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

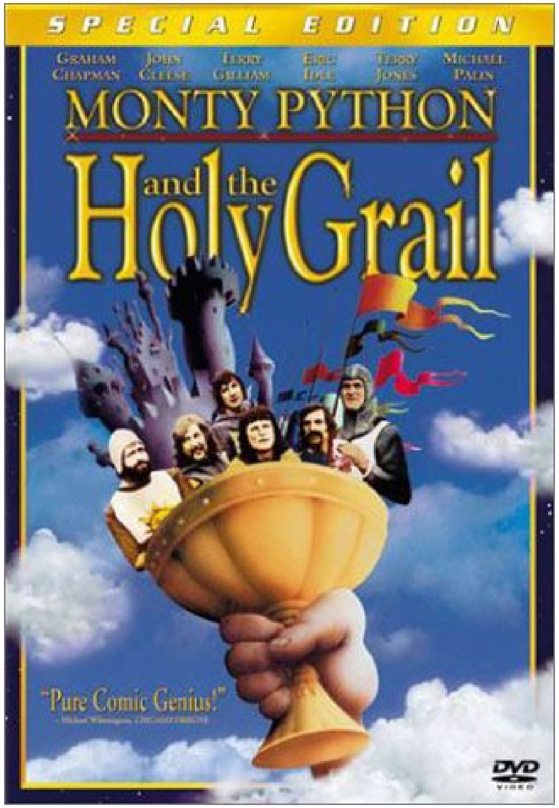
MONTY PYTHON

<u>Keywords</u>:

Holy Grail, Camelot, King Arthur,
Sir Bedevere, The Killer Rabbit®,
Sir Robin*-the-not-quite-so-brave-as-Sir Lancelot*

? - Movie(!)

- *"Monty Python and the Holy Grail"* *(1974)*
  - Scene V: "*The Witch*":



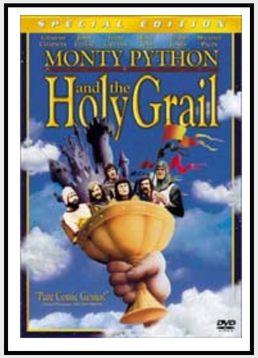
?- The Monty Python Reasoning:

- "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: witch(girl)

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

```
witch(girl)

witch(X) :- female(X) , burns(X).

female(girl)                          burns(girl)

%- by observation -----               burns(X) :- wooden(X).

                                      wooden(girl)

                                      wooden(X) :- floats(X).

                                      floats(girl)

floats(X) :- sameweight(X,Y) , floats(Y).

sameweight(girl,duck)                 floats(duck)

%- by experiment -----                %- King Arthur -----
```

