Chapter 2.
Prolog Syntax and Semantics

- April 20 , 2011 -

Syntax
Model-theoretic semantics (“Logical Consequence”)
Operational semantics (“Derivation / Resolution”)
Negation
Incompleteness of SLD-Resolution
Practical implications
Recursive Programming with Lists
Relations versus Functions
Operators

Prolog

- Prolog stands for "Programming in Logic".
- It is the most common logic program language.

Bits of history

- 1965
  - John Alan Robinson develops the resolution calculus – the formal foundation of automated theorem provers
- 1972
  - Alain Colmerauer (Marseilles) develops Prolog (first interpreter)
- mid 70th
  - David D.H. Warren (Edinburg) develops first compiler
    - ⇨ Warren Abstract Machine (WAM) as compilation target → like Java byte code
- 1981-92
  - „5th Generation Project“ in Japan boosts adoption of Prolog world-wide

Prolog Syntax

Predicates

Clauses, Rules, Facts

Terms, Variables, Constants, Structures

Predicates, Clauses, Rules, Facts

Predicate symbol (just a name)

Predicate definition (set of clauses)

```
isFatherOf(kurt,peter).
isFatherOf(peter,paul).
isFatherOf(peter,hans).

isGrandfatherOf(G,C) :-
        isFatherOf(G,F), isFatherOf(F,C).
isGrandfatherOf(G,C) :-
        isFatherOf(G,M), isMotherOf(M,C).
```

Implication

Literal

Conjunction

```
?- isGrandfatherOf(kurt,paul).
?- isGrandfatherOf(kurt,C).
?- isGrandfatherOf(G,paul).
?- isGrandfatherOf(G,paul),isFatherOf(X,G).
```

Fact

Rule

Goal / Query

Clause

Predicates, Clauses, Rules, Facts

- Predicate symbol (just a name)
- Predicate definition (set of clauses)

```
isFatherOf(kurt,peter).
isFatherOf(peter,paul).
isFatherOf(peter,hans).

isGrandfatherOf(G,C) :-
        isFatherOf(G,F), isFatherOf(F,C).
isGrandfatherOf(G,C) :-
        isFatherOf(G,M), isMotherOf(M,C).
```

- Implication
- Literals
- Conjunction

```
?- isGrandfatherOf(kurt,paul).
?- isGrandfatherOf(kurt,C).
?- isGrandfatherOf(G,paul).
?- isGrandfatherOf(G,paul),isFatherOf(X,G).
```

- Facts
- Rules
- Goals / Querys
- Clauses

Clauses and Literals

- Prolog programs consist of clauses
  - Rules, facts, queries (see previous slide)
- Clauses consist of literals separated by logical connectors.
  - Head literal
  - Zero or more body literals

```
isGrandfatherOf(G,C) :-
        isFatherOf(G,X), ( isFatherOf(X,C) ; isMotherOf(X,C) ).
```

- Logical connectors are
  - implication (:-), conjunction (,) and disjunction (;)
- Literals consist of a predicate symbol, punctuation symbols and arguments
  - Punctuation symbols are the comma "," and the round braces "(" and ")"

```
isGrandfatherOf(G,C)
isFatherOf(peter,hans)
fieldHasType(FieldName, type(basic,TypeName,5) )
```

Rules

- Rules consist of a head and a body.

"head literal"
represents the predicate that is defined by the clause

```
isGrandfatherOf(G,C) :-
        isFatherOf(G,X), ( isFatherOf(X,C) ; isMotherOf(X,C) ).
```

"body literals"
represent goals to be proven in order to prove the head.

- Facts are just syntactic sugar for rules with the body "true".

```
isFatherOf(peter,hans).
```

```
isFatherOf(peter,hans) :- true.
```

these clauses are equivalent

Terms

Terms are the arguments of literals. They may be

- Variables `X`,`Y`, `Father`, `Method`, `Type`, `_type`, `_Type`, `...`
- Constants Numbers, Strings, ...
- Function terms `person(stan,laurel)`, `+(1,*(3,4))`, ...

Terms are the only data structure in Prolog!

The only thing one can do with terms is unification with other terms!

→ All computation in Prolog is based on unification.

Variables: Syntax

- Variables start with an upper case letter or an underscore '_'.

```
Country  Year  M  V  _45  _G107  _europe   _
```

internal naming scheme for variables

- Anonymous Variables ('_')
  - For irrelevant values
  - “Does Peter have a father?” We neither care whether he has one or many fathers nor who the father is:

```
?- isFatherOf(_,peter).
```

Variables: Semantics

- The scope of a variable is the clause in which it appears
- Variables that appear only once in a clause are called singletons.
  - Mostly results of typos
  - SWI Prolog warns about singletons,
  - … unless you suppress the warnings
- All occurrences of the same variable in the same clause must have the same value!
  - Exception: the "anonymous variable" (the underscore)

```
isGrandfatherOf(G,C) :-
        isFatherOf(G,F),
        isFatherOf(F,C).
isGrandfatherOf(G,Child) :-
        isFatherOf(G,M),
        isMotherOf(M,Chil).

loves(romeo,juliet).
loves(john,eve).
loves(jesus,Everybody).

?- classDefT(ID, _, 'Applet', _).
```

```
_Everybody
```

Intentional singleton variable, for which singleton warnings should be supressed.

Constants

- Numbers -17 -2.67e+021 0 1 99.9 512
- Atoms sequences of letters, digits or underscore characters '_' that
  - start with a lower case letter

  OR
  - are enclosed in simple quotes ('). If simple quotes should be part of an atom they must be doubled.

  OR
  - only contains special characters

```
ok:      peter   'Fritz'   new_york   :-   -->  'I don''t know!'
```

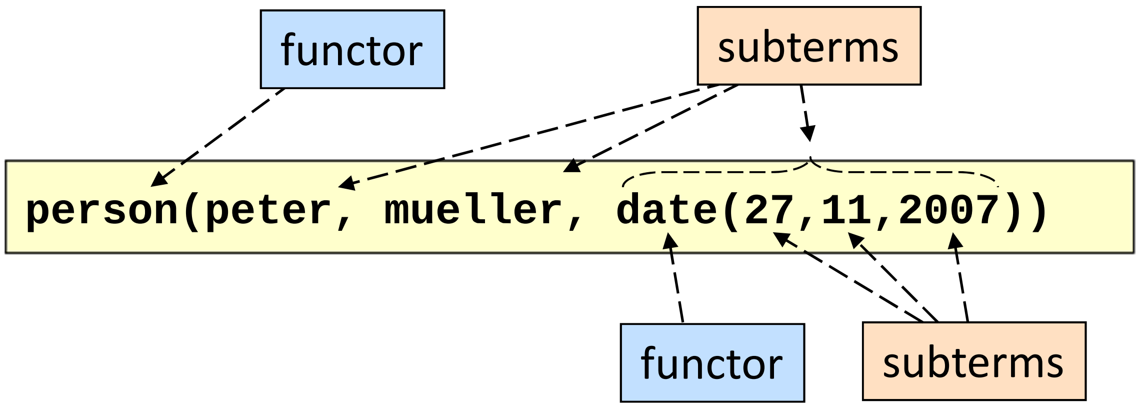
```
wrong:        Fritz    new-york    _xyz     123
```

- Remember: Prolog has no static typing!
  - So it is up to you to make sure you write what you mean.

Function Terms ('Structures')

Function terms (structures) are terms that are composed of other terms

- akin to "records" in Pascal (or objects without any behavior in Java)



- Arbitrary nesting allowed
- No static typing: **`person(1,2,'a')`** is legal!
- Function <u>terms</u> are not function <u>calls</u>! They do not yield a result!!!

Notation for function symbols: Functor/Arity, e.g. **`person/3, date/3`**

Using Function Terms as Data Types

- Function terms are the only “data constructor” in Prolog
- In conjunction with recursive predicates, one can construct arbitrarily deep structures

```
binary_tree(empty).
binary_tree(tree(Left,Element,Right)) :-
    binary_tree(Left),
    binary_tree(Right).

?- binary_tree( Any ).
?- binary_tree( tree(empty,1,Right) ).
```

recursive definition of „binary tree“ data type

Lists – Recursive Structures with special Syntax

- Lists are denoted by square brackets "[]"

```
[]  [1,2,a]  [1,[2,a],c]
```

- The pipe symbol "|" delimits the initial elements of the list from its „tail“

```
[1|[2,a]]  [1,2|[a]]  [Head|Tail]
```

- Lists are just a shorthand for the binary functor ‘.’

```
[1,2,a] = .(1,.(2,.(a,[])))
```

- You can define your own list-like data structure like this:

```
mylist( nil ).
mylist( list(Head,Tail) ) :- mylist( Tail ).
```

Strings

- Strings are enclosed in double quotes (")
  - "`Prolog`" is a string
  - '`Prolog`' is an atom
  - `Prolog` (without any quotes) is a variable

- A string is just a list of ASCII codes

```
"Prolog" = [80,114,111,108,111,103]
         = .(80,.(114,.(111,.(108,.(111,.(103,[]))))))
```

- Strings are seldom useful → Better use atoms!
  - There are many predefined predicates for manipulating atoms the same way as Java uses strings.
  - Prolog strings are useful just for low level manipulation
  - Their removal from the language has often been suggested

Terms, again

- Terms are constanten, variables or structures

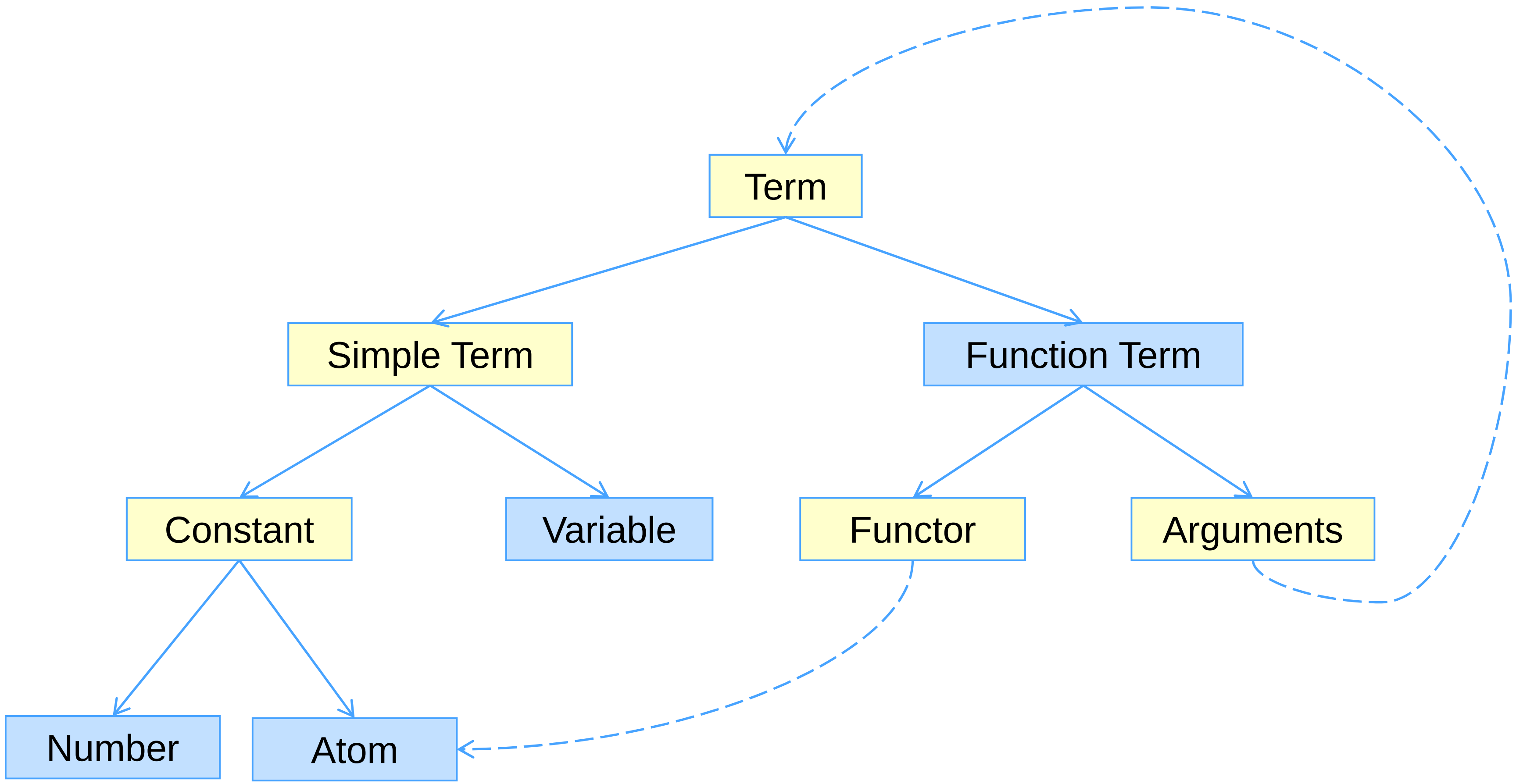
```
peter
27
MM
[europa, asien, afrika | Rest]
person(peter, Nachname, date(27, MM, 2007))
```

- A ground term is a variable free term

```
person(peter, mueller, date(27, 11, 2007))
```

Terms: Summary

Relations between the four different kinds of term



Unification – the only operation on terms

Equality

Variable bindings, Substitutions, Unification

Most general unifiers

Equality (1)

- Testing equality of terms

```
?- europe = europe.                          yes
?- 5 = 2.                                    no
?- 5 = 2 + 3.                                no
?- 2 + 3 = +(2, 3).                          yes
```

- Terms are not evaluated!
- Terms are equal if they are structurally equal!!
- Structural equality for ground terms:
  - functors are equal and …
  - … all argument values in the same position are structurally equal.

Constants are just functors with zero arity!

Equality (2)

- Testing equality of terms with variables:

```
?- person(peter, Name,    date(27, 11, 2007))
   =
   person(peter, mueller, date(27, MM, 2007)) .
```

- These terms are obviously not equal. However, …

Idea

- A variable can take on any value
  - For instance, **`mueller`** for **`Name`** and **`11`** for **`MM`**
  - After applying this substitution, the two person/3 terms will be equal.
- Equality = terms are equal
- Unifiability = terms can be made equal via a substitution.
- Prolog doesn't test equality but unifiability!

Unifiability

- Testing equality of terms with variables:

```
?- person(peter, Name,    date(27, 11, 2007))
   =              ↕                 ↕
   person(peter, mueller, date(27, MM, 2007)) .
```

- Terms T1 and T2 are unifiable if there is a substitution that makes them equal!

Bindings, substitutions and unifiers

- A binding is an association of a variable to a term
  - Two sample bindings: **Name** ← **mueller** and **MM** ← **11**
- A substitution is a set of bindings
  - A sample substitution: {**Name** ← **mueller**, **MM** ← **11**}
- A unifier is a substitution that makes two terms equal
  - The above substitution is a unifier for the two person/3 terms above

Unifiability (2)

- Can you find out the unifiers for these terms?

```
      date(1, 4, 1985) = date(1, 4, Year)
date(Day, Month, 1985) = date(1, 4, Year)
a(b,C,d(e,F,g(h,i,J))) = a(B,c,d(E,f,g(H,i,j)))
          [[the, Y]|Z] = [[X, dog], [is, here]]
                     X = Y + 1
```

{Year ← 1985}

{Year ← 1985, Month ← 4}

{B ← b, C ← c, …, J ← j}

{Y ← dog, X ← the, Z ← [is,here]}

{X ← Y+1}

- What about

```
p(X) = p(q(X))
```

{X ← q(X)}

produces a cyclic substitution

Application of a Substitution to a Term (1)

- Substitutions are denoted by greek letters: γ, σ, τ
  - For instance: $\gamma = \{$`Year` $\leftarrow$ `1985`, `Month` $\leftarrow$ `4`$\}$

- Application of a substitution $\tau = \{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$ to a term T
  - is written Tτ

```
date(Day,Month,1985)γ
        X=Y+1{X ← Y+1}
 f(X,1){Y←2,X←g(Y)}
```

  - replaces all the occurrences of V_i in T by $t_i\tau$, for i = 1..n.

```
date(Day,Month,1985)γ ≡ date(Day,4,1985)
        X=Y+1{X ← Y+1} ≡ Y+1=Y+1
 f(X,1){Y←2,X←g(Y)} ≡ f(g(2),1)
```

Application of a Substitution to a Term (2)

Important

For $\tau = \{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$ and $i = 1..n$

$T\tau$ replaces all the occurrences of V_i in T by $t_i\tau$.

