Logic Programming and Deductive Databases

Chapter 2: Prolog Tutorial

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Summer 2023

`http://www.informatik.uni-halle.de/~brass/lp23/`

Objectives

After completing this chapter, you should be able to:

- use a Prolog system.

  How does one leave the interpreter?

  How does one load a file with a Prolog program?

- explain the syntax of Prolog predicates, constants and variables.

- explain the basic syntax ofafacts and rules in Prolog.

- develop simple Prolog/Datalog programs.

Contents

Basic Prolog Syntax: Identifiers (1)

- In Prolog, identifiers for constants (and predicates) start with a lowercase letter.

  Identifiers for constants and predicates are also called “atoms” in prolog.

- The reason is that variable names are distinguished because they start with an uppercase letter.

  Prolog is a language without declarations (it uses “dynamic typing”).

  The system must know whether an identifier is a constant or a variable.

  Other languages, e.g. PHP and XPath/XQuery, use a special character (a “$”-Sign) to mark variables.

- Otherwise both contain
  - uppercase and lowercase letters,
  - digits, and
  - the underscore symbol “_”.

Basic Prolog Syntax: Identifiers (2)

- In Prolog, one can also use any sequence of characters enclosed in single quotes `'` (apostrophe) as constant or predicate.
  - The constants `c` and `'c'` are treated as identical.

    But it is good style to decide for one spelling. While it is in principle possible to write predicate names in `'...'`, this is never done.
  - The constants `c` and `'C'` are different.

    Prolog is case sensitive.
- Of course, one can also use numeric constants as known in other languages.

  We will look at the details in a later chapter. For the moment, it suffices that integers like `123` are possible.

Example Database (1)

- As a first example in Prolog, it is customary to study family relations.
- In a simple relational database, we could use three tables:
  - `person(`<u>`ID`</u>`, FirstName, LastName, Gender)`

    Gender can be `m` (male), `f` (female) or `d` (other).

    As known from the database course, primary key attributes are underlined. E.g., there cannot be two table rows with the same value in the `ID` column (attribute).
  - `parent(`<u>`Child`</u>` → person, `<u>`Parent`</u>` → person)`

    The arrow marks a foreign key. All values appearing in the `Child` and `Parent` columns must appear also as primary key value, i.e. as `ID`, in the referenced table `person`.
  - `couple(`<u>`Partner1`</u>` → person, `<u>`Partner2`</u>` → person)`

Example Database (2)

person			
ID	FirstName	LastName	Gender
alan	Alan	Smith	m
bianca	Bianca	Smith	f
chris	Christopher	Johnson	m
doris	Doris	Johnson	f
eric	Eric	Smith	m
fiona	Fiona	Smith	f
george	George	Johnson	m
helen	Helen	Johnson	f
ian	Ian	Smith	m
julia	Julia	Smith	f
ken	Kenneth	Smith	m
laura	Laura	Williams	f

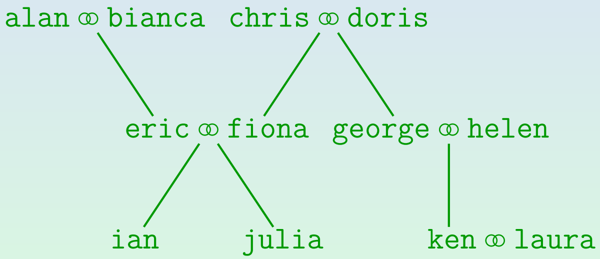
Example Database (3)

parent	
Child	Parent
eric	alan
eric	bianca
fiona	chris
fiona	doris
george	chris
george	doris
ian	eric
ian	fiona
julia	eric
julia	fiona
ken	george
ken	helen

Example Database (4)

couple	
Partner1	Partner2
alan	bianca
chris	doris
eric	fiona
george	helen
ken	laura

Example Database (5)



Predicates (1)

- Logic is the science of statements and their interrelationships, especially consequence.
- Consider a statement with placeholders, e.g. “`C` is child of `P` (parent)”.
- Let us abbreviate this to “`parent(C, P)`”.
- The statement can be true or false if concrete values are given for the placeholders.
- E.g. in the situation described in the above DB:
  - `parent(eric, alan)` is true, and
  - `parent(eric, bianca)` is true, but
  - `parent(eric, fiona)` is false.

