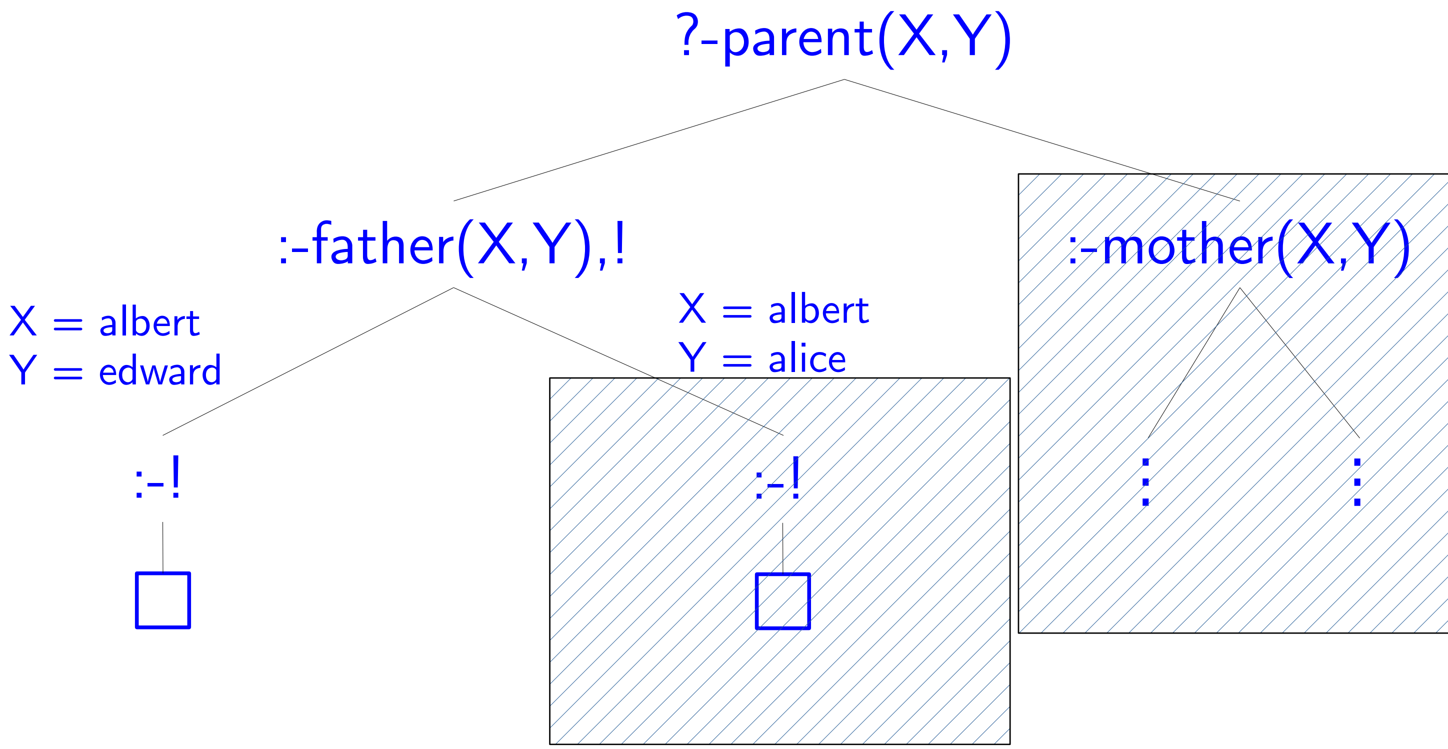
Adding a cut (!) in the first rule achieves this. The cut:

- always succeeds
- cancels all backtracking *from the node selecting it to the node using it*

Cut and side effects

Note that the use of cut can lead to unwanted side effects!

```
parent(X,Y) :- father(X,Y),!.
parent(X,Y) :- mother(X,Y).
father(albert,edward).
father(albert,alice).
mother(victoria,edward).
mother(victoria,alice).
```



The leftmost branch cuts out two other good branches!

We need to think operationally when using cut.

Negation as Failure

Recall that the literals we use in Prolog must be positive – there is no negation!

But we can *simulate* negation using cut:

```
not_q :- q,!,fail
not_q.
```

Thus, proving not_q amounts to first trying to prove q and:

- if that succeeds then not_q fails (fail is a goal that just fails)
- otherwise, not_q succeeds.

