Advanced topics in databases:

Introduction to Prolog as a database language

A course note

Henning Christiansen
Roskilde University, Computer Science Dept.

Version 24 feb 2003

Contents

1 Introduction and overview

This note gives a practical introduction to Prolog under a database perspective. Prolog is interesting for students and researchers in the database area for a number of reasons:

- Prolog itself may serve as a prototype database language as relational algebra can be expressed directly, including tabular relations, views and integrity constraints.

- Prolog is a programming language that is very easy to learn and in which it is fairly straightforward to implement and experiment with different database functionalities.

- Prolog is based on first-order logic so that programs have a clear semantics that can refer to a large body of knowledge and tradition.

- It represents live logic in a way that is much more appealing than a myriad of Greek letters and strange symbols on paper. And a good supplement to or a starting point for the one who wants to dig into the database literature which is inherently loaded with logic.

We do not, in general, advocate to use Prolog for storing really large amounts of data, as traditional database systems are much better for this purpose. The real advantage of Prolog in this context is as an experimental tool, which can give a clearer understanding of underlying concepts and serve as workbench for developing new methods in the shape of running prototypes. It is much easier to play with different variations of some advanced database functionality in a Prolog program than doing so by modifying the source code of a big relational database system!

In what follows, we introduce Prolog to a level which is intended to make it possible for the reader to write interesting and working Prolog programs, and to see the similarity with databases on the one side and logic on the other. We explain also basic semantic notions of Prolog as they are useful for the understanding of databases and their semantics in general. For the interested reader, it should be mentioned, however, that there is much more to the semantics of logic programming than the soft and minimal introduction given here.

This is an early version of this note which has been written in a very short time, so (hopefully only) minor errors can be expected here and there. There are not as many references to the literature yet as will be included in a possible future revision, and no reference list is included. The author kindly asks for his readers patience :)

2 Prolog as a simple database engine

2.1 The subset of Prolog called Datalog

Prolog is a programming language based on a subset of first-order logic and the syntactic and semantic notions of Prolog are more or less taken over from logic. There are, however, a few (and occasionally quite unfortunate) clashes in usage between the two worlds that will be noticed in the sequel, mostly as footnotes.

In the following we introduce the basic elements of the Prolog language, which coincide with the subset called Datalog; it contains sufficient expressive power to express relational algebra and is actually a generalization of it in several directions.

The basic entity in Prolog that corresponds to operations and procedures in programming languages and to relations in relational databases is called a *predicate (symbol).* A predicate takes a number of arguments, and this number is called the *arity* of the predicate. The notation `p/2` refers to a predicate with name `p` and arity 2, and `p(a,b)` is an example of an *atom.*[1]

The different arguments can be either *constant (symbol)s* or *variables.* Syntactically, a variable is distinguished from other items as it starts with a capital letter or an underline. Examples of Prolog variables:

```
X
Y
Monkey
Monkey_17_aBcD
_117
_
```

The variable name spelled as one underline character has a special meaning that is explained later; variables of the form "_*number*" should be avoided as Prolog uses such when printing out internally generated variables.

Constants are the basic data values that Prolog works with; generally a constant denotes itself, i.e., constant `a` is (and refers to) a symbolic value that we can only refer to as `a` and which is different from any other constant, say `b`. Constants can be written in different ways:

- starting with a small letter followed by sequences of letters, underlines, and digits: `monkey`, `mOnKeY`, `m_1o_nkey`,
- numbers: `-1`, `0`, `1`, `117`, `3.14`,
- sequences of one or more special characters, however, excluding brackets, single and double quotes, the vertical bar, and the percentage sign; a few restrictions may apply, consult your Prolog manual in problematic cases: `=`, `========`, `@@*@@`, `<=>`,
- arbitrary sequences of characters[2] surrounded by single quotes: `'Monkey'`, `'monkey'`, `'_'`,
- the specific sequence `[]` which serves a specific purpose in Prolog's list notation (see section 5.2); it is, in fact, possible to write spaces between the square brackets and it is still recognized as this atom.

Notice that a constant which can be written without quotes is the same as the one written the same way with brackets, e.g., `monkey` and `'monkey'` are the same thing. The constant `'Monkey'` can only be written in one way as removing brackets yields a variable `Monkey` that is by no means related to the constant `'Monkey'`.

[1] We use here the vocabulary most often used in the literature on mathematical logic. Standard Prolog usage applies "atom" to mean "constant symbol", and what in standard logic is called an atom (such as `p(a)`) is in Prolog called a (simple) goal.

[2] A single quote inside a quoted constant symbol is indicated writing it twice; for example `''''` represents the constant whose name is a single quote.

The fundamental building brick for Prolog programs is the atom which is a conglomerate of predicate and arguments which are either variables or constants. Example: `p(a,X)`. (A third kind of arguments called structures is introduced in section 5.1.

The first Prolog program we show looks as follows and it does not write *"Hello World!"*.

```
father(john, mary).
```

This program consists of a single *fact*; syntactically a fact is an atom followed by a period. One way to understand such a program is as a database. The logical meaning of this program is simply the set of facts {`father(john, mary)`}. Rephrased in database terminology, the program defines a *relation* referred to by predicate `father/2` which contains the tuple $\{\langle$`john`,`mary`$\rangle\}$.

There is no sense in executing a Prolog program, but we can execute *queries* to a given program. The following sample dialogue with a Prolog system shows first how the program is read into the system ("consulted") and a first query is given; the characters "`?-`" are the system's prompt.

```
?- [family0].
File family0 consulted in 117 ms
?- father(john,mary).
yes
?- father(maggi,frede).
no
```

The program is expected to reside in a file called `family0` in this example. The first query *succeeds*, indicated by the system's reply `yes`, since `father(john,mary)` is in fact known by the database. The second query *fails* indicated by the system's reply `no`, since `father(maggi,frede)` is not known by the database.

Queries may contain variables:

```
?- father(john,X).
X=mary ?;
no
```

The query reads "are there any values of `X` that make `father(john,X)` hold in the database?" The system replies "`X=mary ?`" whose meaning is obvious: the question mark indicates to the user that a response is expected. Typing end-of-line means that the user is satisfied with the answer returned, and semicolon (as in the example) means to ask the system for possible alternative solutions; for this little program there are no more possible values for `X` that satisfy the query, hence the system's second response "`no`".