Predicates (2)

- "`parent`" is an example for a predicate symbol.
  It has two arguments: child and parent.

  The number of arguments is called the arity of a predicate.

- Formally, a predicate is a function that assigns true or false to given values for the arguments.

- A predicate symbol is a name for such a function.

  One could also choose another name, such as `p` or `child_of`. One only has to use one name consistently. Logic and Prolog do not understand the meaning of the name, they only know the specified facts and rules.

- Since logic analyses statements, it carefully distinguishes between symbols and their interpretation.

  Relation names are defined in the DB schema, relations in the state.

Predicates (3)

- The extension of a predicate is the set of argument tuples for which the predicate is true.

  (`eric`, `alan`) belongs to the extension of `parent` (in the situation of the above DB state), while (`eric`, `chris`) does not.

- Predicates (with finite extension) are really the same as (database) relations.

  E.g. given a relation, one can see it as predicate that is true for the tuples in the relation, and false for all other arguments. In the opposite direction, one chooses the extension of the predicate as the relation.

- In Prolog, one can define predicates with infinite extension, e.g. odd(n) is true iff n is an odd number.

Predicates (4)

- In logic and Prolog, the arguments of a predicate are identified by position.

  I.e. one must know that the first argument is the child and the second the parent. The names of the placeholders in the original statement (`C`, `P`) are not important.

- In SQL, the columns of a table (attributes of a relation) are identified by name.

- However, one could also define a logic programming language that uses argument names.

  If there are few arguments, and one applies consistent style rules for ordering the arguments, then the Prolog notation is shorter (more concise). With many arguments, the SQL notation is safer.

More about Prolog Syntax

- In Prolog, every fact (or rule) must be terminated with "`.`" (full stop).

  It is required that the full stop is followed by white space (a space or a line break). The reason is that "`.`" is also an operator in Prolog (list constructor). If it is used that way, it is not followed by white space.

- One should avoid spaces between the predicate and the opening parenthesis "`(`".

  This is used to distinguish operator syntax from the standard syntax. Operator syntax is treated in a later chapter.

- Comments in Prolog start with "`%`" and extend to the end of the line. (Alternative: `/* ... */` as in C.)

No Declarations in Prolog

- Prolog is a concise language: One does not have to declare predicates or constants.

  Predicates are automatically declared by writing facts or rules about them. This is a bit dangerous because typing errors might not be detected. However, most Prolog systems require at least that (1) facts and rules about one predicate are not interrupted by facts/rules about another predicate (2) at least one fact/rule exists for every predicate that is called. This gives already some protection.

- Prolog is untyped. However, many type systems have been proposed and implemented for Prolog.

  It is easy to write a type checker for Prolog in Prolog, because Prolog has good metaprogramming facilities (processing programs as data).

Example DB as Prolog Facts (1)

https://users.informatik.uni-halle.de/~brass/lp23/prolog/family.pl

```
person(alan, 'Alan', 'Smith', m).
person(bianca, 'Bianca', 'Smith', f).
person(chris, 'Christopher', 'Johnson', m).
person(doris, 'Doris', 'Johnson', f).
person(eric, 'Eric', 'Smith', m).
person(fiona, 'Fiona', 'Smith', f).
person(george, 'George', 'Johnson', m).
person(helen, 'Helen', 'Johnson', f).
person(ian, 'Ian', 'Smith', m).
person(julia, 'Julia', 'Smith', f).
person(ken, 'Kenneth', 'Smith', m).
person(laura, 'Laura', 'Williams', f).
```

Example DB as Prolog Facts (2)

```
parent(eric, alan).
parent(eric, bianca).
parent(fiona, chris).
parent(fiona, doris).
parent(george, chris).
parent(george, doris).
parent(ian, eric).
parent(ian, fiona).
parent(julia, eric).
parent(julia, fiona).
parent(ken, george).
parent(ken, helen).
```

Example DB as Prolog Facts (3)

```
couple(alan, bianca).
couple(chris, doris).
couple(eric, fiona).
couple(george, helen).
couple(ken, laura).
```

Contents

Loading Predicate Definitions (1)

- Write the logic program into a file, e.g. "`family.pl`".

  [https://users.informatik.uni-halle.de/~brass/lp23/prolog/family.pl]

  The extension "`.pl`" is usual for Prolog sources. Unfortunately, it is also used for Perl programs (Prolog was first!). Some Prolog systems permit to choose the extension "`.pro`" during installation. Of course, one can use any extension, but if it is not the standard extension, one later has to specify it explicitly.

