An introduction to logic programming and Prolog

For the course "Logic in Computer Science and Artificial Intelligence"

Anders Lundstedt

Stockholm university

October 2019

Aim of this lecture

- Give a brief background to logic programming and Prolog.
- Give an introduction to Prolog, roughly covering chapters 1–2 of "Learn Prolog Now!"[1] (LPN).
  - How to construct Prolog knowledge bases.
  - How to query Prolog knowledge bases.
  - How Prolog's proof search algorithm works.
- Also included in these slides, for your reference:
  - A proof that Prolog's proof search is sound.
  - Prolog's basic syntax.

[1] `http://www.learnprolognow.org`

What is logic programming?

- Deduction as computation: specify axioms, rules and goals; then apply a proof search procedure.
- Thus “declarative” as opposed to “imperative”: emphasis on what to compute rather than on how to compute it.
- In general, deduction is non-deterministic until a proof search procedure is given.

What is logic programming?

- Applications:
  - expert systems,
  - machine learning (via inductive logic programming[2]),
  - natural language processing,
  - querying relational databases.
- Examples:
  - IBM Watson (partly written in Prolog),
  - The Linux distribution NixOS and the package manager Nix (see `https://discourse.nixos.org/t/nixpkgs-nixos-is-an-expert-system-database/671`).

[2] See `https://web.archive.org/web/20151222194228/https://chronicles.mfglabs.com/big-data-small-data-and-the-role-of-logic-in-machine-learning-c5f9796765e9?gi=df1eb37baff8`.

What is logic programming?

Example: Deciding if a natural number is even.

- Let the natural numbers be given by the presentation

$$n := 0 \mid S(n).$$

- One way to decide if a number in this representation is even is to implement the following recursion:

$$E(n) = \begin{cases} \top & \text{if } n = 0, \\ \bot & \text{if } n = 1, \\ E(m) & \text{if } n = S(S(m)), \end{cases}$$

- Instead, in logic programming we specify axioms and inference rules:

$$\frac{}{0 \text{ even}} \qquad \frac{n \text{ even}}{S(S(n)) \text{ even}} \ .$$

What is Prolog?

- A general-purpose logic programming language.
- Based on first-order logic.
- Developed in the early 1970s by Colmerauer and Roussel (Marseilles) and Kowalski (Edinburgh).

Introduction to Prolog

- With Prolog, one constructs a *knowledge base* which consists of *facts* and *rules*. One uses a knowledge base by posing *queries* about the information represented by the knowledge base.
  - Facts:

    $$H\,.$$

    Intended interpration: Under any variable assignment, H is true.
  - Rules:

    $$H \;\texttt{:-}\; B_1, \ldots, B_{n+1}\,.$$

    Intended interpretation: If, under a variable assignment v, each of $B_1, \ldots, B_{n+1}$ is true, then H is true under v.
  - Queries:

    $$\texttt{?-}\; G_1, \ldots, G_n\,.$$

    Intended interpretation: Is there a variable assignment such that each of $G_1, \ldots, G_n$ follows from the facts and rules in the knowledge base?

- Examples of facts in Prolog:

```
playsAirGuitar(jody).
party.
```

  These say that that the unary predicate `playsAirGuitar` holds of the individual `jody` and that the proposition `party` is true, respectively.

- An example of a rule:

```
listens2music(yolanda) :- happy(yolanda).
```

  This rule says that if the unary predicate `happy` holds of the individual `yolanda` then the unary predicate `listens2music` holds of the individual `yolanda`; or put simpler: if Yolanda is happy then she listens to music.

- Another example:

```
jealous(X, Y) :- loves(X, Z), loves(Y, Z).
```

  This rule is formulated with variables—in Prolog any string beginning with an uppercase letter is a variable. It is to be read as: for any three terms u, v and w, if `loves(`u`,`w`)` holds and `loves(`v`,`w`)` holds then `jealous(`u`,`v`)` holds.