A pragmatic note: As it appears, the Prolog system is an interactive environment that (in most cases) resides in an old-fashioned line-oriented operating system. Under MacOs X, one needs to open a Unix terminal in order to run SICStus Prolog, and similarly under Windows where a command-line window needs to be opened. A plain text editor is the tool which is needed to edit Prolog programs. Libraries for writing graphical interfaces in Prolog are available for different versions and it is possible to interface to Java and C, but in general this is a bit difficult.

Another pragmatic note: No explicit declarations are needed for introducing the symbols used in a Prolog program. Predicates, constants, and variables can be used directly and be thought of as pre-existing and available in any program. This creates obvious problems when something is misspelled but most Prolog systems issue certain warnings which catch most but not all cases. The shortest Prolog programs consist of two characters, e.g., "`p.`", which defines a program with a nullary predicate, i.e., a predicate with zero arguments, which is written without a pair of empty brackets.

Let us extend the program above with more facts so that we can show more interesting queries:

```
father(john, mary).
father(john, karen).
father(paul, john).
```

The programmer who wrote the program probably has in mind a part of a family consisting of four persons, Paul be the oldest, father of John who in turn is father of Mary and Karen. According to our experience, Paul is then grandfather of Mary and Karen. However, the system has no everyday knowledge, in fact no real knowledge at all: it is able to play around with symbols and no more. The relation between real world objects and the symbols is a convention and interpretation of the programmer who is responsible for the correctness of this mapping.

About failure, it should be made clear that the answer "`no`" to a query Q does not mean that the real world interpretation of Q is false (in the real world); it simply means that the program (= the database) does not contain information about Q. It may be the case that the database (= the program) contains all information that characterizes the defined predicates in some real world sense and it may be the case that the database is an incomplete or approximate description of the world.

Now back to the sample program and let us pose a *compound* query:

```
?- father(paul,X), father(X,Y).
```

This query consists of two atoms (each of which is referred to as a *subgoal*) and contains two variables, one of which recurs in both atoms. The meaning of the query is to ask for pairs of values for `X` and `Y` so that the query holds in the database. More precisely pairs of values so that, when they are substituted simultaneously into the variables' positions into the entire query, it provides a collection of facts that holds in the database. Thus we get the two answers (notice that the user types twice semicolon):

```
?- father(paul,X), father(X,Y).
X = john, Y = mary ?;
X = john, Y = karen ?;
no
```

It is obvious that the everyday interpretation is to ask for a grandchild `Y` of Paul, and in order to do this we must include the intermediate link `X` in the query.

Any interesting programming language contains one or more *abstraction mechanisms.* We can define an abstraction mechanism informally as a device which makes it possible to put together a compound expression and refer to it by a name in some way. The word "abstraction" means here that once we got used to the new named thing we can forget all about its definition and just apply it; hopefully the name indicates some real notions. In Java we have classes and methods and in Prolog we have *rules.* The grandfather relation inherent in the query above can be abstracted into the following rule:

```
grandfather(X,Z):- father(X,Y), father(Y,Z).
```

The meaning is (informally) that `grandfather(X,Z)` holds for some `X` and `Z` whenever there is some `Y` so that the query `father(X,Y), father(Y,Z)` succeeds. With this clause in the program we can now ask queries involving the `grandfather` predicate:

```
?- grandfather(paul,X).
X = mary ?;
X = karen ?;
no
```

There is an obvious similarity between a Prolog rule and the definition of a view in a database; we consider the similarity between Prolog and SQL in more detail in section 4. It is interesting to observe that the definition of the `grandfather` predicate is correct with respect to the everyday interpretation independently of which `father` facts are in the database (= program).

In general, a new predicate may be defined by more than one rule. Consider the following predicate `ancestor` that is supposed to include father relations, grandfather relations, greatgrandfather and so on.

```
ancestor(X,Y):- father(X,Y).
ancestor(X,Z):- father(X,Y), ancestor(Y,Z).
```

This predicate is recursive and recursion is basically the only sort of looping that is possible in Prolog.[3]

Rules and facts are collectively called *clauses* and the set of clauses defining a given predicate is called the *definition* for that predicate. The meaning is that the predicate holds for some values if either the first clause succeeds, *or* the second one does, *or* the third one, etc. As it went for a program of facts only, so it does for the two-rule definition of `ancestor` above: An `ancestor` relation holds if either a direct father relation holds or a combination of a father relation holds together with another ancestor relation (which in turn may stem from a `father` fact or yet another nested `father-ancestor` combination).

Finally we explain the use of the anonymous variable which is written as an underline character. Here is an example:

```
father(X):- father(X,_).
```

Here, the anomymous variable could equally well have been written here as an ordinary variable, say `Y`. In order to cope with the problem of misspelling, Prolog issues a warning whenever a variable occurs only once in a clause, but this is suppressed in case the anonymous variable is used. Thus the advice is to use the anonymous variable in case you really intend to have a variable that occurs only once in a clause.

To be more precise, each occurrence of "_" stands for a new variable, thus `f(X,X)` and `f(_,_)` mean two very different things; the last one unifies with `f(a,b)` but the first one does not (unification defined below).

[3] There are a few other ways to produce phenomena that are analogous to loops in imperative programming languages, but this belongs to the less logical parts of Prolog that we shall avoid except when there is a good reason for it.

When a predicate is defined by more than one clause, these clauses work together in an "or" fashion. A goal succeeds if it succeeds with the first clause, or with the second clause or, ... Prolog includes a syntax that represents "or" within a clause. This is useful when two clauses are almost identical except for a few details. For example, the clause

```
a(X,Y,Z):- b(X,Y), (c(Y,W,p) ; d(X,Y,Z,W)), e(W).
```

can be understood as an abbreviation for the following two clauses.

```
a(X,Y,Z):- b(X,Y), c(Y,W,p), e(W).
a(X,Y,Z):- b(X,Y), d(X,Y,Z,W), e(W).
```

Semicolon works, thus, as an "or" operator within a clause body; however, as it can be defined as a kind of syntactic sugar (abbreviating a more complicated expression made without semicolon), we need not consider semicolon when describing Prolog's semantics in details.

Summary of Prolog so far: The subset of Prolog we have shown until now has been given the name Datalog and has been used in database research, as it provides sufficient expressive power to express relational algebra (we return to this point in section 4).