- Start the Prolog system (e.g. "`swipl`" or "`pl`" under UNIX).

- It should display the prompt "`?-`".

  This means that it is in query mode.

- Read the file with the command "`[family].`".

  The brackets are an abbreviation for the built-in predicate "`consult`", e.g. "`consult(family).`". Commands are queries to special predicates.

Loading Predicate Definitions (2)

- Do not forget the full stop "`.`" at the end!

  Every Prolog fact, rule, query, or command must be terminated with a full stop. Otherwise, Prolog assumes that the command continues on the next line and either silently waits for more input or displays a prompt like "`|`". Of course, one can then still write the full stop.

- If one has to specify a path (or a filename that is not a Prolog identifier) one must put it in single quotes `'` (to make it a Prolog identifier), e.g.

  ```
  ['C:/stefan/courses/lp23/examples/family.pl'].
  ```

- Note that the backslash "`\`" is usually interpreted as escape symbol, thus it must be doubled: "`\\`".

  SWI Prolog accepts a normal slash "`/`" in filenames also under Windows.

Loading Predicate Definitions (3)

- If one wants to enter rules and facts interactively, one can read the special file "`user`", e.g. "`[user].`".

  The input usually ends with the UNIX end-of-file marker `Crtl+D`.

- Facts and rules can be distributed over several files, e.g. "`[myfacts,myrules1,myrules2].`"

  Most Prolog systems assume that rules about one predicate are stored consecutively in the file. If one loads another file that contain rules about the same predicate, the first rules are forgotten. Normally a warning is printed in this case. However, it is possible (depending on the system) that one reloads a file with the rules about a predicate removed, and the old rules still remain in memory (until one exists from the Prolog system). This is normally no problem, since one will not call the old predicate.

Queries (1)

- Given the above program (“knowledge base”), one can pose queries (goals for the theorem prover), for example:
  - `parent(eric, bianca).`
    $\longrightarrow$ Yes.
  - `parent(eric, chris).`
    $\longrightarrow$ No.
  - `parent(X, bianca).`
    $\longrightarrow$ `X = eric.`
  - `parent(eric, X).`
    $\longrightarrow$ `X = alan.`
    `X = bianca.`

Queries (2)

- Example queries (proof goals), continued:
  - `parent(X, Y).`

    ```
    ⟶ X = eric,   Y = alan.
      X = eric,   Y = bianca.
      X = fiona,  Y = chris.
           ⋮           ⋮
    ```
  - SQL is based on tuple calculus, Prolog on Domain calculus:

    ```
    SELECT X.Parent
    FROM   parent X
    WHERE  X.Child = 'eric'
    ```

    In SQL, variables range over entire table rows (tuples).

    In Prolog, a variable stands for a single data value (domain element).

Using a Prolog System: Queries (1)

- Once facts and rules are defined, one can enter queries (from the "`?-`" prompt), e.g.

  `parent(julia, X).`

- Prolog prints only one solution at a time.
  - If one wants more solutions, one must press the "`;`" key (this stands in Prolog for "or").

    When there are no more solutions, Prolog will print "No". This "a tuple at a time" processing (which may also print duplicates) is also a difference to deductive databases.

  - If one does not want more solutions, one must press the "`Enter`" key.

Using a Prolog System: Queries (2)

- If a query should get into an infinite loop, one can press "`Crtl+C`".

  This normally will enter the Prolog debugger. Pressing "`a`" (for "abort") will stop the query and leave the debugger.

- One can leave the Prolog system with "`halt.`".

  "`quit`" and "`exit`" will not work in most systems. If one really wants, one can of course define them by a rule. Again: Don't forget the full stop "`.`" at the end.

- For predicates with 0 arguments (like `halt`), one does not write "`()`" in Prolog.

  "`halt().`" is a syntax error.

Using a Prolog System: Getting Help

- Most systems have an online manual which documents at least all built-in predicates, e.g. try

  `help(consult/1).`

- Note that the number of arguments usually has to be specified in the notation `p/n` (predicate `p` with `n` args).

  In Prolog, different predicates can have the same name if they have a different number of arguments. E.g. in SWI-Prolog, one can also call "`help.`" to bring up the online manual. This is documented in "`help(help/0).`".