Substitutions are applied to their own right-hand-sides too!

Therefore: `f(X,1){Y←2,X←g(Y)} ≡ f(g(2),1)`

This would be wrong: `f(X,1){Y←2,X←g(Y)} ≡ f(g(Y),1)`

Forgot to apply Y←2

Resulting Problem

- Application of cyclic substitutions creates infinite terms

`p(X){X←q(X)} ≡ p(q(q(q(q(q(q(q(q(...)...)`

- Prevention: Don't create cyclic substitutions in the first place!
  - "Occurs Check" verifies whether unification would create cyclic substitutions

„Occurs Check“ (1)

Theory

- Unification must fail if it would create substitutions with cyclic bindings

```
p(X) = p(q(X))                              // must fail
```

Problem

- Unification with “occurs-check” has exponential worst-case run-time
- Unification without “occurs-check” has linear worst-case run-time

Practical Prolog implementations

- Prolog implementations do not perform the occurs check

```
p(X) = p(q(X))                              // succeeds
```

- … unless you explicitly ask for it

```
unify_with_occurs_check(p(X), p(q(X)) )     // fails
```

„Occurs Check“ (2)

- No occurs check when binding a variable to another term

```
?- X=f(X).
X = f(**).
```

  - ➢ Circular binding is flagged (**)

```
?- X=f(X), write(X).
... printing of infitinte term never terminates ...
```

  - ➢ Printing of infinite term never terminates

```
?- X=f(X), X=a.
fail.
```

  - ➢ Circular reference is checked by second unification, so the goal fails gracefully

- SWI-Prolog has an occurs-check version of unification available

```
?- unify_with_occurs_check(X,f(X)).
fail.
```

Unification (2)

- Unification of terms T1 and T2
  - finds a substitution σ for the variables of T1 and T2 such that …
  - … if σ is applied to T1 and T2 then the results are equal

- Unification satisfies equations
- … but only if possible

```
?- p(X,f(Y),a) = p(a,f(a),Y).
X = a, Y = a.
?- p(X,f(Y),a) = p(a,f(b),Y).
fail.
```

Question

- How to unify two variables?
  - Problem: Infinitely many unifying substitutions possible!!!

```
?- p(X) = p(Y).
X = a, Y = a;
X = b, Y = b;
…
```

Solution

- Unification finds the most general unifying substitution
  - “most general unifier” (mgu)

```
?- p(X) = p(Z).
X = _G800, Y = _G800;
true.
```

Unification yields Most General Unifier (MGU)

- Unification of terms T1 and T2
  - finds a substitution σ for the variables of T1 and T2 such that …
  - … if σ is applied to T1 and T2 then the results are equal
  - if σ is a most general substitution

Theorem (Uniqueness of MGU): The most general unifier of two terms T1 and T2 is uniquely determined, up to renaming of variables.

- If there are two different most general unifiers of T1 and T2, say σ and τ, then there is also a renaming substitution γ such that $\boxed{T1\sigma\gamma \equiv T2\tau}$

- A renaming substitution only binds variables to variables

$$\boxed{f(A)\{A\leftarrow B,\ B\leftarrow C\} \equiv f(C)}$$

Computing the Most General Unifier mgu(T1,T2)

- Input: two terms, T_1 and T_2
- Output: σ, the most general unifier of T_1 and T_2 (only if T_1 and T_2 are unifiable)
- Algorithm

1. If T_1 and T_2 are the same constant or variable then $\sigma = \{\}$
2. If T_1 is a variable not occurring in T_2 then $\sigma = \{T_1 \leftarrow T_2\}$
3. If T_2 is a variable not occuring in T_1 then $\sigma = \{T_2 \leftarrow T_1\}$
4. If $T_1 = f(T_{11},\ldots,T_{1n})$ and $T_2 = f(T_{21},\ldots,T_{2n})$ are function terms with the same functor and arity
   1. Determine $\sigma_1 = \text{mgu}(T_{11}, T_{21})$
   2. Determine $\sigma_2 = \text{mgu}(T_{12}\sigma_1, T_{22}\sigma_1)$
   3. . . .
   4. Determine $\sigma_n = \text{mgu}(T_{1n}\sigma_1\ldots\sigma_{n-1}, T_{2n}\sigma_1\ldots\sigma_{n-1})$
   5. If all unifiers exist then $\sigma = \sigma_1\ldots\sigma_{n-1}\sigma_n$ (otherwise T_1 and T_2 are not unifiable)
5. Occurs check: If σ is cyclic fail, else return σ

Semantics

How do we know what a goal / program means?

→ Translation of Prolog to logical formulas

How do we know what a logical formula means?

→ Models of logical formulas (Declarative semantics)

→ Proofs of logical formulas (Operational semantics)

Question

Question

- What is the meaning of this program?

```
bigger(elephant, horse).
bigger(horse, donkey).
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

Rephrased question: Two steps

1. How does this program translate to logic formulas?
2. What is the meaning of the logic formulas?

Semantics: Translation

How do we translate a Prolog program to a formula in First Order Logic (FOL)?

→ Translation Scheme

Can any FOL formula be expressed as a Prolog Program?

→ Normalization Steps

Translation of Programs (repeated)

- A Prolog program is translated to a set of formulas, with each clause in the program corresponding to one formula:

  { bigger(elephant, horse),
  bigger(horse, donkey),
  $\forall x.\forall y.(\ bigger(x, y) \rightarrow is\_bigger(x, y)\ ),$
  $\forall x.\forall y.(\ \exists z.(bigger(x, z) \wedge is\_bigger(z, y)) \rightarrow is\_bigger(x, y)\ )$
  }

- Such a set is to be interpreted as the conjunction of all the formulas in the set:

  bigger(elephant, horse) $\wedge$
  bigger(horse, donkey) $\wedge$
  $\forall x.\forall y.(\ bigger(x, y) \rightarrow is\_bigger(x, y)\ ) \wedge$
  $\forall x.\forall y.(\ \exists z.(bigger(x, z) \wedge is\_bigger(z, y)) \rightarrow is\_bigger(x, y)\ )$

Translation of Clauses

- Each predicate remains the same (syntactically).
- Each comma separating subgoals becomes $\wedge$ (conjunction).
- Each :- becomes $\leftarrow$ (implication)
- Each variable in the head of a clause is bound by a $\forall$ (universal quantifier)
  - `son(X,Y) :- father(Y,X),male(X).`
  - $\forall x.\forall y \; son(x, y) \leftarrow father(y, x) \wedge male(x)$
- Each variable that occurs only in the body of a clause is bound by a $\exists$ (existential quantifier)
  - `grandfather(X):-      father(X,Y) , parent(Y,Z).`
  - $\forall x. \; (grandfather(x) \leftarrow \exists y.\exists z. \; father(x, y) \wedge parent(y, z))$

Translating Disjunction

- Disjunction is the same as two clauses:

```
disjunction(X) :-
    ( ( a(X,Y), b(Y,Z) )
    ; ( c(X,Y), d(Y,Z) )
    ).
```

⇔

```
disjunction(X) :-
    a(X,Y), b(Y,Z).
disjunction(X) :-
    c(X,Y), d(Y,Z) .
```

- Variables with the same name in different clauses are different
- Therefore, variables with the same name in different disjunctive branches are different too!
- Good Style: Avoid accidentally equal names in disjoint branches!
    - Rename variables in each branch and use explicit unification

```
disjunction(X) :-
    ( (X=X1, a(X1,Y1), b(Y1,Z1) )
    ; (X=X2, c(X2,Y2), d(Y2,Z2) )
    ).
```

⇔

```
disjunction(X1) :-
    a(X1,Y1), b(Y1,Z1).
disjunction(X2) :-
    c(X2,Y2), d(Y2,Z2) .
```

Declarative Semantics – in a nutshell

Meaning of Programs (in a nutshell)

Meaning of a program

Meaning of the equivalent formula.

$\text{bigger(elephant, horse)}$

$\wedge$

$\text{bigger(horse, donkey)}$

$\wedge$

$\forall x.\forall y.(\ \text{bigger}(x, y) \rightarrow$
$\qquad \text{is\_bigger}(x, y)\ )$

$\wedge$

$\forall x.\forall y.(\ \exists z.(\text{bigger}(x, z) \wedge$
$\qquad \text{is\_bigger}(z, y)) \rightarrow$
$\qquad \text{is\_bigger}(x, y)\ )$

Meaning of a formula

Set of logical consequences

$\text{bigger(elephant, horse)}$

$\wedge$

$\text{bigger(horse, donkey)}$

$\wedge$

$\text{is\_bigger(elephant, horse)}$

$\wedge$

$\text{is\_bigger(horse, donkey)}$

$\wedge$

$\text{is\_bigger(elephant, donkey)}$

Meaning of Programs

Model =
Set of logical consequences =
What is true according to the formula

Meaning of a program

Meaning of the equivalent formula.

bigger(elephant, horse)

∧

bigger(horse, donkey)

∧

$\forall x.\forall y.($ bigger(x, y) $\rightarrow$
is_bigger(x, y))

∧

$\forall x.\forall y.(\exists z.($bigger(x, z) $\wedge$
is_bigger(z, y)) $\rightarrow$
is_bigger(x, y))

Meaning of a formula

Set of logical consequences

bigger(elephant, horse)

∧

bigger(horse, donkey)

∧

is_bigger(elephant, horse)

∧

is_bigger(horse, donkey)

∧

is_bigger(elephant, donkey)

Semantics of Programs and Queries (in a nutshell)

Program

```
bigger(elephant,horse).

bigger(horse,donkey).

is_bigger(X,Y) :-
   bigger(X,Y).

