LOGIC-PROGRAMMING IN PROLOG

Claus Brabrand
brabrand@itu.dk
IT University of Copenhagen

[ http://www.itu.dk/people/brabrand/ ]
Plan for Today

- Scene V: "Monty Python and The Holy Grail"
- Lecture: "Relations & Inf. Sys." (10:15 – 11:00)
- Exercise 1 (11:15 – 12:00)
- Lunch break (12:00 – 12:30)
- Lecture: "PROLOG & Matching" (12:30 – 13:15)
- Lecture: "Proof Search & Rec" (13:30 – 14:15)
- Exercises 2+3 (14:30 – 15:15)
- Exercises 4+5 (15:30 – 16:15)
Outline (three parts)

Part 1:
- "Monty Python and the Holy Grail" (Scene V)
- Relations & Inference Systems

Part 2:
- Introduction to PROLOG (by-Example)
- Matching

Part 3:
- Proof Search (and Backtracking)
- Recursion
Keywords:
Movie(!)

  - Scene V: "The Witch":
    - Image of the Monty Python DVD cover.

CLAUS BRABRAND

PROGRAMMING PARADIGMS

OCT 30, 2007
"Axioms" (aka. "Facts"):

- female(girl). %- by observation
- floats(duck). %- King Arthur
- sameweight(girl,duck). %- by experiment

"Rules":

- witch(X) :- female(X), burns(X).
- burns(X) :- wooden(X).
- wooden(X) :- floats(X).
- floats(X) :- sameweight(X,Y), floats(Y).
Inductive Reasoning: \texttt{witch(girl)}

"Induction": (aka. "bottom-up reasoning")

- \texttt{witch(X) :- female(X), burns(X).}
- \texttt{burns(X) :- wooden(X).}
- \texttt{wooden(X) :- floats(X).}
- \texttt{floats(X) :- sameweight(X,Y), floats(Y).}
- \texttt{sameweight(girl,duck)}

\% by observation

\texttt{female(girl)}

\texttt{burns(girl)}

\texttt{wooden(girl)}

\texttt{floats(girl)}

\texttt{floats(duck)}

\% by experiment

\% King Arthur
"Deduction": (aka. "top-down reasoning")

- By observation:
  - female(girl)
  - burns(girl)

- By experiment:
  - wooden(girl)
  - floats(girl)

- Sameweight:
  - sameweight(girl, duck)
  - floats(duck)

- King Arthur:
  - floats(duck)

- Deduction:
  - witch(girl)
**Induction vs. Deduction**

- **Induction**
  - (aka. “bottom-up reasoning”):
    - Specific → General
    - (or: concrete → abstract)

- **Deduction**
  - (aka. “top-down reasoning”):
    - General → Specific
    - (or: abstract → concrete)

“Same difference” (just two different directions of reasoning...)

**Deduction ↔ Induction**
(just swap directions of arrows)
"DEDUCTIVE vs. INDUCTIVE REASONING"

**LEVIN:**

"You in my office discussed, I think, a very interesting approach, which is the difference between starting with a conclusion and trying to prove it and instead starting with digging into all the facts and seeing where they take you.

Would you just describe for us that difference and why [...]?"

**HAYDEN:**

"Yes, sir. And I actually think I prefaced that with both of these are legitimate forms of reasoning,

- that you've got **deductive** [...] in which you begin with, first, [general] principles and then you work your way down the specifics.

- And then there's an **inductive approach** to the world in which you start out there with all the data and work yourself up to general principles.

*They are both legitimate.*
Inference Systems

Keywords:
relations, axioms, rules, fixed-points
### Relations

**Example 1:** "even" relation: \( l_{\text{even}} \subseteq \mathbb{Z} \)

- Written as: \( l_{\text{even}} 4 \)
- … and as: \( \text{even} 5 \)

\[ \begin{align*}
4 & \in l_{\text{even}} \\
5 & \not\in l_{\text{even}}
\end{align*} \]

**Example 2:** "equals" relation: \( '=' \subseteq \mathbb{Z} \times \mathbb{Z} \)

- Written as: \( 2 = 2 \)
- … and as: \( 2 \neq 3 \)

\[ \begin{align*}
(2,2) & \in '=' \\
(2,3) & \not\in '='
\end{align*} \]

**Example 3:** "DFA transition" relation: \( '\rightarrow' \subseteq \mathbb{Q} \times \Sigma \times \mathbb{Q} \)

- Written as: \( q \xrightarrow{\sigma} q' \)
- … and as: \( p \xrightarrow{\sigma} p' \)

\[ \begin{align*}
(q, \sigma, q') & \in '\rightarrow' \\
(p, \sigma, p') & \not\in '\rightarrow'
\end{align*} \]
Inference System

- Inference System:
  - Inductive (recursive) specification of relations
  - Consists of axioms and rules

Example: $\vdash \text{even} \subseteq \mathbb{Z}$

Axiom: $\vdash \text{even} \ 0$

- “0 (zero) is even”!

