10
Logic Programming and Prolog: paradigm, engine, first examples

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Magistrale in Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2021/2022

Outline

This week:

- The logic programming (and Prolog) paradigm
- Computational model (resolution + unification)
- Basic, idiomatic programming

Later:

- some more advanced programming with Prolog
- Java/Scala integration with Prolog, by tuProlog

Outline

Logic Programming

Literally: using mathematical logic for computer programming

- logic used as a declarative representation language, and as a theorem-prover (for problem-solving)

However it is comparable to imperative programming

- to achieve efficiency, some imperative mechanisms are often used to "control" program execution
- still, a declarative interpretation is possible which is higher-level (focus on what, not on how), and helps ensuring correctness
- actually, logic programming is quintessential declarative programming
  - modern functional languages borrow some techniques from logic programming, which is in a sense "more declarative"

Logic Programming mechanisms

Essential/distinctive elements

1. computation is about establishing in how many ways a goal can be solved (0,1, many solutions)
2. mechanisms of trial-and-error (backtracking) are used to search all solutions
3. a goal is a relation (either 0-ary, 1-ary, binary, ternary,..) over first-order terms
4. terms, i.e., arguments of goals, can be used as inputs and/or outputs
5. terms are untyped trees, possibly "incomplete" due to the use of logical variables
6. data and programs have essentially the same syntax, facilitating meta-programming

Prolog origins

History

- late 60's problem: of how to represent plans in primitive AI
  - Stanford school: declarative representation, leading to knowledge representation
  - MIT school: procedural representation, leading to functional paradigm
- Knowledge representation languages (Planner, QA-4, ...) evolved to Prolog (1972, A.Colmerauer)

Prolog

- started as theorem prover, now the reference for logic programming (LP)
- ISO Prolog (1995) is still the reference language and library

Prolog in SW Engineering

- very high hype initially, then mostly faded
- now back as the basis for "symbolic AI", not for general programming
- some aspects of Prolog impacted other areas: DBs, XML, pattern matching

Traditional applications

Areas

- Rapid prototyping of algorithms
- Rapid prototyping of DSLs
- Dynamically evolving structures
- Reasoning-like computation (planning)
- Rule-based computation
- Semantic and symbolic reasoning
- Agent-based programming languages

Domains

- Speech applications (NASA)
- Configuring IP Backbone networks (Ericsson)
- Logistics, Data Mining, Bioinformatics
- Robotics and autonomous systems
- Coordination models

Why Prolog in this course?

It covers the "remaining" paradigm: LP (OOP, FP, LP, ...)

- if/when you grasp it, it is fun!
- the "feeling" is completely different from mainstream programming of Java, C, C++, Scala
- programming as "searching into a space of solutions"

Practicing polyglotism (Java/Scala/Prolog)

- Java/Scala as the part handling more "in-the-large" aspects

→ s/w organisation, connection with the O.S. and libraries

- Prolog as the engine to handle certain data & algorithms

→ reasoning, space exploration features

A preview: permutations in Structured Programming

Permutations in idiomatic imperative programming (C/Java)

- at each call, the input is updated

```
static boolean nextperm(int[] a){
    int i,k;
    for (i=a.length-2; i>=0 && a[i]>a[i+1]; i--);
    if (i<0){
        return false;
    }
    for (k=a.length-1;a[i]>a[k];k--);
    swap(a,i,k);
    k=0;
    for (int j=i+1;j<(a.length+i)/2+1;j++){
        swap(a,j,a.length-k-1);
        k++;
    }
    return true;
}
```

A preview: permutations in LP

Permutations in idiomatic logic programming (Prolog)

- the goal is to seek any permutation of a list
- `member/3` relates a list with any element in it and the rest of the list
- `permutation/2` relates a list with any permutation of it
  - empty list is permutation of an empty list
  - given a list `L`, let `H` be any element of it, `T` the rest, and `TP` any permutation of `T`, than `L` has as permutation a list starting with `H` and having tail `TP`

```
member([H|T], H, T).
member([H|T], E, [H|T2]):- member(T, E, T2).

permutation([], []).
permutation(L, [H | TP]) :-
    member(L, H, T),
    permutation(T, TP).
```

A preview: permutations in modern FP

Permutations in idiomatic, modern FP (Scala)

- producing a stream of permutations
- solution highly-inspired by idiomatic LP
- `filter` plays the role of `member/3`
- note that LP somewhat inherently deals with "stream of results"

```
def member[A](l: List[A]): List[(A, List[A])] = l match
  case a :: Nil => List((a, Nil))
  case a :: t => (a, t) :: (for (a2, l2) <- member(t) yield (a2, a :: l2))

def permutations[A](l: List[A]): Iterable[List[A]] = l match
  case Nil => Iterable(List())
  case _ =>
    for
      (a, l2) <- member(l)
      p <- permutations(l2)
    yield a :: p
```

Conciseness of Prolog

Prolog allows you to code certain programs with much less (and more idiomatic) code than Java, C, C#, and Scala

Pros:

- if you master Prolog, you can directly and simply capture desired complex behaviour
- recall that learning a new paradigm means learning new computational patterns
- Prolog syntax and semantics are incredibly succinct

Cons:

- if you do not correctly understand it, difficulties arise
- there's no true school of clean coding for Prolog
- incrementality is key to control programs – debugging is very difficult

Prolog Technologies

Free implementations

- Gnuprolog: `http://www.gprolog.org`
- SWIProlog: `http://www.swi-prolog.org`

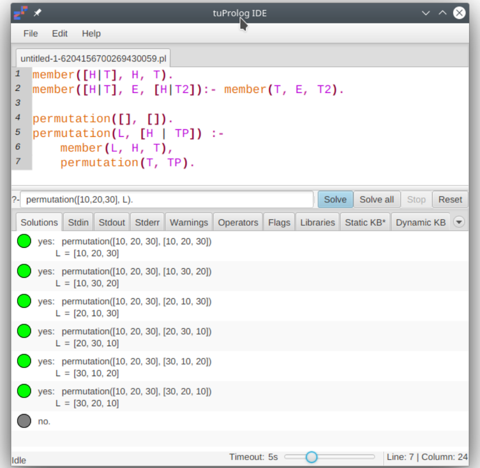
Commercial implementations

- SICStus: `https://sicstus.sics.se`

tuProlog: `https://apice.unibo.it/xwiki/bin/view/Tuprolog/`

- an academic open-source framework originally written/combined with Java, developed by "us" (i.e., Ricci + Omicini)
- now rewritten in Kotlin (i.e., Ciatto + Omicini)
- features library, API, IDE, web playground
- we will adopt it, for exercises and polyglotism

tuProlog IDE



Learning Prolog incrementally

This week: core Prolog

- Two mechanisms: resolution + unification
- Basic goal resolution examples
- Programming with lists