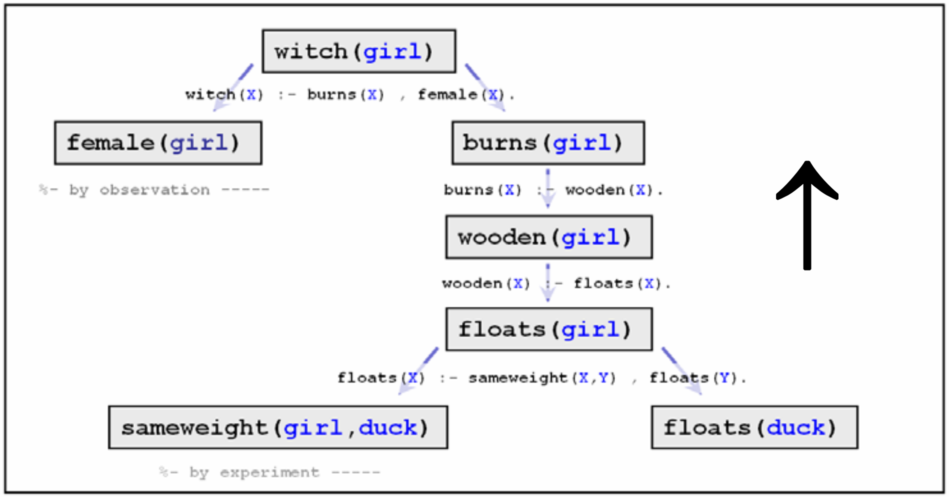
?- Deductive Reasoning: `witch(girl)`

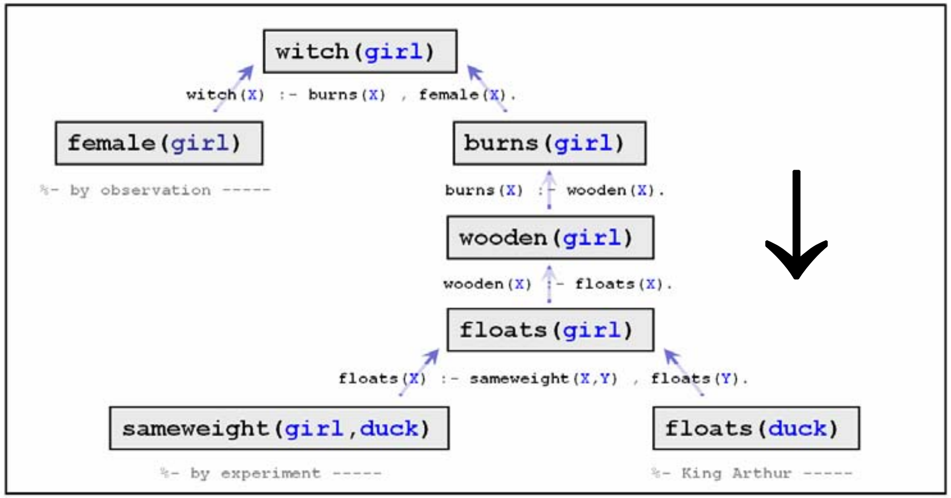
- ***"Deduction":*** (aka. *"top-down reasoning"*)

```
witch(girl)

witch(X) :- female(X) , burns(X).

female(girl)

%- by observation -----

burns(girl)

burns(X) :- wooden(X).

wooden(girl)

wooden(X) :- floats(X).

floats(girl)

floats(X) :- sameweight(X,Y) , floats(Y).

sameweight(girl,duck)

%- by experiment -----

floats(duck)

%- King Arthur -----
```

?- *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)

? - *Hearing:* Nomination of CIA Director, General Michael Hayden (USAF).

LEVIN: U.S. SENATOR CARL LEVIN (D-MI)
HAYDEN: GENERAL MICHAEL B. HAYDEN (USAF),
NOMINEE TO BE DIRECTOR OF CIA

CQ Transcriptions
Thursday, May 18, 2006; 11:41 AM

"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: $\vdash_{even} \subseteq \mathbf{Z}$
  - Written as: $\vdash_{even} 4$ as a short-hand for: $4 \in \vdash_{even}$
  - … and as: $\nvdash_{even} 5$ as a short-hand for: $5 \notin \vdash_{even}$

- Example[2]: "*equals*" relation: $\text{'='} \subseteq \mathbf{Z} \times \mathbf{Z}$
  - Written as: $2 = 2$ as a short-hand for: $(2,2) \in \text{'='}$
  - … and as: $2 \neq 3$ as a short-hand for: $(2,3) \notin \text{'='}$

- Example[3]: "*DFA transition*" relation: $\text{'}\rightarrow\text{'} \subseteq \mathbf{Q} \times \Sigma \times \mathbf{Q}$
  - Written as: $q \xrightarrow{\sigma} q'$ as a short-hand for: $(q, \sigma, q') \in \text{'}\rightarrow\text{'}$
  - … and as: $p \stackrel{\sigma}{\nrightarrow} p'$ as a short-hand for: $(p, \sigma, p') \notin \text{'}\rightarrow\text{'}$

? - Inference System

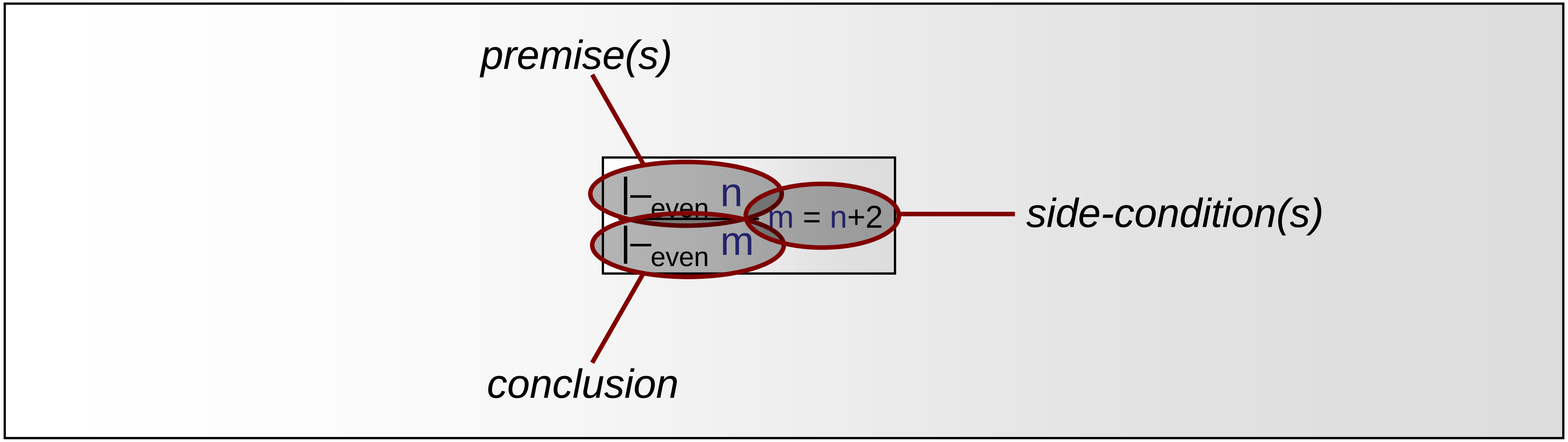
- Inference System:
  - Inductive (recursive) *specification* of relations
  - Consists of *axioms* and *rules*