Rule: $\vdash \text{even} \ n \ m = n+2$

- “If $n$ is even, then $m$ is even (where $m = n+2$)”
Terminology

- **Meaning:**
  - **Inductive:**
    "If \( n \) is even, then \( m \) is even (provided \( m = n+2 \))"; or
  - **Deductive:**
    "\( m \) is even, if \( n \) is even (provided \( m = n+2 \))"
?– Abbreviation

- Often, rules are *abbreviated*:

  - **Rule:** \[
    \begin{array}{c}
    \text{even } n \\
    \hline \\
    \text{even } m \\
    \hline \\
    m = n+2
    \end{array}
  \]
  
  - “If \( n \) is even, then \( m \) is even (provided \( m = n+2 \))”; or
  
  - “\( m \) is even, if \( n \) is even (provided \( m = n+2 \))”

  - **Abbreviated rule:** \[
    \begin{array}{c}
    \text{even } n \\
    \hline \\
    \text{even } n+2
    \end{array}
  \]
  
  - “If \( n \) is even, then \( n+2 \) is even”; or
  
  - “\( n+2 \) is even, if \( n \) is even”
Relation Membership? $x \in \mathbb{R}$

- **Axiom:**
  
  $\bot \vdash \text{even } 0$
  
  “0 (zero) is even”!

- **Rule:**
  
  $\vdash \text{even } n$  
  $\vdash \text{even } n + 2$
  
  “If $n$ is even, then $n + 2$ is even”

- **Is 6 even?!?**

  ![Inference Tree]

  The inference tree proves that: $\vdash \text{even } 6$
**Example:** “less-than-or-equal-to”

- **Relation:** 
  \[ \leq \subseteq \mathbb{N} \times \mathbb{N} \]

- ** Activation exercise**

- **Is ”1 ≤ 2” ? (why/why not)!?**
  - Yes, because there exists an inference tree:
    - In fact, it has two inference trees:
Activation Exercise:

1. Specify the signature of the relation: '<<'
   - x << y  "y is-double-that-of x"

2. Specify the relation via an inference system
   - i.e. axioms and rules

3. Prove that indeed:
   - 3 << 6  "6 is-double-that-of 3"
Activation Exercise:

1. Specify the signature of the relation: '

\[ x \div y \quad \text{"x is-half-that-of y"} \]

2. Specify the relation via an inference system
   i.e. axioms and rules

3. Prove that indeed:
   \[ 3 \div 6 \quad \text{"3 is-half-that-of 6"} \]

Syntactically different
Semantically the same relation
Relation vs. Function

A function...

- \( f : A \rightarrow B \)

...is a relation

- \( R_f \subseteq A \times B \)

...with the special requirement:

- \( \forall a \in A, b_1, b_2 \in B: R_f(a, b_1) \land R_f(a, b_2) \implies b_1 = b_2 \)

i.e., "the result", \( b \), is uniquely determined from "the argument", \( a \).
The (2-argument) \textit{function} ' + ' ...

\vspace{-1em}

\[ + : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \]

...induces a (3-argument) \textit{relation}

\[ \mathbb{R}_+ \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N} \]

...that obeys:

\[ \forall n, m \in \mathbb{N}, \ r_1, r_2 \in \mathbb{N}: \quad R_+ (n, m, r_1) \land R_+ (n, m, r_2) \Rightarrow r_1 = r_2 \]

\[ i.e., \ "the \ result", \ r, \ is \ \textit{uniquely} \ determined \ from \ "the \ arguments", \ n \ and \ m \]
Example: “add”

Relation: \( + \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N} \)

Is \(2 + 2 = 4\) ?!

Yes, because there exists an inf. tree for \((2,2,4)\):

\[
\begin{align*}
+(0,2,2) \\
+(1,2,3) \\
+(2,2,4)
\end{align*}
\]
- **Relation Definition (Interpretation)**

- Actually, an inference system: \( \vdash_R \subseteq \mathbb{Z} \)

  - [axiom₁] \( \vdash_R 0 \)
  - [rule₁] \( \vdash_R n \Rightarrow \vdash_R n+2 \)

  - **Demand specification** for a relation:

    \[
    \begin{align*}
    (0 \in \{R\}' & \land (\forall n \in \{R\}' \Rightarrow n+2 \in \{R\}'))
    \end{align*}
    \]