Next Week: full Prolog

- Additional non-core mechanisms
- Additional programming techniques

Later

- Lab with other exercises and tuProlog/Java/Scala integration

Prolog as a programming language

Resolution

- computing in Prolog means finding a solution to a list of "goals"
- start from first goal G, and find in the program rules whose head matches G, and for each try to solve the body B of the rule (again a list of goals, possibly empty)
- since many rules can match, at each step we have a choice, hence we intrinsically have to explore alternatives via "backtracking", and possibly get many results (0, 1, 2, ..., or even infinite ones)

Unification

- a goal expresses a (mathematical) relation (0-ary, 1-ary, 2-ary, ...) between first-order terms
- terms are the "data values" processed by Prolog: basically, untyped trees with atomic values or logic variables in leaves
- match between terms is done by the "unification algorithm", and gives a substitution that is incrementally refined during solution exploration
- each result of computation is actually a substitution

Outline

Resolution (without matching): technical details

Syntax of resolution system: a grammar abstracting Goal

$\texttt{Resolvent} ::= \texttt{Goal}_1, \ldots, \texttt{Goal}_n \quad (n \geq 0)$
$\texttt{Clause} ::= \texttt{Fact} \mid \texttt{Rule}$
$\texttt{Fact} ::= \texttt{Goal}$
$\texttt{Rule} ::= \texttt{Goal :- Resolvent}$
$\texttt{Program} ::= \texttt{Clause}_1 \ldots \texttt{Clause}_k \quad (k > 0)$
will sometime assume a fact is a rule with empty resolvent...

Semantics of resolution system: transition relation

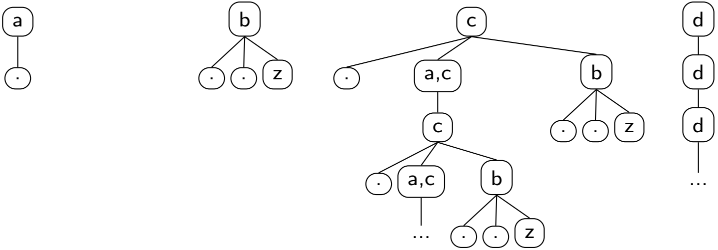
- Start with an initial "input" resolvent $\texttt{R}_0$
- A valid computation step is a transition $\texttt{R} \to \texttt{R}'$, defined as follows
- If clause (fact/rule) $\texttt{G}' \texttt{ :- } \texttt{G}'_1, \ldots, \texttt{G}'_m$ is defined in Program, and $\texttt{G}' = \texttt{G}_1$, then:
  $\texttt{G}_1, \texttt{G}_2, \ldots, \texttt{G}_n \to \texttt{G}'_1, \ldots, \texttt{G}'_m, \texttt{G}_2, \ldots, \texttt{G}$

Resolution tree

- For each resolvent, its first goal can match the head of many clauses (always considered from top to bottom), hence several "child" resolvents could be generated
- This naturally induces a (possibly infinite) tree of resolvents, to be read from left to right
- A resolution is considered successful/complete if there is a leaf with empty resolvent
- The process of going-up the tree to find a (new) solution is called "backtracking"

Example program, and resolution trees

```
a.             % a clause with empty body is called a "fact"
b.             % multiple copies of rules/facts can occur
b.
b :- z.        % b is the rule "head", z is the "body"
c.             % different clauses can have same "head"
c :- a, c.     % a sort of recursive rule
c :- b.
d :- d.        % .. recall the order of clauses is relevant
```



Outcomes of resolution

Predicative

- has it one solution (reaching a . leaf)? – don't care after first solution

Stream-like

- how many solutions?

Loop-aware

- will computation terminate?

Output-oriented

- what is the output of a solution? is it related to the sequence of resolvents? what is the order of results?

Inference/knowledge with resolutions

Facts

- though they are atomic, they can be used to state knowledge we take as true, e.g., an axiom
- similarly to propositional symbols in propositional logic

Rule

- by a resolvent, we check composition of goals as sort of higher-level knowledge
- a rule is a way of giving such a composition a name (head), and a definition (body)
- essentially: name-based abstraction, with key possibility of recursion

```
father_abraham_isaac.
father_terach_abraham.
grandfather_terach_isaac :- father_abraham_isaac,
    father_terach_abraham.
```

Shortcomings

Empowering resolution

- goals might have a structure, not just be atomic symbols
- goals seem to express a relationship between "elements"
- might want to express grandfather relation in the general case
- might want to have an explicit notion of "result" (who is abraham's father?)
- need to express computations in a Turing-complete way

Roadmap

- giving a concrete, structured syntax to goals
- providing an advanced mechanism of goal-rule matching
- adding information to the status of computation beyond mere resolvents

$\Rightarrow$ will extend resolution analogously to the transition from "proposition logic" to "first-order logic"

Outline

Empowering resolution framework

Ingredients

- goals are 0-ary, 1-ary, 2-ary, . . . , relations between terms
- terms are the "values" of the Prolog language, and are (finite) trees
- terms can have in leaves logic variables
- goal matching is done by so-called "unification", essentially defined as "a substitution of variable to terms making two goals identical"
- computation evolves a pair of a resolvent + substitution "grown so far"
- substitution is considered to be the "result" of computation

Prolog terms

Goals/terms: Prolog syntax

`Goal` ::= `Predicate` | `Predicate`(`Term`$_1$, …, `Term`$_n$)
`Term` ::= `Variable` | `Number` | `Functor` | `Functor`(`Term`$_1$, …, `Term`$_n$)

- `Predicates` and `Functors` names are literals starting with lower-case
- `Predicates` and `Functors` are said to have arity 0, 1, 2,… n
- `Variables` are literals starting with upper-case
- Note that goals have same syntax of terms, not vice-versa
- A term with no variables in it is called *ground*

```
father(abraham, isaac).
father(terach, abraham).
grandfather(GF, GS) :- father(GF, F), father(F, GS).

element(H, cons(H, T)).
element(E, cons(H, T)) :- element(E, T).

odd(1).           % 1 is odd
odd(3).           % 3 is odd
sum(2, 3, 5).     % 2,3,5 are in the sum relation
```

Interpretations of Prolog programs

Logic interpretation: the classical one

- a program as a "logic theory", computing as "proving a goal is a theorem under that theory"
- e.g.: to prove that `GF` is grandfather of `GS`, first prove that...