- Example: $\vdash_{\text{even}} \subseteq \mathbf{Z}$

- Axiom: $\dfrac{}{\vdash_{\text{even}} 0}$
  - *"0 (zero) is even"!*

- Rule: $\dfrac{\vdash_{\text{even}} n}{\vdash_{\text{even}} m}\ m = n+2$
  - *"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: $\dfrac{\vdash_{even}\ n}{\vdash_{even}\ m}\ m = n+2$
  - *"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*: $\dfrac{\vdash_{even}\ n}{\vdash_{even}\ n+2}$
  - *"If n is even, then n+2 is even"; or*
  - *"n+2 is even, if n is even"*

Even so, this is what we mean

? - Relation *Membership?* $x \stackrel{?}{\in} \mathcal{R}$

- Axiom: $\dfrac{}{\vdash_{even} 0}$
  - *"0 (zero) is even"!*
- Rule: $\dfrac{\vdash_{even} n}{\vdash_{even} n+2}$
  - *"If n is even, then n+2 is even"*
- Is 6 even?!?

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{even} 0}[axiom_1]}{\vdash_{even} 2}[rule_1]}{\vdash_{even} 4}[rule_1]}{\vdash_{even} 6}[rule_1]$$

inference tree

- The *inference tree* ***proves*** that: $\vdash_{even} 6$

? - *Example:* "less-than-or-equal-to"

- Relation: $\text{'}\leq\text{'} \subseteq \mathbf{N} \times \mathbf{N}$

[axiom$_1$] $\dfrac{}{0 \leq 0}$ [rule$_1$] $\dfrac{n \leq m}{n \leq m+1}$ [rule$_2$] $\dfrac{n \leq m}{n+1 \leq m+1}$

- Is "$1 \leq 2$" **?** (why/why not)!? [*activation exercise*]
  - Yes, because there exists an inference tree:
    - In fact, it has *two* inference trees:

$$\dfrac{\dfrac{\dfrac{}{0 \leq 0}}{0 \leq 1}}{1 \leq 2}$$ [axiom$_1$] [rule$_1$] [rule$_2$]

$$\dfrac{\dfrac{\dfrac{}{0 \leq 0}}{1 \leq 1}}{1 \leq 2}$$ [axiom$_1$] [rule$_2$] [rule$_1$]

? - *Activation Exercise 1*

- 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 2

- Activation Exercise:
  - **1.** Specify the signature of the relation: '`//`'
    - **x** `//` **y**   *"**x** is-half-that-of **y**"*
  - **2.** Specify the relation via an inference system
    - i.e. axioms and rules
  - **3.** Prove that indeed:
    - **3** `//` **6**   *"3 is-half-that-of 6"*

Syntactically different
Semantically the same relation

? - Relation vs. Function

- A ***function***...
  - $f : A \to B$

...is a ***relation***

  - $R_f \subseteq A \times B$

  ...with the special requirement:

  - $\forall a \in A,\ b_1, b_2 \in B:$
    $R_f(a, b_1) \wedge R_f(a, b_2) \;\Rightarrow\; b_1 = b_2$
    - *i.e., "the result", **b**, is **uniquely** determined from "the argument", **a**.*

? - Relation vs. Function (Example)

- The (2-argument) ***function*** '+'...
  - $+ : N \times N \rightarrow N$

...induces a (3-argument) ***relation***

- $R_+ \subseteq N \times N \times N$

...that obeys:

- $\forall n,m \in N,\ r_1,r_2 \in N:$
  $R_+(n,m,r_1) \wedge R_+(n,m,r_2) \Rightarrow r1 = r2$
  - *i.e., "the result", **r**, is **uniquely** determined from "the arguments", **n** and **m***

? - *Example:* "add"

- Relation: $'+' \subseteq \mathbf{N} \times \mathbf{N} \times \mathbf{N}$

$$[\text{axiom}_1] \quad \frac{}{+(0,m,m)} \qquad\qquad [\text{rule}_1] \quad \frac{+(n,m,r)}{+(n+1,m,r+1)}$$

- Is "2 + 2 = 4" ?!?
  - Yes, because there exists an inf. tree for "+(2,2,4)":

$$\dfrac{\dfrac{\dfrac{}{+(0,2,2)}\ [\text{axiom}_1]}{+(1,2,3)}\ [\text{rule}_1]}{+(2,2,4)}\ [\text{rule}_1]$$

? - Relation Definition (Interpretation)

- Actually, an inference system: $\vdash_{\mathcal{R}} \subseteq \mathbf{Z}$

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

…is a *demand specification* for a relation:

$$(0 \in \text{‘}\vdash_{\mathcal{R}}\text{’}) \;\wedge\; (\forall\, n \in \text{‘}\vdash_{\mathcal{R}}\text{’} \;\Rightarrow\; n+2 \in \text{‘}\vdash_{\mathcal{R}}\text{’})$$

- The three relations:
  - R = {0, 2, 4, 6, …} (aka., 2**N**)
  - R’ = {0, 2, 4, 5, 6, 7, 8, …}
  - R’’ = {…, -2, -1, 0, 1, 2, …} (aka., **Z**)

…all *satisfy* the (above) specification!