- The three relations:
  - \( R = \{0, 2, 4, 6, \ldots\} \) (aka., \( 2\mathbb{N} \))
  - \( R' = \{0, 2, 4, 5, 6, 7, 8, \ldots\} \)
  - \( R'' = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \) (aka., \( \mathbb{Z} \))

  - **Satisfy** the (above) specification!
Inductive Interpretation $(\star)$

A inference system: $\vdash_{\mathcal{R}} \subseteq \mathbb{Z} \iff \vdash_{\mathcal{R}} \in P(\mathbb{Z})$

- [axiom$_1$] $\vdash_{\mathcal{R}} 0$
- [rule$_1$] $\frac{\vdash_{\mathcal{R}} n}{\vdash_{\mathcal{R}} n+2}$

...induces a function: $F_{\mathcal{R}} : P(\mathbb{Z}) \to P(\mathbb{Z})$

From rel. to rel.

\[ F_{\mathcal{R}}(\mathcal{R}) = \{0\} \cup \{ n+2 \mid n \in \mathcal{R} \} \]

Definition:

\[ \vdash_{\text{even}} := \text{lfp}(F_{\mathcal{R}}) = \bigcup_n F_{\mathcal{R}}^n(\emptyset) \]

'ifp' (least fixed point) ~ least solution:

\[ F(\emptyset) = \{0\} \bigcup F^2(\emptyset) = F(\{0\}) = \{0,2\} \bigcup F^3(\emptyset) = F^2(\{0\}) = F(\{0,2\}) = \{0,2,4\} \bigcup \ldots = 2\mathbb{N} \]

\[ F^n(\emptyset) \sim \text{“Anything that can be proved in ‘n’ steps”} \]
Exercise 1:

11:15 – 12:00
Purpose:

Learn how to describe relations via inference systems (in Prolog)

Exercise 1 (relations via inference systems in Prolog)

Purpose: to learn how to describe relations via inference systems (in Prolog):

- **Unary: odd/1**
  - a) Determine the arity and signature of the odd relation, written \( \text{odd}(n) \) ("N is an odd number"), on natural numbers.
  - b) Define the relation formally via an inference system (using only constant addition in the rules).
  - c) Prove that: \( \text{odd}(5) \) (in terms of your definition).
  - d) Rewrite your inference system so that it instead uses a unary \( \text{suc}(\cdot) \) encoding of numerals (cf. Section 3.1.3).
  - e) Implement this inference system in Prolog as a predicate \( \text{odd/1} \).
  - f) Prove that: \( \text{odd}(\text{suc}(\text{suc}(\text{suc}(\text{suc}(0)))))) \) (using Prolog) and explain how Prolog establishes this.

- **Binary: double/2**
  - a) Repeat steps a)-f) but for the binary double relation, written \( X \ll Y \) ("Y is double that of X"), on natural numbers (using only constant addition in the rules).
  - b) Prove that: \( 2 \ll 4 \) and \( \text{double}(\text{suc}(\text{suc}(0)), \text{suc}(\text{suc}(\text{suc}(\text{suc}(0)))))) \), respectively.

- **Ternary: congruent/3 [hardish]**
  - a) Repeat steps a)-f) but for the binary congruent relation, written \( X \equiv Z \) ("X is congruent with Y modulo Z (for Y < Z)", on natural numbers (using only less-than-or-equal, constant, and/or binary addition in the rules).
  - b) Prove that: \( 5 \equiv 1 \) and \( \text{congruent}(\text{suc}(\text{suc}(\text{suc}(\text{suc}(\text{suc}(\text{suc}(0)))))), \text{suc}(0), \text{suc}(\text{suc}(\text{suc}(\text{suc}(0)))))) \), respectively.
  - c) [I give up; give me a hint]
INTRODUCTION TO PROLOG
(by example)

Keywords:
Logic-programming, Relations, Facts & Rules, Queries, Variables, Deduction, Functors, & Pulp Fiction :(
We'll use the **on-line** material:

"Learn Prolog Now!"

[Patrick Blackburn, Johan Bos, Kristina Striegnitz, 2001]

[http://www.coli.uni-saarland.de/~kris/learn-prolog-now/]
A French programming language (from 1971):
- "Programmation en Logique" (="programming in logic")

A **declarative, relational** style of programming based on **first-order logic**:
- Originally intended for **natural-language processing**, but has been used for many different purposes (esp. for programming **artificial intelligence**).