LPN's knowledge base 1:

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
```

Five facts about six atomic terms: two unary predicates (`woman`, `playsAirGuitar`), one proposition (`party`) and three individuals (`mia`, `jody`, `yolanda`).

LPN's knowledge base 2:

```
happy(yolanda).
listens2music(mia).
listens2music(yolanda) :- happy(yolanda).
playsAirGuitar(mia) :- listens2music(mia).
playsAirGuitar(yolanda) :- listens2music(yolanda).
```

Two facts and three rules. Does Mia play air guitar? Does Yolanda? Is there someone who plays air guitar and is happy?

LPN's knowledge base 5:

```
loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
jealous(X, Y) :- loves(X, Z), loves(Y, Z).
```

Is there someone who is jealous of Vincent?

- We can query a knowledge base by loading it in a Prolog interpreter.
- A query consists of a list of atomic predicates $G_1, \ldots, G_{n+1}$, possibly containing variables, constants and complex terms.
- A query is intended to be interpreted existentially: is there a variable assignment such that each of $G_1, \ldots, G_{n+1}$ follows from the facts and rules in the knowledge base?

- In Prolog, Queries are written:

$$?- \ G_1, \ldots, G_{n+1}.$$

- When given a query, Prolog will perform a proof search. Possible results are:
  - `false.`—for unsatisfiable queries ("negation by failure"),
  - `true.`—for satisfiable queries without variables,
  - a satisfying variable assignment—for satisfiable queries with variables,
  - non-termination.
- If satisfiable, Prolog will return a separate answer for each distinct proof. Thus Prolog returns all satisfying variable assignments.

Consider LPN's knowledge base 1:

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
```

Some queries:

```
?- woman(mia).
true.

?- playsAirGuitar(jody).
true.

?- playsAirGuitar(mia).
false.
```

Consider LPN's knowledge base 2:

```
happy(yolanda).
listens2music(mia).
listens2music(yolanda) :- happy(yolanda).
playsAirGuitar(mia) :- listens2music(mia).
playsAirGuitar(yolanda) :- listens2music(yolanda).
```

Does Mia play air guitar?

```
?- playsAirGuitar(mia).
true.
```

Is there someone who plays air guitar and is happy?

```
?- playsAirGuitar(X), happy(X).
X = yolanda.
```

Consider LPN's knowledge base 5:

```
loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
jealous(X, Y) :- loves(X, Z), loves(Y, Z).
```

Is there someone who is jealous of Vincent?

```
?- jealous(X, vincent).
X = vincent  ;
X = marcellus.
```

- We will limit ourselves to the fragment of Prolog restricted to *Horn clauses*. "Pure Prolog" sometimes (but not always) refers to this fragment.
- A Horn clause is a disjunction of literals with at most one unnegated literal:

$$P \vee \neg Q_1 \vee \cdots \vee \neg Q_n$$

or

$$\neg Q_1 \vee \cdots \vee \neg Q_n$$

($n \in \{0, 1, \dots\}$).

- Note: The empty disjunction (i.e. $\bot$) is a Horn clause.
- Note: In first-order logic as well as in Prolog, literals may contain variables and complex terms.

- A Horn clause

$$P \vee \neg Q_1 \vee \cdots \vee \neg Q_n$$

with exactly one unnegated literal is a *definite clause*. Only definite Horn clauses occur in a Prolog knowledge base.

- A definite clause with no negated literals ($n = 0$) is simply an atom P. Such clauses are called *facts*.
- A definite clause with at least one negated literal ($n > 0$) is called a *rule*. A rule can equivalently be written as the implication

$$P \leftarrow Q_1 \wedge \cdots \wedge Q_n.$$

In Prolog syntax:

$$P \ :- \ Q_1, \ldots, Q_{n+1}.$$

P is called the *head* of the rule and $Q_1, \ldots, Q_{n+1}$ is called the *body* of the rule.

- A Horn clause

$$\neg Q_1 \vee \cdots \vee \neg Q_n$$

  with no unnegated literal is a *goal clause*. In particular, the empty disjunction is a goal clause.
- In Prolog, a goal clause corresponds to a query. The above goal clause corresponds to the query

$$?- \ Q_1, \ldots, Q_n.$$

Classification of Horn clauses

		disjunction form	implication form
definite clauses	fact	P	$P \leftarrow \top$
	rule	$P \vee \neg Q_1 \vee \cdots \vee \neg Q_{n+1}$	$P \leftarrow Q_1 \wedge \cdots \wedge Q_{n+1}$
goal clause		$\neg Q_1 \vee \cdots \vee \neg Q_n$	$\bot \leftarrow Q_1 \wedge \cdots \wedge Q_n$

- Given a query

$$?- \ G_1, \ldots, G_{n+1}.$$

the corresponding goal clause is

$$\neg G_1 \vee \cdots \vee \neg G_{n+1}.$$

- Prolog performs proof search by resolution. By systematically applying the resolution inference rule it tries to derive the empty clause from the goal clause corresponding to the query together with the definite Horn clauses corresponding to the rules and facts in the knowledge base.