Relational interpretation: the idiomatic one – shall use this

- a program as a set of "predicates over terms", computing as "querying for a relation"
- e.g.: 10 is in `element` relation with `cons(10, nil)`

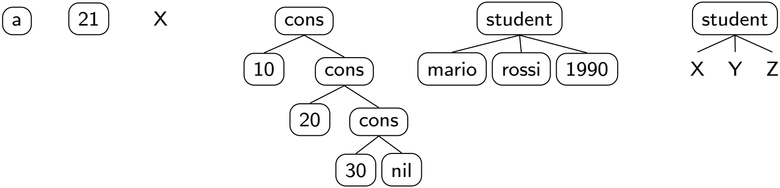
Procedural interpretation: the unproper one

- a program as a "set of procedures", computing as "calling predicates"
- call `element(10, cons(20, nil))` causes call `element(10, nil)`

Terms as trees

Example terms

- `a`: an atom
- `21`: a number
- `X`: a variable
- `cons(10,cons(20,cons(30,nil)))`: a compound (ground) term
- `student(mario, rossi, 1990)`: a compound (ground) term
- `student(X, Y, Z)`: a compound (non-ground) term



Terms(/Goals) Substitution

Definition

- a substitution $\theta = \{\mathtt{X}_1/\mathtt{T}_1, \ldots, \mathtt{X}_n/\mathtt{T}_n\}$ maps variables to terms
- terms $\mathtt{T}_j$ should contain no variable $\mathtt{X}_k$
- e.g. $\{\}$, $\{\mathtt{Y}/\mathtt{10}\}$, $\{\mathtt{X}/\mathtt{a(1,Z)}, \mathtt{Y}/\mathtt{10}, \mathtt{W}/\mathtt{Z}\}$
- e.g. $\{\mathtt{X}/\mathtt{a(1,Z)}, \mathtt{Y}/\mathtt{10}, \mathtt{Z}/\mathtt{W}\}$ is invalid, should rewrite: $\{\mathtt{X}/\mathtt{a(1,Z)}, \mathtt{Y}/\mathtt{10}, \mathtt{W}/\mathtt{Z}\}$

Related concepts

- application to terms:
  - `p(X,a)`$\{\mathtt{X}/\mathtt{10}\}$ is `p(10,a)`
  - `p(X,a)`$\{\mathtt{Y}/\mathtt{X}\}$ is `p(X,a)` (equivalent to `p(Y,a)` "under that substitution")
- equivalence of substitutions: $\{\mathtt{X}/\mathtt{Y}, \mathtt{Z}/\mathtt{10}\} \equiv \{\mathtt{Y}/\mathtt{X}, \mathtt{Z}/\mathtt{10}\}$
- generality: $\{\mathtt{X}/\mathtt{10}\}$ more general than $\{\mathtt{X}/\mathtt{10}, \mathtt{Y}/\mathtt{2}\}$
- composition of substitutions:
  - $\{\mathtt{X}/\mathtt{10}\}\{\mathtt{Y}/\mathtt{X}\} \equiv \{\mathtt{X}/\mathtt{10}, \mathtt{Y}/\mathtt{10}\}$
  - $\{\mathtt{X}/\mathtt{10}\}\{\mathtt{X}/\mathtt{5}\}$ impossible
- term/goal instance: `p(10, b)` is an instance of `p(X, b)`
- term cloning: *clone*(`p(10,X,X,Y)`) = `p(10,X2,X2,Y2)`, with (`X2`, `Y2` fresh)

Most general unifier

Definition

- *mgu*$(\mathtt{T}_1, \mathtt{T}_2)$ is any substitution θ such that $\theta\mathtt{T}_1 = \theta\mathtt{T}_2$, and such that the same does not hold for a substitution more general that θ
- an *mgu* might not exist, if it exists there might be equivalent results

Examples

- *mgu*$(\mathtt{a(1,2)}, \mathtt{a(X,Y)}) = \{\mathtt{X}/1, \mathtt{Y}/2\}$
- *mgu*$(\mathtt{a(1,2)}, \mathtt{a(X,X)}) = \perp$
- *mgu*$(\mathtt{a(1,b(X))}, \mathtt{a(Y,b(Z))}) = \{\mathtt{X}/\mathtt{Z}, \mathtt{Y}/1\}$
- *mgu*$(\mathtt{a(X,Y,A,b(W))}, \mathtt{a(Z,Z,B,b(1))}) = \{\mathtt{X}/\mathtt{Z}, \mathtt{Y}/\mathtt{Z}, \mathtt{W}/1, \mathtt{A}/\mathtt{B}\}$

What is unification

- a symmetrical pattern matching mechanism
- binding variables to non-variable terms, and some other variables into groups
- e.g. in last case above: `W/1`, and groups (`X, Y, Z`) and (`A,B`)
- MGU can be simply tested in Prolog: `?- p(X,1) = p(2,Y).`

Unification algorithm (by Martelli, Montanari, 1982)