The programmer writes a **"database"** of **"facts"** and **"rules"**;

```
%- FACTS -----------
female(girl).
floats(duck).
sameweight(girl,duck).

%- RULES -----------
witch(X)  :- burns(X) , female(X).
burns(X)  :- wooden(X).
wooden(X) :- floats(X).
floats(X) :- sameweight(X,Y) , floats(Y).
```

The user then supplies a **"goal"** which the system **attempts to prove deductively** (using **resolution** and **backtracking**); e.g., `witch(girl)`.
Operational vs. Declarative Programming

- **Operational Programming:**
  - The programmer specifies *operationally*:
    - *how* to obtain a solution
  - Very dependent on operational details

- **Declarative Programming:**
  - The programmer *declares*:
    - *what* are the properties of a solution
  - (Almost) Independent on operational details

**PROLOG:**

"The programmer describes the logical properties of the result of a computation, and the interpreter searches for a result having those properties".
Facts, Rules, and Queries

There are only 3 basic constructs in PROLOG:

- **Facts** \{ "knowledge base" (or "database") \}
- **Rules**
- **Queries** (goals that PROLOG attempts to prove)

*Programming in PROLOG is all about writing knowledge bases.*

*We use the programs by posing the right queries.*
Introductory Examples

Five example (knowledge bases)

...from "Pulp Fiction":

...in increasing complexity:

- **KB1**: Facts only
- **KB2**: Rules
- **KB3**: Conjunction ("and") and disjunction ("or")
- **KB4**: N-ary predicates and variables
- **KB5**: Variables in rules
KB1: Facts Only

KB1:

% FACTS:
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).

Basically, just a collection of facts:
- Things that are unconditionally true;
- e.g. "mia is a woman"

We can now use KB1 interactively:

?- woman(mia).
  Yes

?- woman(jody).
  Yes

?- playsAirGuitar(jody).
  Yes

?- playsAirGuitar(mia).
  No
Rules

Syntax: \texttt{head} :- \texttt{body}.

Semantics:

"If the body is true, then the head is also true"

To express \textit{conditional} truths:

- e.g., \texttt{playsAirGuitar(mia) :- listensToMusic(mia)}.
- i.e., "Mia plays the air-guitar, \textit{if she listens to music}"

\textbf{Prolog} then uses the following deduction principle (called: "modus ponens"):

\begin{align*}
H :- B & \quad \text{// If } B, \text{ then } H \text{ (or } "H \leq B") \\
B & \quad \text{// } B. \\
\hline \text{Therefore, } H.
\end{align*}
KB2 contains 2 facts and 3 rules:

- **Facts:**
  - listensToMusic(mia).
  - happy(yolanda).

- **Rules:**
  - playsAirGuitar(mia) :- listensToMusic(mia).
  - playsAirGuitar(yolanda) :- listensToMusic(yolanda).
  - listensToMusic(yolanda) :- happy(yolanda).

Which define 3 predicates: (listensToMusic, happy, playsAirGuitar)

Prolog is now able to deduce...

?- playsAirGuitar(mia).
   Yes

?- playsAirGuitar(yolanda).
   Yes

...using "modus ponens":

\[
\begin{align*}
\text{playsAirGuitar}(\text{mia}) & \quad \text{:-} & \quad \text{listensToMusic}(\text{mia}). \\
\text{listensToMusic}(\text{mia}) & \quad \text{:-} & \quad \text{listensToMusic}(\text{mia}). \\
\hline
\text{playsAirGuitar}(\text{mia}) & \quad \text{:-} & \quad \text{listensToMusic}(\text{mia}). \\
\end{align*}
\]

...combined with...

\[
\begin{align*}
\text{listensToMusic}(\text{yolanda}) & \quad \text{:-} & \quad \text{happy}(\text{yolanda}). \\
\text{happy}(\text{yolanda}) & \quad \text{:-} & \quad \text{listensToMusic}(\text{yolanda}). \\
\hline
\text{playsAirGuitar}(\text{yolanda}) & \quad \text{:-} & \quad \text{listensToMusic}(\text{yolanda}). \\
\end{align*}
\]
？– **Conjunction and Disjunction**

- Rules may contain multiple bodies (which may be combined in two ways):

  - **Conjunction** (aka. "and"):
    - \[ \text{playsAirGuitar(} \text{vincent} \text{)} :- \text{listensToMusic(} \text{vincent} \text{)}, \text{happy(} \text{vincent} \text{)}. \]
    - i.e., "Vincent plays, if he listens to music and he's happy".

  - **Disjunction** (aka. "or"):
    - \[ \text{playsAirGuitar(} \text{butch} \text{)} :- \text{listensToMusic(} \text{butch} \text{)}, \text{happy(} \text{butch} \text{)}. \]
    - i.e., "Butch plays, if he listens to music or he's happy".

  ...which is the same as (**preferred**):

    - \[ \text{playsAirGuitar(} \text{butch} \text{)} :- \text{listensToMusic(} \text{butch} \text{)}. \]
    - \[ \text{playsAirGuitar(} \text{butch} \text{)} :- \text{happy(} \text{butch} \text{)}. \]
KB3 defines 3 predicates:

- happy(vincent).
- listensToMusic(butch).

playsAirGuitar(vincent) :- listensToMusic(vincent), happy(vincent).
playsAirGuitar(butch) :- happy(butch).
playsAirGuitar(butch) :- listensToMusic(butch).