is_bigger(X,Y) :-
   bigger(X,Z),
   is_bigger(Z,Y).
```

Formula

bigger(elephant, horse)
$\wedge$
bigger(horse, donkey)
$\wedge$
$\forall x. \forall y. (\text{is\_bigger}(x, y) \leftarrow \text{bigger}(x, y))$
$\wedge$
$\forall x. \forall y. (\exists z. (\text{is\_bigger}(x, y) \leftarrow \text{bigger}(x, z) \wedge \text{is\_bigger}(z, y)))$

Model

bigger(elephant, horse)
$\wedge$
bigger(horse, donkey)
$\wedge$
is_bigger(elephant, horse)
$\wedge$
is_bigger(horse, donkey)
$\wedge$
is_bigger(elephant, donkey)

Query

?-
bigger(elephant, X)
$\wedge$
is_bigger(X, donkey)

Translation

Interpretation
(logical consequence)

Matching

Declarative Semantics – the details

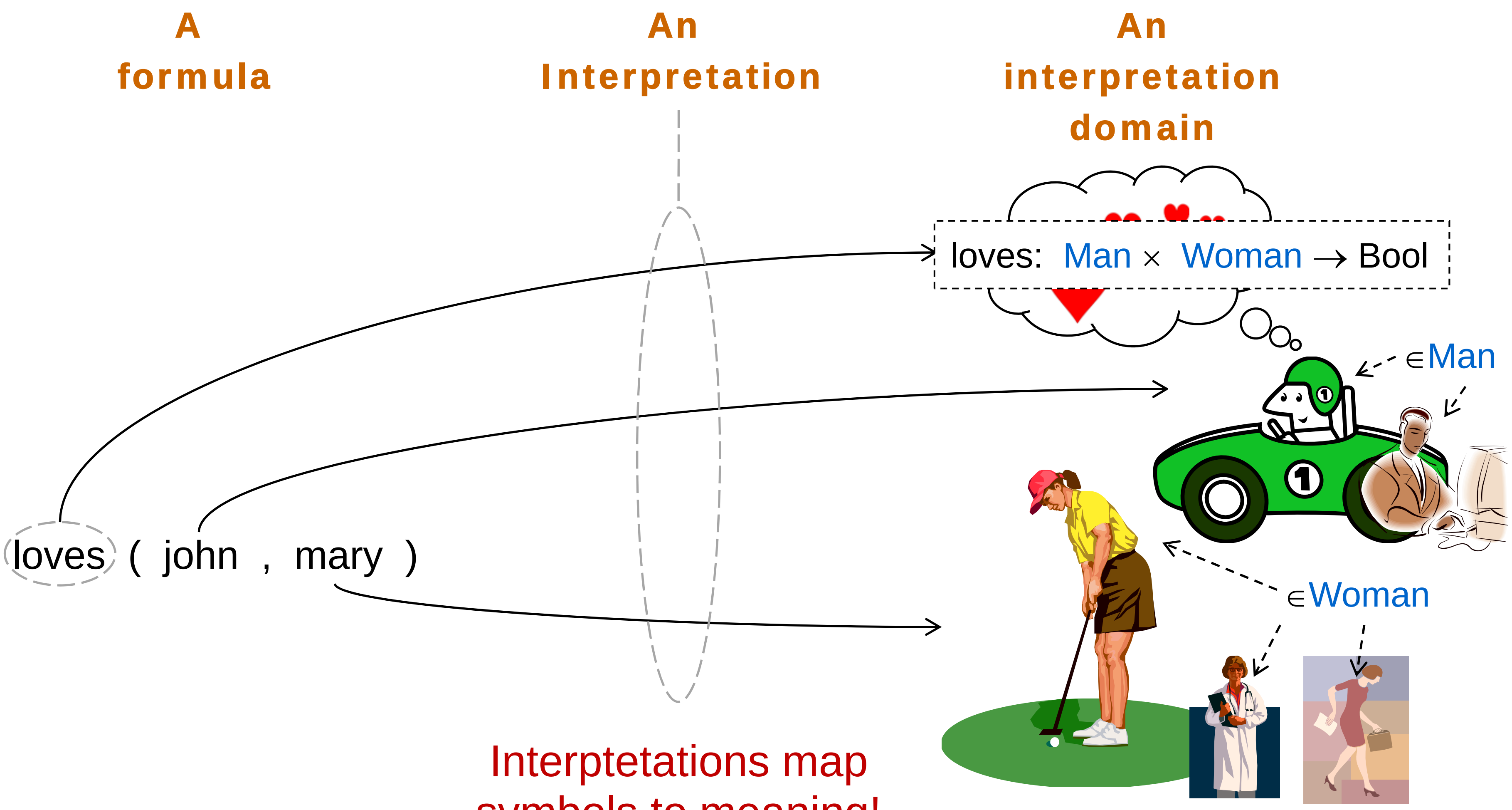
→ Interpretations of formulas

→ Herbrand Interpretations

→ Herbrand Model

→ Logical Consequence

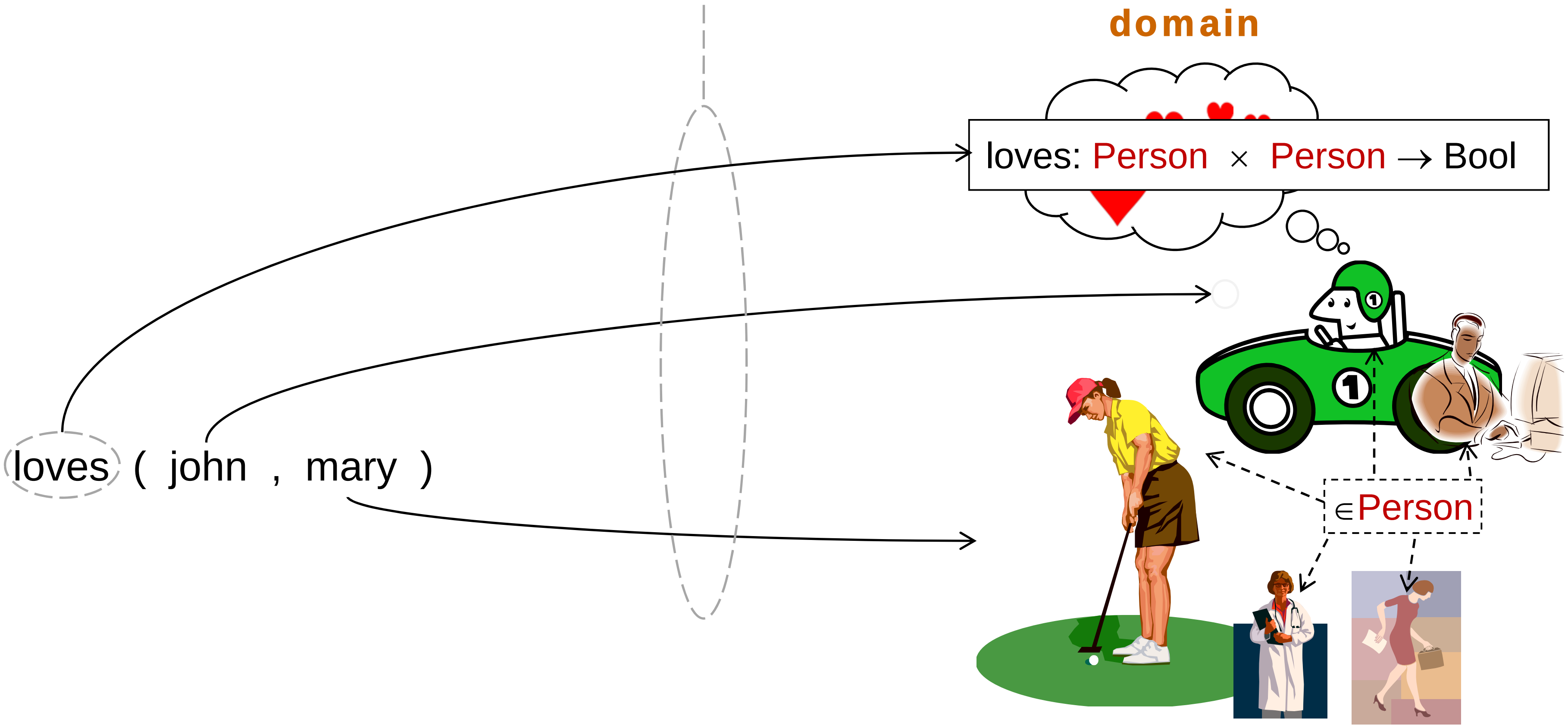
Interpretations of Formulas



Interptetations map symbols to meaning!

Interpretations of Formulas

Same formula | **Slightly different Interpretation** | **Slightly different interpretation domain**



Interpretations of Formulas

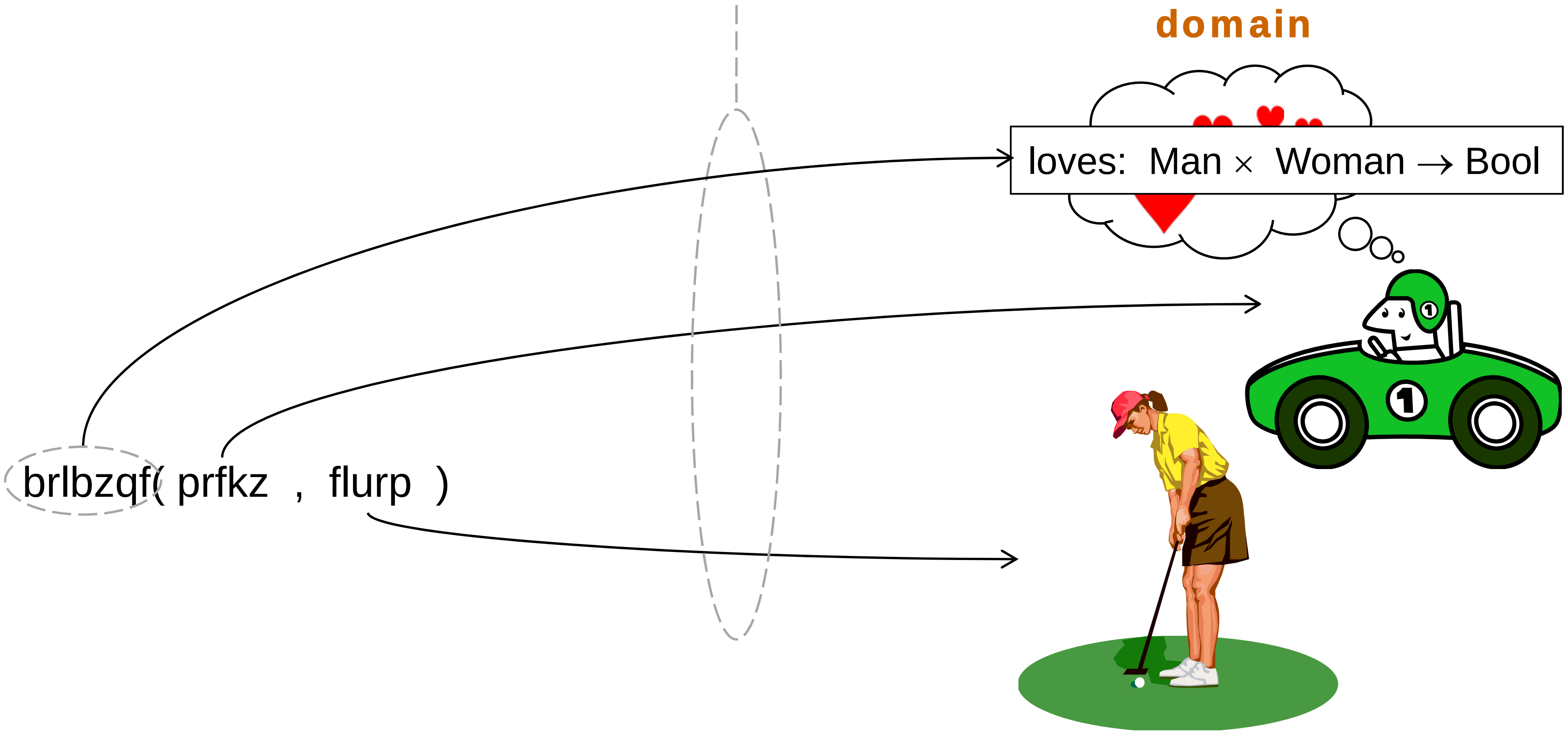
Same formula | **Other Interpretation** | **Other interpretation domain**

targetOf: Spy × Secret → Bool

loves (john , mary)

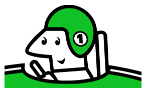
Interpretations of Formulas

A formula | An Interpretation | Our initial interpretation domain



What does this tell us?

Observations

- Formulas only have a meaning with respect to an interpretation
- An interpretation maps formulas to elements of an interpretation domain
  - constants to constant in the domain
    - ⇨ "john" to 
  - function symbols to functions on the domain
    - ⇨ no example
  - predicates to relations on the domain
    - ⇨ "loves/2" to "targetOf: Spy $\times$ Secret $\rightarrow$ Bool"
  - formulas to truth values
    - ⇨ "loves(john,mary)" to "true"

What does this tell us?

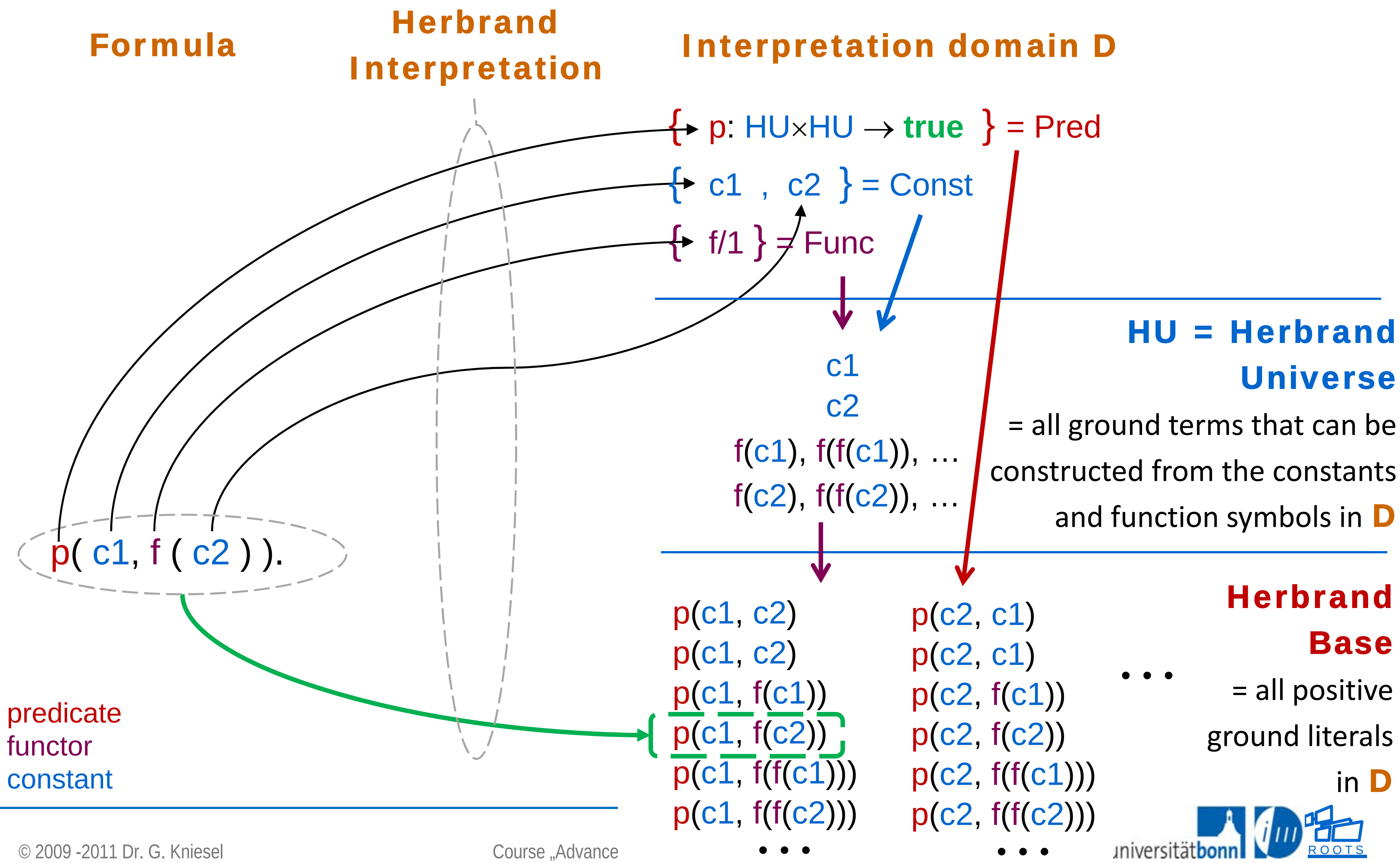
Dilemma

- Too many possible interpretations!
- Which interpretation to use for proving truth?

Solution

- For universally quantified formulas there is a “standard” interpretation, the “Herbrand interpretation” that has two nice properties:
  - If any interpretation satisfies a given set of clauses *S* then there is a Herbrand interpretation that satisfies them
    - ⇨ It suffices to check satisfiability for the Herbrand interpretation!
  - If *S* is unsatisfiable then there is a finite unsatisfiable set of ground instances from the Herbrand base defined by *S*.
    - ⇨ Unsatisfiability can be checked finitely

Herbrand Interpretations



universitätbonn ROOTS

Herbrand Interpretations of Formulas with Variables

Formula | **Herbrand Interpretation** | **Interpretation domain D**

{ p: HU×HU×HU → **true** } = Pred

{ c1 , c2 } = Const

{ f/1 } = Func

p(c1, f (X), c2).

c1
c2
f(c1), f(f(c1)), …
f(c2), f(f(c2)), …

HU = Herbrand Universe
= all ground terms that can be constructed from the constants and function symbols in **D**

p(c1, c1,c2) p(c2, c1,c1)
p(c1, c2,c2) p(c2, c2,c1)
p(c1, f(c1),c2) p(c2, f(c1),c1) . . .
p(c1, f(c2),c2) p(c2, f(c2),c1)
p(c1, f(f(c1)),c2) p(c2, f(f(c1)),c1)
p(c1, f(f(c2)),c2) p(c2, f(f(c2)),c1)
. . . . . .

Herbrand Base
= all positive ground literals in **D**

predicate
functor
constant

Herbrand Models (1)

- The Interpretation Domain (D) of a program P consists of three sets:
  - Const contains all constants occurring in P
  - Func contains all function symbols occurring in P
  - Pred contains a predicate $p\colon \underbrace{HU \times \ldots \times HU}_{\text{arity } n} \rightarrow true$

  for each predicate symbol p of arity n occurring in the program P

- The Herbrand Universe (HU) of a program P is the set of all ground terms that can be constructed from the function symbols and constants in P

- The Herbrand Base of a program P is the set of all positive ground literals that can be constructed by applying the predicate symbols in P to arguments from the Herbrand Universe of P

Herbrand Models (2)

- A Herbrand Interpretation maps each formula in P to the elements of the Herbrand Base that are its logical consequences
  - Each ground fact is mapped to true.
  - Each possible ground instantiation of a non-ground fact is mapped to true.
  - Each instantiation of the head literal of a rule that is a logical consequence of the rule body is mapped to true

- The Herbrand Model of a program P is the subset of the Herbrand Base of P that is true according to the Herbrand Interpretation.
  - It is the set of all logical consequences of the program

- The Herbrand Model can be constructed by fixpoint iteration:
  - Initialize the model with the ground instantiations of facts in P
  - Add all new facts that follow from the intermediate model and P
  - … until the model does not change anymore (= fixpoint is reached)

Constructing Models by Fixpoint Iteration

Program

```
p :- q.
q :- p.
p :- r.
r.
```

Formula

$p \leftarrow q \land$
$q \leftarrow p \land$
$p \leftarrow r \land$
r

Model

Fixpoint

$M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M_4 = M_3$

M_0	M_1	M_2	M_3	$M_4=M_3$
	r.	r. p.	r. p. q.	r. p. q.