? - Inductive *Interpretation* (*)

- A inference system: $\vdash_{\mathcal{R}} \subseteq \mathbf{Z}$ $\Leftrightarrow$ $\vdash_{\mathcal{R}} \in P(\mathbf{Z})$

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

- …induces a function: $F_{\mathcal{R}}: P(\mathbf{Z}) \to P(\mathbf{Z})$ *From rel. to rel.*

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

- <u>Definition</u>: $\vdash_{even} \;:=\; \mathrm{lfp}(F_{\mathcal{R}}) \;=\; \bigcup_n F_{\mathcal{R}}^{\mathbf{n}}(\emptyset)$
  - 'lfp' (*least fixed point*) ~ least solution:

$F(\emptyset) = \{0\}$ $\cup$ $F^2(\emptyset) = F(\{0\}) = \{0,2\}$ $\cup$ $F^3(\emptyset) = F^2(\{0\}) = F(\{0,2\}) = \{0,2,4\}$ $\cup \ldots =$ **2N**

$F^{\mathbf{n}}(\emptyset)$ ~ *"Anything that can be proved in 'n' steps"*

Exercise 1:

11:15 – 12:00

? - 1. Relations via Inf. Sys. (in Prolog)

- **Purpose:**
  - *Learn how to describe relations via inf. sys. (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 $|\!-_{odd}\ N$ (*"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: $|\!-_{odd}\ 5$ (in terms of your definition).
  - **d)** *Rewrite* your inference system so that it instead uses a unary `succ` encoding of numerals (cf. Section 3.1.3).
  - **e)** *Implement* this inference system in Prolog as a predicate `odd/1`.
  - **f)** *Prove* that: `odd(succ(succ(succ(succ(succ(0))))))` (using Prolog) and explain how Prolog establishes this.

- Binary: **`double/2`**
  - **-)** *Repeat* steps **a)**-**f)** but for the binary `double` relation, written `X << Y` (*"Y is double that of X"*), on natural numbers (using only constant addition in the rules).
    - In steps **c)** and **f)**; prove that `2 << 4` and `double(succ(succ(0)),succ(succ(succ(succ(0)))))`, respectively.
- Ternary: **`congruent/3`** [hard'ish]
  - **-)** *Repeat* steps **a)**-**f)** but for the binary `congruent` relation, written $X \equiv_Z Y$ (*"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).
    - In steps **c)** and **f)**; prove that $5 \equiv_2 1$ and `congruent(succ(succ(succ(succ(succ(0))))),succ(0),succ(succ(0)))`, respectively.
  - [I give up; give me a hint]

INTRODUCTION TO **PROLOG**

(by example)

<u>Keywords</u>:

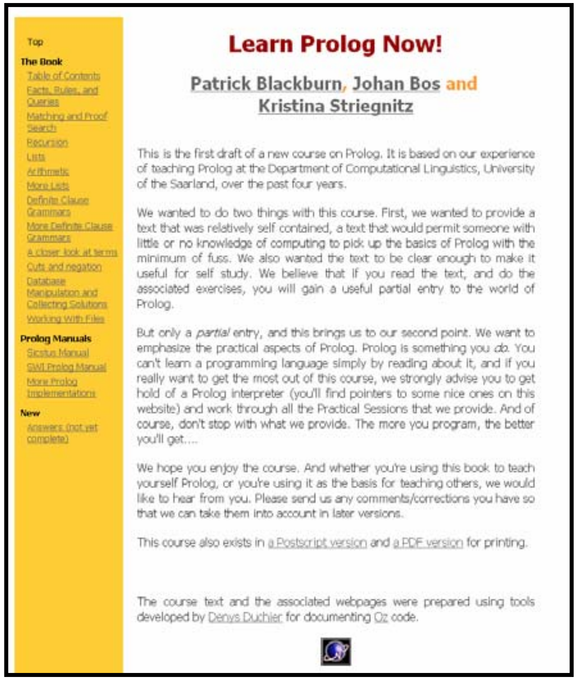
Logic-programming, Relations,
Facts & Rules, Queries, Variables,
Deduction, Functors, & Pulp Fiction :)

? - PROLOG Material

- 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/]

?- Prolog

- 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"*; e.g.:

```
%- 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

- C
- Java
- ...

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

- Prolog
- Haskell
- ...

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***
  - ***Rules***

    }***"knowledge base"*** (or *"database"*)

  - ***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
```

```
?- tatooed(joey).
No

?- playsAirGuitar(marcellus).
No

?- attends_dProgSprog(marcellus).
No

?- playsAirGitar(jody).
No
```