- A program is a finite set of clauses.
- A clause is either a fact of the form "*head*." or a rule of the form "*head*:-*body*.".
- A head is an atom and a body is a sequence of atoms separated by commas.
- In the context of trying to prove a query, the notion of a *goal* refers to a conjunction of one or more atoms to be proved (or disproved), originating either from the query or an expression resulting from the application of a rule; each atom in a goal is called a *subgoal*.
- A goal has the form *predicate-name*(*argument*, ..., *argument*) where each argument is a term, which in Datalog is either a variable or a constant symbol.
- We have not introduced this above, but it is customary in Datalog to split the predicates of a program into extensional and intensional ones; extensional predicates are defined by ground facts only and the intensional ones by at least one rule,[4] ...
- ... where a ground fact is one without variables; in general, "ground" means variable-free. We may occasionally use the term "ground fact" also to refer to a ground atom.
- Informally the semantics of a Datalog program is a database in which the extensional predicates correspond to tables and the intensional ones to views.
- Programs are executed by posing queries which are compound goals, perhaps with variables. The system returns as result, if possible, values for the variables with which the query can be seen as a member of the database.

[4] Facts with variables may be useful in Prolog programs, but are problematic in database applications as we will see later.

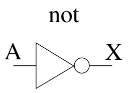
2.2 Example: Logical circuits in Prolog

This example is not so much related to database applications of Prolog but serves to illustrate the diversity of problems that can be described elegantly in Prolog.

We consider logical circuits, which are a graphical formalism that serves as an abstract model of a class of electronic circuits composed by conductors and gates that can be thought of as performing logical operations. A physical component corresponding to the “not” gate below behaves in the following way: If a potential of about five volts is imposed on the input connector to the left in the diagram below, a potential of about zero volts can be observed at the output connector to the right (and the other way round for an input of about zero volts). The actual technology may be based on another pair of voltages than roughly five/roughly zero; the only interesting property is that the gates behave in a consistent way with respect to the logical behaviour. The presentation is more or less self-contained; if a more detailed introduction is needed, we may refer to (Tanenbaum 1999, chap. 3).

We represent the value corresponding to logical “truth” by the constant symbol `1` and logical “falsity” by `0`. The fact that these symbols in some context may serve as numbers in Prolog is of no interest here; we could in principle have used any other pair of two distinct constant symbols.

A given gate (or circuit) can be defined as a predicate whose argument represents the gate's (or circuit's) input and output connectors. A “not” gate, for example, can be specified by the following table.

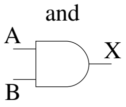


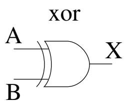
A	X
0	1
1	0

The corresponding definition in Prolog is the following sequence of facts, one for each row in the table.

```
not(0, 1).
not(1, 0).
```

“And” and “exclusive-or” gates are specified in similar ways, and so on for gates corresponding to other logical functions.





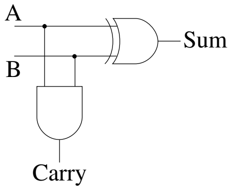
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

The corresponding predicates and(A, B, X) og xor(A, B, X) are defined as follows.

```
    and(0, 0, 0).          xor(0, 0, 0).
    and(0, 1, 0).          xor(0, 1, 1).
    and(1, 0, 0).          xor(1, 0, 1).
    and(1, 1, 1).          xor(1, 1, 0).
```

In the graphical language, gates are put together by connecting the components by means of conductors. In Prolog, we can do very much the same thing, except that we use variables instead of conductors. The following picture shows a so-called half-adder circuit.



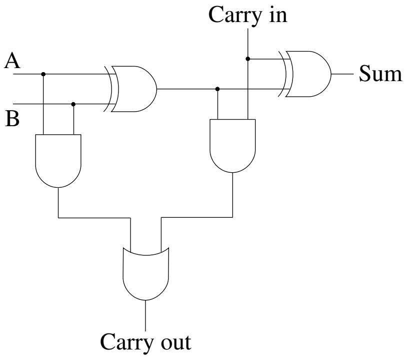
It can be described by a predicate defined as follows.

```
    halfadder(A, B, Carry, Sum):-
        and(A, B, Carry),
        xor(A, B, Sum).
```

It is interesting to recall electric science's definition of a(n ideal) conductor as a body which has the same potential everywhere: x volts in one end means x volts in the other end as well as in any intermediate point. A Prolog variable works exactly the same way within a rule: In an application of a rule (defined formally as "an instance" later), each variable has the same value throughout the rule. In principle, we could replace the rule by all possible such instances that may arise when putting values for variables, e.g.,

```
    halfadder(1,0,0,1):-      halfadder(1,1,1,0):-
        and(1,0,0),               and(1,1,1),
        xor(1,0,1).               xor(1,1,0).
```

The following more complicated circuit is known as a full adder.



It involves some internal conductors that are not connected to the circuit's external connectors. In the Prolog version, these conductors are replaced by variables that recur in the subgoals of the body but do not occur in the head, here X, Y, and Z.

```
fulladder(A, B, Carryin, Sum, Carryout):-
   xor(A, B, X),
   and(A, B, Y),
   and(X, Carryin, Z),
   xor(Carryin, X, Sum),
   or(Y, Z, Carryout).
```

The program explained in this section is a model of a physical system and we can use the program to predict properties of this system. We may, for example, pose a query that for a given set of input values (abstract potentials) calculates the output values.

```
?- fulladder(0, 1, 1, S, C).
S = 0,  C = 1 ? ;
no
```

This shows that the circuit for adding a 0 and a 1 given a previous carry of 1 produces sum 0 with new carry 1, and it appears that this is the only solution.

Contrary to the physical system, we can also evaluate which inputs are necessary in order to produce given output values.

```
?- fulladder(X, Y, Z, 0, 1).
X = 0,  Y = 1,  Z = 1 ? ;
X = 1,  Y = 0,  Z = 1 ? ;
X = 1,  Y = 1,  Z = 0 ? ;
no
```

2.3 How Prolog answers queries

In general, we distinguish between Prolog's declarative and procedural semantics. Declarative semantics means a logical specification of what answers Prolog ideally should produce; it is based on an understanding of clauses as first-order formulas, considered in detail in section 2.4. Procedural semantics, that we consider here, means an abstract description of how, and in which order, the system performs simpler operations to achieve an answer.

A central notion used in the description of both the declarative and procedural semantics is that of a *substitution*. A substitution is a mapping of variables to terms and is used for (among other things) specializing clauses so that they fit specific goals. By tradition, we use Greek letters to refer to substitutions and postfix notation for applying. So we may have a substitution σ that maps variable X to term a and Y to b; this can be specified $\texttt{X}\sigma = \texttt{a}$, $\texttt{Y}\sigma = \texttt{b}$.