?- playsAirGuitar(vincent).
No

?- playsAirGuitar(butch).
Yes

...because we cannot deduce:
listensToMusic(vincent).

playsAirGuitar(butch) :- listensToMusic(butch).
listensToMusic(butch).

?- playsAirGuitar(butch).
playsAirGuitar(butch).

...using the last rule above
Interaction with **Variables** (in upper-case):

PROLOG tries to *match* `woman(X)` against the rules (from top to bottom) using `X` as a placeholder for anything.

More complex query:

```prolog
?- loves(marcellus,X), woman(X).
X = mia
```
KB5: Variables in Rules

i.e., "X is jealous-of Y, if there exists someone Z such that X loves Z and Y also loves Z".

(statement about everything in the knowledge base)

Query: ?- jealous(marcellus,Who).
Who = vincent

Q: Any other jealous people in KB5?
Terms:

- **Atoms** (first char lower-case or is in quotes):
  - a, vincent, vincentVega, big_kahuna_burger, ...
  - 'a', 'Mia', 'Five dollar shake', '#!%@*', ...

- **Numbers** (usual):
  - ..., -2, -1, 0, 1, 2, ...

- **Variables** (first char upper-case or underscore):
  - X, Y, X_42, Tail, _head, ...  
    ("_" special variable)

- **Complex terms** (aka. "structures"):
  - \[ f(term_1, term_2, ..., term_n) \]
    (f is called a "functor")
  - a(b), woman(mia), woman(X), loves(X,Y), ...
  - father(father(jules)), f(g(X),f(y)), ... (nested)
Implicit Data Structures

- **Prolog** is an *untyped* language.

- Data structures are *implicitly defined* via constructors (aka. "functors"): e.g. \[ \text{cons}(x, \text{cons}(y, \text{cons}(z, \text{nil}))) \]

- **Note**: these functors don't *do* anything; they just *represent* structured values.
  - e.g., the above might *represent* a three-element list: \[ [x, y, z] \]
MATCHING

Keywords:
Matching, Unification, "Occurs check", Programming via Matching...
Matching: simple rec. def. ($\cong$)

**Matching:** $\cong \subseteq \text{TERM} \times \text{TERM}$

- **Constants:**
  - $c \cong c'$ iff $c, c'$ same atom/number (c,c' constants)
  - e.g.; $\text{mia} \cong \text{mia}$, $\text{mia} \not\cong \text{vincent}$, 'mia' $\cong \text{mia}$, ...
  - $0 \cong 0$, $-2 \cong -2$, $4 \not\cong 5$, $7 \not\cong '7'$, ...

- **Variables:**
  - $X \cong t$
  - $t \cong X$
  - $X \cong Y$

  _always match_ (X,Y variables, t any term)

- e.g.; $X \cong \text{mia}$, $\text{woman}(\text{jody}) \cong X$, $A \cong B$, ...

- **Complex Terms:**
  - $f(t_1, \ldots, t_n) \cong f'(t'_1, \ldots, t'_m)$

    iff $f = f'$, $n=m$, $\forall i$ _recursively:_ $t_i \cong t'_i$

    - e.g., $\text{woman}(X) \cong \text{woman}(\text{mia})$, $f(a,X) \cong f(Y,b)$, $\text{woman}(\text{mia}) \not\cong \text{woman}(\text{jody})$, $f(a,X) \not\cong f(X,b)$.

  Note: all vars matches compatible $\forall i$
?- 
"=/2" and QUIZzzzz...

- In PROLOG (built-in matching pred.): "=/2":
  - \( = (2, 2) \); may also be written using infix notation:
    - i.e., as "2 = 2".

- Examples:
  - Yes
    - mia = mia ?
  - No
    - mia = vincent ?
  - Yes
    - \(-5 = -5 ?\)
  - X=5
    - \(5 = x ?\)
  - J...=v...
    - vincent = Jules ?
  - No
    - \(X = mia, X = vincent ?\)
  - X=...,Y=...
    - \(kill(shoot(gun), Y) = kill(X, stab(knife)) ?\)
  - No
    - \(loves(X, X) = loves(marcellus, mia) ?\)
?- Variable Unification ("fresh vars")

Variable Unification:

?- X = Y.
X = _G225
Y = _G225

"_G225" is a "fresh" variable (not occurring elsewhere)

Using these fresh names avoids name-clashes with variables with the same name nested inside