? - Rules

- ***Rules***:
  - **Syntax**: `head :- body.`
  - **Semantics**: ~ $\frac{\text{body}}{\text{head}}$ *inf.sys.*
    - *"**If** the body is true, **then** the head is also true"*
  - To express ***conditional*** *truths:*
    - e.g., `playsAirGuitar(mia) :- listensToMusic(mia).`
    - i.e., *"Mia plays the air-guitar, **if** she listens to music".*
  - **PROLOG** then uses the following deduction principle (called: *"modus ponens"*):

```
H :- B    // If B, then H (or "H <= B")
B         // B.
-----------------------------------------
□ H       // Therefore, H.
```

?- KB2: Rules

- **KB2** contains *2 facts* and ***3 rules:***

```
% FACTS:
listensToMusic(mia).
happy(yolanda).
```

```
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":

```
playsAirGuitar(mia)  :-  listensToMusic(mia).
listensToMusic(mia).
------------------------------------------------
□ playsAirGuitar(mia).
```

```
listensToMusic(yolanda)  :-  happy(yolanda).
happy(yolanda).
------------------------------------------------
□ listensToMusic(yolanda).
```

...combined with...

```
playsAirGuitar(yolanda)  :-  listensToMusic(yolanda).
listensToMusic(yolanda).
------------------------------------------------
□ playsAirGuitar(yolanda).
```

?- Conjunction and Disjunction

- Rules may contain multiple bodies (which may be combined in two ways):
  - ***Conjunction*** (aka. "and")***:***
    - ```
      playsAirGuitar(vincent)  :-  listensToMusic(vincent),
                                   happy(vincent).
      ```
    - *i.e., "Vincent plays, **if** he listens to music **and** he's happy".*
  - ***Disjunction*** (aka. "or")***:***
    - ```
      playsAirGuitar(butch)  :-  listensToMusic(butch);
                                 happy(butch).
      ```
    - *i.e., "Butch plays, if he listens to music **or** he's happy".*

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

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

?- KB3: Conjunction and Disjunction

- **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
```

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

```
?- playsAirGuitar(butch).
Yes
```

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

...using the last rule above

?- KB4: N-ary Predicates and Variables

- **KB4:**

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

Defining unary predicate: `woman/1`

```
loves(vincent,mia).
loves(marcellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).
```

Defining binary predicate: `loves/2`

- Interaction with ***Variables*** (in upper-case):
  - ```
    ?- woman(X).
    X = mia
    ?- ;          // ";" ~ are there any other matches ?
    X = jody
    ?- ;          // ";" ~ are there any other matches ?
    X = yolanda
    ?- ;          // ";" ~ are there any other matches ?
    No
    ```
  - **PROLOG** tries to ***match*** **`woman(X)`** against the rules (from top to bottom) using **`X`** as a placeholder for anything.
- More complex query:

```
?- loves(marcellus,X), woman(X).
X = mia
```
```

?- *KB5: Variables in Rules*

- **KB5:**

```
loves(vincent,mia).
loves(marcellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).

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

<u>NB</u>: (implicit) *existential quantification* (i.e., "∃ Z")

- i.e., *"X is-jealous-of Y, **<u>if there exists</u>** 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
```

  - (they both love Mia).
  - **Q:** ***Any other jealous people in KB5?***

?- Prolog Terms

- **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`, ...

  constants (Atoms and Numbers)

  - **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. `cons(x, cons(y, cons(z, 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

<u>Keywords</u>:

Matching, Unification, "Occurs check", Programming via Matching...

? - Matching: simple rec. def. (≅)

- ***Matching***: '≅' ⊆ TERM × TERM

constants

  - `c ≅ c'` **iff** c,c' same atom/number (c,c' constants)
    - e.g.; `mia ≅ mia`, `mia ≇ vincent`, `'mia' ≅ mia`, ...
      `0 ≅ 0`, `-2 ≅ -2`, `4 ≇ 5`, `7 ≇ '7'`, ...

variables

  - `X ≅ t`
  - `t ≅ X`
  - `X ≅ Y`

  } ***always match*** (X,Y variables, t any term)

    - e.g.; `X ≅ mia`, `woman(jody) ≅ X`, `A ≅ B`, ...

complex terms

  - $f(t_1,\ldots,t_n) \cong f'(t'_1,\ldots,t'_m)$
    iff `f=f'`, `n=m`, ∀i ***recursively***: $t_i \cong t'_i$
    - e.g., `woman(X) ≅ woman(mia)`, `f(a,X) ≅ f(Y,b)`, `woman(mia) ≇ woman(jody)`, `f(a,X) ≇ f(X,b).`

__Note__: all vars matches compatible ∀i

?- "=/2" and QUIZZzz...