Applying a substitution to an expression means to apply it simultaneously to all variables in that expression. Let, for example, σ refer to the substitution above and let the expression in question be the clause p(X,Y):- q(X,Y). In this case we may write as follows:

$$(\texttt{p(X,Y):- q(X,Y)})\sigma = (\texttt{p(a,b):- q(a,b)})$$

In this example, we can say that σ is applied for specializing the clause so that it fits perfectly for the processing of the query `?- p(a,b)`, i.e., in order to have `p(a,b)` to succeed, the specialized clause says that we might obtain this if we can have `q(a,b)` to succeed (involving whatever clauses might be available concerning `q`).

For any expression E and substitution σ, we refer to the image $E\sigma$ as an *instance* of E.

We need to introduce the *composition* of two substitutions which means no more that applying them one after another. The composition of substitutions σ and θ is written simply as $\sigma\theta$ ("first σ then θ"). So if σ maps a variable `X` to some term t, and another substitution θ applies to the variables of t, we have that $\texttt{X}\sigma\theta = t\theta$.

Let, for example, substitution α be defined by $\texttt{X}\alpha = \texttt{a}$ (and leaving all other variables untouched), and substitution β similarly by $\texttt{Y}\beta = \texttt{b}$. We have, then, the following:

$$\texttt{p(X,Y)}\alpha = \texttt{p(a,Y)} \text{ and } \texttt{p(X,Y)}\alpha\beta = \texttt{p(a,Y)}\beta = \texttt{p(a,b)}.$$

Some special kinds of substitutions are useful. A *renaming substitution* for an expression E is one that replaces each variable occurring in E by another distinct variable. So one renaming substitution for the clause shown above as one that we may call ρ with $\texttt{X}\rho = \texttt{X1}$ and $\texttt{Y}\rho = \texttt{Y1}$. With this we have

$$(\texttt{p(X,Y):- q(X,Y)})\rho = (\texttt{p(X1,Y1):- q(X1,Y1)})$$

In a case like this, we refer to the new clause as a *variant* of the original clause. In case the variables substituted in by ρ have not been used before, we talk about a *fresh* variant. The ability to draw new, fresh versions of a given clause is similar to what we have in a traditional programming language with recursive procedures or methods: Whenever a procedure is called, a new stack frame is allocated for the local variables and parameters of that procedure. In this way, each procedure call works in its own local variable space without mixing up different calls.

We talk also about a *grounding substitution* for a clause (or any other expression) if it replaces any variable in the clause by a ground term. So, for example, the substitution given by $\texttt{X}\sigma = \texttt{a}$, $\texttt{Y}\sigma = \texttt{b}$ is grounding for the clause `p(X,Y):- q(X,Y)`.

Finally, we introduce what is called a *unifier* of two expressions E and F, which is a substitution σ such that $E\sigma = F\sigma$. Unifiers are relevant when we compare a given goal G with the head H of a clause H`:-` B. The existence of a unifier of G and H means that the clause might be a candidate for having G to succeed.

In general, we are only interested in the *most general* unifier of two terms. The definition of this is a bit technical: A unifier σ of E and F is *most general* if, for any other unifier θ of E and F, it holds that $\theta = \sigma\gamma$ for some substitution γ. The common instance $E\text{mgu}(E,F) = F\text{mgu}(E,F)$ is called the *unification* of E and F. If no $\text{mgu}(E,F)$ exists, we may say that the unification of E and F *fails.*

A unifier σ being most general simply means that any other unifier is a specialization or a variant of σ.

Let us consider a few examples. The most general unifier of two goals mgu(`p(X,a)`, `p(Y,a)`) exists and is a substitution which maps `X` and `Y` to the same variable (or, alternatively, `X` to `Y` or the other way round). There is no mgu(`p(X,a)`, `p(Y,b)`) nor an mgu(`p(X,X)`, `p(a,b)`).

The fundamental step in the execution of a Prolog program is the unification of a goal to be evaluated with the head of a clause. As a very simple example of this, consider a program consisting of one fact `p(a).` The query `p(X)` succeeds with answer `X=a` because $mgu($`p(X)`,`p(a)`$)$ maps `X` to `a`.

When there is more than one clause matching a given goal, the different clauses are tried out one by one starting with the textually first one. For simplicity, however, we describe the execution of a goal in a nondeterministic fashion, so that an arbitrary clause is chosen. When some clause H`:-`B clause is chosen for the execution of A, the body B is executed similarly to a recursive procedure: The subgoals of B are executed in sequence, and the control is returned to whatever goals follow A.

To execute a query `?-` $G_1, \ldots, G_n$ we assume a state containing a substitution referred to by α which gets more and more specialized as the execution continues and eventually holds a substitution that provides an answer (unless some unification fails along the way). The overall execution of the query is initiated in the following way.

> $\alpha :=$ *the empty substitution*;
>
> execute the query
>
> > $G_1, \ldots, G_n$,print-solution,
>
> where print-solution is a pseudo-goal

The details of execution of composite and primitive goals are described as follows.

1. To execute a sequence of atoms $G_1, \ldots, G_n$ means to execute first G_1 with the current α producing a new substitution that we call α_1; now G_2 is executed starting with α_1 producing α_2 and so on, until G_n has produced α_n which is the resulting, new value of α.

2. To execute the pseudo-goal, print-solution, print out α and stop.

3. To execute an atom G, locate a clause with a fresh variant H`:-`B (or a fact H in which case B refers to an empty body) with $\mathrm{mgu}(G\alpha, H) = \sigma$; set $\alpha := \alpha\sigma$ and execute B; however if there is no such clause for which the specified unifier exists, stop execution.

This nondeterministic algorithm describes all possible ways a query can be executed. A branch is said to fail if the current subgoal does not unify with the head of any program clause. If all branches fail, it means that the query fails. The existence of one successful branch means that the query succeeds.

In general, a Prolog system executes according to a more detailed algorithm that tries different choices of clause (in case 3) in textual order. When a given clause is tried, Prolog sets up a so-called *choice point* indicating that there may be other clauses that might unify with the given goal (H in case 3 above). When a branch fails, Prolog searches back to the most recent choice point, re-establishes the value of α to what it was before a unification was made at that choice point, and execution continues with the next possible clause if such one exists. This strategy is called *backtracking.*

The feature that Prolog asks the user if he or she wants to see another solution (the "`?`" following the variable substitution), is implemented by simulating a failure. So if the user types "`;`", the algorithm treats this as a failure and goes back to the most recent choice point and continues from there.