- Resolution of a definite Horn clause with a goal clause produces a new goal clause:

$$\frac{G_1 \vee \neg B_1 \vee \cdots \vee \neg B_m \qquad \neg G_1 \vee \cdots \vee \neg G_n}{\neg B_1 \vee \cdots \vee \neg B_m \vee \neg G_2 \vee \cdots \vee \neg G_n} .$$

- In Prolog syntax for rules:

$$\frac{G_1 \texttt{ :- } B_1, \ldots, B_m . \qquad \texttt{?- } G_1, \ldots, G_n .}{\texttt{?- } B_1, \ldots, B_m, G_2, \ldots, G_n .} .$$

  For facts:

$$\frac{G_1 . \qquad \texttt{?- } G_1, \ldots, G_n .}{\texttt{?- } G_2, \ldots, G_n .} .$$

- When given a query

$$?- \; G_1, \ldots, G_n .$$

  Prolog searches the knowledge base from the top for a fact or head of a rule that can be unified with G_1. Applying resolution this either derives the empty clause, in which case a satisfying assignment has been found, or a new nonempty goal clause, in query form

$$?- \; G'_1, \ldots, G'_{n'} .$$

  to which the procedure can be repeated.
- When unification fails or when the user asks for additional solutions, Prolog backtracks: it forgets the last unification and continues searching for another fact or head that unifies with G_1. If no other unifications exists then Prolog returns `false`.

Suppose we have the following knowledge base.

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

Suppose we ask the query

```
?- k(Y), h(Y).
```

Prolog's proof search will execute as follows.

- Start with the list of goals `k(Y), h(Y).`
- Scan the knowledge base down from the top for a head of a rule or a fact that unifies with the first goal `k(Y)`.
- First match: `k(X) :- f(X), g(X).` Unifier: [X/Y].
- Replace `k(Y)` in the list of goals with `f(X), g(X)` and perform the substitution [X/Y] in the remaining goals.

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

New list of goals: `f(X), g(X), h(X).`

- Scan the knowledge base down from the top for a head of a rule or a fact that unifies with the first goal `f(X)`.
- First match: `f(a).` Unifier: [a/X].
- Remove `f(X)` from the list of goals and perform the substitution [a/X] in the remaining goals.

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

New list of goals: `g(a), h(a).`

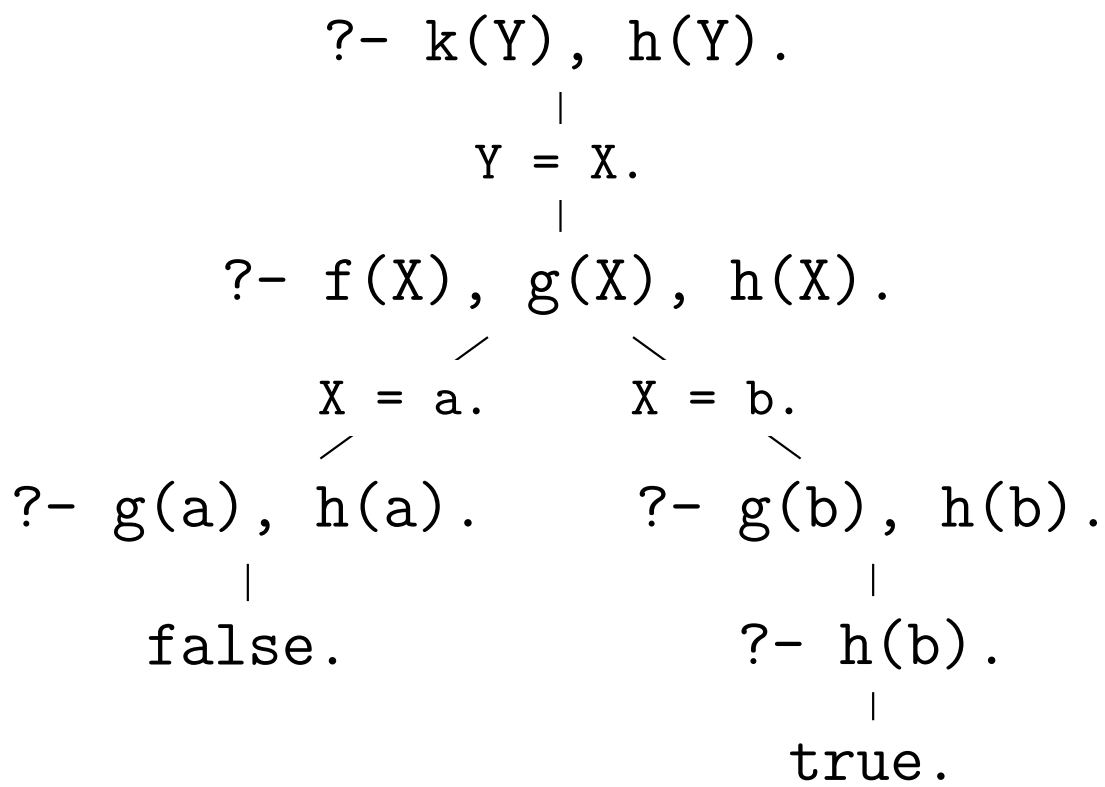
- Scan the knowledge base down from the top for a head of a rule or a fact that unifies with the first goal `g(a)`.
- No match: backtrack. Undo the last unification and add back the goal `f(X)`. New list of goals: `f(X), g(X), h(X).`
- Scan for next head of a rule or fact that unifies with `f(X)`.
- First match: `f(b)`. Unifier: [b/X].
- Remove `f(X)` from the list of goals and perform the substitution [b/X] in the remaining goals.