[ More on this later... ]
PROLOG: Non-Standard Unification

- **PROLOG** does **not** use "standard unification":
  - It uses a "short-cut" algorithm (**w/o** cycle detection for speed-up, saving so-called "occurs checks"):

Consider **(non-unifiable)** query:

- `?- father(X) = X.`

...on **older** versions of PROLOG:

- `?- father(X) = X.`
  - Out of memory! // on older versions of Prolog
  - `X = father(father(father(father(father(father(father(father(father(father(**))... (// on newer versions of Prolog

...on **newer** versions of PROLOG:

- `?- father(X) = X.`
  - `X = father(**)` // on newer versions of Prolog

...representing an **infinite term**
Consider the following knowledge base:

- vertical(line(point(X,Y),point(X,Z)).
- horizontal(line(point(X,Y),point(Z,Y)).

Almost looks too simple to be interesting; however...!

?- vertical(line(point(1,2),point(1,4))). // match
Yes
?- vertical(line(point(1,2),point(3,4))). // no match
No
?- horizontal(line(point(1,2),point(3,Y))). // var match
Y=2
?- ; // <-- ";" are there any other lines ?
No
?- horizontal(line(point(1,2),P)). // any point?
P = point(_G228,2) // i.e. any point w/ Y-coord 2
?- ; // <-- ";" other solutions ?
No

We even get complex, structured output:
"point(_G228,2)".

Note: scope rules:
the X,Y,Z's are all different in the (two) different rules!
Short Break:

15 mins
Keywords:
Proof Search *Order*,
Deduction, Backtracking,
Non-termination, ...
Consider the following knowledge base:

\[
\begin{align*}
f(a) . \\
f(b) . \\
g(a) . \\
g(b) . \\
h(b) . \\
k(X) \leftarrow f(X), g(X), h(X).
\end{align*}
\]

...and query:

\[
?- k(X).
\]

We (homo sapiens) can "easily" figure out that \( x=b \) is the (only) answer but how does PROLOG go about this?
Resolution:

1. **Search** knowledge base (from top to bottom) for (axiom or rule head) matching with (first) goal: \( k(X) \)
   - **Axiom match**: remove goal and process next goal \( [\rightarrow 1] \)
   - **Rule match**: (as in this case): \( k(X) : - f(X), g(X), h(X) \). \( [\rightarrow 2] \)
   - **No match**: backtrack (undo; try next choice in 1.) \( [\rightarrow 1] \)

2. "\( \alpha \)-convert" variables (to avoid later name clashes):
   - Goal': \( k(_G225) \) (unifying goal and match)
   - Match': \( k(_G225) : - f(_G225), g(_G225), h(_G225) \). \( [\rightarrow 3] \)

3. **Replace** goal with rule body: \( f(_G225), g(_G225), h(_G225) \).
   - Now resolve new goals (from left to right); \( [\rightarrow 1] \)

Possible outcomes:
- **success**: no more goals to match (all matched w/ axioms and removed)
- **failure**: unmatched goal (tried all possibilities: exhaustive backtracking)
- **non-termination**: inherent risk (same / bigger-and-bigger / more-and-more goals)
KB: f(a), f(b), g(a), g(b), h(b).
k(X) :- f(X), g(X), h(X).

; goal: k(X)

Search Tree (Visualization)

X = _G225

f(_G225), g(_G225), h(_G225)

_G225 = a

choice point

g(a), h(a)

h(a)

backtrack

axiom_1

axiom_2

_G225 = b

g(b), h(b)

h(b)

axiom_4

axiom_5

Yes

rule_1

k(X)
Keywords:
Recursion (numerals, addition),

**Careful** with Recursion:

(Prolog vs. inf.sys.)
Recursion (in Rules)

- **Declarative** (recursive) specification:

  ```prolog
  just_ate(mosquito, blood(john)).
  just_ate(frog, mosquito).
  just_ate(stork, frog).
  
  is_digesting(X,Y) :- just_ate(X,Y).
  is_digesting(X,Y) :- just_ate(X,Z),
  is_digesting(Z,Y).
  ```

- What does **Prolog** do (operationally) given query:

  ```prolog
  ?- is_digesting(stork, mosquito).
  ```

  ...same algorithm as before (works fine w/ recursion)
Do we really need Recursion?

Example: *Descendants*

"\(X\) descendant-of \(Y\)" ~ "\(X\) child-of, child-of, ..., child-of \(Y\)"

```
child(anne, brit).
child(brit, carol).

descend(A,B) :- child(A,B).
descend(A,C) :- child(A,B),
               child(B,C).
```

Okay for above knowledge base; but what about...:

```
child(anne, brit).
child(brit, carol).
child(carol, donna).
child(donna, eva).

?- descend(anne, donna).
No :(
```
Need Recursion? (cont'd)

Then what about...

```
descend(A, B) :- child(A, B).
descend(A, C) :- child(A, B), child(B, C).
descend(A, D) :- child(A, B), child(B, C), child(C, D).
```

Now works for...

```
?- descend(anne, donna).
   Yes :)  
```

...but now what about:

```
?- descend(anne, eva).
   No :(
```

Our "strategy" is:

- extremely redundant; and
- only works up to finite $K$!
Solution: Recursion!