As it appears, Prolog uses a left-to-right execution order for sequences of goals and sequential order for trying out different clauses. An experienced Prolog programmer is always aware of this. Consider again the clauses at page 7 that define a recursive ancestor predicate. The first clause is the simplest one without recursion: an ancestor relationship between two individuals is by a father fact, so this possibility is tested as the first one. If that one fails, the next clause starts calling a father goal in order to provide some additional information before the recursive call is launched. The best way to illustrate this point is perhaps to show a very bad way of ordering the clauses and subgoals in the body:

```
ancestor(X,Z):- ancestor(Y,Z), father(X,Y).
ancestor(X,Y):- father(X,Y).
```

Logically, it has the same meaning as the original version, but consider the execution of query ?- ancestor(adam,eminem). The first clause is selected, and its first subgoal (with variables replaced) becomes ancestor(Y17, eminem) (Y17 a fresh variable). This new goal hits recursively the first clause, and yet another ancestor(Y18, eminem) and so on, leading to an infinite loop.

In the original program, ?- ancestor(adam, eminem) would probably fail with the first clause ancestor$\cdots$:- father$\cdots$ and enter the second one, giving rise to a call from its body father(Y17,eminem); if eminem has no father, the whole execution fails; otherwise a recursive call of the form ancestor(adam, eminemsFather) is launched.

Unless the program contains a logical bug with someone being ancestor of himself (or the program is a database concerning a world in which time travels are possible and in which case it might not be a logical bug), it is obvious that recursion cannot continue further than the number of facts in the program.

2.4 A logical semantics for Prolog

Here we give a very brief introduction to the declarative semantics of Prolog which is based on first-order logic; for a more detailed presentation, we refer to Lloyd's book from 1987, a survey paper by Apt, and others.

A Prolog clause is understood as an abbreviation for a first-order formula with any variable universally quantified, with ":-" read as implication in reversed order, and the comma as conjunction, i.e. logical "and". For example, the Prolog clause ancestor(X,Z):- father(X,Y), ancestor(Y,Z) is a way of writing the formula:

$$\forall x, y, z(f(x, y) \wedge a(y, z) \rightarrow a(x, z))$$

Quite often, clauses are written with an arrow in the other direction:

$$\forall x, y, z(a(x, z) \leftarrow f(x, y) \wedge a(y, z))$$

Clauses can also be written as a disjunction where body goals are negated; it follows from standard logic equivalences that the two presentation forms are equivalent. The sample clause can also be written as follows:

$$\forall x, y, z(a(x, z) \vee \neg f(x, y) \vee \neg a(y, z))$$

In the literature, a clause is generally defined so that it can have more than one subgoal in its head, i.e., more than one positive goal in the disjunctive format. Clauses with a single goal in the head are called definite clauses; we consider only definite clauses in this note. (Clauses with more than one subgoal in the head are often called disjunctive clauses and are useful for describing imprecise knowledge; however, they are more difficult to treat semantically.)

A useful exchange of quantifiers is possible for variables in the body that do not occur in the head. It is possible to show by standard logic equivalences that the sample clause is equivalent to the following:

$$\forall x, z(a(x,z) \leftarrow \exists y(f(x,y) \wedge a(y,z)))$$

Intuitively this means that in order to prove $a(x,z)$ we need to find just one y so that the body holds; and there is not much use in trying out different y's.

Consider another clause `father(X):- father(X,_)`. It can be written in standard logic notation in the following ways:

$$\forall x, y(f(x) \leftarrow f(x,y)) \qquad\qquad \forall x(f(x) \leftarrow \exists y f(x,y))$$

Thus, anonymous variables in the body can be understood as existentially quantified locally to the atom in which they appear.

The logical reading of a whole program is the logical conjunction of the logical reading of each of its clauses.

So, for example, a program consisting of rules for grandfathers, ancestors plus a few father facts should be read logically as the following formula.

$$\forall x, y, z(g(x,z) \leftarrow f(x,y) \wedge f(y,z))$$

$$\bigwedge \forall x, y(a(x,y) \leftarrow f(x,y))$$

$$\bigwedge \forall x, y, z(f(x,y) \wedge a(y,z) \rightarrow a(x,z))$$

$$\bigwedge f(j,m) \bigwedge f(j,k) \bigwedge f(p,j)$$

In the following we take basic notions of first-order logic for granted; for an introduction, see any introductory book on mathematical logic or Lloyd (1987).

The symbol $\models$ refers to logical consequence, and for any program P (e.g., the formula above), we say that a ground goal g follows from P whenever $P \models g$. It is possible to prove that the procedural semantics of section 2.3 is *sound* and *complete*: Let P be a program and q a query; any substitution σ produced by the procedural semantics is called a *computed answer substitution*; notice that σ does not necessarily ground q.

- **Soundness:** Whenever σ is a computed answer substitution for q in program P, we have that $P \models \forall \bar{x}(q\sigma)$ where $\bar{x}$ are the variables in $q\sigma$.

- **Completeness** Whenever $P \models q$ for some ground atom q, there exists a computed answer substitution for the query q in P.

It is important to make precise that the completeness result requires a nondeterministic semantics that tries out all possible execution paths. Obviously, the infinite loops that may arise with the depth-first, left-to-right strategy applied in any "real" Prolog system, destroy completeness.

We recall that $P \models q$ by definition means that q is satisfied in any model for P, that is any assignment of meanings to the predicates in P that satisfies P. Such "predicate meanings" are mappings from tuples to true or false, which is the same as a relation. When referring to a *Herbrand* model of a program, we refer to a model represented as a set of ground facts.

As an example, let P_0 be program written as a logical program above. Then the following set is a Herbrand model of P_0.

$$M_0 = \{f(j,m), f(j,k), f(p,j), g(p,m), g(p,k), a(j,m), a(j,k), a(p,j), a(p,m), a(p,k)\}$$

How can we check that this is really a model? Well, consider all possible ground instances of clauses of P; for each of them, assign true or false to each atom depending on whether or not it is member of M_0, and check that the whole clause instance evaluates to true with the usual interpretation of $\wedge$ and $\leftarrow$. One such instance is the following.

$$a(p,k) \leftarrow f(p,j) \wedge a(j,k)$$

The body evaluates to true and so does the head wrt. M_0, and as "true implies true" is true, this instance holds. Another instance is

$$a(b,c) \leftarrow f(d,e) \wedge a(j,k)$$

Both head and body contain facts that do not occur in M, so both evaluate to false, and as "false implies false" is true, this instance also holds.