```
r.                 % r
```

```
r.                 % r
p :- r.            % p
```

```
r.                 % r
p :- r.            % p
q :- p.            % q
```

```
r.                 % r
p :- r.            % p
q :- p.            % q
p :- q.            % p
```

Clauses contributing model elements in the respective iteration

Declarative Semantics → Algorithm

- Model-based semantics
  - Herbrand interpretations and Herbrand models
  - Basic step: “Entailment” (Logical consequence)
  - A formula is true if it is a logical consequence of the program

Program	Formula	Model	Query

`bigger(elephant,horse).`
`bigger(horse,donkey).`
`...`
	bigger(elephant, horse)
	^
	bigger(horse, donkey)
	^
	…
		bigger(elephant, horse)
		^
		bigger(horse, donkey)
		^
		…
			?-
			bigger(elephant, X)
			^
			is_bigger(X, donkey)

Translation (Program → Formula)

Interpretation (Formula → Model)
(logical consequence)

Matching (Query → Model)

- Algorithm = Logic + Control
  - Logic = Clauses
  - Control = Bottom-up fixpoint iteration to build the model
  - Control = Matching of queries to the model

Declarative Semantics Assessed

Pro

- Simple
  - Easy to understand
- Thorough formal foundation
  - implication (entailment)

Perfect for understanding the meaning of a program

Contra

- Inefficient
  - Need to build the whole model in the worst case
- Inapplicable to infinite models
  - Never terminates if the query is not true in the model

Bad as basis of a practical interpreter implementation

Operational Semantics

Horn clauses

Normalization

SLD-Resolution

Negation as failure

Translation of Programs (repeated)

- A Prolog program is translated to a set of formulas, with each clause in the program corresponding to one formula:

  { bigger(elephant, horse),
  bigger(horse, donkey),
  $\forall x.\forall y.(\ bigger(x, y) \rightarrow is\_bigger(x, y)\ ),$
  $\forall x.\forall y.(\ \exists z.(bigger(x, z) \wedge is\_bigger(z, y)) \rightarrow is\_bigger(x, y)\ )$
  }

- Such a set is to be interpreted as the conjunction of all the formulas in the set:

  bigger(elephant, horse) $\wedge$
  bigger(horse, donkey) $\wedge$
  $\forall x.\forall y.(\ bigger(x, y) \rightarrow is\_bigger(x, y)\ ) \wedge$
  $\forall x.\forall y.(\ \exists z.(bigger(x, z) \wedge is\_bigger(z, y)) \rightarrow is\_bigger(x, y)\ )$

Horn Clauses

- The formula we get when translating a Prolog clause has the structure:

$$a_1 \wedge a_2 \wedge \cdot\cdot\cdot \wedge a_n \rightarrow B$$

- Such a formula can be rewritten as follows:

$$a_1 \wedge a_2 \wedge \cdot\cdot\cdot \wedge a_n \rightarrow B \qquad \text{by law } a \rightarrow B \equiv \neg a \vee B \text{ we get}$$

$$\neg(a_1 \wedge a_2 \wedge \cdot\cdot\cdot \wedge a_n) \vee B \qquad \text{by law } \neg(a \wedge B) \equiv \neg a \vee \neg B \text{ we get}$$

$$\neg a_1 \vee \neg a_2 \vee \cdot\cdot\cdot \vee \neg a_n \vee B$$

- Hence, every Prolog clause can be translated as a disjunction of negative literals with at most one positive literal.

- This is called a Horn clause.

Horn Clauses: Relevance

- Expressiveness
  - Every (closed) first order logic formula can be translated to Horn clause form.
  - This translation preserves (un)satisfiability: If the original formula is (un)satisfiable, the translated one is (un)satisfiable too and vice versa.

- Efficiency
  - Satisfiability is the problem of determining if the variables of a Boolean formula can be assigned in such a way as to make the formula true.
    - ⇨ Satisfiability is an NP-complete problem. ☹
  - There exists an efficient automated way to prove the unsatisfiability of a set of Horn clauses: SLD-Resolution.
  - This is the basis for practical implementations of Prolog interpreters and compilers.

- SLD-Resolution is only applicable to Horn Clauses

Normalization: Translation of Formulas to Horn Clauses

Start: Closed First Order Formula (FOF)

- "Closed" means that each variable is in the scope of a quantifier
  - ⇨ "in the scope of" a quantifier = "bound by" a quantifier.

1. Disjunct Variable Form (VDF)
   - Rename variables bound by quantifiers so that they are unique!

2. Elementary Junctor Form (EJF)
   - Reduce $\Rightarrow$, $\Leftrightarrow$, etc. to $\vee$, $\wedge$ and $\neg$ according to the following rules:

$\phi \Longrightarrow \psi$	$\rightleftharpoons$	$\neg\phi \vee \psi$
$\phi \Longleftarrow \psi$	$\rightleftharpoons$	$\phi \vee \neg\psi$
$\phi \Longleftrightarrow \psi$	$\rightleftharpoons$	$(\neg\phi \vee \psi) \wedge (\phi \vee \neg\psi)$ oder $(\phi \wedge \psi) \vee \neg(\phi \vee \psi)$
$\phi \not\Longleftrightarrow \psi$	$\rightleftharpoons$	$\neg(\phi \Longleftrightarrow \psi)$

3. Negation form (NF)
   - EJF and all negations in front of atomic formulas (= literals) according to the following rules:

$\neg\neg\phi$	$\rightleftharpoons$	ϕ
$\neg(\phi \vee \psi)$	$\rightleftharpoons$	$\neg\phi \wedge \neg\psi$
$\neg(\phi \wedge \psi)$	$\rightleftharpoons$	$\neg\phi \vee \neg\psi$
$\neg\forall_x \phi$	$\rightleftharpoons$	$\exists_x \neg\phi$
$\neg\exists_x \phi$	$\rightleftharpoons$	$\forall_x \neg\phi$

Normalization Steps (cont.)

We illustrate the previous steps on a formula from our translated program:

- A formula in Disjunct Variable Form

$$\forall x.\forall y.(\ \exists z.(bigger(x, z) \wedge is\_bigger(z, y)) \rightarrow is\_bigger(x, y)\ )$$

- Its Elementary Junctor Form is

$$\forall x.\forall y.(\ \neg\exists z.(bigger(x, z) \wedge is\_bigger(z, y)) \vee is\_bigger(x, y)\ )$$

- Its Negation Form is

$$\forall x.\forall y.(\ \forall z.\neg(bigger(x, z) \wedge is\_bigger(z, y)) \vee is\_bigger(x, y)\ ) \quad \Leftrightarrow$$

$$\forall x.\forall y.(\ \forall z.(\neg bigger(x, z) \vee \neg is\_bigger(z, y)) \vee is\_bigger(x, y)\ )$$

Normalization Steps (cont.)

4. Prenex Normal Form (PNF): $Q_1x_1 \cdots Q_nx_n\psi$
   - Move all quantifiers to the prefix(= the left-hand-side)
   - The matrix (= remaining right-hand-side part of formula) is quantifier-free
   - Each formula in VDF can be translated to PNF using the following rules:

Introduction:	$\forall_x\phi \rightleftharpoons \phi$ if x not free in Φ	$\exists_x\phi \rightleftharpoons \phi$ if x not free in Φ
Negation:	$\forall_x\neg\phi \rightleftharpoons \neg\exists_x\phi$	$\exists_x\neg\phi \rightleftharpoons \neg\forall_x\phi$
Conjunction:	$\forall_x(\phi \wedge \psi) \rightleftharpoons (\forall_x\phi) \wedge (\forall_x\psi)$	$\exists_x(\phi \wedge \psi) \rightleftharpoons (\exists_x\phi) \wedge \psi$ if x not free in Φ $\exists_x(\phi \wedge \psi) \rightleftharpoons \phi \wedge (\exists_x\psi)$ if x not free in Ψ
Disjunction:	$\forall_x(\phi \vee \psi) \rightleftharpoons (\forall_x\phi) \vee \psi$ if x not free in Φ $\forall_x(\phi \vee \psi) \rightleftharpoons \phi \vee (\forall_x\psi)$ if x not free in Ψ	$\exists_x(\phi \vee \psi) \rightleftharpoons (\exists_x\phi) \vee (\exists_x\psi)$
Implication:	$\forall_x(\phi \Longrightarrow \psi) \rightleftharpoons (\exists_x\phi) \Longrightarrow \psi)$ if x not free in Φ $\forall_x(\phi \Longrightarrow \psi) \rightleftharpoons \phi \Longrightarrow (\forall_x\Psi)$ if x not free in Ψ	$\exists_x(\phi \Longrightarrow \psi) \rightleftharpoons (\forall_x\phi) \Longrightarrow (\exists_x\psi)$
Commutativity:	$\forall_x\forall_y\phi \rightleftharpoons \forall_y\forall_x\phi$	$\exists_x\exists_y\phi \rightleftharpoons \exists_y\exists_x\phi$

Normalization Steps (cont.)

We illustrate the PNF by continuing our example:

- Its Negation Form was

$$\forall x.\forall y.(\ \forall z.(\neg bigger(x, z) \vee \neg is\ bigger(z, y)) \vee is\_bigger(x, y)\ )$$

- Its Prenex Normal Form is

$$\forall x.\forall y.(\ \forall z.(\neg bigger(x, z) \vee \neg is\_bigger(z, y) \vee \forall z.is\_bigger(x, y)\ )$$

$$\forall x.\forall y.(\ \forall z.(\neg bigger(x, z) \vee \neg is\_bigger(z, y) \vee is\_bigger(x, y))\ )$$

$$\forall x.\forall y.\forall z.(\neg bigger(x, z) \vee \neg is\_bigger(z, y) \vee is\_bigger(x, y))$$

Normalization Steps (cont.)

4. Skolem Form (SF)
   - Replace in PNF formula all occurrences of each existential variable by a unique constant
   - Skolemization does not preserve truth but preserves satisfiability.
   - This is sufficient since resolution proves truth of F by proving unsatisfiability of $\neg$F.

5. Conjunctive Normal Form (CNF)
   - Transform quantor-free matrix of formulas in PNF into a conjunction of disjunctions of atoms or negated atoms
   - A formula can be translated to CNF if and only if it is quantor-free.

6. Clausal Form
   - A formula in PNF, SF and with matrix in CNF is said to be in clausal form.
   - Each conjunct of a formula in clausal form is one clause.

Normalization Steps (cont.)

Our previous example already was in Skolem form (no existential quantifiers).

- Here is another formula, which is in Prenex but not Skolem form:

$$\exists x \exists v \forall y \forall w.(R(x,y) \land \neg R(w,v))$$

- Its Skolem Form is

$$\forall y \forall w.(R(c_1,y) \land \neg R(w,c_2))$$

- Skolem form is often written without quantifiers, abusing the implicit knowledge that all variables are universally quantified

$$R(c_1,y) \land \neg R(w,c_2)$$

Translation of Queries: Basics

- Undecidability of first order logic
  - There is no automated proof system that always answers yes if a goal is provable from the available clauses and answers no otherwise.

- Semi-decidability of first order logic
  - It is possible to determine unsatisfiability of a formula by showing that it leads to a contradiction (an empty clause)

Implication of Semi-Decidability

- We cannot prove a goal directly but must show that adding the negation of the goal to the program $\mathscr{P}$ makes $\mathscr{P}$ unsatisfiable

$$\mathscr{P} \models G \quad \text{is proven by showing that} \quad (\mathscr{P} \cup \neg G) \models \{\}$$

- Proving a formula by showing that its negation is wrong is called proof by refutation.

Translation of Queries

The query

?- is_bigger(elephant, X), is_bigger(X, donkey).

corresponds to the rule

fail :- is_bigger(elephant, X), is_bigger(X, donkey).

and to the formula

$$\forall x.\ \neg(\text{is\_bigger}(\text{elephant}, x) \wedge \text{is\_bigger}(x, \text{donkey})) \quad \rightarrow \quad \text{false}$$

Operational Semantics

Equality ✓
Variable bindings, Substitutions, Unification ✓
Most general unifiers ✓
Clause translation ✓
Normalization ✓
SLD-Resolution
Negation as failure

Proof by Refutation via Resolution

Negation

- Formula that we want to prove $(\exists x \forall y.\ R(x,y)) \rightarrow \forall y \exists x.\ R(x,y)$
- Its negation $\neg\Big((\exists x \forall y.\ R(x,y)) \rightarrow \forall y \exists x.\ R(x,y)\Big)$

Normalization

possible for all closed formulas (which only contain variables bound by quantifiers)

- Variable Disjunct Form $\neg\Big((\exists x \forall y.\ R(x,y)) \rightarrow \forall v \exists w.\ R(w,v)\Big)$
- Elementary Junctor Form $(\exists x \forall y.\ R(x,y)) \wedge \neg \forall v \exists w.\ R(w,v)$
- Prenex Normal Form $\exists x \boxed{\exists v} \forall y \boxed{\forall w}.\ R(x,y) \wedge \neg R(w,v)$
- Skolemized Form (implicit ∀) $R(c_0,y) \wedge \neg R(w,c_1)$
- (Horn) Clause Form (implicit ∀) $\{\{R(c_0,y)\},\{\neg R(w,c_1)\}\}$

Refutation

- Unification with mgu $\{w \leftarrow c_0,\ y \leftarrow c_1\}$ $\{\{R(c_0,c_1)\}\ \{\neg R(c_0,c_1)\}\}$
- Resolution of clause 2 with clause 1 $\{\}$

Why is unification so important?

Unification is the basic operation

of any Prolog interpreter.

- Resolution is the process by which Prolog derives answers (successful substitutions) for queries
- During resolution, clauses that can be used to prove a goal are determined via unification

```
?- isFatherOf(paul,Child).

   isFatherOf( F  ,  C  ) :- isMarriedTo(F,M), isMotherOf(M,C).
```

Resolution

- Resolution Principle

The proof of the goal G	`?- P, L, Q.`
if there exists a clause	L_0 `:-` L_1`, ...,` L_n ($n \geq 0$)
such that	σ = mgu(L, L_0)
can be reduced to prooving	`?-` $P\sigma$, $L_1\sigma$, ... , $L_n\sigma$, $Q\sigma$.

- Informal "resolution algorithm"

To proof of the goal G	`?- P, L, Q.`
select one literal in G, say	`L`,
select a copy of a clause	L_0 `:-` L_1`, ...,` L_n ($n \geq 0$)
such that there exists	σ = mgu(L, L_0)
apply σ to the goal	`?-` $P\sigma$, $L\sigma$, $Q\sigma$
apply σ to the clause	$L_0\sigma$ `:-` $L_1\sigma$, ..., $L_n\sigma$
replace $L\sigma$ by the clause body	`?-` $P\sigma$, $L_1\sigma$, ..., $L_n\sigma$, $Q\sigma$

Resolution

- Resolution Principle

The proof of the goal G	`?- P, L, Q.`
if there exists a clause	L_0 `:-` L_1, ..., L_n ($n \geq 0$)
such that	σ = mgu(L, L_0)
can be reduced to prooving	`?-` $P\sigma$, $L_1\sigma$, ... , $L_n\sigma$, $Q\sigma$.

- Graphical illustration of resolution by “derivation trees”

Initial goal --> `?- P, L, Q.`

Copy of clause with renamed variables (different from variables in goal!) --> L_0 `:-` L_1, L_2, ..., L_n

Unifier of selected literal and clause head --> σ = mgu(L, L_0)

Derived goal --> `?-` $P\sigma$, $L_1\sigma$, ... , $L_n\sigma$, $Q\sigma$.

Resolution reduces goals to subgoals

For	"$Goal_2$ results from $Goal_1$ by resolution"
we also say	"$Goal_2$ is derived from $Goal_1$"
or	"$Goal_1$ is reducible to $Goal_2$"
and write	"$Goal_1$ |-- $Goal_2$"

Goal / $Goal_1$ --> `?- P, L, Q.`

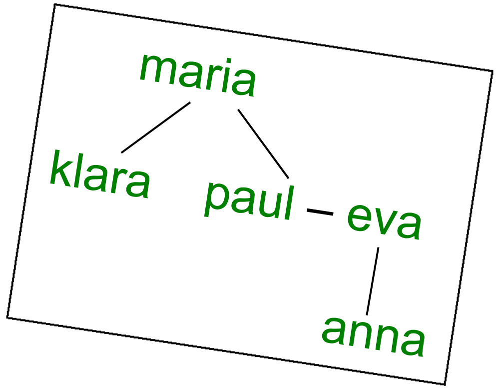
Derivation / reduction step -->

Subgoal / $Goal_2$ --> `?-` $P\sigma$, $L_1\sigma$, ... , $L_n\sigma$, $Q\sigma$.

Resolution Example: Program and Goal

- Program

```
isMotherOf(maria, klara).
isMotherOf(maria, paul).
isMotherOf(eva, anna).

isMarriedTo(paul, eva).
```

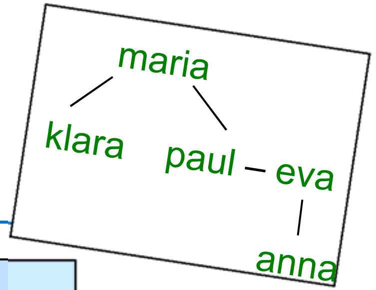



```
isGrandmaOf(G, E) :- isMotherOf(G, M), isMotherOf(M, E).
isGrandmaOf(G, E) :- isMotherOf(G, V), isFatherOf(V, E).

isFatherOf(V, K) :- isMarriedTo(V, M), isMotherOf(M,K).
...
```

- Goal

```
?- isGrandmaOf(maria,Granddaughter).
```

Resolution Example: Derivation




```
?- isGrandmaOf(maria,Granddaughter).
```

```
isGrandmaOf(G1, E1) :- isMotherOf(G1, V1),isFatherOf(V1, E1).
```

$\sigma_1 = \{\texttt{G1} \leftarrow \texttt{maria},\ \texttt{E1} \leftarrow \texttt{Granddaughter}\}$

```
?- isMotherOf(maria,V1),isFatherOf(V1,Granddaughter).
```

```
isMotherOf(maria, paul).
```

$\sigma_2 = \{\texttt{V1} \leftarrow \texttt{paul}\}$

```
?- isFatherOf(paul,Granddaughter).
```

```
isFatherOf(V2, K2) :- isMarriedTo(V2, M2),isMotherOf(M2, K2).
```

$\sigma_3 = \{\texttt{V2} \leftarrow \texttt{paul},\ \texttt{K2} \leftarrow \texttt{Granddaughter}\}$

```
?- isMarriedTo(paul,M2),isMotherOf(M2,Granddaughter).
```

```
isMarriedTo(paul, eva).
```

$\sigma_4 = \{\texttt{M2} \leftarrow \texttt{eva}\}$

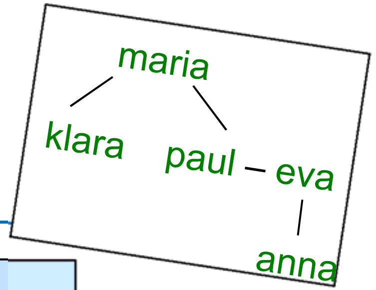
```
?- isMotherOf(eva,Granddaughter).
```