Contents

Logical Formulas (1)

- If there were only such elementary statements, logic would not be very interesting.
- However, one can combine statements with logical connectives, e.g.:
  - $\wedge$: logical "and" (conjunction)
  - $\vee$: logical "or" (disjunction)
  - $\neg$: logical "not" (negation)
  - $\leftarrow$: logical "if"
  - $\leftrightarrow$: logical "iff" (if and only if)

Logical Formulas (2)

- One can also introduce variables:
  - $\forall X$: “for all X” (universal quantification)
  - $\exists X$: “there is an X” (existential quantification)
- In SQL, such formulas are used as query language.

  SQL has no universal quantifier, except in a specific context: `>= ALL`. However, one can simulate it with `EXISTS`-subqueries. Actually, it is a result of mathematical logic that one kind of quantifier suffices.
- Prolog is a restricted automated theorem prover: Knowledge can be specified not only as facts (as in RDBs), but also as rules (special kind of formulas).

Rules (1)

- Predicates can be defined also by "if-then" rules:

  $$\texttt{man(X)} \leftarrow \texttt{person(X, Y, Z, m)}.$$

  "If there is a person with ID X, first name Y, last name Z, and gender "m", then X is a man".

  Note the difference between variables (X, Y, Z), for which any value can be inserted, and constants, such as m, which stand for a single value.

- In Prolog, one writes ":-" instead of "←".
- A rule has two parts:
  - Rule Head: The left hand side, the conclusion.
  - Rule Body: The right hand side, the condition.
- If the rule body is satisfied (for certain values of X, Y, Z), the rule head can be derived (with the same values of X, Y, Z).

Rules (2)

- The above rule defines the predicate "`man`" and uses the predicate "`person`".

  I.e. it assumes that there is information about `person` that can be used to derive information about `man`. Prolog does not require a specific sequence of declaration: In the source file, could also lists the facts for "`person`" below the rule for "`man`". This is important because two predicates can reference each other with mutual recursion (see below). In predicate logic, there is no such distinction between definition and use.

- Derived predicates correspond to database views.
- A rule with the predicate p in the head is called a "rule about p".

  E.g. the above rule is a rule about `man`.

Rules (3)

- Names starting with a capital letter are variables: One can insert any value for a variable.

  I.e. the variables are universally quantified ("for all", $\forall$) in front of the rule. Of course, during a single rule application, one must replace different occurrences of the same variable by the same value.

- E.g. when one replaces `X` with `alan`, `Y` with `'Alan'`, and `Z` with `'Smith'`, one gets:

  ```
  man(alan) ← person(alan, 'Alan', 'Smith', m).
  ```

- The right hand side of the rule is true (it is given as a fact), thus the left hand side can be derived.

Rules (4)

- Suppose one substitutes e.g. `X` with `laura`, `Y` with `'Laura'`, and `Z` with `'Williams'`:

  `man(laura)` ← `person(laura, 'Laura', 'Williams', m).`

- The right hand side cannot be proven, thus nothing can be derived with this rule instance (the condition is false, nothing follows about the head).

  This does not mean that the rule head must be false: There might be another rule / rule instance that permits to derive it (see below).
  In this example, the rule head is false (there is no other way to derive it).

- Prolog and deductive databases do not simply try all possible values for the variables (to reach better performance).

  "All possible values" might also be infinitely many.

Rules (5)

- Of course, one can choose better variable names (they only have to start with an uppercase letter):

  $$\texttt{man(ID)} \leftarrow \texttt{person(ID, FirstName, LastName, m)}.$$

- This renaming of variables does not change the meaning of the rule in any way.
- Variables are implicitly $\forall$-quantified in front of each rule. I.e. the scope of each variable is the rule.
- Two different rules can have variables with the same name, but there is no connection between them.

Anonymous Variables (1)

- When a variable appears only once in a rule, its name is not important.
- Prolog then permits to use an underscore "_" instead of the variable name ("anonymous variable").

  I.e. each occurrence of the underscore stands for a new variable.

  Even if the underscore appears twice in a rule, it is not the same variable.
- E.g. the rule about `man` can be written as:

  ```
  man(ID) :- person(ID, _, _, m).
  ```
- I.e. the underscore can be used to fill in arguments that are not needed.

  This is necessary since arguments are identified by position. It corresponds to a projection in databases.