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

New list of goals: `g(b), h(b).`

- Scan the knowledge base down from the top for a head of a rule or a fact that unifies with the first goal `g(b)`.
- Match: `g(b)`. Unifier: []. New goal: `h(b)`.
- Scan the knowledge base down from the top for a head of a rule or a fact that unifies with the remaining goal `h(b)`.
- Match: `h(b)`. Unifier: []. No remaining goal. Solution given by composition of computed unifiers: $[\mathtt{X}/\mathtt{Y}][\mathtt{b}/\mathtt{X}][][] = [\mathtt{b}/\mathtt{X}, \mathtt{b}/\mathtt{Y}]$; thus $\mathtt{Y} = \mathtt{b}$.

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



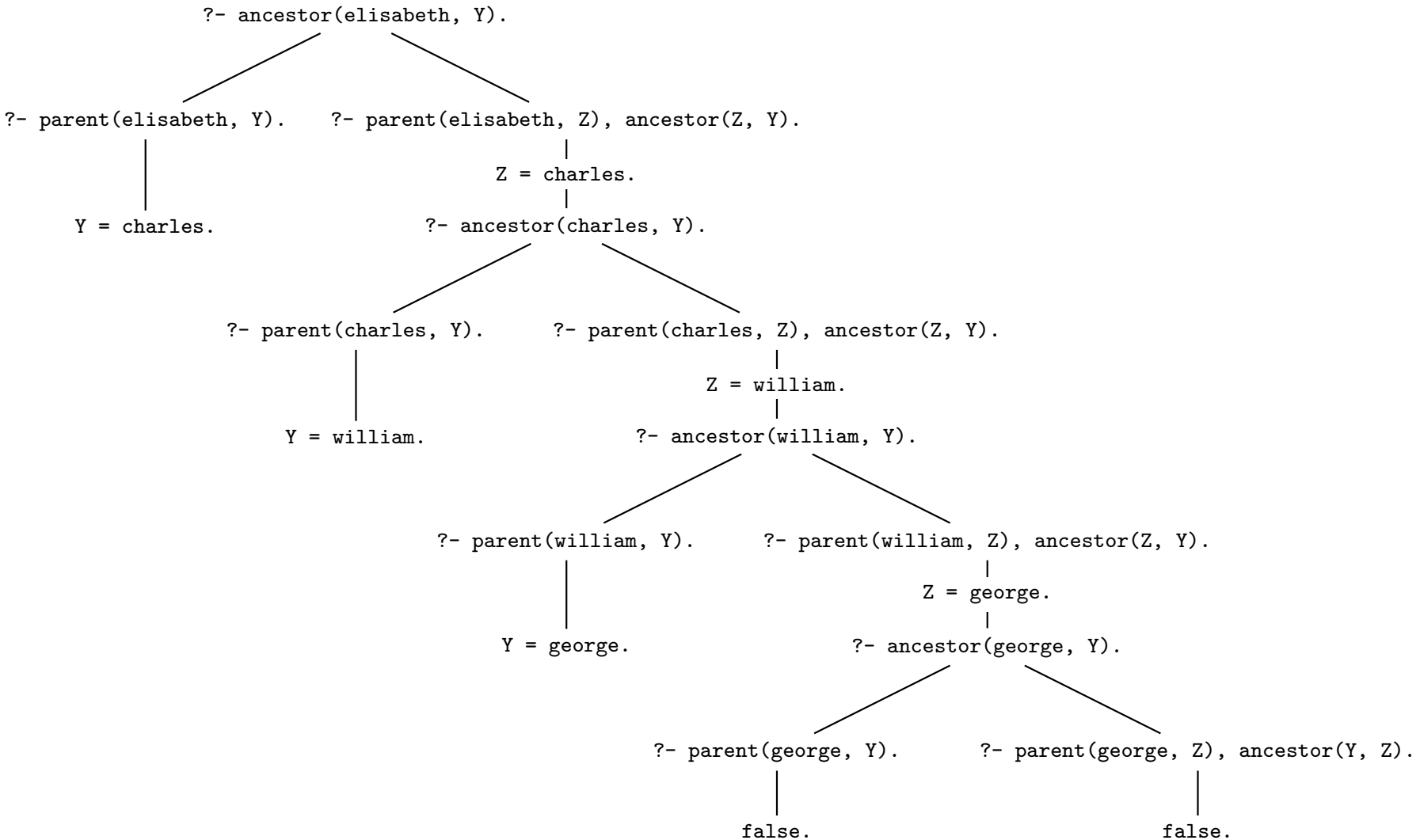
The knowledge base of royal relations:

```
parent(elisabeth, charles).
parent(charles, william).
parent(diana, william).
parent(diana, harry).
parent(william, george).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

- Note: Two rules with head `ancestor(X, Y)`. The second rule is recursive: `ancestor` with arity 2 occurs both in head and body.
- Who are Elisabeth's descendants?