```
isMotherOf(eva, anna).
```

$\sigma_5 = \{\texttt{Granddaughter} \leftarrow \texttt{anna}\}$

$\square$

Resolution Example: Result



`?- isGrandmaOf(maria,Granddaughter).`

σ_1 = {G1←maria, E1←Granddaughter}

σ_2 = {V1←paul}

σ_3 = {V2←paul, K2←Granddaughter}

σ_4 = {M2←eva}

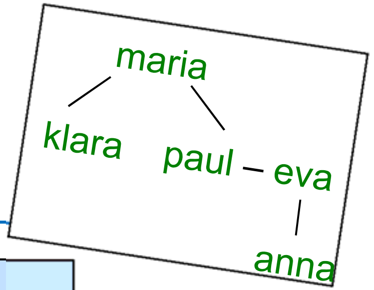
σ_5 = {Granddaughter ← anna}

So what is the result?

→ the last substitution?

→ the substitution(s) for the variable(s) of the goal?

Resolution Example: Derivation with different variable bindings



?- isGrandmaOf(maria,Granddaughter).

isGrandmaOf(G1, E1) :- isMotherOf(G1, V1),isFatherOf(V1, E1).

σ_1 = {G1←maria, Granddaughter←E1}

?- isMotherOf(maria,V1),isFatherOf(V1,E1).

isMotherOf(maria, paul).

σ_2 = {V1←paul}

?- isFatherOf(paul,E1).

isFatherOf(V2, K2) :- isMarriedTo(V2, M2),isMotherOf(M2, K2).

σ_3 = {V2←paul, E1←K2}

?- isMarriedTo(paul,M2),isMotherOf(M2,K2).

isMarriedTo(paul, eva).

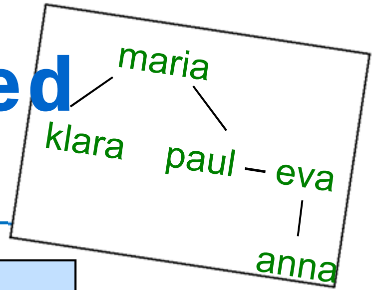
σ_4 = {M2←eva}

?- isMotherOf(eva,K2).

isMotherOf(eva, anna).

σ_5 = {K2← anna}

Resolution Example: Result revisited




```
?- isGrandmaOf(maria,Granddaughter).
```

σ_1 = {G1←maria, Granddaughter←E1}

σ_2 = {V1←paul}

σ_3 = {V2←paul, E1←K2}

σ_4 = {M2←eva}

σ_5 = {K2← anna}

Observation

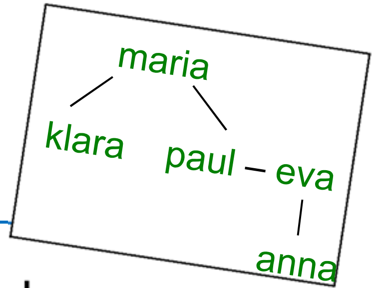
The result is not

→ the last substitution

→ the substitution(s) for the variable(s) of the goal

→ We need to „compose" the substitutions!

Resolution Example: Result



- The result is the composition of all substitutions computed along a derivation path

σ_1 = {G1←maria, E1←Granddaughter}
σ_2 = {V1←paul}
σ_3 = {V2←paul, K2←Granddaughter}
σ_4 = {M2←eva}
σ_5 = {Granddaughter ← anna}

$\sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5$ ={G1←maria, E1←Granddaughter, V1←paul, V2←paul, K2←Granddaughter, M2←eva, Granddaughter ← anna}

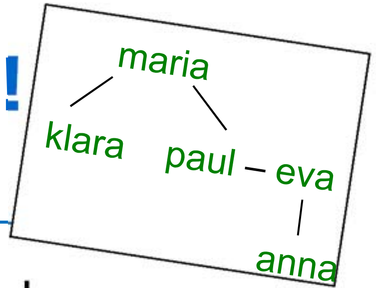
- … restricted to the bindings for variables from the initial goal

$\sigma = \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 \,|\, Vars($ **isGrandmaOf(maria,Granddaughter)** $)$

$= \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 \,|\,$ { Granddaughter }

= {Granddaughter ← anna}

Note: One can also bind differently!



- The result is the composition of all substitutions computed along a derivation path

σ_1 = {G1←maria, Granddaughter←E1}
σ_2 = {V1←paul}
σ_3 = {V2←paul, E1←K2}
σ_4 = {M2←eva}
σ_5 = {K2←anna}

Different bindings than on the previous page!

Does that mean we get a different result???

No, because during composition, later substitutions are applied to the previous ones!

$\sigma_1\,\sigma_2\,\sigma_3\,\sigma_4\,\sigma_5$ ={G1←maria, Granddaughter←anna, V1←paul, V2←paul, E1←anna, M2←eva, K2←anna}

- … restricted to the bindings for variables from the initial goal

σ = $\sigma_1\,\sigma_2\,\sigma_3\,\sigma_4\,\sigma_5$ | *Vars*(isGrandmaOf(maria,Granddaughter))

= $\sigma_1\,\sigma_2\,\sigma_3\,\sigma_4\,\sigma_5$ | { Granddaughter }

= {Granddaughter ← anna}

Same result substitution as on the previous page!

Composition Defined

Let $\sigma_1 = \{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$ and $\sigma_2 = \{W_1 \leftarrow u_1, \ldots, W_m \leftarrow u_m\}$ be two substitutions.

- Then $\sigma_1\sigma_2 = \{V_1 \leftarrow t_1\,\sigma_2, \ldots, V_n \leftarrow t_n\,\sigma_2, W_1 \leftarrow u_1, \ldots, W_m \leftarrow u_m\}$
- Terminology: $\sigma_1\sigma_2$ is called the composition of σ_1 and σ_2
- Informally: The composition $\sigma_1\sigma_2$ is obtained by
  a) applying σ_2 to the right-hand-side of σ_1
  b) and appending σ_2 to the result of step a)
- Note the difference
  - $t_1\,\sigma_2$ is the application of a substitution to a term
  - $\sigma_1\sigma_2$ is the composition of two substitutions

Restriction Defined

Let $\sigma = \{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$ be a substitution and V be a set of variables.

- Then $\sigma|V = \{V_i \leftarrow t_i \mid V_i \leftarrow t_i \wedge V_i \in V\}$
- Terminology: $\sigma|V$ is called the restriction of σ to V
- Informally: The restriction $\sigma|V$ is obtained by eliminating from σ all bindings for variables that are not in V

Resolution Result Defined

Let σ_1 , … , σ_n be the mgus computed along a successful derivation path for the goal G and let *Vars*(G) be the set of variables in G.

- Then the result substitution is $\sigma_1 \ldots \sigma_n \mid Vars(G)$
- Informally: The result substitution for a successful derivation path (= a proof) of goal G is obtained by
  a) Composing all substituions computed during the proof of the goal
  b) …and restricting the composition result to the variables of the goal.

Operational Semantics: Resolution (cont.)

OK, we've seen how resolution finds one answer. But how to find more answers?

→ Backtracking!

Derivation with Backtracking

```
✓f(a).  ✓g(a).
 f(b).   g(b).   h(b).
```

```
?- f(Y), g(Y), h(Y).
```

?- f(X), g(X), h(X).

#1, X=a | choicepoint: #2

?- g(a), h(a).

#1 | choicepoint: #2

?- h(a).

☹

- The subgoal h(a) fails because there is no clause whose head unifies with it.
- The interpreter backtracks to the last "choicepoint" for g(a)

Derivation with Backtracking

```
✓f(a).  ✓g(a).
 f(b).   g(b).   h(b).
```

```
?- f(Y), g(Y), h(Y).
```

?- f(X), g(X), h(X).

#1, X=a | choicepoint: #2

?- g(a), h(a).

☹

- The subgoal g(a) fails because there is no remaining clause (at the choicepoint or after it) whose head unifies with it.
- The interpreter backtracks to the last "choicepoint" for f(X)

Derivation with Backtracking

```
 f(a).   g(a).
 f(b).   g(b).   h(b).
```

```
?- f(Y), g(Y), h(Y).
Y=b;
no
```

?- f(X), g(X), h(X).

#2, X=b   choicepoint: ---

?- g(b), h(b).

#2   choicepoint: ---

?- h(b).

#1   choicepoint: ---

?- true.

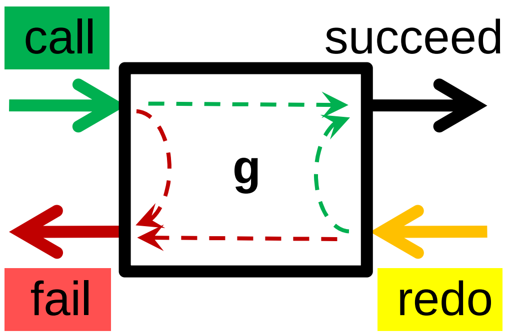
- The derivation is successful (it derived the subgoal “true”).
- The interpreter reports the successful substitutions

SLD-Resolution with Backtracking: Summary

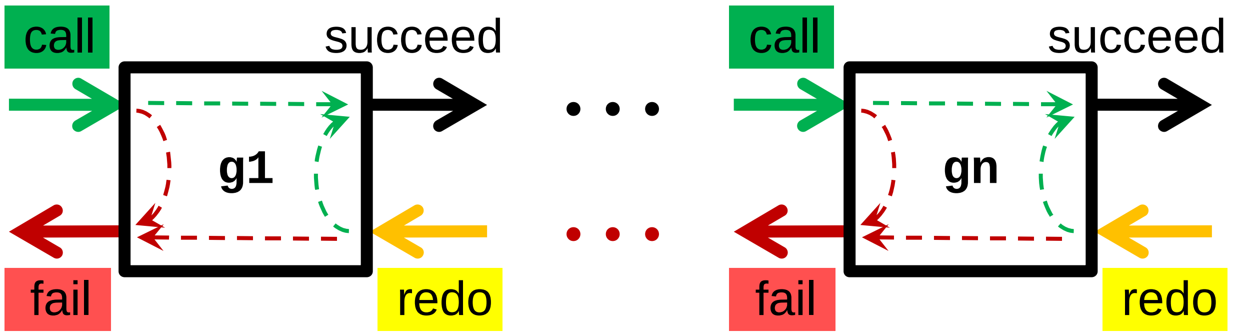
- SLD-Resolution always selects the
  - the leftmost literal in a goal as a candidate for being resolved
  - the topmost clause of a predicate definition as a candidate for resolving the current goal
- If a clause's head is not unifiable with the current goal the search proceeds immediately to the next clause
- If a clause's head is unifiable with the current goal
  - the goal is resolved with that clause
  - the interpeter remembers the next clause as a choicepoint
- If no clause is found for a goal (= the goal fails), the interpreter undoes the current derivation up to the last choicepoint .
- Then the search for a candidate clause continues from that choicepoint

Box-Model of Backtracking

- A goal is a box with four ports: call, succeed, redo, fail

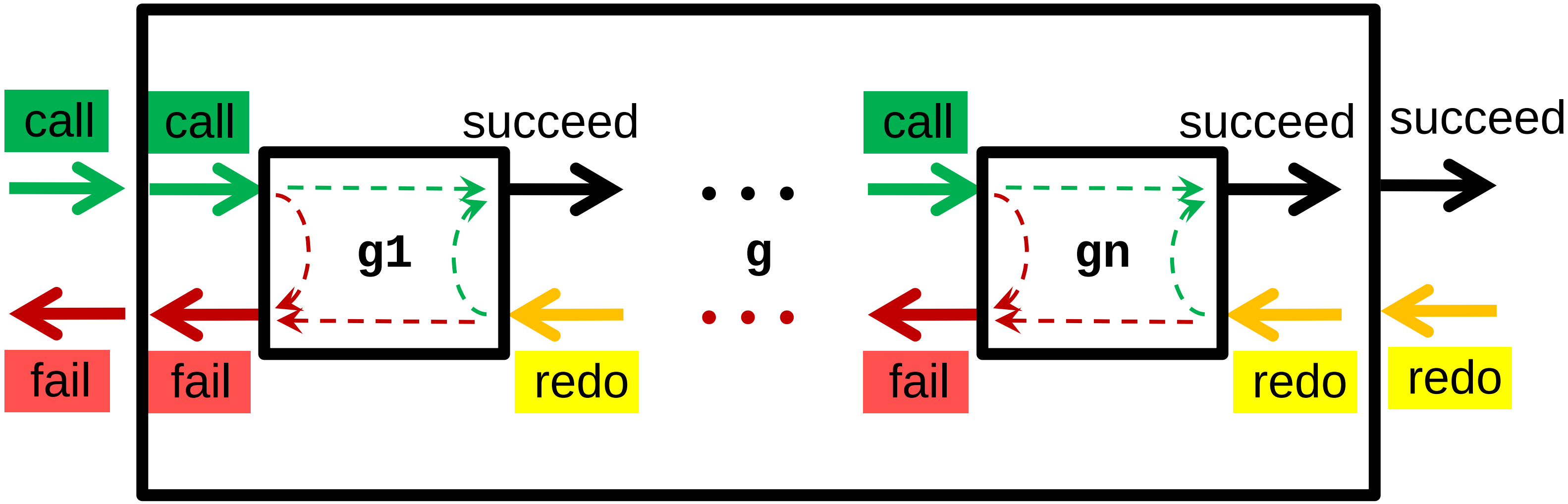


- A conjunction is a chain of connected boxes
  - the “succeed” port is connected to the “call” port of the next goal
  - the “fail” port is connected to the “redo” port of the previous goal



Box-Model of Backtracking

- Subgoals of a clause are boxes nested within the clause box, with outer and inner ports of the same kind connected
  - clause's call to first subgoal's call
  - last subgoal's suceed to clause's suceed
  - clause's redo to last subgoal's redo
  - first subgoal's fail to the fail of the clause

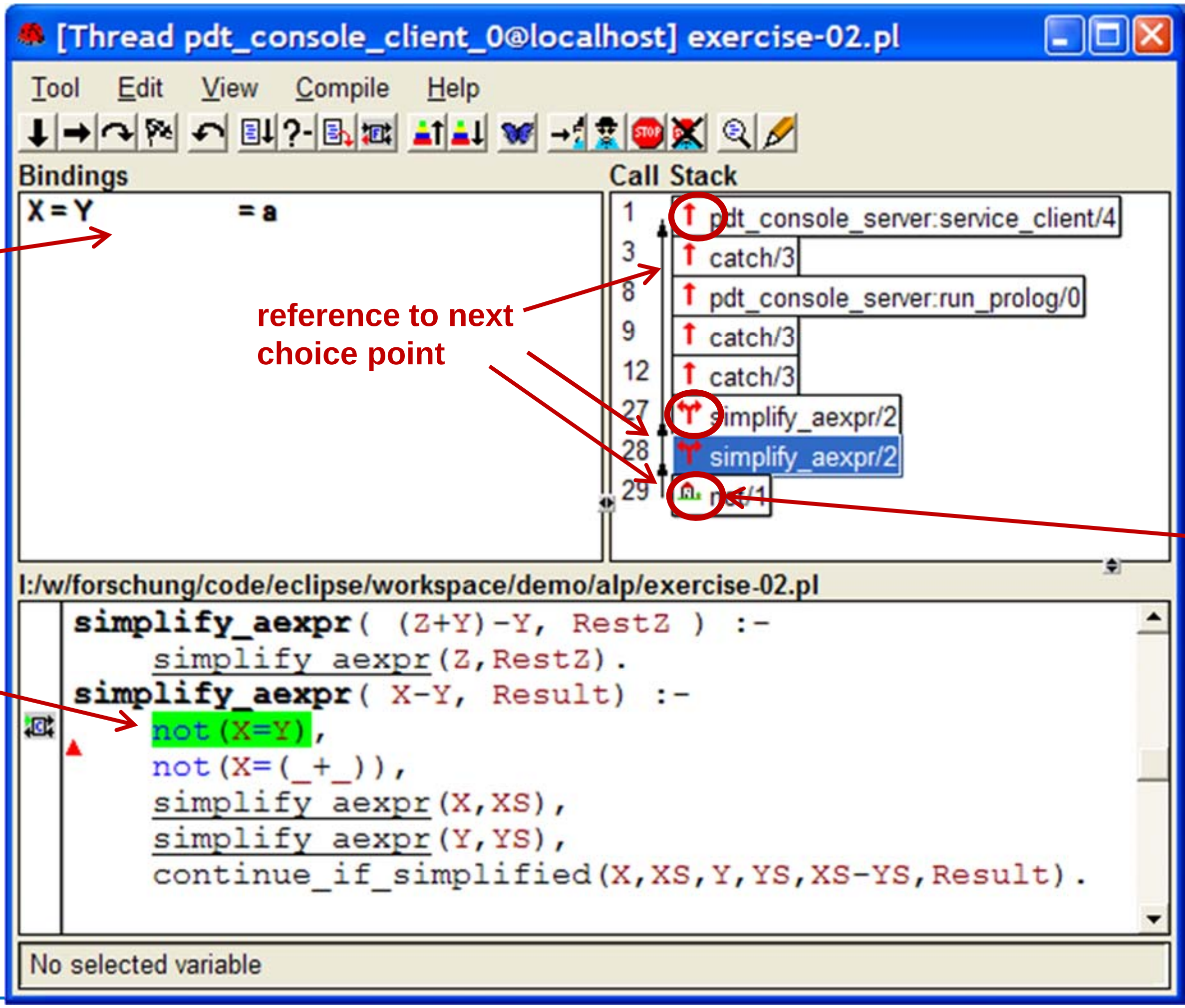


Viewing Backtracking in the Debugger (1)