Anonymous Variables (2)

- Most Prolog systems give a warning (“singleton variables”) if a non-anonymous variable appears only once in a rule.
  - This is intended to catch typing errors in variables.
  - Variables do not have to be declared, but a typing error will yield a variable that appears only once.
  - If one wants to have a meaningful name, although the variable appears only once, one can start that name with an underscore to switch off the warning.

    Thus, the underscore counts as an uppercase letter with this special treatment if it starts a variable name.

Multiple Body Literals

- A rule can have several conditions which are conjunctively connected (logical "and"):

$$\begin{aligned}\texttt{grandparent}(\texttt{X},\texttt{Z}) \leftarrow\ & \texttt{parent}(\texttt{X},\texttt{Y}) \land \\ & \texttt{parent}(\texttt{Y},\texttt{Z}).\end{aligned}$$

- E.g. one successful application of the rule is:

$$\begin{aligned}\texttt{grandparent}(\texttt{ian},\texttt{bianca}) \leftarrow\ & \texttt{parent}(\texttt{ian},\texttt{eric}) \land \\ & \texttt{parent}(\texttt{eric},\texttt{bianca}).\end{aligned}$$

- Both conditions in the body ("body literals") follow from the given facts and rules.

- Then this rule can be applied and permits to derive that `bianca` is a grandparent of `ian`.

Prolog Rule Syntax (1)

- A Prolog rule consists of
  - A rule head, on the left:
    A single atomic formula (“head literal”).
  - A rule body, on the right:
    A conjunction of atomic formulas (“body literals”).
- In Prolog, one writes
  - “`:-`” instead of “$\leftarrow$” (between head and body),
  - a comma “,” instead of “$\wedge$” (between body literals).
- An atomic formula (used as head or body literal) consists of a predicate symbol and a list of argument terms.
- Argument terms are (for the moment) variables or constants.

Prolog Rule Syntax (2)

- E.g. the predicate "`grandparent`" is defined by the rule:

  ```
  grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
  ```

  It has the head literal `grandparent(X, Z)`, and the two body literals `parent(X, Y)` and `parent(Y, Z)`.

- Where a space is permitted, one can use any sequence of newline, spaces, tabs (Prolog is free format):

  ```
  grandparent(X, Z) :-
          parent(X, Y),
          parent(Y, Z).
  ```

  I recommend this formatting with one literal per line and the body literals indented below the head literal (by one tab).

Using Derived Predicates

- Derived predicates can be used in the definition of additional derived predicates:

$$\begin{array}{lll} \texttt{mother(X, Y)} & \leftarrow & \texttt{parent(X, Y)} \land \texttt{woman(Y).} \\ \texttt{woman(X)} & \leftarrow & \texttt{person(X, \_, \_, f).} \end{array}$$

  As mentioned above, it is permissible to use predicates defined later. Only when queries to `mother` are executed, the predicate `woman` must be defined, or one will get an error message. However, it might be better style to make the program readable in sequential order and define predicates before they are used. But a top-down approach for program construction als has its merits (if the meaning of the called predicates is clear).

- It is no problem that `woman` is called with variable `Y` as argument, and defined with variable `X`.

  Since variables have a meaning only within a rule, the variables would anyway be different, even if they had the same name. During the call, they are matched.

Multiple Rules (1)

- The sequence of the two person IDs in the predicate for the married couples is arbitrary (e.g., alphabetic).
- We can define a symmetric version with two rules:

$$\texttt{married\_with}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{couple}(\texttt{X}, \texttt{Y}).$$
$$\texttt{married\_with}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{couple}(\texttt{Y}, \texttt{X}).$$

- Both rules can be used to derive facts about `married`:
    - The first rule gives facts with the partners in the same order as in `couple`.

      E.g. `married_with(alan, bianca)` follows with this rule.
    - The second rule permits to derive facts in the inverse order.

      E.g. `married_with(bianca, alan)` can be derived with this rule.

Multiple Rules (2)

- Suppose that in the first rule, `X` is replaced by `bianca`, and `Y` by `alan`:

  `married_with(bianca, alan)` ← `couple(bianca, alan)`.

- The condition (rule body) is false.

  Only `couple(alan, bianca)` is given as a fact.

- However, the consequence (rule head) is true: `married_with(bianca, alan)` follows from the other rule.