However, M is not the only Herbrand model for P. The following

$$M_0^+ = M_0 \cup \{f(\mathit{mickey}, \mathit{goofy}), a(\mathit{mickey}, \mathit{goofy})\}$$

is also a model of P_0. However, there is something wrong intuitively with M_0^+. There seems to be no good reason to include facts concerning *mickey* and *goofy*, as P_0 does not state anything about them. The problem is that M_0^+ is not minimal.

We define a *minimal* Herbrand model for some program P as one that does not contain another, smaller Herbrand model for P as a proper subset, i.e., minimality is understood according to subset ordering. The following properties can be shown:

- Any Prolog program (without negation as introduced later) has a unique, minimal Herbrand model.

- The minimal Herbrand model for such a program is given as the intersection of all its Herbrand models.

- Let M be the minimal Herbrand model for P. Then $P \models q$ if and only if $q \in M$.

The last property is interesting as it indicates that an implementation of Prolog needs only to consider one model, namely the minimal Herbrand model, in order to be correct. This is basically what the procedural semantics that we gave is doing and what is a basis for its soundness and completeness.

There is an alternative and a bit more constructive way to characterize the minimal Herbrand model for a given program, which is often used as an intermediate characterization for actually proving soundness and completeness. It is based on an *intermediate consequence operator* that maps a set of facts F into a set of other facts F' so to speak by one application of the program clauses. So if F represents what we know at a certain point, F' gives what the program gives of new knowledge by single deductive steps. The operator T_P for program P is defined as follows:

$$T_P(F) = \{A \mid \text{there is an instance } A\texttt{:-}B_1, \ldots, B_n \text{ of a clause in } P \text{ with } B_1, \ldots, B_n \in F\}$$

For a Datalog program (subset of Prolog considered so far), we can generate the minimal Herbrand model in the following way:

```
$M := \emptyset$;
while $M$ grows, do $M := T_P(M)$;
```

For Datalog programs, this algorithm is guaranteed to terminate, and a straightforward proof can show that the final value for M is the minimal Herbrand model for P.

In a more general class of Prolog programs with function symbols and structures (section 5.1), the iteration may go on forever, however, still converging to a well-defined model in the limit. Also here, this model characterizes the semantics of the program.

Let us go back to consider simple Prolog programs with only variables and constants as arguments. There are cases where the model is infinite despite the fact that the iteration terminated in a finite number of steps. Consider the program consisting of the following single clause.

```
equal(X,X).
```

The program defines in a quite reasonable way a predicate `equal` stating that everything is equal to itself (and nothing else). Thus `equal(monkey,monkey)` is a logical consequence of the program whereas `equal(monkey,ape)` is not.

Seen as a program in a programming language called Prolog, it is quite sensible, but when we consider Prolog as a database engine, it is a bit problematic. A database engine is supposed to be able to answer queries. When we give it a query such as `?- equal(monkey,X)`, it should be able to produce the set of all those values of `X` for which the query is satisfies; in this case it is the set {`monkey`}. (The procedural semantics we described earlier will need to backtrack in order to produce the full result in case of a program with several clauses, but other evaluation mechanisms could work in a way that resembles the iterative characterization of the minimal Herbrand model).

But consider the query `?- equal(X,Y)`. Here the result is the infinite set

$$\{\langle x, x\rangle \mid x \text{ is a Prolog constant symbol}\}.$$

In other words, the analogy to a database is gone if we think of a database relation as table of a finite number of tuples, each of which is representing some entity in the real world.

The problem with the `equal` predicate is the presence of a variable in the head of a clause that does not occur in the body. Consider the following "typed" version of the equal predicate.

```
equal(X,X):- domain_object(X).
domain_object(cat).
domain_object(dog).
```

Here the `equal` predicate denotes a finite relation $\{\langle\texttt{cat}, \texttt{cat}\rangle, \langle\texttt{dog}, \texttt{dog}\rangle\}$.

Definition 1 *A Datalog program is* range-restricted *whenever no clause has a variable in its head that does not occur in its body.*

It is easy to prove that the Herbrand model of a range-restricted Datalog program is finite; when later we introduce negation, we need to revise the definition so we can keep this desired property.

2.5 Predefined predicates

In the same way as relational algebra can refer to various comparators in selection expressions, it is also useful to include some auxiliaries in Prolog/Datalog to express such things.

If, for example, the argument of some predicate is known always to represent a number, it may be useful to compare such numbers in various standard ways. Assume a predicate `girl` with two arguments, the first one giving the name of a girl, the second her age, and consider the following predicate definition:

```
older_sister(X,Y):-
    girl(X,AgeX), girl(Y,AgeY),
    X \== Y,
    parent(Z,X), parent(Z,Y),
    AgeX > AgeY.
```

Predicates such as "`>`" and "`\==`" are called *predefined*, as they have no definition in the program but have an a priori meaning which is a possibly infinite relation. For "`>`" the following relation serves as definition.

$$\{\langle x, y\rangle \mid x \text{ is a number greater than } y\}$$

Prolog has several predicates expressing that terms are not equal. The simplest one that we have used here, "`\==`", has an a priori meaning given by the following relation:

$$\{\langle x, y\rangle \mid x \text{ and } y \text{ are Prolog terms that are different}\}$$

We use the term *program defined* about a predicate that appears as the head of one or more clauses in the current program as to distinguish them from predefined ones.

Obviously, predefined predicates serve other purposes than the program defined ones that we think of as database predicates. We must take care when using predefined predicates in our database programs so we do not destroy what we obtained by the introduction of range-restrictedness. The following definitions are no good:

```
big_number(X):- X > 4.
some_number(X):- X > Y.
```

The following generalization of the previous definition is sufficient.

Definition 2 *A Datalog program extended with predefined predicates in rule bodies is* range-restricted *whenever*

- *no clause has a variable in its head that does not occur in its body,*
- *any variable which is argument to a predefined predicate in the body of some rule appears also as argument of a predicate defined in the program.*

The `older_sister` rule above illustrates this definition perfectly. We assume that a suitable collection of predefined predicates is available and use those we may need without explanation when they are named by some standard comparison operator.

Such predicates are standard in implemented versions of Prolog. However, they should be used with care. Most such predicates have the unfortunate property that their arguments must be instantiated at the moment the predicate is called in order to work correctly. This means that a variable should occur in a defined normal predicate in a rule body textually before it is involved with a predefined predicate.

Again the older_sister predicate illustrates this principle. If "AgeX > AgeY" were placed as the first subgoal in the body, the logical semantics as we have specified it is preserved, but the procedural semantics in a running Prolog system would not work.

Prolog includes some predefined predicates that behave in a more logical way.

- A=B, a predicates that unifies terms A and B; it could in principle have been defined by means of the program "=(X,X)".