```
?- gtrace, simplify_aexpr(a-a+b-b, Simple).
```

call the graphical tracer …

… for this goal.



variable bindings in selected stack frame

reference to next choice point

goals without choice points

goals with choice points

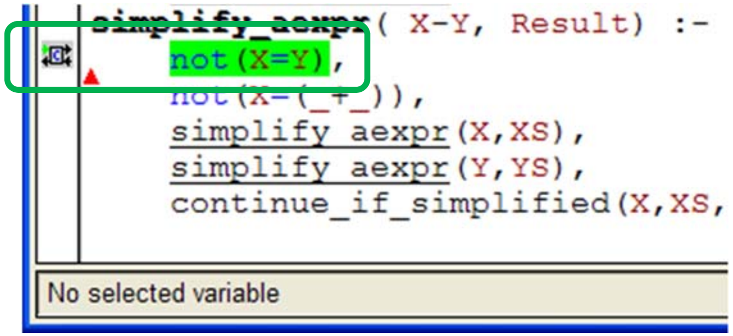
call of “built-in” predicate (has no choicepoint)

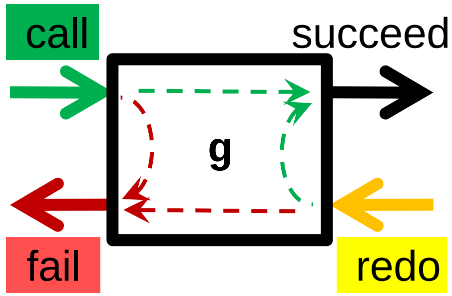
the only exception is “repeat”

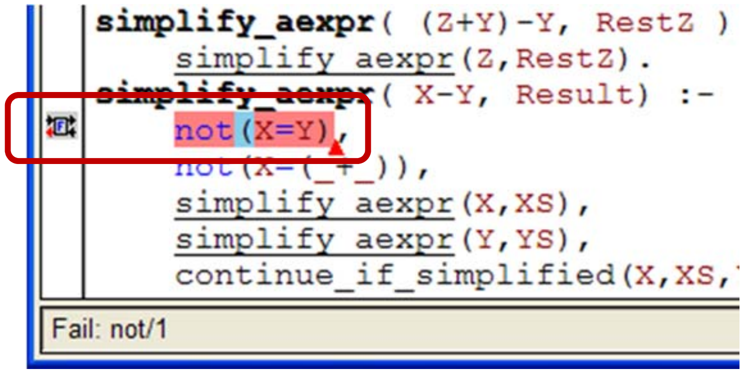
source code view of goal associated to selected stack frame

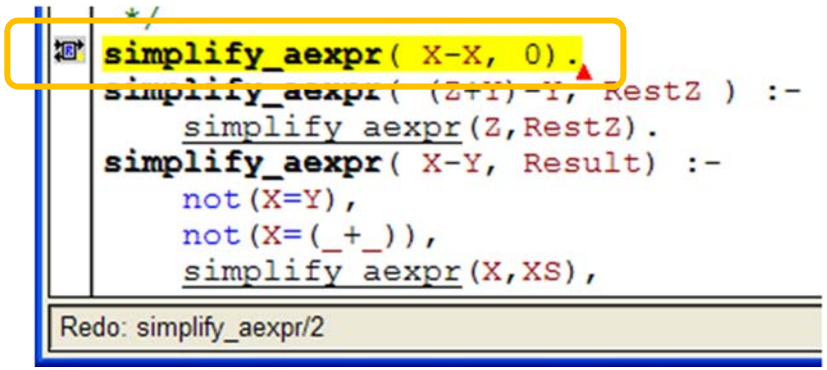
Viewing Backtracking in the Debugger (2)

The debugger visualizes the port of the current goal according to the box model.









Recursion

- Prolog predicates may be defined recursively

- A predicate is recursive if one or more rules in its definition refer to itself.

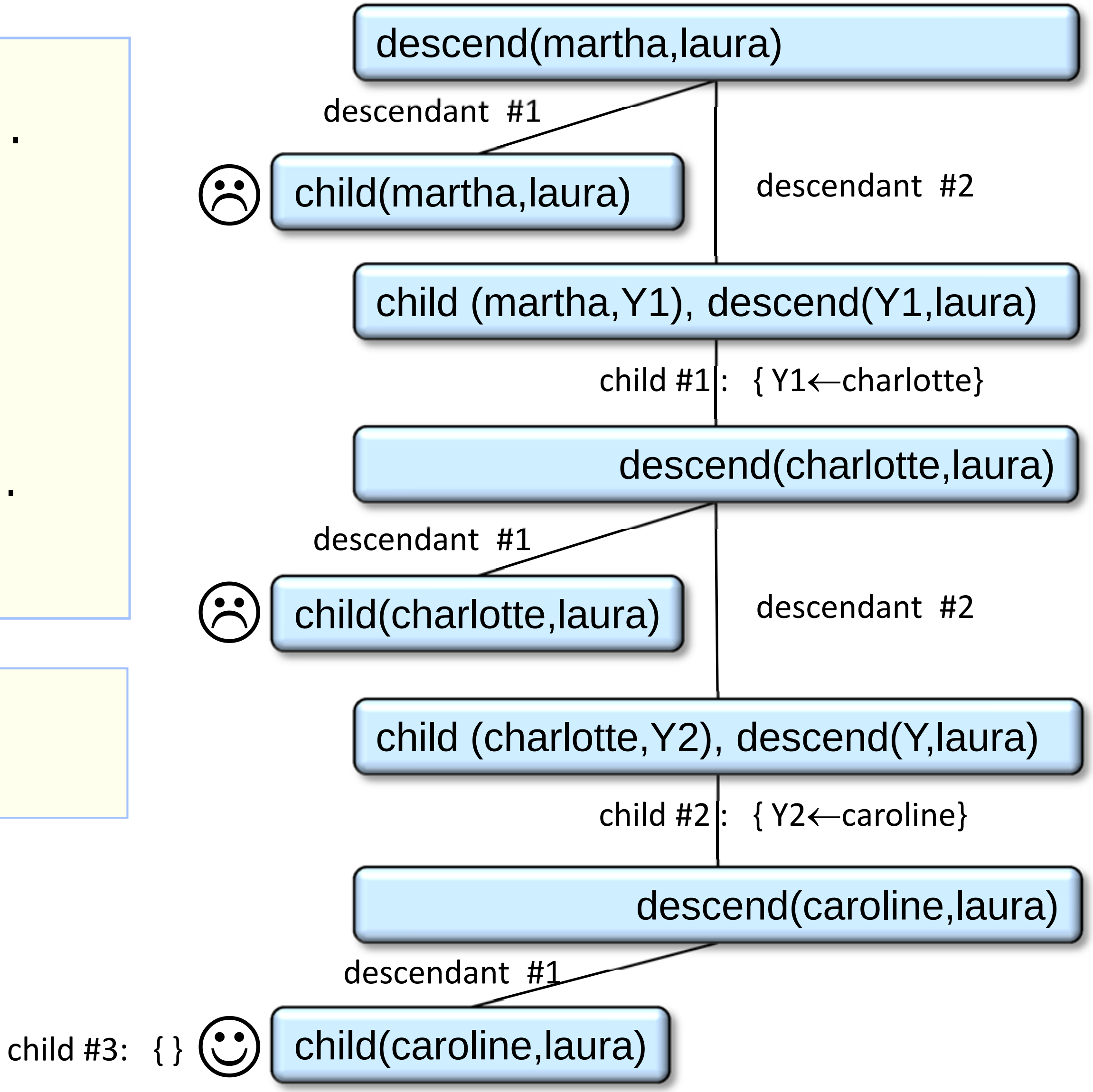
```
descendant(C,X):- child(C,X).
descendant(C,X):- child(C,D), descendant(D,X).
```

- What does the descendant/2 definition mean?
  1. *if* C is a child of X, *then* C is a descendant of X
  2. *if* C is a child of D, *and* D is a descendant of X, *then* C is a descendant of X

Recursion: Derivaton Tree for “descend”

```
child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).

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

```
?- descend(martha, laura)
yes
```



Example: Derivation and Recursion

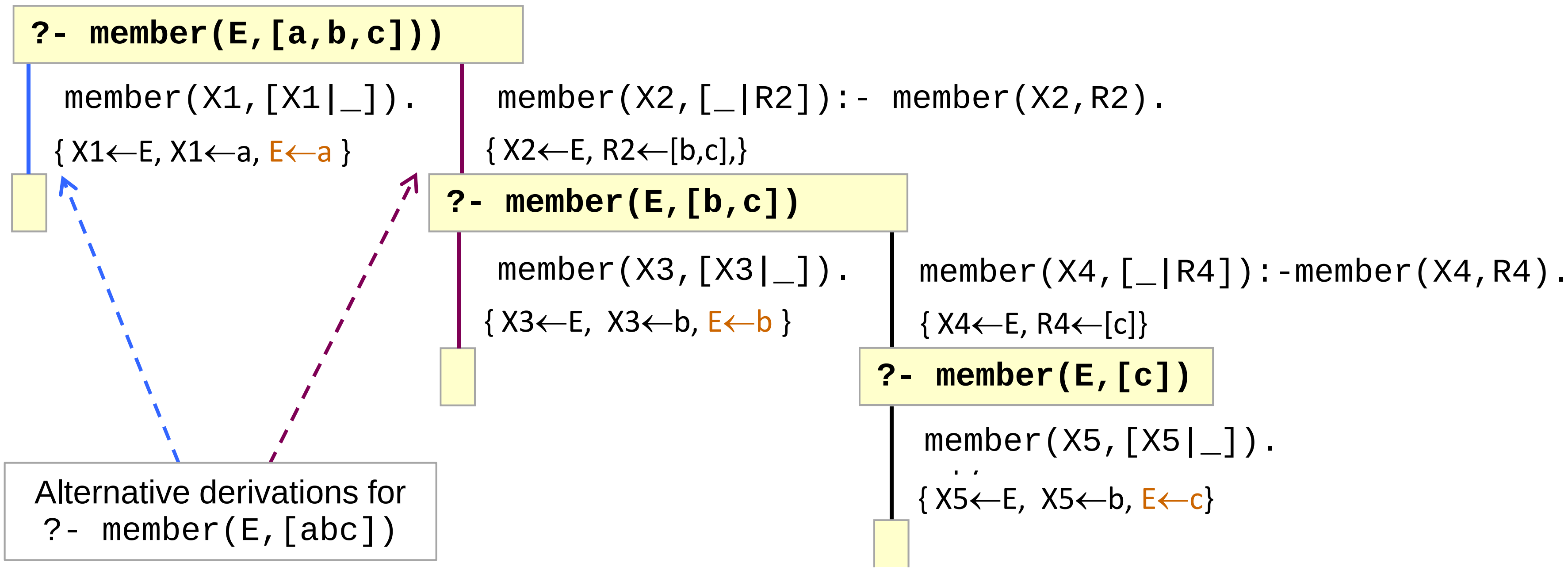
- A program (List membership: Arg1 is a member of the list Arg2)

```
member(X,[X|_]).                  % clause #1
member(X,[_|R]):- member(X,R).   % clause #2
```

- A query, its successful substitutions …

```
?- member(E,[a,b,c]).
E = a ; E = b ; E = c ; fail.
```

- … and its derivation tree



Recursion: Successor

- Suppose we want to express that
  - 0 is a numeral
  - If X is a numeral, then succ(X) is a numeral

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

- Let's see how this behaves:

```
?- numeral(X).

X = 0 ;

X = succ(0)  ;

X = succ(succ(0))  ;

X = succ(succ(succ(0)))  ;

…
```

Two different ways to give meaning to logic programs

Operational Semantics

- Proof-based approach
  - Algorithm to find a proof
  - Refutation proof using SLD resolution
  - Basic step: Derivation
- To prove a goal prove each of its subgoals
- Algorithm = Logic + Control
  - Logic = Clauses
  - Control = Top-down resolution process

Declarative Semantics

- Model-based semantics
  - Mathematical structure
  - Herbrand interpretations and Herbrand models
  - Basic step: Entailment (Logical consequence)
- A formula is true if it is a logical consequence of the program
- Algorithm = Logic + Control
  - Logic = Clauses
  - Control = Bottom-up fixpoint iteration

Semantics (cont.): Negation

OK, we've seen how to prove or conclude what is true. But what about negation?

→ Closed world assumption

→ Negation as failure

→ "Unsafe negation" versus existential variables

Closed World Assumption

- We cannot prove that something is false.
- We can only show that we cannot prove its contrary.

```
isFatherOf(kurt,peter).

?- isFatherOf(adam,cain).
no.    ← means: we cannot prove that "isFatherOf(adam,cain)" is true
```

- If we assume that everything that is true is entailed by the program, we may then conclude that what is not entailed / provable is not true.
- This assumption is known as the "Closed World assumption" (CWA)
- The conclusion is known as "Negation by Failure" (NF)

```
?- not( isFatherOf(adam,cain) ).
 yes.
← means: we conclude that "not(isFatherOf(adam,cain) )" is true because we
cannot prove that "isFatherOf(adam,cain)" is true
```

Negation with Unbound Variables (1)

Deductive Databases

```
isFatherOf(kurt,peter).

?- ∀X.isFatherOf(adam,X).
no.

?- ∀X.not(isFatherOf(adam,X)).
     ← unsafe, infinite result set!
```

- Deductive databases consider all variables to be universally quantified.
- However, the set of values for X for which isFatherOf/2 fails is infinite and unknown because it consists of everything that is not represented in the program.
- So it is impossible to list all these values!
- Therefore, the above negated query with universal quantification is unsafe.

Negation with Unbound Variables (2)

Prolog

```
isFatherOf(kurt,peter).

?- isFatherOf(adam,X).
no.

?- not( isFatherOf(adam,X) ).
yes.      ← no substitution for X returned!
```

Prolog (behind the scenes)

```
isFatherOf(kurt,peter).

?- ∀X.isFatherOf(adam,X).
no.

?- ∃X.not(isFatherOf(adam,X)).
yes.    ← safe
```

- Prolog treats free variables in negated goals as existentially quantified. So it does not need to list all possible values of X.
- It shows that there is some value for which the goal G fails, by showing that G does not succeed for any value

$$\boxed{\exists x.\neg G \Leftrightarrow \neg\forall x.G}$$

- This is precisely negation by failure!

Negation with Unbound Variables (3)

Existential variables can also occur in clause bodies:

- The clause
- means

`single(X) :- human(X), not(married(X,Y)).`	
$\forall$X.$\exists$Y. human(X) $\wedge$ not(married(X,Y)) $\rightarrow$ single(X)	

Take care: The following is different from the above:

- The clause
- is the same as
- Both mean

`single(X) :- not(married(X,Y)) human(X).`	
`single(X) :- not(married(X1,Y)), human(X).`	
$\forall$X.$\exists$X1.$\exists$Y human(X) $\wedge$ not(married(X1,Y)) $\rightarrow$ single(X).	