$G \cup \{t \doteq t\} \Rightarrow G$		**delete**
$G \cup \{f(s_0, \ldots, s_k) \doteq f(t_0, \ldots, t_k)\} \Rightarrow G \cup \{s_0 \doteq t_0, \ldots, s_k \doteq t_k\}$		**decompose**
$G \cup \{f(s_0, \ldots, s_k) \doteq g(t_0, \ldots, t_m)\} \Rightarrow \bot$	if $f \neq g$ or $k \neq m$	**conflict**
$G \cup \{f(s_0, \ldots, s_k) \doteq x\} \Rightarrow G \cup \{x \doteq f(s_0, \ldots, s_k)\}$		**swap**
$G \cup \{x \doteq t\} \Rightarrow G\{x \mapsto t\} \cup \{x \doteq t\}$	if $x \notin \mathrm{vars}(t)$ and $x \in \mathrm{vars}(G)$	**eliminate**[note 8]
$G \cup \{x \doteq f(s_0, \ldots, s_k)\} \Rightarrow \bot$	if $x \in \mathrm{vars}(f(s_0, \ldots, s_k))$	**check**

Outline

Resolution with unification: Prolog semantics

Semantics of Prolog resolution system: transition relation

- Use a tree of pairs of resolvent + substitution as computation state
- Start with an initial "input" configuration $C = \langle \mathtt{R}_0 : \{\} \rangle$
- A valid computation step is a transition $\mathtt{C} \rightarrow \mathtt{C}'$, defined as follows
- If $\mathtt{G}'$ `:-` $\mathtt{G}'_1, \ldots, \mathtt{G}'_m$ is the clone of a clause in `Program`, and $\theta' = mgu(\mathtt{G}', \mathtt{G}_1)$ exists, then:
  $\langle \mathtt{G}_1, \mathtt{G}_2, \ldots, \mathtt{G}_n : \theta \rangle \rightarrow \langle (\mathtt{G}'_1, \ldots, \mathtt{G}'_m, \mathtt{G}_2, \ldots \mathtt{G}_n)\theta' : \theta\theta' \rangle$
- For simplicity, of a θ in a configuration only the part that mentions variables in the resolvent and in the input resolvent is needed

Resolution tree

- The transition relation induces as usual a possibly infinite tree
- A "solution" is the substitution θ^s we have in a leaf with empty resolvent

Resolution with unification: example 1

Resolution step

- If $\mathtt{G}'$ `:-` $\mathtt{G}'_1, \ldots, \mathtt{G}'_m$ is the clone of a clause in `Program`, and $\theta' = mgu(\mathtt{G}', \mathtt{G}_1)$ exists, then: $\langle \mathtt{G}_1, \mathtt{G}_2, \ldots, \mathtt{G}_n : \theta \rangle \rightarrow \langle (\mathtt{G}'_1, \ldots, \mathtt{G}'_m, \mathtt{G}_2, \ldots \mathtt{G}_n)\theta' : \theta\theta' \rangle$
- For simplicity, of a θ in a configuration only the part that mentions variables in the resolvent and in the input resolvent is needed

```
father(abraham, isaac).
father(terach, abraham).
grandfather(GF, GS) :- father(GF, F), father(F, GS).
```

```
C1:  grandfather(terach, X), father(X, Y) : {}
-->
C2:  father(terach, F′), father(F′, X), father(X,Y) : {}
```

Explanation

- cloned rule: `grandfather(GF', GS') :- father(GF', F'), father(F', GS').`
- $\theta' = \{GF'/terach, GS'/X\}$
- new resolvent: `father(GF', F'), father(F', GS'), father(X, Y)`
- applying θ': `father(terach, F'), father(F', X), father(X,Y)`
- no part of θ' must be recalled for subsequent steps

Resolution with unification: example 2

Resolution step

- If $\mathtt{G}' \text{ :- } \mathtt{G}'_1, \ldots, \mathtt{G}'_m$ is the clone of a clause in `Program`, and $\theta' = mgu(\mathtt{G}', \mathtt{G}_1)$ exists, then: $\langle \mathtt{G}_1, \mathtt{G}_2, \ldots, \mathtt{G}_n : \theta \rangle \rightarrow \langle (\mathtt{G}'_1, \ldots, \mathtt{G}'_m, \mathtt{G}_2, \ldots \mathtt{G}_n)\theta' : \theta\theta' \rangle$
- For simplicity, of a θ in a configuration only the part that mentions variables in the resolvent and in the input resolvent is needed

```
sum(X, s(Y), s(Z)) :- sum(X, Y, Z).
```

```
C1:   sum(s(zero), s(zero), N).
-->
C2:   sum(s(zero), zero, Z′)  :  N/s(Z′)
```

Explanation

- cloned rule: `sum(X', s(Y'), s(Z')) :- .`
- $\theta' = \{X'/s(zero), Y'/zero, N/s(Z')\}$
- new resolvent: `sum(X', Y', Z')`
- applying θ': `sum(s(zero), zero, Z')`
- the part of θ' that must be recalled is `N/s(Z′)`

Logic programming patterns

On Prolog syntax and semantics

- it is essentially all there...
- will now show some useful programming(/design) patterns
- will also show few additional elements (operators/syntax/library)
- will later extend a bit the whole framework

Patterns

- querying facts
- existential queries
- querying universal facts
- working with records
- wilcard variables
- deriving knowledge with rules
- programming math
- programming with ADTs and lists
- list syntax
- full relationality
- math operators

Outline

A Prolog program as a DB

Querying facts

- A Prolog program with only ground facts can be seen as a DB
- All facts of a certain predicate (name + arity) as a table
- A single, ground goal can be used to query the DB for a "tuple"

Existential queries

- if the goal has variables, you are really asking if there exists an instantiation of variables (a substitution) equating your goal with one or more facts
- this is like searching multiple tuples with a query
- composing goals make Prolog find combinations

Universal facts

- variables in facts are quantified universally instead, hence a fact with variables is like an infinite set of facts

Working with records

- a fact can connect terms which are structured, e.g., in the form of records

Use of wildcard variable

- to avoid mentioning a variable once in a clause

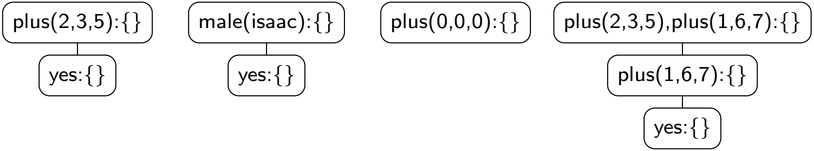
Querying facts

Program

```
male(isaac).
plus(2,3,5).
plus(1,6,7).
plus(0,0,5).
```

Goals

```
?- plus(2,3,5). Yes:{}
?- male(isaac). Yes:{}
?- plus(0,0,0). No
?- plus(2,3,5),plus(1,6,7). Yes:{}
```