Negation as Failure – Side Effects Example

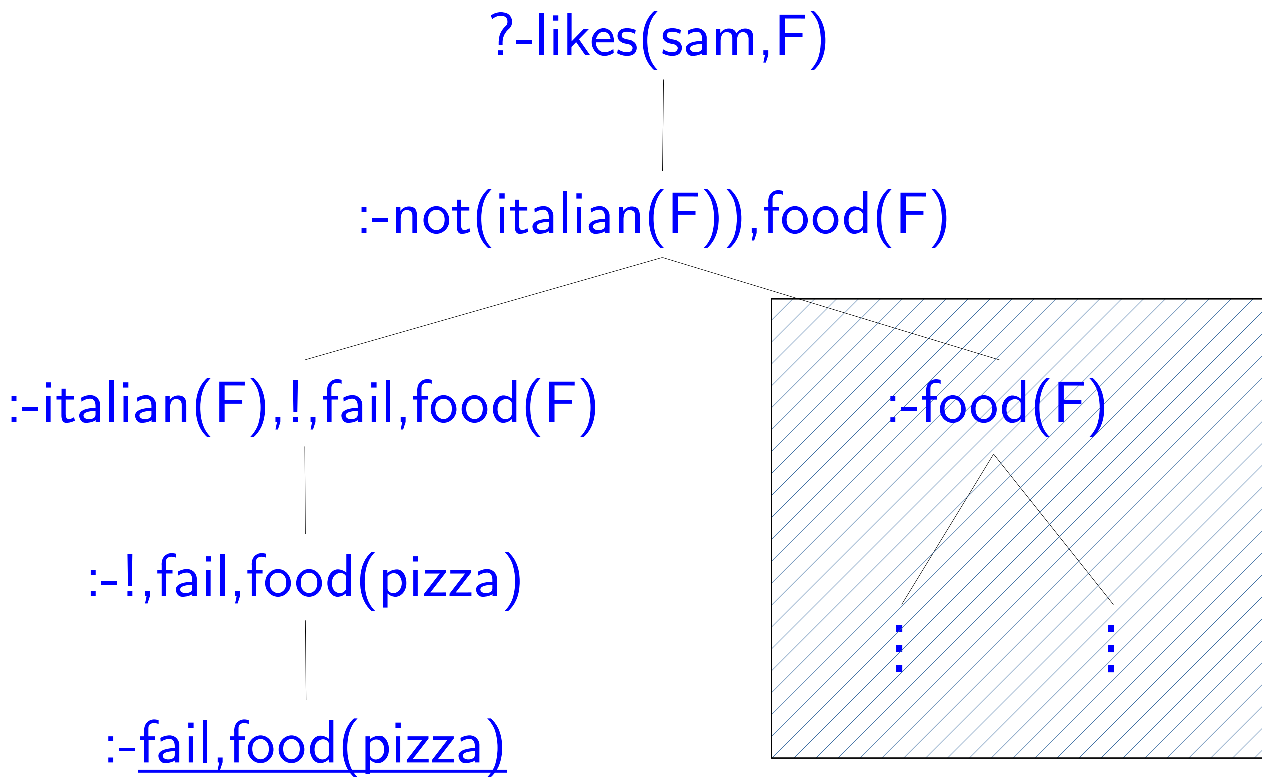
Suppose Sam likes any food that is not Italian.

```
food(F) :- indian(F).
food(F) :- greek(F).
food(F) :- italian(F).
indian(dahl).
indian(tandoori).
indian(kurma).
greek(souvlaki).
italian(pizza).
italian(spaghetti).
italian(lasagne).

likes(sam,F) :- not(italian(F)),food(F).
```

Negation as Failure – Side Effects Example

Suppose Sam likes any food that is not Italian.

```
food(F) :- indian(F).
food(F) :- greek(F).
food(F) :- italian(F).
indian(dahl).
indian(tandoori).
indian(kurma).
greek(souvlaki).
italian(pizza).
italian(spaghetti).
italian(lasagne).

likes(sam,F) :- not(italian(F)),food(F).
```