  If the rule body is true, the head must be true, too. If the rule body is false, this rule alone does not say anything about the head (unless we know that there is no other rule about the predicate).

Multiple Rules (3)

- Several rules about one predicate are a way to encode disjunctive preconditions in Prolog.
- Consider again the two rules about `married_with`:

$$\texttt{married\_with}(X, Y) \leftarrow \texttt{couple}(X, Y).$$
$$\texttt{married\_with}(X, Y) \leftarrow \texttt{couple}(Y, X).$$

  A fact is true if it can be derived with the first rule or it can be derived with the second rule.

- This is equivalent to:

$$\begin{aligned}\texttt{married\_with}(X, Y) \leftarrow\ & \texttt{couple}(X, Y)\\ \vee\ & \texttt{couple}(Y, X).\end{aligned}$$

  In Prolog, the disjunction symbol "∨" is written "`;`". But, as this example shows, disjunction is not really necessary. Especially at the beginning, it is a complication that is better avoided.

Multiple Rules (4)

- It is not required that several rules about a predicate all have the same head.

  The facts for a database relation can be seen as different rules with empty body for the same predicate.

- One could write the two rules also as:

$$\mathtt{married\_with}(X, Y) \leftarrow \mathtt{couple}(X, Y).$$
$$\mathtt{married\_with}(Y, X) \leftarrow \mathtt{couple}(X, Y).$$

  Variable names are local to rules, so swapping the two variables in the second rule does not change the meaning of the program in any way.

- The following solution is problematic:

$$\mathtt{married\_with}(X, Y) \leftarrow \mathtt{couple}(X, Y).$$
$$\mathtt{married\_with}(Y, X) \leftarrow \mathtt{married\_with}(X, Y).$$

  The second rule causes an infinite recursion.

Example: Summary (1)

Facts (Database):

```
person(alan, 'Alan', 'Smith', m).
person(bianca, 'Bianca', 'Smith', f).
   ⋮

parent(eric, alan).
parent(eric, bianca).
   ⋮

couple(alan, bianca).
   ⋮
```

Example: Summary (2)

Rules (Derived Predicates, Views):

$$
\begin{array}{lll}
\mathtt{man(X)} & \leftarrow & \mathtt{person(X,\_,\_,m).} \\
\mathtt{woman(X)} & \leftarrow & \mathtt{person(X,\_,\_,f).} \\
\mathtt{father(X,Y)} & \leftarrow & \mathtt{parent(X,Y)} \land \mathtt{man(Y).} \\
\mathtt{mother(X,Y)} & \leftarrow & \mathtt{parent(X,Y)} \land \mathtt{woman(Y).} \\
\mathtt{grandparent(X,Z)} & \leftarrow & \mathtt{parent(X,Y)} \land \mathtt{parent(Y,Z).} \\
\mathtt{married\_with(X,Y)} & \leftarrow & \mathtt{couple(X,Y).} \\
\mathtt{married\_with(X,Y)} & \leftarrow & \mathtt{couple(Y,X).}
\end{array}
$$

Queries

- Syntactically, queries are the same as rule bodies (a conjunction of literals).
- Queries (Goals) are not very powerful, e.g. they normally do not permit disjunction.

  Actually, modern Prolog systems have disjunction in rule bodies and queries. However, this is not really necessary.
- However, one can extend the knowledge base with new rules that define temporary predicates.
  These new predicates can also be used in the query.

  SQL-99 permits to define temporary views in queries (`WITH`-clause).

Contents

Recursive Rules (1)

- It is possible to use a predicate in its own definition:

  $$\texttt{ancestor}(X, Y) \leftarrow \texttt{parent}(X, Y).$$
  $$\texttt{ancestor}(X, Z) \leftarrow \texttt{parent}(X, Y) \land \texttt{ancestor}(Y, Z).$$

- Initially, no facts about `ancestor` are known, thus only the first rule is applicable.
- Then, $\texttt{ancestor}(X, Y)$ is known if Y is parent of X.
- This can be inserted in the second rule, and it is derived that grandparents are also ancestors.
- Another application of the second rule yields that great-grandparents are ancestors, too. And so on.

Recursive Rules (2)

- Finally, all ancestor relationships that hold in the database are derived.

  The example DB contains only three generations, so there are already no great-grandparents. But the recursion works with any number n of generations: After $n-2$ iterations, no new facts are derived.