Notes

- “yes” is used to mean empty resolvent (sometimes avoided at all)
- will sometimes avoid “: {}” at all

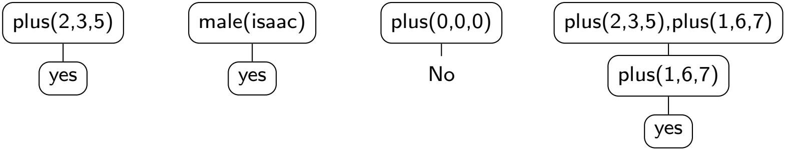
Querying facts: notation for trees

Program

```
male(isaac).
plus(2,3,5).
plus(1,6,7).
plus(0,0,5).
```

Goals

```
?- plus(2,3,5).  Yes:{}
?- male(isaac).  Yes:{}
?- plus(0,0,0).  No
?- plus(2,3,5),plus(1,6,7).  Yes:{}
```



Notes

- "yes" is used to mean empty resolvent (sometimes avoided at all)
- when there's no solution will add "no" label
- will sometimes avoid ": {}" at all

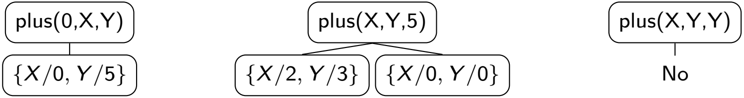
Existential queries

Program

```
plus(2,3,5).
plus(1,6,7).
plus(0,0,5).
```

Goals

```
?- plus(0,X,Y).
     -> {X/0,Y/3}
?- plus(X,Y,5).
     -> {X/2,Y/5}; {X/0,Y/0}
?- plus(X,Y,Y).
     -> No
```



Notes

- recall that we have one branch for rule/fact unifying with the goal
- the unifier is taken as solution

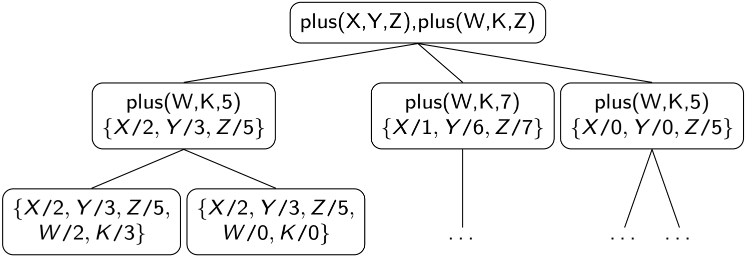
Existential queries and inherent exploration

Program

```
plus(2,3,5).
plus(1,6,7).
plus(0,0,5).
```

Goals

```
?- plus(X,Y,Z),plus(W,K,Z).
     -> {X/2,Y/3,Z/5,W/2,K/3};
     -> {X/2,Y/3,Z/5,W/0,K/0};
     ...
```



Notes

- multiple goals to be solved inherently explore combinations
- a bit like multiple clauses in for-comprehension

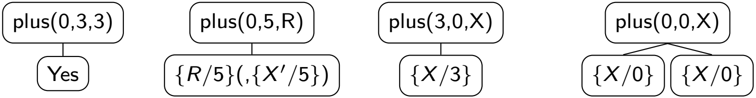
Universal facts

Program

```
plus(0,X,X).
plus(X,0,X).
```

Goals

```
?- plus(0,3,3). -> Yes
?- plus(0,5,R). -> {R/5}
?- plus(3,0,X). -> {X/3} % no clash!
?- plus(0,0,X). -> {X/0}; {X/0}
```



Notes

- the unification with a universal fact creates a substitution mentioning variables of a cloned clause (see the (,{X'/5}))
- however, they are not showed as final result

Working with records

Program

```
manager(person(john,smith,1283)).
clerk(person(jim,white,3475)).
clerk(person(george,red,8765)).
chief(person(john,smith,1283), person(george,red,8765)).
```

Goal

```
?-chief(X,person(Y,Z,8765)).
yes.
X/person(john,smith,1283) Y/george Z/red
chief(person(john,smith,1283), person(george,red,8765)).
```

Notes

- usage of facts as relations (1-ary, 2-ary) between records
- recall `manager` is a predicate, `person` a functor
- note Prolog outputs minimal substitution and instantiated goal.

Working with records

Program

```
manager(person(john,smith,1283)).
clerk(person(jim,white,3475)).
clerk(person(george,red,8765)).
chief(person(john,smith,1283), person(george,red,8765)).
```

Goal

```
?-chief(X,person(Y,Z,8765)).
yes.
X/person(john,smith,1283) Y/george Z/red
chief(person(john,smith,1283), person(george,red,8765)).
```

Notes

- usage of facts as relations (1-ary, 2-ary) between records
- recall `manager` is a predicate, `person` a functor
- note Prolog outputs minimal substitution and instantiated goal.

Wildcard variable

Program

```
p(_,1).
p(1,2).
q(_,a(1,_)).
r(X,a(1,X)).
```

Goals

```
?- p(X,1). Yes
?- p(1,Y). {Y/1}; {Y/2}
?- p(1,_). Yes; Yes
?- q(1,a(1,2)). Yes
?- r(1,a(1,2)). No
```

Semantics

- in programs: a variable used once in the clause
- in goals: a variable we do not need to occur in solutions

Notes

- it is good Prolog practice to never use singleton variables in a clause, but wildcards instead
- this is used to avoid copy-and-paste errors in variable names

A recap example: modelling propositional logic

Goals

```
b_not(b_true, b_false).
b_not(b_false, b_true).
b_and(B, b_true, B).
b_and(B, b_false, b_false).
b_or(B, b_false, B).
b_or(B, b_true, b_true).
```

Goals

```
?- b_not(b_false, B).
    -> {B/b_true}
?- b_and(b_false, b_true, B).
    -> {B/b_false}
?- b_or(b_false, b_true, B).
    -> {B/b_true}
?- b_or(b_true, B2, B). -> ???
?- b_or(B1, B2, B). -> ???
```

Modelling idea: this is a simple ADT!