- 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 Unificat°

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

PROLOG Design Choice:
trading safety for efficiency
(rarely a problem in practice)

- 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(
      ```
    - ...on ***newer*** versions of PROLOG:
    - ```
      ?- father(X) = X.
      X = father(**)      // on newer versions of Prolog
      ```
    - *...representing an* ***infinite term***

?- Programming via Matching

- Consider the following knowledge base:
  - ```
    vertical(line(point(X,Y),point(X,Z)).
    horizontal(line(point(X,Y),point(Z,Y)).
    ```

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

    - 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)".

Short Break:

15 mins

PROOF SEARCH ***ORDER***

Keywords:

Proof Search ***Order***,
Deduction, Backtracking,
Non-termination, ...

?- Proof Search *Order*

- Consider the following *knowledge base*:
  - ```
    f(a).
    f(b).

    g(a).
    g(b).

    h(b).

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

PROLOG's Search Order

- **Resolution:**

```
f(a).  f(b).      g(a).  g(b).      h(b).        } axioms (5x)
k(X)  :-  f(X),g(X),h(X).                        } rule (1x)
```

rule head (k(X)) · *rule body* (f(X),g(X),h(X))

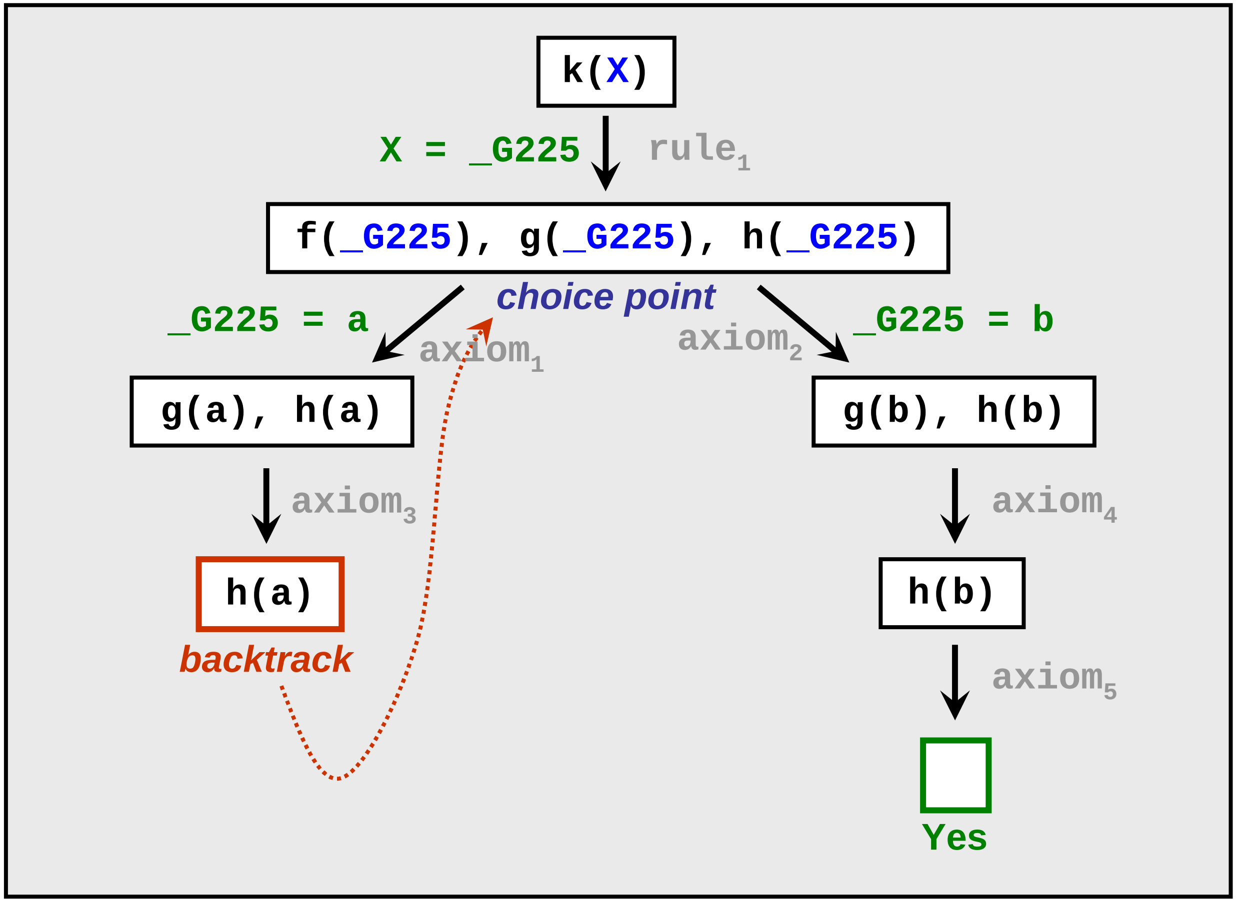
- **1. *Search*** knowledge base (*from* ***top*** *to* ***bottom***) for (***axiom or rule head***) ***matching*** with (first) ***goal***: `k(X)`
  - ***Axiom match*:** remove goal and process ***next goal*** [→1]
  - ***Rule match*:** (as in this case): `k(X) :- f(X),g(X),h(X).` [→2]
  - ***No match*:** ***backtrack*** (***undo***; try next choice in **1.**) [→1]
- **2.** "***α-convert***" variables (to avoid later name clashes):
  - Goal': `k(_G225)` (***unifying*** *goal and match*)
  - Match': `k(_G225) :- f(_G225),g(_G225),h(_G225).` [→3]
- **3. *Replace*** goal with ***rule body***: `f(_G225),g(_G225),h(_G225).`
  - *Now* ***resolve new goals*** (*from* ***left*** *to* ***right***); [→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)

?- Search Tree (Visualization)

- **KB:** `f(a).  f(b).  g(a).  g(b).  h(b).`
  `k(X) :- f(X),g(X),h(X).` ; goal: `k(X)`



RECURSION

<u>Keywords</u>:

Recursion (numerals, addition),
Careful w/ Recursion:
(**PROLOG** ***vs.*** inf.sys.)