- Of course, a recursive rule like

  $$\texttt{p(X)} \leftarrow \texttt{p(X)}.$$

  is useless: It never yields anything new.

  In Prolog, such a rule would actually create an infinite loop. This shows that Prolog is not an ideal logic programming language. In logic, the rule is a tautology: It is always trivially satisfied. Deductive databases can process such rules without problems.

Recursive Rules (3)

- The important point is that although one of the rules that defines "`ancestor`" uses "`ancestor`", it never refers to the same fact as it tries to prove.
- As in other programming languages, in Prolog one has to reduce the "problem size" in the recursive call, or the recursion will not come to an end.
- E.g. given the query "`ancestor(julia, bianca)`", Prolog will first try the nonrecursive rule.

  Prolog tries the rules in the order they are written down. This dependence on the rule order again violates the ideal of logic programming. Deductive DBs are again better, but at the expense of performance.

Recursive Rules (4)

- Using the nonrecursive rule, Prolog has to prove

  $$\texttt{parent(julia, bianca)},$$

  but this fails.

- Now it uses the recursive rule. It inserts the data from the query and finds that it has to prove

  $$\texttt{parent(julia, Y)} \wedge \texttt{ancestor(Y, bianca)}.$$

- Thus, it searches for the parents of `julia` (`eric` and `fiona`), and processes the recursive calls:

  - `ancestor(eric, bianca).`

    `eric` happens to be found first, because the fact was written first in the Prolog program. Only when the ancestor search for `eric` is done, Prolog searches for another parent (and finds `fiona`).

  - `ancestor(fiona, bianca).`

Recursive Rules (5)

- The recursive call `ancestor(eric, bianca)` is proven with the nonrecursive rule: `bianca` is mother of `eric`.

  Thus, the answer "Yes" is printed.

- The recursive call `ancestor(fiona, bianca)` fails.

  Prolog first tries to prove that `bianca` is parent of `fiona`. This fails. Then it creates again two recursive calls by inserting `fiona`'s parents: `ancestor(chris, bianca)` and `ancestor(doris, bianca)`. These immediately fail since there are no parents of `chris` and `doris` in the database.

- The problem size is reduced because every recursive call goes one generation up in the database and somewhere, there are no further data.

Recursion in SQL-99

- Ancestors cannot be computed in SQL-92
  (one needs one more join for every generation).
- However, SQL-99 permits recursion:

```
WITH
    RECURSIVE ancestor(Child, Anc) AS
       (SELECT Child, Par FROM parent
        UNION
        SELECT P.Child, A.Anc
        FROM   parent P, ancestor A
        WHERE  P.Par = A.Child)
SELECT Anc FROM ancestor
WHERE  Child = 'julia'
```

Contents

Exercises (1)

- The Table DEPT has the columns
  - DEPTNO (Department Number),
  - DNAME (Department Name),
  - LOC (Location):

DEPT		
DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

- How would these data look as Prolog facts?

  [https://users.informatik.uni-halle.de/~brass/lp23/prolog/empdept.pl]

Exercises (2)

EMP

EMPNO	ENAME	JOB	MGR	SAL	DEPTNO
7369	SMITH	CLERK	7902	800	20
7499	ALLEN	SALESMAN	7698	1600	30
7521	WARD	SALESMAN	7698	1250	30
7566	JONES	MANAGER	7839	2975	20
7654	MARTIN	SALESMAN	7698	1250	30
7698	BLAKE	MANAGER	7839	2850	30
7782	CLARK	MANAGER	7839	2450	10
7788	SCOTT	ANALYST	7566	3000	20
7839	KING	PRESIDENT		5000	10
7844	TURNER	SALESMAN	7698	1500	30
7876	ADAMS	CLERK	7788	1100	20
7900	JAMES	CLERK	7698	950	30
7902	FORD	ANALYST	7566	3000	20
7934	MILLER	CLERK	7782	1300	10

Exercises (3)

Formulate These Queries in Prolog and in SQL:

- Print number and name of the department in Boston.
- List number and name of all employees in the research department.
- List the names of all employees who are manager or president of the company.
- List all employees who earn more than their direct supervisor. One can use a condition like `X > Y`.

  The requirement is that `X` and `Y` must occur in a body literal to the left of `X > Y`, so that they are "bound" to a number when this condition is evaluated.
- List all employees who are directly or indirectly managed by "`JONES`".