- we have two booleans, modelled as atomic terms `b_true, b_false` (functors with arity 0)
- a unary function is modelled as a binary `p(I,O)` predicate
- a binary function is modelled as a ternary `p(I1,I2,O)` predicate
- use universal facts to somewhat address DRY

⇒ can we derive a general approach for ADTs?

Outline

Programming ADTs

An algebraic data type

- Construction + algorithms
- Construction defined by deciding functors (name + arity) for terms
- Algorithms by predicates (with function to relation mapping)

Example: programmed booleans

- improvement with rules

Example: booleans with built-in engine

- functions yielding a boolean as a predicate

Example: Peano arithmetics

- showcasing recursive rules, and recursion

Example: lists as recursive ADTs

- a playground for exercises

Algebraic data types in Prolog

Recall features of an ADT

- A name of the type
- A set of values, expressed by "sum" of "products"
- A set of I/O pure functions: often using recursion

ADT in Prolog

We have no types, hence no enforcement

- Name: can only impact names of values/functions
- Values:
  - in terms of a set of functors, e.g.: (`cons/2` and `nil/0`)
  - use an extra-argument (typically the last) as output
- Functions:
  - hopefully using matching on functors
  - $f : I_1, I_2, \ldots, I_m \mapsto O$ becomes `p(I1,I2,...,In,O)`
  - define functions (in the body) as composition of (goals) relations
  - `h(X)=f(g(X))` becomes `h(X,Y) :- g(X,Z), f(Z,Y).`
  - functions returning booleans might have special treatment

Example 1: Improving booleans with rules

```
b_not(b_true, b_false).
b_not(b_false, b_true).
b_and(B, b_true, B).
b_and(B, b_false, b_false).
b_or(B1, B2, B) :-           % (a and b) = !(!a or !b)
    b_not(B1, NB1), b_not(B2, NB2),  b_and(NB1, NB2, NB), b_not(NB, B).
b_implies(B1, B2, B) :-      % a --> b = !a or b
    b_not(B1, NB1), b_or(NB1, B2, B).
```

b_implies(b_true, b_false,O)

b_not(b_true, NB1'), b_or(NB1', b_false, O)

b_or(b_false, b_false, O)

b_not(b_false, NB1'),b_not(b_false, NB2'),b_and(NB1', NB2', NB'),b_not(NB', O)

b_not(b_false, NB2'),b_and(b_true, NB2', NB'),b_not(NB', O)

b_and(b_true, b_true, NB'),b_not(NB', O)

b_not(b_true, O)

$\{O/b\_false\}$

Example 2: naturals by “Peano numbers”

Encoding natural numbers

- traditionally, the next step after booleans while studying a language expressiveness
- an easy encoding is the unary, Peano one: Z,SZ,SSZ,SSSZ,SSSSZ,...
- hence we would have, e.g.: SSZ + SSSZ = SSSSSZ
- Prolog will actually have an ad-hoc management of numbers (and booleans)

Ideas

- use functors `s/1` and `zero/0`, e.g.: `s(s(s(zero)))`
- `succ` function is easily obtained by `s/1` construction
- `sum` function is obtained recursively from `succ` or `s/1`
  - $a + 0 = a$, $a + s(b) = s(a + b)$
- `mul` function is obtained recursively from `sum`
  - $a * 0 = 0$, $a * s(b) = a + (a * b)$
- e.g., we could be able to implement `factorial`
- output is last argument as usual

Example 2: solution

```
succ(X, s(X)).
sum(X, zero, X).
sum(X, s(Y), s(Z)) :- sum(X, Y, Z).
mul(X, zero, zero).
mul(X, s(Y), Z) :- mul(X, Y, W), sum(W, X, Z).
dec(s(X), X).
factorial(zero, s(zero)).
factorial(s(X), Y):-factorial(X, Z), mul(s(X), Z, Y).
```

How to read the specification in the relational interpretation

- the successor of `X` is `s(X)`
- the sum of `X` and `zero` is `X`
- the sum of `X` and successor of `Y` is the successor of `Z` provided the sum of `X` and `Y` is `Z`
- etcetera...

Example 2: resolution

```
succ(X, s(X)).
sum(X, zero, X).
sum(X, s(Y), s(Z)) :- sum(X, Y, Z).
mul(X, zero, zero).
mul(X, s(Y), Z) :- mul(X, Y, W), sum(W, X, Z).
dec(s(X), X).
factorial(zero, s(zero)).
factorial(s(X), Y):-factorial(X, Z), mul(s(X), Z, Y).
```

```
?- succ(s(s(zero)), N). -> N/s(s(s(zero)))
?- sum(s(s(s(zero))), s(s(zero)), N). -> N/s(s(s(s(s(zero)))))
?- mul(s(s(zero)), s(s(zero)), N). -> N/s(s(s(s(zero))))
?- dec(s(s(zero)), N). -> N/s(zero)
?- dec(zero), N). -> No
?- sum(N, M, s(s(s(zero)))). -> ???
```

sum(s(s(s(zero))), s(s(zero)), N)

sum(s(s(s(zero))), s(zero), Z') : $\{N/s(Z')\}$

sum(s(s(s(zero))), zero, Z'') : $\{N/s(Z'), Z'/s(Z'')\}\equiv\{N/s(s(Z''))\}$

$\{N/s(Z'), Z'/s(Z''), Z''/s(s(s(zero)))\}\equiv\{N/s(s(s(s(s(zero)))))\}$

Example 2: compare to modern FP solution

```
succ(X, s(X)).
sum(X, zero, X).
sum(X, s(Y), s(Z)) :- sum(X, Y, Z).
mul(X, zero, zero).
mul(X, s(Y), Z) :- mul(X, Y, W), sum(W, X, Z).
dec(s(X), X).
factorial(zero, s(zero)).
factorial(s(X), Y):-factorial(X, Z), mul(s(X), Z, Y).
```

```
enum Nat:
  case Zero
  case S(n: Nat)

object Nat:
  def succ(n: Nat): Nat = S(n)
  def sum(n1: Nat, n2: Nat): Nat = n2 match
    case Zero => n1
    case S(n) => S(sum(n1, n))
  def mul(n1: Nat, n2: Nat): Nat = n2 match
    case  Zero => Zero
    case S(n) => sum(n1, mul(n1, n))

@main def mainPeano() =
  import Nat.*
  val one = S(Zero)
  val two = S(S(Zero))
  val three = S(S(S(Zero)))
  println(succ(one))
  println(sum(two, three))
  println(mul(two, three))
```