Recursion to the rescue:

- \texttt{descend(X,Y) :- child(X,Y).}
- \texttt{descend(X,Y) :- child(X,Z),
\hspace{1em} \texttt{descend(Z,Y).}}

Works:

- \texttt{?- descend(anne, eva).}
  \hspace{1em} \texttt{Yes :)}

...for structures of \textit{arbitrary size}:

- ...even for "zoe":

- \texttt{?- descend(anne, zoe).}
  \hspace{1em} \texttt{Yes :)}

...and is very \textit{concise}!
Search tree for query:

?- descend(a,d).
Yes :)
Example: Successor

Mathematical definition of numerals:

- "Unary encoding of numbers"
  - Computers use *binary encoding*
  - Homo Sapiens agreed (over time) on *decimal encoding*
  - (Earlier cultures used other encoding: base 20, 64, ...)

In **PROLOG**:

```
numeral(0).
numeral(succ(N)) :- numeral(N).
```

typing in the inference system
"head under the arm"
(using a Danish metaphor).
Backtracking (revisited)

Given:

numeral(0).

numeral(succ(N)) :- numeral(N).

Interaction with PROLOG:

?- numeral(0). // is 0 a numeral?
Yes
?- numeral(succ(succ(succ(0)))). // is 3 a numeral?
Yes
?- numeral(X). // okay, gimme a numeral?
X=0
?- ; // please backtrack (gimme the next one?)
X=succ(0)
?- ; // backtrack (next?)
X=succ(succ(0))
?- ; // backtrack (next?)
X=succ(succ(succ(0)))
... // and so on...
**Example: Addition**

Recall addition inference system (~3 hrs ago):

```
'+' ⊆ N x N x N
[axiom₁]  + (0, M, M)
[rule₁]   + (N, M, R) → + (N+1, M, R+1)
```

**In Prolog:**

```
add(0, M, M).
add(succ(N), M, succ(R)) :- add(N, M, R).
```

Again:
typing in the inference system
"head under the arm"
(using a Danish metaphor).

**However, one extremely important difference:**

<table>
<thead>
<tr>
<th>math. ∃ inf.tree</th>
<th>vs. fixed search alg.</th>
<th>top-to-bottom backtracking</th>
<th>loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>no ¬/ loops</td>
<td>add(X, succ(succ(0)), succ(0))</td>
<td>add(0, M, M)</td>
<td></td>
</tr>
<tr>
<td>x?: + (x, 2, 1)</td>
<td>vs. Prolog</td>
<td>add(succ(N), M, R) :- add(N, succ(M), R)</td>
<td></td>
</tr>
</tbody>
</table>

[axiom₁]  + (0, M, M)
[rule₁]   + (N, M+1, R) + (N+1, M, R)
Clas Brabrand

OCT 30, 2007

Be Careful with Recursion!

Original:

just_ate(mosquito, blood(john)).
just_ate(frog, mosquito).
just_ate(stork, frog).

Query:

?- is_digesting(stork, mosquito).

rule bodies:

is_digesting(A,B) :- just_ate(A,B).
is_digesting(X,Y) :- just_ate(X,Z),
is_digesting(Z,Y).

rules:

is_digesting(X,Y) :- just_ate(X,Z),
is_digesting(Z,Y).
is_digesting(A,B) :- just_ate(A,B).

bodies+rules:

is_digesting(X,Y) :- is_digesting(Z,Y),
just_ate(X,Z).
is_digesting(A,B) :- just_ate(A,B).

EXERCISE:

What happens if we swap...
Exercises 2+3:

14:30 – 15:15
2. Finite-State Search Problems

Purpose:

- Learn to solve encode/solve/decode search problems

Exercise 2 (Finite-State Problem Solving in Prolog)

Purpose: to learn how to encode problems, solve, and decode (answers) finite search problems (in Prolog).

- Consider the following crossword puzzle and Prolog knowledge base representing a lexicon of six-letter words:

<table>
<thead>
<tr>
<th>Crossword Puzzle</th>
<th>Information Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>V1</td>
</tr>
<tr>
<td>H2</td>
<td></td>
</tr>
<tr>
<td>H3</td>
<td></td>
</tr>
</tbody>
</table>

- word (abalone, a, b, a, l, n, e).
- word (abandon, a, b, a, n, d, o).
- word (anagram, a, n, a, g, e, a).
- word (connect, c, o, n, e, r).
- word (elegant, e, i, c, r, t).
- word (enhance, e, n, h, e, r).