?- Recursion (in Rules)

- ***Declarative*** (*recursive*) specification:
  - ```
    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:
    - `?- 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).

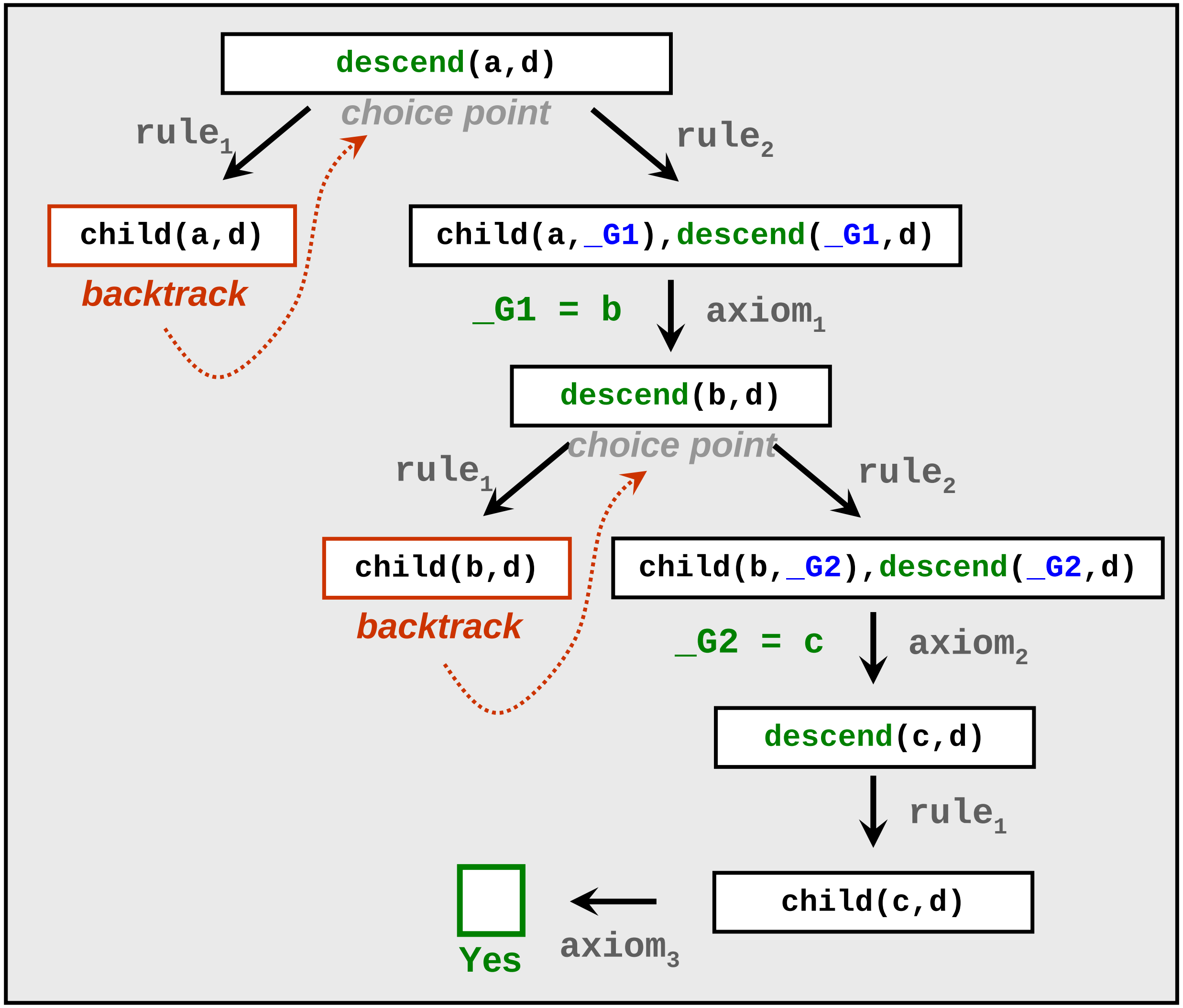
descend(A,B) :- child(A,B).
descend(A,C) :- child(A,B),
                child(B,C).
```