Dealing with functions returning a boolean

Typical Prolog approach

- Not to have an additional argument being `b_true` or `b_false`, but rather...
- $f : I_1, I_2, \ldots, I_m \mapsto bool$ becomes `p(I1,I2,...,In)`, the result is whether call to the predicate fails or succeeds
- in fact, one such function is a predicate
- this way, we ca only implement what should happen in positive cases

```
greater(s(_), zero).
greater(s(N), s(M)) :- greater(N, M).
```

```
?- greater(s(zero), s(zero)). -> No
?- greater(s(s(zero)), s(zero)). -> Yes
```

Dealing with multiple output arguments

Typical Prolog approach

- Not to have an output of type `Pair`, but rather...
- $f : I_1, I_2, \ldots, I_m \mapsto O1 \times O2$ becomes `p(I1,I2,...,In,O1,O2)`
- in fact, we can conceptually have many inputs and many outputs

```
nextprev(s(N), N, s(s(N))).
```

```
?- nextprev(s(s(zero)), Prev, Next)).
     -> {Prev/s(zero), Next/s(s(s(zero)))
?- nextprev(zero, Prev, Next)).
     -> No
```

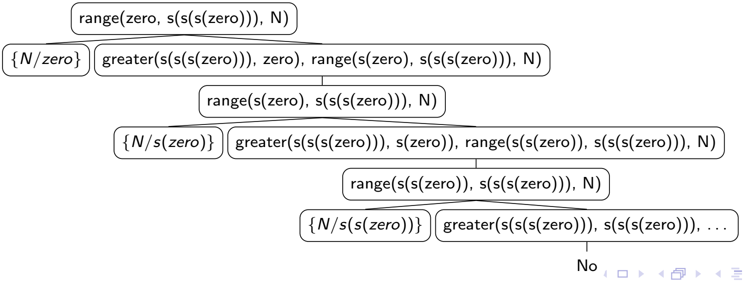
Dealing with multiple results

Typical Prolog approach

- Not to have a result as a sort of lazy list, but rather...
- $f : I_1, I_2, \ldots, I_m \mapsto \mathit{LazyList}$ becomes `p(I1,I2,...,In,O)` such that it provides multiple solutions due to the inherent Prolog resolution mechanism
- in fact, we should always be prepared that goals admit many solutions

```
range(N1, N2, N1).
range(N1, N2, N) :- greater(N2, N1), range(s(N1), N2, N).
```

```
?- inrange(zero, s(s(s(zero))), N).
   -> {N/zero}; {N/s(zero)}; {N/s(s(zero))}
```



Example 3: (linked)lists by `cons/nil` construction

Ideas

- use functors `cons/2` (with head and tail as args) and `nil/0` for empty lists
- e.g.: `cons(a, nil)`, `cons(a, cons(b, nil))` – note they are trees
- generally implement functions/predicates by matching, through different clauses
- recursive functions by base cases implemented as facts, and then by recursive rules

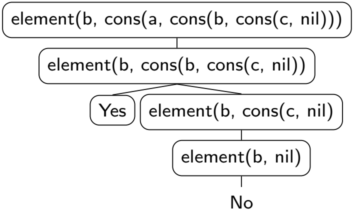
Example 3: `element` over lists

```
% relates an element E with a list that contains it
element(E, cons(E, _)).
element(E, cons(_, T)) :- element(E, T).
```

How to read the specification in the relational interpretation

- `E` is found in a list with head `E`
- `E` is found in a list with tail `T` provided `E` is found in `T`

```
?- element(b, cons(a, cons(b, cons(c, nil)))). -> Yes; No
?- element(a, cons(a, cons(b, cons(c, nil)))). -> Yes; No
?- element(40, cons(a, cons(b, cons(c, nil)))). -> No
```



Built-in list syntax

Syntax for cons/nil construction

1. libraries have list functions assuming you actually use functors:
   - `'.'/2` instead of `cons`
   - `[]/0` instead
   - e.g.: `append('.'(a, '.'(b, [])), '.'(c, []), '.'(a, '.'(b, '.'(c, []))))` succeeds
2. ad-hoc syntax to avoid verbose right-associative construction:
   - write `[H1,H2,...,Hn|T]` instead of `'.'(H1, '.'(H2, ...,'.'(Hn,T)...))`
   - write `[H1,H2,...,Hn]` instead of `[H1,H2,...,Hn|[]]`
   - `[H|T]`, `[]`, `[E1,E2]`, `[_|T]`, `[E1,_|_]` as special cases
   - this is the syntax alwas used!

```
?- append([a, b], [c], L). -> L/[a, b, c]
?- append([a, b], [c], [H | T]). -> H/a, T/[b, c]
?- append([a, b], [c], [_ | _]). -> Yes
?- append([a, b], [c], [_, _, _]). -> Yes
?- append([a, b], [c], [_, _, _, _]). -> No
?- append([a, b], [c], [_, _, E]). -> E/c
?- append([a, b], [c], [_, _ | T]). -> T/[c]
?- append([a, b], [c], [_, _, _ | T]). -> T/[]
```