```
?- ancestor(elisabeth, Y).
Y = charles ;
Y = william ;
Y = george.
```

Search tree of `?- ancestor(elisabeth, Y).`



Why does Prolog's proof search method answer the query (if it terminates)?

- Recall the intended interpretation of the query: is there a variable assignment such that each of $G_1, \ldots, G_{n+1}$ follows from the facts and rules in the knowledge base?
- Let Γ be the set of (universally quantified) definite Horn clauses corresponding to the rules and facts of the knowledgebase.
- Let $X_1, \ldots, X_m$ be the variables occurring in $G_1, \ldots, G_{n+1}$.
- With the query we are equivalently asking if

$$\Gamma \models \exists X_1 \cdots \exists X_m (G_1 \wedge \cdots \wedge G_{n+1}).$$

- By soundness and completeness of resolution this is equivalent to

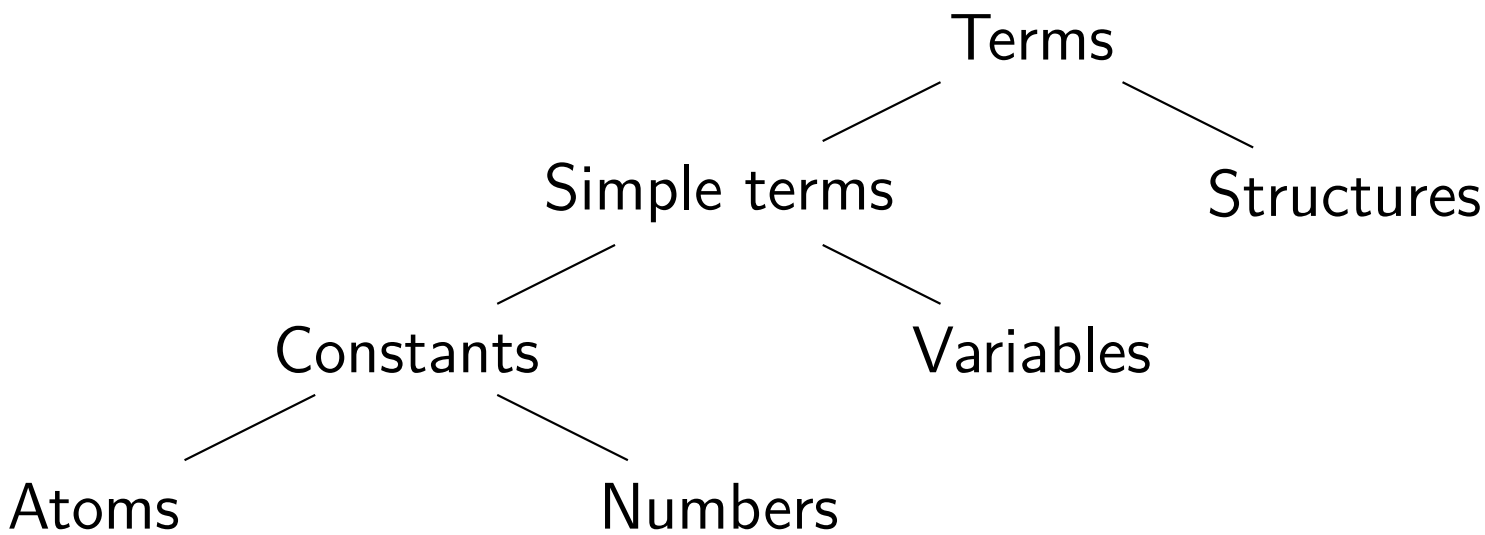
$$\Gamma \vdash_{\mathrm{RES}} \exists X_1 \cdots \exists X_m (G_1 \wedge \cdots \wedge G_{n+1}).$$

  Negating the conclusion and transforming it to normal form we get

$$\begin{aligned} \neg\exists X_1 \cdots \exists X_m (G_1 \wedge \cdots \wedge G_{n+1}) &\equiv \forall X_1 \cdots \forall X_m \neg (G_1 \wedge \cdots \wedge G_{n+1}) \\ &\equiv \forall X_1 \cdots \forall X_m (\neg G_1 \vee \cdots \vee \neg G_{n+1}), \end{aligned}$$

  i.e. (the universal closure of) the goal clause corresponding to the query.
- Thus, if we derive the empty clause via resolution from the goal clause together with the definite clauses corresponding to the facts and rules of the knowledge base then we have proved the query satisfiable. Furthermore, the composition of computed unifiers provide a satisfying assignment.

- All expressions in a knowledge base or in a query are built from *terms*. A term is either an *atom*, a *number*, a *variable* or a *structure* (complex term).
- Atoms and numbers are called *constants*. Constants and variables are called *simple* terms.



- For our purposes, an atom is a string of characters made up of upper-case letters, lower-case letters, digits and the underscore character, starting with a lower-case letter. Examples:
  - `butch`
  - `big_kahuna_burger`
  - `listens2Music`
  - `playsAirGuitar`
- A number is an integer or a floating point number written in decimal form. (Numbers are not used in this lecture.)
- A variable is a string of characters made up of upper-case letters, lower-case letters, digits and the underscore character, starting with either an upper-case letter or an underscore. Examples:
  - `X`
  - `Butch`
  - `_tag`
  - `_518`

- A structure, or complex term, is built from an atom immediately followed by comma-separated list of terms enclosed in parenthesis. This is an inductive definition:

$$\frac{A \text{ atom} \qquad t_1, \ldots, t_n \text{ terms}}{A(t_1, \ldots, t_n) \text{ structure}} .$$

- The atom A above is the structure's *functor*.
- The number n of terms above is the structure's *arity*.
- Examples:
  - `listens2Music(yolanda)`
  - `loves(X, Y)`
  - `even(s(s(z)))`
- Note: Any atom can occur both as a functor and as an argument to a functor. Thus Prolog does not enforce a syntactic distinction between function symbols and predicate symbols (contrary to first-order logic).

- A fact is a term t followed by a dot:

$$t\,.$$

- A rule is a term h (the head of the rule) followed by `:-` and a comma-separated list of terms $b_1, \ldots, b_{n+1}$ (the body of the rule) followed by a dot:

$$h \;\texttt{:-}\; b_1, \ldots, b_{n+1}\,.$$

- A knowledge base is a sequence of facts and rules separated by whitespace (typically newlines).
- A query is the prefix `?-` followed by a comma-separated list of terms $g_1, \ldots, g_{n+1}$ followed by a dot:

$$\texttt{?-}\; g_1, \ldots, g_{n+1}\,.$$

Learn Prolog

- Prolog is a programming language. The best way to learn a programming language is to write code.
  - SWI Prolog is a free Prolog interpreter: `http://www.swi-prolog.org/`.
  - Online version (runs directly in browser) available at: `http://swish.swi-prolog.org/`.
  - Many other implementations of Prolog exist.
- For further introductory reading: "Learn Prolog Now!", `http://www.learnprolognow.org/`.