Remember: Free variables in negated goals are existentially quantified.

- They do not “return bindings” outside of the scope of the negation.
- They are different from variables outside of the negation that accidentally have the same name.

Negation with Unbound Variables (4)

Explanations for the previous slide

- The clause `single(X) :- human(X), not(married(X,Y)).`
- means $\forall X.\exists Y.\ human(X) \wedge not(married(X,Y)) \rightarrow single(X)$
- because X is is already bound by human(X) when the negation is entered.

- The clause `single(X) :- not(married(X,Y)) human(X).`
- is the same as `single(X) :- not(married(X1,Y)), human(X).`
- Both mean $\forall X.\exists X1.\exists Y\ human(X) \wedge not(married(X1,Y)) \rightarrow \ldots(X)$
- because the red X in the first clause is not bound when the negation is reached. So it is existentially quantified, whereas the blue X is universally quantified. Thus both are actually different variables since the same variable cannot be quantified differently in the same scope.

Eliminate accidentally equal names!

Last slide shown on 10.5.2010

Remember: Free variables in negated goals are existentially quantified.

- They do not "return bindings" outside of the scope of the negation.
- They are different from variables outside of the negation that accidentally have the same name.

```
nestedneg1(Y) :-                 % INST
    q(Y),                        % -
    not( ( p(X,Y),               % -+
           not( ( f(X,Z),        % +-
                  g(Z)           % +
                )
           ),
           q(X,Z)                  % -
         )
    ),
   q(X) .                        % -
```

⇒

```
nestedneg1(Y) :-                 % INST
    q(Y),                        % -
    not( ( p(X,Y),               % -+
           not( ( f(X,Z),        % +-
                  g(Z)           % +
                )
           ),
           q(X,Z1)               % -
         )
    ),
   q(X1) .                       % -
```

A Test

- Predict what this program does!

```
f(1,a).
f(2,b).
f(2,c).
f(4,c).

q(1).
q(2).
q(3).
```

```
negation(X) :-
  not(
      ( f(X,c),
        output(X),
        g(X)
       )
  ),
  q(X).
```

```
output(X) :-
  format('Found f(~a,c) ', [X]).

output(X) :-
  format('but no g(~a)~n', [X]).
```

- This is what it does (try it out):

```
?- negation(X).
Found f(2,c) but no g(2)
Found f(4,c) but no g(4)
X=1 ;
X=2 ;
X=3 ;
fail.
```

- Homework:

  If you don't understand the result reread the slides about negation (and eventually also those about backtracking if you do not understand why output/1 has two clauses).

Operational Semantics (cont.)

Can we prove truth or falsity of <u>every</u> goal?

→No, unfortunately!

Incompleteness of SLD-Resolution

- Provability
  - If a goal can be reduced to the empty subgoal then the goal is provable.
- Undecidability
  - There is no automated proof system that always answers yes if a goal is provable from the available clauses and answers no otherwise.
  - Prolog answers yes, no or does not terminate.

Incompleteness of SLD-Resolution

- The evaluation strategy of Prolog is incomplete.
  - Because of non-terminating derivations, Prolog sometimes only derives a subset of the logical consequences of a program.

- Example
  - `r`, p, and q are logical consequences of this program

```
p :- q.    % 1
q :- p.    % 2
p :- r.    % 3
r.         % 4
```

  - However, Prolog's evaluation strategy cannot derive them. It loops indefinitely:

```
?- p.
   |---     1st clause
   q
   |---     2nd clause
   p
  ...        etc.
```

Practical Implications

- Need to understand both semantics
  - The model-based (declarative) semantics is the “reference”
    - ⇨ We can apply bottom-up fixpoint iteration to understand the set of logical consequences of our programs
  - The proof-based (operational) semantics is the one Prolog uses to prove that a goal is among the logical consequences
    - ⇨ SLD-derivations can get stuck in infinite loops, missing some correct results
- Need to understand when these semantics differ
  - When do Prolog programs fail to terminate?
    - ⇨ Order of goals and clauses
    - ⇨ Recursion and “growing” function terms
    - ⇨ Recursion and loops in data
  - Which other problems could prevent the operational semantics match the declarative semantics?
    - ⇨ The cut!
    - ⇨ Non-logical features
    - ⇨ …

General Principles

- Try to match both semantics!
  - Your programs will be more easy to understand and maintain

- Write programs with the model-based semantics in mind!
  - If they do not behave as intended change them so that they do!

Practical Implications (Part 1)

Order of goals and clauses

Recursion and cyclic predicate definitions

Recursion and cycles in the data

Recursion and “growing” function terms

Order of Clauses in Predicate Definition

Ensure termination of recursive definitions by putting non-recursive clauses before recursive ones!

- Loops infinitely for ?-p:

```
p :- q.          % 1
p :- r.          % 2
q :- p.          % 3
r.               % 4
```

?-p succeeds (infinitely often):

```
p :- r.          % 1
p :- q.          % 2
q :- p.          % 3
r.               % 4
```

In spite of same Herbrand Model:

```
r.
p.
q.
```

- Traces:

```
?- p.

... nothing happens
...
```

```
?- p.

true ;
true ;
...
```

Order of Literals in Clause

Ensure termination of recursive definitions by putting non-recursive goals before recursive ones!

- Succeeds twice (and then loops infinitely) for ?-p(X):

```
p(0).
p(X) :- p(Y), a(X,Y).
a(1,0).
```

Succeeds exactly twice for ?-p(X):

```
p(0).
p(X) :- a(X,Y), p(Y).
a(1,0).
```

In spite of same Herbrand Model:

```
p(0).
p(1).
a(1,0).
```

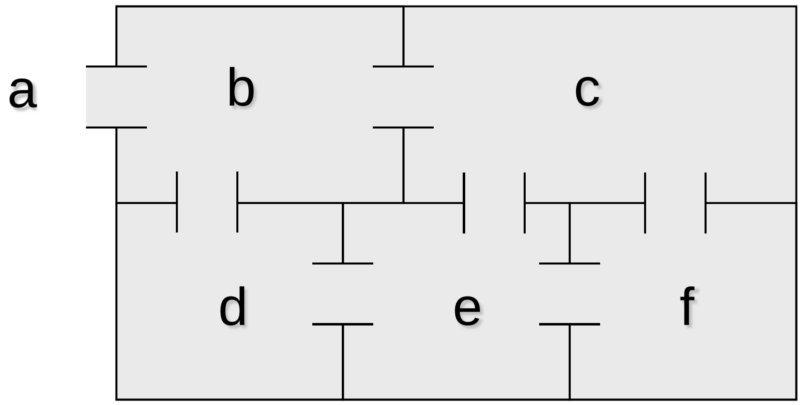
- Traces:

```
?- p(X).

X = 0 ;
X = 1 ;
ERROR: Out of local
stack
```

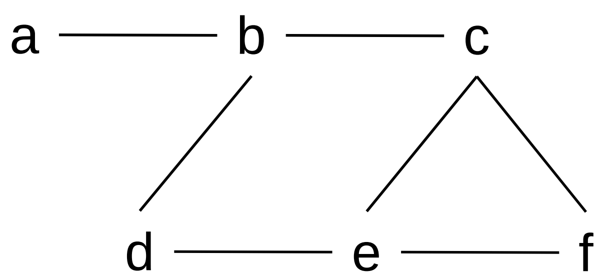
```
?- p(X).

X = 0 ;
X = 1 ;
false.
```

Cycles in the data (1)

- Given: The following floor plan



- … or its graph representation

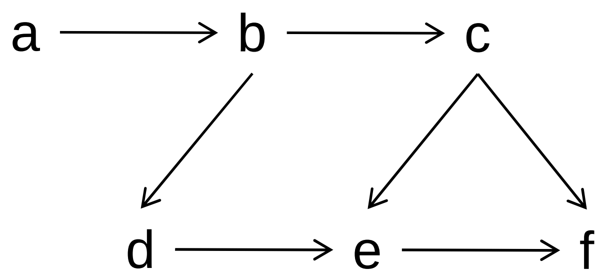


- A possible Prolog representation:

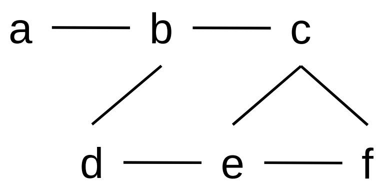
```
door(a,b).
door(b,c).
door(b,d).
door(c,e).
```

```
door(c,f).
door(d,e).
door(e,f).
```

- … for a directed graph



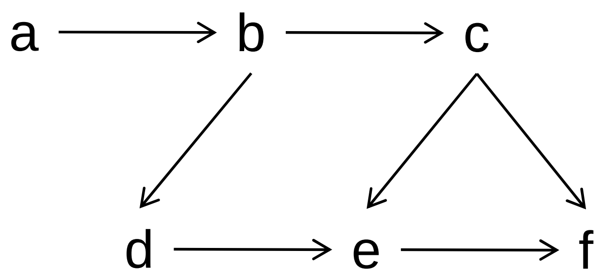
Cycles in the data (2)



- Question: How to represent symmetry of doors?

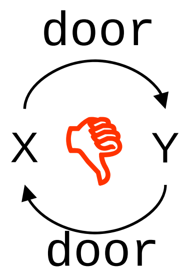
```
door(a,b).
door(b,c).
door(b,d).
door(c,e).
```

```
door(c,f).
door(d,e).
door(e,f).
```



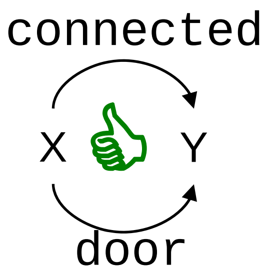
- 1. Attempt: Recursive definition

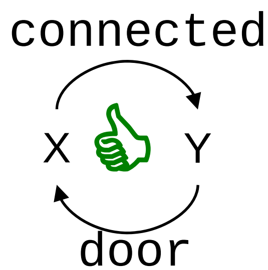
```
door(X, Y) :- door(Y, X).
```



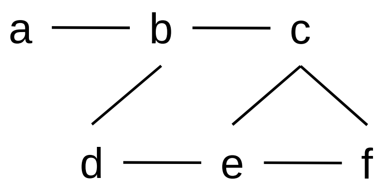
- 2. Attempt: Split definition into two predicates

```
connected(X, Y) :- door(X, Y).
connected(Y, X) :- door(X, Y).
```





Cycles in the data (3)



- Question: Is there a path from room X to room Y?

- 1. Attempt:

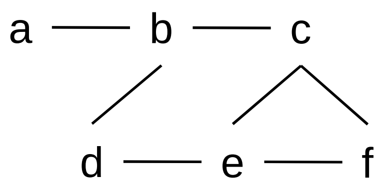
```
connected(X, Y) :- door(X, Y).
connected(X, Y) :- door(Y, X).
```

```
path(X, Y) :- connected(X, Y).
path(X, Y) :- connected(X, Z), path(Z, Y).
```

  - ➢ Declaratively OK, but will loop on cycles induced by definition of connected/2!
  - ➢ Derives the same facts infinitely often:

```
?- path(X,Y).
X = a, Y = b ;
...
X = a, Y = b ;
...
```

Cycles in the data (4)



- Question: Is there a path from room X to room Y?

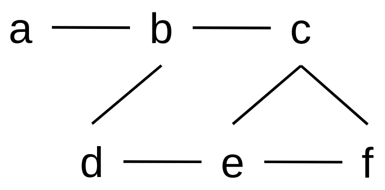
- 2. Attempt: Avoid looping through cycles in data by “remembering”

```
connected(X, Y) :- door(X, Y).
connected(X, Y) :- door(Y, X).
```

```
path(X, Y) :- path(X, Y, [X]).        // don‘t visit start node again

path(X, Y, Visited) :- connected(X, Y),
                       not( element(Y, Visited) ).
path(X, Y, Visited) :- connected(X, Z),
                       not( element(Z, Visited) ),
                       path(Z, Y, [Z|Visited]).
```

  - Remember each visited room in additional list parameter
  - Never visit the same node twice

Cycles in the data (5)



- Question: Is there a path from room X to room Y?

- 2. Attempt: Avoid looping through cycles in data by “remembering”

```
connected(X, Y) :- door(X, Y).
connected(X, Y) :- door(Y, X).

path(X, Y) :- assert(visited(X)),  // don‘t visit start node again
              path__(X, Y).

path__(X, Y) :- connected(X, Y),
                not( visited(Y) ).
path__(X, Y) :- connected(X, Z),
                not( visited(Z) ),
                assert( visited(Z) ),
                path(Z, Y).
```

Constant time check

‘assert’ adds a clause at run-time

- ◆ Remember visited rooms in dynamically created facts
- ◆ Never visit the same node twice

Keep in Mind!

Prolog predicates will loop infinitely if

- there is no matching non-recursive clause before a recursive one
- there is no non-recursive literal before a recursive invocation
- there are cycles in the data traversed by a recursive definition
  - either cycles in the data itself
  - or cycles introduced by rules
- there is divergent construction of terms
  - We'll see examples of this in the following section about lists!

Recursive Programming with Lists

List notation

Head and Tail

Recursive list processing

Lists in Prolog

Prolog lists may be heterogeneous: They may contain elements of different „types“

- Example: Homogeneous lists

`[1, 2, 3]`	List of integers
`['a', 'b', 'c']`	List of characters
`[ ]`	Empty list
`[[1,2], [ ], [5]]`	List of lists

- Example: Homogeneous only at the top level

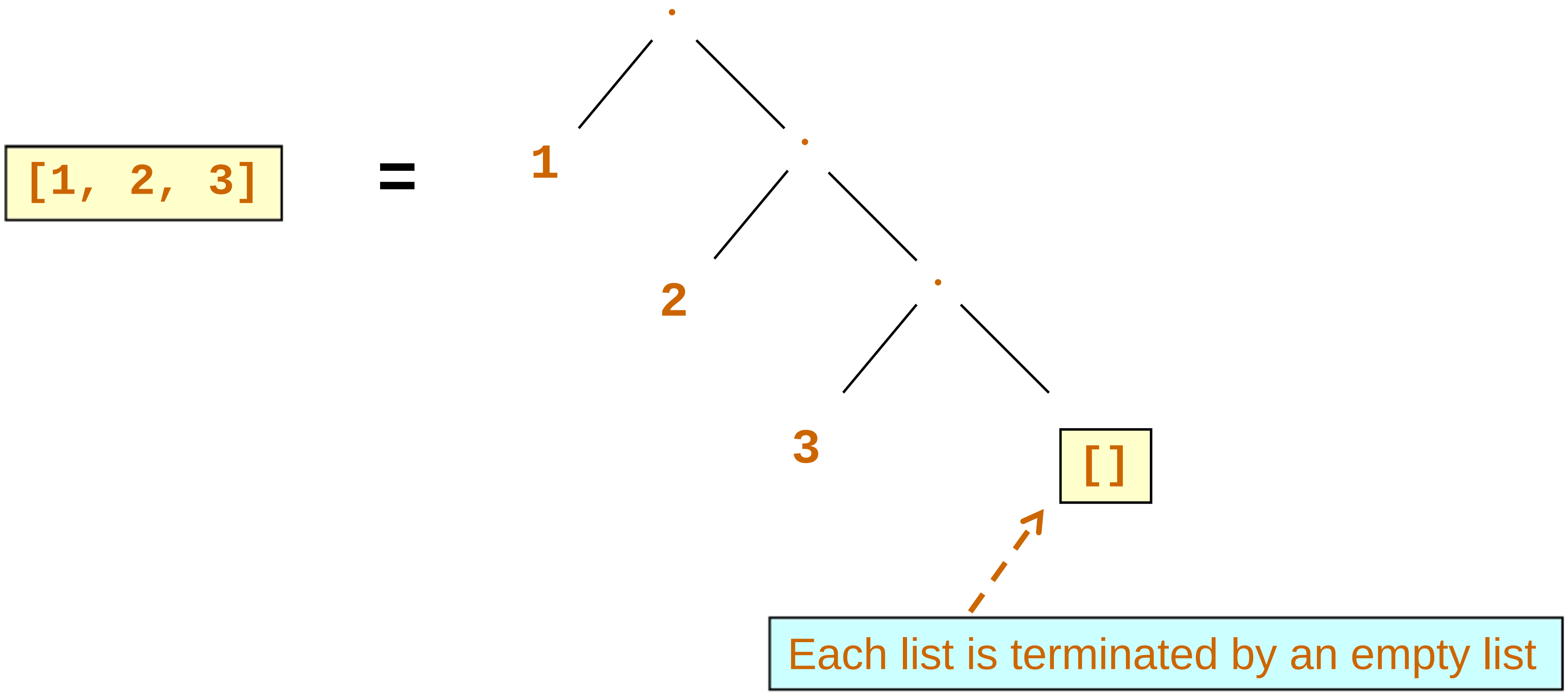
`[[1,2], [ ], ['a']]`	List of lists but the element types differ