  - 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:
  - ```
    descend(X,Y) :- child(X,Y).
    descend(X,Y) :- child(X,Z),
                    descend(Z,Y).
    ```
  - ***Works:***
    ```
    ?- descend(anne, eva).
    Yes                                          :)
    ```
  - ...for structures of ***arbitrary size:***
    - *...even for "zoe":*
    ```
    ?- descend(anne, zoe).
    Yes                                          :)
    ```
    - ...and is very ***concise!***
    ```

?- Operationally (in PROLOG)

```
child(a,b).
child(b,c).
child(c,d).
child(d,e).

descend(X,Y) :- child(X,Y).
descend(X,Y) :- child(X,Z),
                descend(Z,Y).
```

- Search tree for query:

```
?- descend(a,d).
Yes                           :)
```



? - *Example: Successor*

- Mathematical definition of numerals:
  - [axiom$_1$] $\dfrac{}{\vdash_{num} 0}$ [rule$_1$] $\dfrac{\vdash_{num} N}{\vdash_{num} \textbf{\textit{succ}}\ N}$
  - "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):

$`+' \subseteq \mathbf{N} \times \mathbf{N} \times \mathbf{N}$ [axiom$_1$] $\dfrac{}{+(0,M,M)}$ [rule$_1$] $\dfrac{+(N,M,R)}{+(N+1,M,R+1)}$

- 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:***

`X?: +(X,2,1)` inf. sys. *VS.* PROLOG `?- add(X,succ(succ(0)),succ(0)).`

no ☺ math. ∃ inf.tree *VS.* fixed search alg. - *top-to-bottom* - *left-to-right* - *backtracking* ☹ *loops*

[axiom$_1$] $\dfrac{}{+(0,M,M)}$ [rule$_1$] $\dfrac{+(N,M+1,R)}{+(N+1,M,R)}$ *VS.*

```
add(0,M,M).

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

?- Be Careful with Recursion!

- Original:

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

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

Query:

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

EXERCISE: *What happens if we swap...*

- ***rule bodies***:

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

- ***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).
```

Exercises 2+3:

14:30 – 15:15

? - 2. Finite-State Search Problems

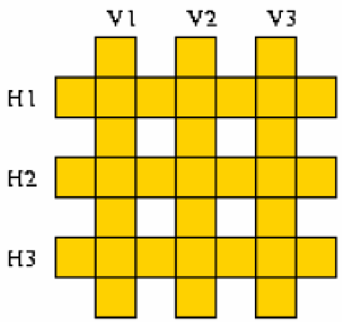
- **Purpose:**
  - *Learn to solve encode/solve/decode search problems*

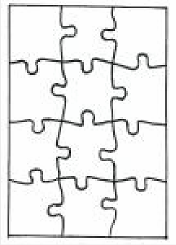
EXERCISE 2 (FINITE-STATE PROBLEM SOLVING IN PROLOG)

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 six-letter words:

Crossword Puzzle	Information Representation

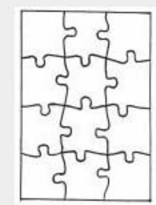
	`word(abalone,a,b,a,l,o,n,e).`
	`word(abandon,a,b,a,n,d,o,n).`
	`word(anagram,a,n,a,g,r,a,m).`
	`word(connect,c,o,n,n,e,c,t).`
	`word(elegant,e,l,e,g,a,n,t).`
	`word(enhance,e,n,h,a,n,c,e).`



- **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 `(C1,C2)` 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 named `(COL,ECU,PER,BRA)` and tells us whether or not the colors constitute a valid coloring for the map of south-america (restricted to Columbia, 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:

Axioms	Rules

`noun(john).`
`noun(jane).`
 	
`verb('to like',likes,liked).`
	`verb(Present) :- verb(Infinitive,Present,Past).`
	`verb(Past) :- verb(Infinitive,Present,Past).`
	
	`sentence(S,V,O) :- noun(S), verb(V), noun(O).`

- **a)** How many relations are defined and what are their repective 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.

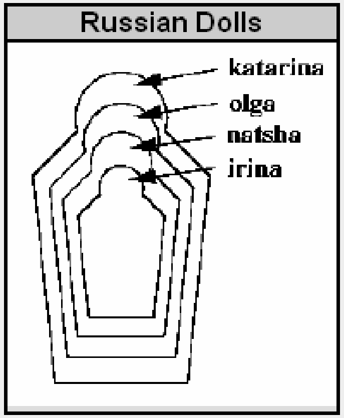
? - 5. Recursion in Prolog

- **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):



- **a)** *Define* a predicate `in/2` that tells which doll is (directly or indirectly) inside which; 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** expain what happened and try to fix the problem
  - **ELSE** good solution, move on to exercise 4...

? - Hand-in #5

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

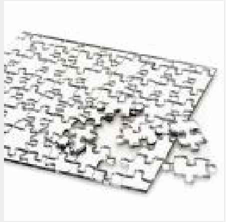
HAND-IN EXERCISE: ("The Ant, the Ant-Eater, and the Ant-Eater-Eater Problem")

"THE ANT-EATER^0, THE ANT-EATER^1, AND THE ANT-EATER^2 PROBLEM":

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!*