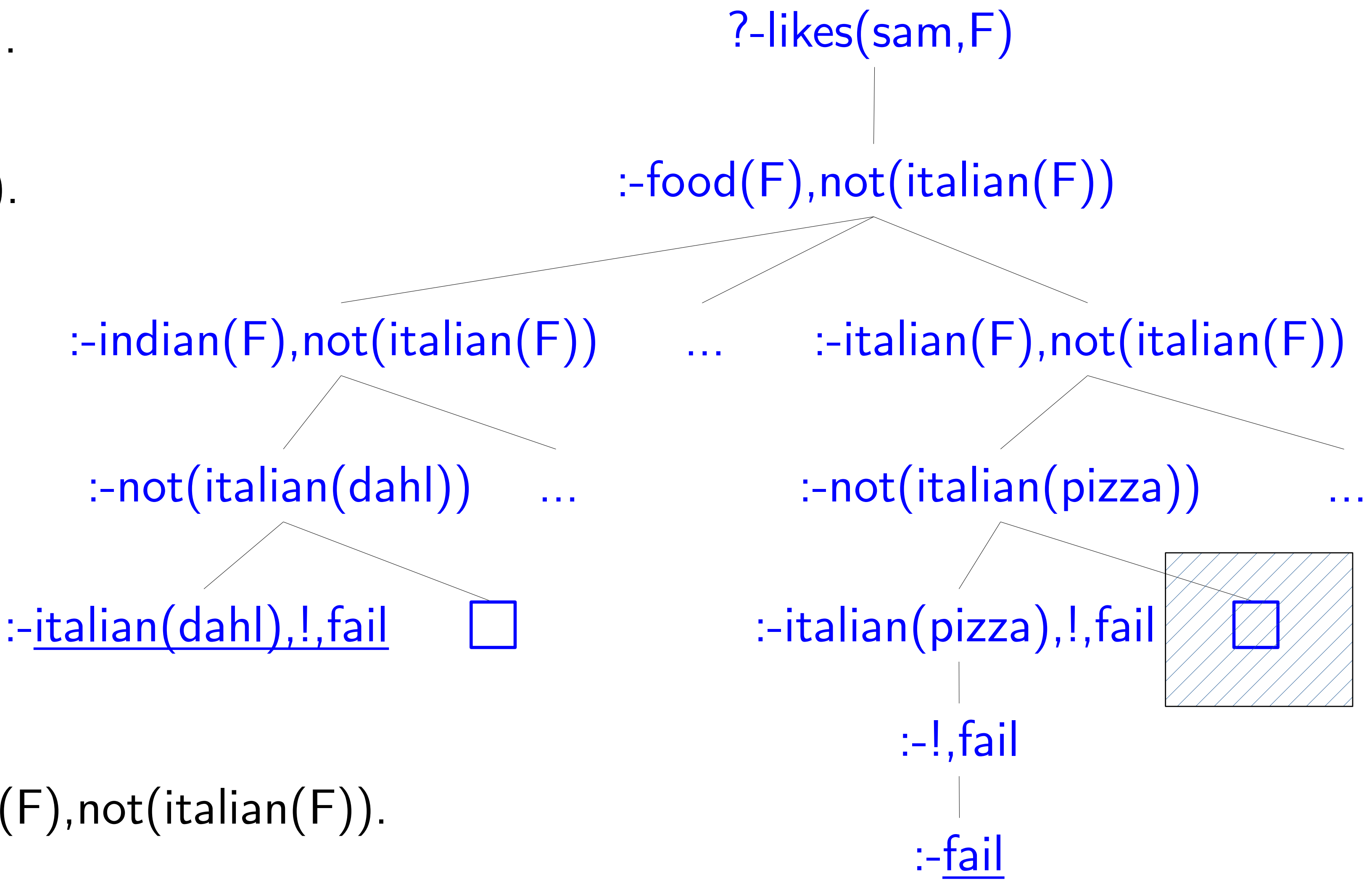


The above is actually saying that Sam likes any food so long as there are *no* Italian foods (in our axioms).

Negation as Failure – Side Effects Example

Suppose Sam likes any food that is not Italian.

```
food(F) :- indian(F).
food(F) :- greek(F).
food(F) :- italian(F).
indian(dahl).
indian(tandoori).
indian(kurma).
greek(souvlaki).
italian(pizza).
italian(spaghetti).
italian(lasagne).

likes(sam,F) :- food(F),not(italian(F)).
```



Swapping the order above we have: Sam likes any food so long as *it* is not Italian.

Negation as Failure

Recall that the literals we use in Prolog must be positive – there is no negation!

But we can *simulate* negation using cut:

```
not_q :- q,!,fail
not_q.
```

Thus, proving not_q amounts to first trying to prove q and:

- if that succeeds then not_q fails (fail is a goal that just fails)
- otherwise, not_q succeeds.

This looks like *the law of excluded middle* in classical logic:

- if φ is not true then $\neg\varphi$ must hold.

But there are two important differences. Prolog does proof search, so:

- not being able to prove q does not mean that q is false (*negation as failure*)
- the use of a cut can have unwanted side effects.

In Prolog, there is notation for this: not(G). (for any *goal G*)

Lists and recursion

Prolog is a powerful (Turing complete) programming language.
Its style of programming (declarative / recursive) favours the use of *lists*.

Lists in Prolog:	[],	[H|T]
These stand for:	empty list,	list with head element H and tail list T

For example: [1|[2|3|[]]] is the list [1,2,3] (which is valid Prolog notation)

Lists and recursion

Prolog is a powerful (Turing complete) programming language.
Its style of programming (declarative / recursive) favours the use of *lists*.

Examples (lists.pl):

- member(X,Y) *% Y is a member of list X*
- size(X,Y) *% the size of list X is Y*
- sum(X,Y) *% the sum of list X is Y*
- append(X,Y,Z) *% append Y to list X yields Z*

Applications

Some applications of Prolog:

- Knowledge representation and reasoning (e.g. semantic web)
- Automated planning and scheduling
- Constraint satisfaction problems
- Natural language processing
- Software verification
- Databases (e.g. Datalog)
- Games
- etc.

Summary

We briefly show the proof theory of predicate logic, and in particular the use of *resolution*.

We then focussed on Prolog: a logic programming language built on (predicate logic and) Horn clauses and resolution.

Programming in Prolog amounts to stating rules and facts (axioms) and proving queries by proof search.

Proof search in Prolog is based on DFS. It is efficient but may get stuck in recursion loops. Proof search can be neatly depicted using SLD trees.

Prolog uses the cut operation to prune the proof search operationally.

The cut allows us to encode (an idiomatic kind of) logical negation.

For more: *Simply Logical* by Peter Flach (available online) and swi-prolog.org