- Example: Fully heterogeneous

```
[[1,2], 'a', 3]
```

List are Binary Trees Encoded as Terms (1)

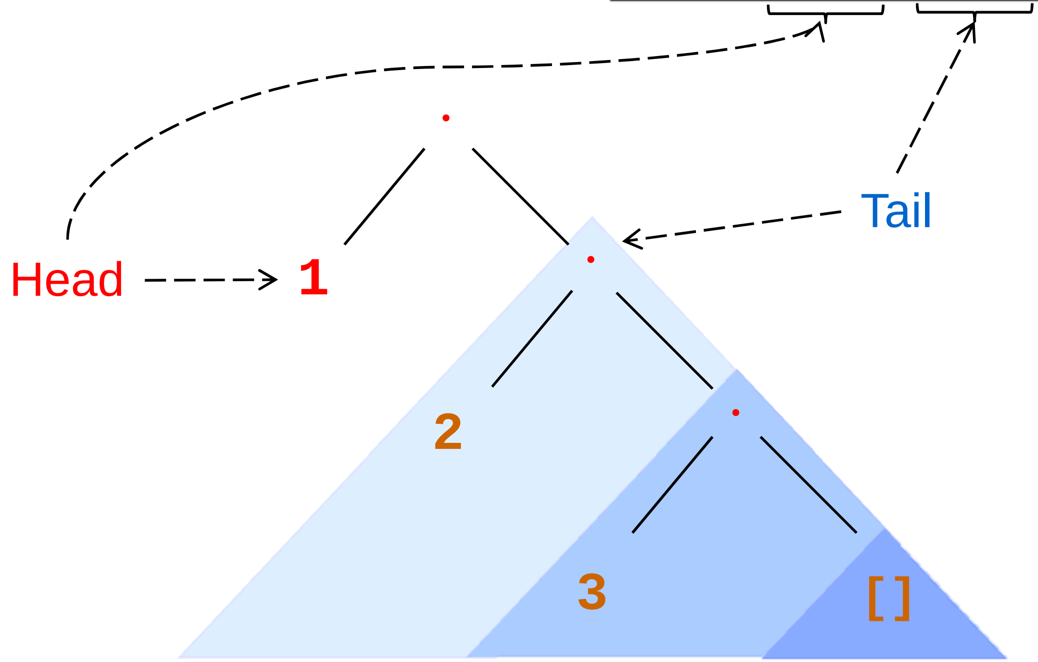
- Internally, lists are binary trees whose leaves are the lists elements:



List are Binary Trees Encoded as Terms (2)

- The functor '.' is the list constructor:

```
?- .(  1  ,[2,3]) = [1, 2, 3]
```



- The first element is the „head“ the second is the „tail“.
- The „tail“ is the list of all the other elements.

Accessing Head and Tail

- Notation [Head | Tail]

```
?- [1,2,3,4]=[H|T].              ------> H=1, T=[2,3,4]

?- [1,2,3,4]=[H1,H2|T].          ------> H1=1, H2=2, T=[3,4]

?- [1,2,3,4]=[_,_,H|_].          ------> H=3

?- [1,2,3,4]=[_,_,_,_|T].        ------> ???

?- []=[H|T].                     ------> ???

?- [1,2,3,4]=[_,_,_,_,_|T].      ------> ???
```

```
?- X = [Y,2,3,4], Y=1.           ------> X=[1,2,3,4], Y=1

?- T = [2,3,4], X=[1|T].         ------> ???
```

Length of a List

- Usually predefined:

```
/**
 * The predicate length(List, Int) suceeds iff Arg2 is
 * the number of elements in the list Arg1.
 */
 length([ ],0).
 length([X|Xs],N) :- length(Xs,N1), N is N1+1.
```

Head

Tail

- Tracing an invocation of 'length' with input on first argument:

```
?- length([1,2],N).
       Call  length([2],N1)
       Call  length([],N2)
       Exit  lenght([],0)
       Creep N2 = 0
       Creep N1 is N2+1
       Creep N is N1+1
   N=2
```

Length of a List

- Usually predefined:

```
/**
 * The predicate length(List, Int) suceeds iff Arg2 is
 * the number of elements in the list Arg1.
 */
 length([ ],0).
 length([X|Xs],N) :- length(Xs,N1), N is N1+1.
```

Head

Tail

- Tracing an invocation of 'length' without input on first argument:

```
?- length(X,N).
       Exit  lenght([],0)
   X=[], N=0 ;
        Call  length(X1,N1)
        Exit  lenght([],0)
        Creep N1 = 0
        Creep N is N1+1
   X=[G100], N=1 ;
       ... produces infinitely many results ...
```

Concatenating Lists

- Predicate definition

```
/**
 * The predicate append(L1, L2, L12) suceeds iff Arg3 is
 * the list that results from concatenating Arg2 and Arg1.
 */
append([], L, L).
append([H|T], L, [H|TL]) :- append(T, L, TL).
```

- Execution trace

```
?- append([a, b], [c, d], L).
    -> append([b], [c, d], TL1]).
           -> append([], [c, d], TL2).
            -> true
```

σ_1 = {L ← [a|TL1]}
σ_2 = {TL1 ← [b|TL2]}
σ_3 = {TL2 ← [c, d]}

The result is the composed substitution $\sigma_1\sigma_2\sigma_3 = \sigma_3(\sigma_2(\sigma_1))$ restricted to the bindings for L:

```
L = [a| [b| [c, d]]] = [a, b, c, d]
```

Testing List Membership (1)

- Predicate definition:

```
/**
 * The predicate member(Elem, List) suceeds iff Arg1 is an element
 * of the list Arg2 or unifiable with an element of Arg2.
 */
member(H, [H|_]).
member(E, [_|T]) :- member(E, T).
```

- Execution trace

```
?- member(2,[12, 2, 2, 3]).
     Call member(2, [2, 2, 3]).
     Exit member(2, [2, 2, 3]).
     Redo member(2, [2, 2, 3]).
     Call member(2, [2, 3]).
     Exit member(2, [2, 3]).
     Redo member(2, [2 ,3]).
     Call member(2, [3]).
     Call member(2, []).
     Fail
```

backtracking inititated by entering ;

backtracking inititated by entering ;

Testing List Membership (2)

- The member/2 predicate can be used in many differen “input modes”:

`?- member(a, [a,b,c,d]).`	Is a an element of [a,b,c,d]?
`?- member(X, [a,b,c,d]).`	Which elements does [a,b,c,d] have?
`?- member(a, Liste).`	Which lists contain the element a?
`?- member(X, Liste).`	Which lists contain the variable X?

Accessing List Elements

- First element of a list

```
first([X|_],X).
```

- Last element of a list:

```
last([X],X).
last([_|Xs],X) :- last(Xs,X).
```

- N-th element of a list:

```
nth(1,[X|_],X).
nth(N,[_|Xs],X) :- N1 is N-1, nth(N1,Xs,X).
```

Splitting Lists

```
/**
 * The split/4 predicate succeeds if
 *   Arg3 is a list that contains all elements of Arg2
 *   that are smaller than Arg1
 * and
 *   Arg4 is a list that contains all elements of Arg2
 *   that are bigger or equal to Arg1
 */

split(_,    [],    [],    []).
split(E, [H|T], [H|S],     B):- H <  E, split(E,T,S,B).
split(E, [H|T],    S , [H|B]):- H >= E, split(E,T,S,B).
```

Sorting Lists

- Naïve test for list membership via member/3 has linear complexity: O(n)
  - But if lists are sorted, membership testing is faster on the average
  - So sorting is very useful

- Quicksort-Algorithm in Prolog

```
/**
 * Quicksort/2 suceeds if the second argument is a sorted
 * version of the list in the first argument. Duplicates
 * are kept.
 */
quicksort([], []).
quicksort([Head|Tail], Sorted) :-
      split(Head,Tail,Smaller,Bigger),
      quicksort(Smaller,SmallerSorted),
      quicksort(Bigger,BiggerSorted),
      append(SmallerSorted,[Head|BiggerSorted], Sorted).
```

Doing Something with all Elements

- Sum of list elements:

```
sum([],0).
sum([H| T], S) :- sum(T, ST), S is ST+H.
```

- Normal Execution:

```
?- sum([12, 4], X).
     Call sum([4], ST])
     Call sum([],ST1)
     Exit sum([],0)
     Exit ST is 4+0=4
     Exit X is 12+ST=16
```

- Goals with illegal modes or type errors:

```
?- sum(X,3).
ERROR: is/2: Arguments are not sufficiently instantiated

?- sum(X,Y).
X = [],
Y = 0 ;
ERROR: is/2: Arguments are not sufficiently instantiated

?- sum([1,2,a],Res).
ERROR: is/2: Arithmetic: `a/0' is not a function
```

Relations versus Functions

Difference of relations and functions

How to document relations?

How to document predicates that have different “input modes”?

Relations versus Functions (1)

- In the functional programming language Haskell the following definition of the **isFatherOf** relation is illegal:

```
isFatherOf x | x==frank  = peter
isFatherOf x | x==peter  = paul
isFatherOf x | x==peter  = hans
           x | otherwise = dummy
```

✘

- In a functional language relations must be modeled as boolean functions:

```
isFatherOf x y| x==frank y==peter = True
isFatherOf x y| x==peter y==paul  = True
isFatherOf x y| x==peter y==hans  = True
           x y| otherwise         = False
```

✓

Relations versus Functions (2)

- Function application in Haskell must not contain any variables!
- Only the following "checks" are legal:

```
isFatherOf frank peter      ──→  True
isFatherOf kurt peter       ──→  False
```

- In Prolog each argument of a goal may be a variable!
- So each predicate can be used / queried in many different input modes:

```
?- isFatherOf(kurt,peter).  ──→  Yes

?- isFatherOf(kurt,X).      ──→  Yes
                                 X = paul;
                                 X = hans

?- isFatherOf(paul,Y).      ──→  No

?- isFatherOf(X,Y).         ──→  Yes
                                 X = frank, Y = peter;
                                 X = peter, Y= paul;
                                 X = peter, Y=hans;
                                 No
```

Relations versus Functions (3)

- Haskell is based on functions
  - Length of a list in Haskell

```
length([ ])   = 0
length(x:xs) = length(xs) + 1
```

- Prolog is based on relations
  - Length of a list in Prolog:

```
length([ ], 0).
length([X|Xs],N) :- length(Xs,M), N is M+1.
```

```
?- length([1,2,a],Length).
   Length = 3

?- length(List,3).
   List = [_G330, _G331, _G332]
```

List with 3 arbitrary (variable) elements

Documenting Predicates Properly

- Predicates are more general than functions
  - There is not one unique result but many, depending on the input

- So resist temptation to document predicates as if they were functions!
  - Don't write this:

```
/**
 * The predicate length(List, Int) returns in Arg2
 * the number of elements in the list Arg1.
 */
```

  - Better write this instead:

```
/**
 * The predicate length(List, Int) succeeds iff Arg2 is
 * the number of elements in the list Arg1.
 */
```

Documenting Invocation Modes

- Documenting the behaviour of a predicate thoroughly, including behaviour of special "invocation modes":
  - "–" means "always a free variable at invocation time"
  - "+" means "not a free variable at invocation time"
    - ⇨ Note: This is weaker than "ground at invocation time"
  - "?" means "don't care whether free or not at invocation time"

```
/**
 * length(+List, ?Int) is deterministic
 * length(-List, -Int) has infinite success set
 *
 * The predicate length(List, Int) succeeds iff Arg2 is
 * the number of elements in the list Arg1.
 */
length([ ],0).
length([X|Xs],N) :- length(Xs,N1), N is N1+1.
```

Operators

Operators are part of the syntax but the examples used here already use “unification”, which is explained in the next subsection. So you might want to fast forward to “Equality” / “Unification” if you do not understand something here.

Operators

Operators are just syntactic sugar for function terms:

- 1+3*4 is the infix notation for +(1,*(3,4))
- head :- body is the infix notation for ':-'(head,body)
- ?- goal is the prefix notation for '?-'(goal)

Operator are declared by calling the predicate

op(precedence, notation_and_associativity, operatorName)

'?-' has higher precedence than '+'

```
:- op(1200, fx, '?-').
:- op( 500, yfx, '+').
```

← prefix notation

← infix notation, left associative

- "f" indicates position of functor (→ prefix, infix, postfix)
- "x" indicates non-associative side
  - ⇨ argument with precedence strictly lower than the functor
- "y" indicates associative side
  - ⇨ argument with precedence equal or lower than the functor

Operator Associativity

In Java, the assignment operator is right-associative. That is, the statement "a = b = c;" is equivalent to "(a = (b = c));". It first assigns the value of c to b, then assigns the value of b to a.

Left associative operators

are applied in left-to-right order
1+2+3 = ((1+2)+3)

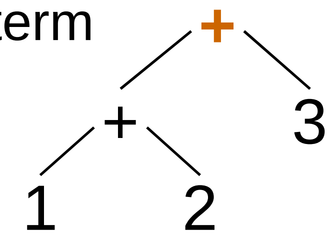
- Declaration

```
:- op(500, yfx, '+').
```

- Effect

```
?- T = 1+2+3, T = A+B.
T = 1+2+3,
A = 1+2,
B = 3.
```

- Structure of term



Right associative operators

are applied in right-to-left order
a,b,c = (a,(b,c))

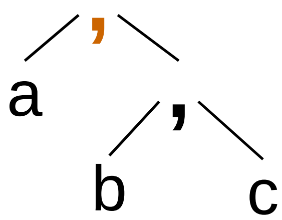
- Declaration

```
:- op(1000, xfy, ',').
```

- Effect

```
?- T = (a,b,c), T =(A,B).
T = (a,b,c),
A = a,
B = (b,c).
```

- Structure of term



Operator Associativity

Non-associative operators

must be explicitly bracketed

- Declaration

```
:- op( 700, xfx, '=').
:- op(1150,  fx, dynamic).
```

- Effect

```
?- A=B=C.
Syntax error:
Operator priority clash
```

```
?- A=(B=C).
A = (B=C).
```

Associative prefix operators

may be cascaded

- Declaration

```
:- op( 700,  fy, '+').
:- op(1150,  fy, '-').
```

- Effect

```
anything(_).
?- anything(+ - + 1).
true.
```

Three associative prefix operators!

```
anything(_).
?- anything(+-+ 1).
Syntax error:
Operator expected
```

One atom, not three operators!

Example from page 5 rewritten using infix operators

```
% Declare infix operators:
:- op(500,xfy,isFatherOf).
:- op(500,xfy,isMotherOf).
:- op(500,xfy,isGrandfatherOf).

% Declare predicates using the operator notation:
kurt  isFatherOf peter.
peter isFatherOf paul.
peter isFatherOf hans.

G isGrandfatherOf C :- G isFatherOf F, F isFatherOf C.
G isGrandfatherOf C :- G isFatherOf M, M isMotherOf C.

% Ask goals using the operator notation:
?- kurt isGrandfatherOf paul.
?- kurt isGrandfatherOf C.
?- isGrandfatherOf(G,paul).
?- isGrandfatherOf(G,paul), X isFatherOf G.
```

any combination of function term notation with operator notation is legal

Chapter Summary

- Prolog Syntax
  - Programs, clauses, literals
  - Terms, variables, constants
- Semantics: Basics
  - Translation to logic
- Operational / Proof-theoretic Semantics
  - Unification, SLD-Resolution
  - Incompleteness because of non-termination
  - Dealing with non-terminating programs:
    - Order of literals / clauses
    - shrinking terms
    - loop detection
- Declarative / Model-based Semantics
  - Herbrand Universe
  - Herbrand Interpretation
  - Herbrand Model
- Negation as Failure
  - Closed World Assumption
  - Existential Variables
- Disjunction
  - Equivalence to clauses
  - Variable renaming