- dif(A,B) a condition that A and B are different; it is executed in a special way so that it does not, so to speak, make mistakes if the arguments are not instantiated. If any of A or B is uninstantiated, the call to dif is delayed; at the time both are instantiated, the predicate wakes up and either succeeds or fails depending on the values of A and B.
  NB: The dif predicate is particular to SICStus Prolog and not included the ISO standard for Prolog.

Prolog includes a large collection of other standard built-in predicates that we introduce later when we extend the language with structures. Both "=" and dif work also correctly when the arguments are structures.

2.6 Exercises

Exercise 2.1 Consider the following program of mother and father facts plus two rules defining a parent predicate.

```
father(paul, john).       mother(jane, john).
father(john, mary).       mother(ann, karen).
father(john, karen).      mother(karen, peter).

parent(X,Y):- father(X,Y).
parent(X,Y):- mother(X,Y).
```

Evaluate by hand and check by running the program, the possible answers that Prolog gives for the following queries:

```
?- parent(john, X).
?- parent(X, john).
?- parent(parent, X), parent(X, Y), parent(Y,Z).
```

Define a predicate aunt_or_uncle(x,y) which holds if and only if x is an aunt or an uncle of y.

Exercise 2.2 Consider the example of section 2.2 showing a Prolog implementation of logical circuits.

- Use the half adder and full adder predicates for putting together another predicate that adds two 3-bit numbers and produces a 4-bit number.

- Apply the predicate you have just constructed in order to define a new predicate that performs subtraction.

Exercise 2.3 Consider the example of section 2.2 showing a Prolog implementation of logical circuits. In this exercise, we are interested in building a logical circuit with three input pins A, B, and C and one output pin X. The relationship between A, B, C, and X is the following: If $C = 0$, then X is "A or B"; otherwise X is "A exclusive-or B".

- Write down a truth table for the logical function computed by such a circuit.
- Draw a diagram for a logical circuit that computes this function.
- Write this diagram as a Prolog program and test it.

Exercise 2.4 Section 2.3 gave a semiformal description of Prolog's procedural semantics in the shape of an abstract interpreter. This interpreter is nondeterministic in choice of clause and each possible execution path may be either successful and result in an "answer substitution", or it is failed.

- Extend the program of exercise 2.1 with a `grandparent` predicate, and write down the successful execution paths for the query `?- grandparent(X,karen)` with indication, in each step, of the current substitution (i.e., current value of α).
- Consider the definition of the `aunt_or_uncle` predicate that you gave as part of the solution to exercise 2.1. Write down a successful execution path for the query `?- aunt_or_uncle(X,peter)`. NB: If your `aunt_or_uncle` definition is correct, it includes a test (an application of a predefined predicate) that is not treated by the procedural semantics described in section 2.3; explain how you need to extend the procedural semantics so that you can execute `aunt_or_uncle` queries.

Exercise 2.5 Section 2.4 described a logical semantics for a subset of Prolog and the purpose of this exercise is to apply it to the circuit program of section 2.2. Write down how the program should be read with logical symbols. Evaluate the minimal Herbrand model using the T_P construction.

Notice the similarity between this model and the truth tables for the half and full adder predicates. Beware that there are two different levels of truth involved here.

3 Negation-as-failure in Prolog

Prolog includes a sort of negation which is intuitively suitable for database applications but is a bit problematic with respect to its semantics.

Negation of a goal G in Prolog is written in the following idiosyncratic way: `\+` G. There are some fundamental differences between negation in Prolog and the way negation is normally conceived in logic; this is a very delicate topic so we will avoid going into details. We simply state how it works in Prolog and ask the reader to be aware that things are not exactly as negation in classical first-order logic. There are also some additional problems with the actual implementation in Prolog of negation that the programmer needs to be aware of.

Negation in Prolog is *negation-as-failure*, which means that a ground, negated goal `\+` G is supposed to be true wrt. a program P if and only if G fails in the program.

In a database, this is fine. Everything in the database is considered to be true, everything else to be false. Maybe some people would start a philosophical and moral argument that this is a dangerous way to define truth: What has been observed and entered into the database is considered true, everything that the subject has not observed in considered false. "... like closing your eyes before crossing the street, you can't see any cars, thus there are no cars, thus you can always safely cross the street with closed eyes". What can we say to this argument? Well, it is somehow irrelevant if it is made clear that the database's notion of truth is an arbitrary notion, and that it should be understood as "known by the database". So in this view, if the closed eye pedestrian is run over by a car, then it is a car that he did not know about.[5] So anyone, including Prolog programmers, should have their eyes open when crossing the street.

As we mentioned, there are some problematic issues with Prolog's way of handling negation, but let us ignore that for a moment and give an example which applies negation in a fully sensible way. We define a predicate `orphan` to hold for any person without a father or mother; compared with the previous examples, we introduce a predicate defining which persons exist.

```
person(adam).
person(abel).
father(adam,abel).
orphan(X):- person(X), \+ father(_,X), \+ mother(_,X).
```

The query `?- orphan(X)` succeeds with one answer `X=adam`. This seems to be a sensible answer. Asking `?- orphan(eve)` results in a failure because the subgoal `person(eve)` fails and in this case the negations need not be considered.

Negation in Prolog works as a kind of test in the sense that it cannot instantiate variables. Consider the query `?- \+ person(X)`, with the intended meaning "give me a list of non-persons", or perhaps "give me a representation of all those values of `X` for which `?- person(X)` fails." Intuitively, this should be any constant value except `adam` and `abel`; a different technology than Prolog might return and answer such as `X`≠`adam` ∧ `X`≠`abel`. An implementation that works like this has been suggested under the name of constructive negation (Chan, 1988) but is not used in Prolog for reasons of efficiency.

What answer will Prolog produce, then? Well, the principle when calling a goal `\+`G is to call G; if it fails, `\+`G succeeds without binding variables in G; if G succeeds, i.e., if some instantiation of the variables of G can be shown to hold in the program, `\+`G fails. Thus for `?- \+person(X)`, the goal `?- person(X)` is called, which succeeds with `X=adam`, so the original query `?- \+person(X)` fails.

In other words, the query `?- \+person(X)` is treated as the logical formula $\neg(\exists x\ p(x))$. In general, any variable X in a negation which is not instantiated at the time of the call is treated as existentially quantified inside the negation.