- a) Specify a Prolog predicate crossword/6 that takes six arguments (V1, V2, V3, H1, H2, H3) (formatted as in the Prolog knowledge base above) and tells us whether or not the puzzle is correctly filled in.
- b) Now specify a search query that extracts the solution to the puzzle, given the particular six words in the knowledge base above.
- c) Are there any other solutions?
3. Finite-State Problem Solving

**Purpose:**

- Learn to solve encode/solve/decode search problems

- a) Specify a Prolog predicate `different/2` that takes two arguments \((C_1, C_2)\) and tells us whether or not the arguments are different colors.
- b) Use the above predicate to specify a Prolog predicate `three_coloring_b/4` that takes four color arguments \((red, ECU, PER, BRA)\) and tells us whether or not the colors constitute a valid coloring for the map of South America (restricted to Colombia, Ecuador, Peru, and Brazil).
- c) Now specify a search query that extracts the solution to the problem (if it exists).
- d) Are there any other solutions?
- e) Now repeat steps b-d for the map restricted to Brazil, Bolivia, Paraguay, and Argentina (i.e., specify a Prolog predicate `three_coloring_e/4`).
Exercises 4+5:

15:30 – 16:15
4. Multiple Solutions & Backtracking

**Purpose:**

- Learn how to deal with mult. solutions & backtracking

**Exercise 4 (Multiple Solutions and Backtracking in Prolog)**

Purpose: to learn how to deal with multiple solutions and backtracking (in Prolog)

- Consider the following Prolog program:

<table>
<thead>
<tr>
<th>Axioms</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>noun(john).</td>
<td>verb(Present</td>
</tr>
<tr>
<td>noun(jane).</td>
<td>verb(Past</td>
</tr>
<tr>
<td>verb('to like','likes','liked').</td>
<td>sentence(S,V,U) :- noun(S), verb(V), noun(U).</td>
</tr>
</tbody>
</table>

- **a)** How many relations are defined and what are their respective arities (i.e., number of arguments)?
- **b)** Query the knowledge base to find a valid sentence (containing the verb *liked*).
- **c)** Draw a search tree (cf. Section 2.2) for your query from **a**.
- **d)** Explain EXACTLY what is going on and in what order.
Recursion in Prolog

**Purpose:** Learn how to be careful with recursion

**Exercise 5 (Recursion in Prolog)**

Purpose: to learn to be careful with recursion (in Prolog)

- Consider the following illustration of the “Russian dolls” (smaller dolls inside bigger dolls):

```
<table>
<thead>
<tr>
<th>Russian Dolls</th>
</tr>
</thead>
<tbody>
<tr>
<td>katarina</td>
</tr>
<tr>
<td>nina</td>
</tr>
<tr>
<td>olga</td>
</tr>
<tr>
<td>natasha</td>
</tr>
</tbody>
</table>
```

- **a)** Define a predicate `in/A` that tells which doll is (directly or indirectly) inside another, e.g., `in(katarina, natasha)` should be true, while `in(olga, katarina)` false.
- **b)** Find all dolls inside `katarina`.
- **c)** [Conditionally]:
  - *If* you got a global stack overflow in **b)** above (after the third possible answer)
  - *Then* explain what happened and try to fix the problem
  - *Else* good solution, move on to exercise 4...
Hand-in:

To **check** that you are able to solve problems in Prolog

**Hand-in Exercise:** ("The Ant, the Ant-Eater, and the Ant-Eater-Eater Problem")

<table>
<thead>
<tr>
<th>THE ANT-EATER(^0), THE ANT-EATER(^1), AND THE ANT-EATER(^2) PROBLEM:</th>
</tr>
</thead>
</table>
|A Zookeeper is travelling with an **ant**, an **ant-eater**, and an **ant-eater-eater** and has come to a river which he has to cross in a small boat.

The Zookeeper has to take good care of his animals as the ant-eater would eat the ant (if left alone with it) and the ant-eater-eater would eat the ant-eater (if left unattended). The boat is only big enough for him to take at most one animal across the river (yes, the ant is, in fact, rather big).

How can the Zookeeper cross the river with all his animals?

- a) **Encode** the problem in Prolog (and explain carefully how you represented the problem).
- b) **Solve** the problem in Prolog (and explain carefully what is going on, how Prolog is solving the problem).
- c) **Decode** the answer to obtain the solution to the problem.
- d) **Try it out** [Online Game]

**HINTS**

**explain carefully** how you repr. & what **PROLOG** does!