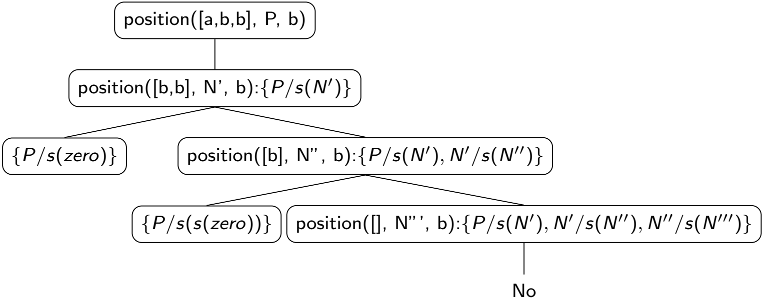
Programming find/position/join

```
% relates a list with one of its elements
find([E|_],E).
find([_|T],E) :- find(T,E).

% relates a list with with one of its elements and its Peano position
position([E|_],zero,E).
position([H|T],s(N),E) :- position(T,N,E).

% relates two lists with their concatenation (similar to append)
join([],L,L).
join([H|T],L,[H|M]):- join(T,L,M).
```

```
?- find([a,b,c], b). -> Yes
?- find([a,b,c], 40). -> No
?- position([a,b,c], zero, a). -> Yes
?- position([a,b,c], s(zero), b). -> Yes
?- position([a,b,b], P, b).  -> P/s(zero); P/s(s(zero))
?- join([a,b], [c], L). -> L/[a,b,c]
```

Resolution with position/3

```
position([E|_],zero,E).
position([H|T],s(N),E) :- position(T,N,E).
```



Full relationality of a Prolog predicate

Informal notion and definition

- a Prolog predicate is said to be *fully relational* if all its arguments could be handled as either input or output
- most specifically, when called with a variable in an argument, resolution successfully attempts to iterate over all inputs that would satisfy the predicate
- often this behaviour can also be obtained for groups of argument, or all arguments

⇒ often this property cannot be achieved, especially for complex algorithms

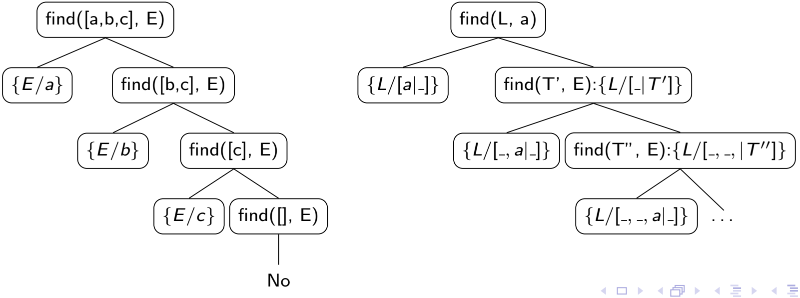
Implication

- when achieved, this property really allows you to obtain "many functions with a single predicate"
- `find` can be used to find in lists, iterate lists, or generate lists

Full relationality of find/2 at work

```
% relates a list with one of its elements
find([E|_],E).
find([_|T],E) :- find(T,E).
```

```
?- find([a,b,c], E). -> ?
?- find(L, a).          -> ?
?- position([a,b,c], N, E). -> ?
?- join(L, M, [a,b,c]). -> ?
?- sum(N1, N2, s(s(s(zero))). -> ?
?- mul(N1, N2, s(s(s(s(zero)))). -> ?
```



Ad-hoc math in Prolog

Prolog operators: the case of unification operator

- a Prolog operator is a binary predicate that can be used in infix notation
- e.g.: =/2 can be used to unify two terms

Values and operators

- can use 10, -20.1, 1.3e-4 as ground terms
- operators =:=, =\=, >=, =< (not <=!!), >, < are modelled as 2-ary predicates working on numbers as expected – they are NOT relational!
- can build terms using +, -, *, / as 2-ary functors, also possibly in infix notation
- operator is/2 can be used to evaluate second argument to a number, unified with first argument
  - watch out: do not use is/2 to unify terms!

Operators at work

```
?- '='(p(1,2), p(X,Y)). -> X/1; Y/2      % standard notation
?- p(1,2) = p(X,Y).     -> X/1; Y/2      % infix notation
?- p(1,2) = p(_,3).     -> No            % no unification

?- '>'(20,10).          -> Yes
?- 20 > 10.             -> Yes
?- 10 > 20.             -> No
?- 10 =:= 20.           -> No
?- 10 = 10.             -> Yes

?- is(X, '+'(10,20)).   -> X/30          % evaluation
?- X is 10+20.          -> X/30          % evaluation
?- 30 is 10+20.         -> X/30
?- 10 is p(20).         -> HALT!         % p(20) not an expression
?- 10 is X+5.           -> HALT!         % x+5 has a variable
```

Working with math: sum/2

```
% relates a list with the sum of its elements
sum([], 0).
sum([H|T], S) :- sum(T, N), S is H + N.
```

```
?- sum([10,20,30], S).   -> S/60
?- sum([], S).           -> S/0
?- sum([10,20,30], 60).  -> Yes
```

sum([10,20,30], S)

sum([20,30], S'), S is 10 + S'

sum([30], S"), S' is 20 + S", S is 10 + S'

sum([], S"'), S" is 30 + S"', M' is 20 + S", S is 10 + S'

S" is 30 + 0, M' is 20 + S", S is 10 + S'

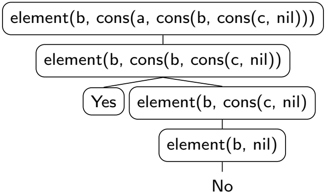
M' is 20 + 30, S is 10 + S'

S is 10 + 50

$\{S/60\}$

Wrap-up on terminology

```
1 element(E, cons(E, _)).
2 element(E, cons(_, T)) :- element(E, T).
3 ?- element(b, cons(a, cons(b, cons(c, nil)))). -> Yes; No
```



Terminology

- Terms: `E`: variable; `_`: wildcard variable; `a`: atom; `cons`: functor name; `cons(E,_)` compound non-ground term; `cons(a,nil)` compound ground term.
- Program: lines 1 and 2: clauses; line 1: fact; line 2: `rule`; part before `:-`: head; part after `:-`: body; part after `?-`: resolvent, a list of goals; `element`: predicate.
- Resolution: root: initial resolvent; arc: resolution step; "yes": solution; "no": failure, traversing from a node to one at a higher-level in the tree: backtracking