Let us examine what the programmer did in order to have the *orphan* clause work right. When `orphan(X)` is called, or `orphan(`*name*`)` for some constant *name*, the first thing that happens is that `person(X)` (or `person(`*name*`)`) is called; if it does not fail, we are sure that subsequently `X` has a value that we call *name* in both cases. Next, `\+father(_,`*name*`)` is called; as we have seen, it

[5] 7-9-13

represents the logical condition $\neg(\exists y \texttt{father(}y\texttt{,}\mathit{name}\texttt{)})$, i.e., it succeeds if and only if there is no father for the given *name*. The last call `\+mother(_,`*name*`)` works in a similar way.

To summarize, the logical meaning expressed by the clause for `orphan` is the following.

$$\forall x(o(x) \leftarrow p(x) \wedge \neg(\exists y\ f(y,x)) \wedge \neg(\exists z\ m(z,x)))$$

In order to express this meaning in Prolog, the programmer carefully considered the following:

- A variable such as x universally quantified at the level of the clause is touched by a call that either fails or instantiates x before (i.e., textually to the left of) any possible negations in the clause.
- A variable existentially quantified at level of the atom being negated is a new variable that does not occur elsewhere in the clause, and thus conveniently written as Prolog's anonymous variable.

It is easy to see what can go wrong; try to consider a version of the `orphan` clause in which the content of the body has been interchanged.

```
orphan(X):- \+ father(_,X), \+ mother(_,X), person(X).
```

In case `X` is instantiated when this clause is invoked, the subgoal `\+ father(_,X)` executes with the same meaning as in the original clause. However, if `X` is not instantiated, this subgoal executes as $\neg(\exists x, y\ f(y,x))$ which is something quite different.

In order to formalize "good behaviour" we extend the definition of range-restrictedness to cover negation in the following way.

Definition 3 *A* literal *is either an atom or an expression of the form* $\neg(\exists \bar{x}\ A)$ *where* A *is an atom and* $\bar{x}$ *a sequence of variables; these forms are called, respectively,* positive *and* negative literals*. A variable in a negative literal not covered by the existential quantifier is called a free variable (relative to that literal). A clause is* range-restricted *whenever*

- *Any variable in its head occurs also in a positive, program-defined atom in the body.*
- *Any variable which is argument to a predefined predicate occurs also in a positive, program-defined atom in the body.*
- *Any free variable in a negative literal occurs also in a positive, program-defined atom in the body.*

A program is range-restricted *if all its clauses are range-restricted.*

In actual Prolog programs, we need also apply a suitable left-to-right order of the subgoals in a clause. A safe way to write a range-restricted clause, is to place all positive, defined literals in the beginning of the clause and all predefined and negated ones at the end; however, it may be advantageous for reasons of efficience in some cases to deviate from this strict rule. All quantifiers are implicit in Prolog clauses, so care should be taken with choice of variables used in negative literals, so that those thought of as existentially quantified should not occur elsewhere in the clause (the anonymous variable "_" is useful here).

Yet another difficult problem concerns recursive programs which include negation. The semantics is difficult when a predicate is defined directly or indirectly in terms of its own negation. To avoid this, it is common to require programs to be *stratified.* This notion is named after the latin word “stratum”, which means a layer. The predicates in a program should be layered in such a way that the definition of a predicate at one level only refers negatively to a predicate at a lower level (directly or indirectly). We skip the formal definition. The logical semantics described in section 2.4 and the characterization of its minimal Herbrand model is easily generalized to range-restricted and stratified programs, but without this requirement, more complicated machinery is needed.

An important notion often referred to in the literature is the *Clark completion* of a program. In order to express negation-as-failure, the logical reading of a clause is now taken as an if-and-only-if reading of the disjunction of all its clauses. As an example, consider the `ancestor` predicate definition in Prolog at page 7 that we repeat here:

```
ancestor(X,Y):- father(X,Y).
ancestor(X,Z):- father(X,Y), ancestor(Y,Z).
```

Its two clauses are read together as the following formula.

$$\forall x_1, x_2 \left(a(x_1,x_2) \leftrightarrow \Big(\exists x,y\,(x_1 = x \wedge x_2 = y \wedge f(x,y)) \bigvee \exists x,y,z\,(x_1 = x \wedge x_2 = z \wedge f(x,y) \wedge a(y,z)) \Big) \right)$$

In this way, we have a definition which gives information about both when `ancestor` (written as a) is true and when it is false. This is due to the only-if part (i.e., the → part of ↔). The Clark completion of a program gives a semantics of negation that is consistent with negation-as-failure.

This sort of negation is also called *default negation* and the principle referred to as the *closed world assumption.* (An open world assumption is one that says “don’t know” about things that are not implied by or provable in the program.)

The final remark in this short survey of negation in logic programming is that Prolog really applies a principle called negation-as-finite-failure. It may be the case that a goal G loops in which case it cannot be determined whether or not we should have \+G. Only in case the failure (or success) of G can be determined in a finite number of steps, we can say something about \+G. We leave this topic, but now the reader should have some idea of what negation-as-finite-failure refers to when he or she finds it in the literature.

((Note for next version of this note: Include a fixpoint construction for stratified programs; refer to standard notions of well-founded and stable model semantics.))

3.1 Exercises

Exercise 3.1 This exercise is concerned with the use of negation-as-failure in Prolog. Consider the program of exercise 2.1 and extend it with relevant `male`/1 and `female`/1 facts, a rule for the `grandparent` predicate, and the `sibling` predicate.

- Define a `sibling` predicate by means of the `\==` test.

- Define a `cousin`/2 predicate according to the principle that two persons are cousins if they have a common grandparent, unless .... (fill in the rest yourself).

- Define other family relations in this way.

Test your solutions on the computer. Argue for in each case that the clauses you construct are range-restricted and that Prolog will execute them correctly.

4 Translating relational algebra and integrity constraints into Prolog

There is a straightforward mapping of expressions of relational algebra into Datalog programs that we will illustrate by means of examples. Assume for this example that we have four tabular relations in our algebra, $R(A, B)$, $S(A, B)$, $R(A, B, C, D)$, and $S(C, D, E, F)$. In the following we show how some standard operators can be implemented by defining a new predicate; each of the tabular predicates are supposed to be represented by sets of Prolog facts for the predicates `r/2`, `s/2`, `r/4`, and `s/4`.

The only disadvantage with this translation method is that the programmer needs to keep track of relational algebra's attribute names as Prolog does not have such names but identifies each argument by its position in the argument list.

Intersection $R(A, B) \cap S(A, B)$:

```
r_intersect_s(A,B):- r(A,B), s(A,B).
```

Union $R(A, B) \cup S(A, B)$:

```
r_union_s(A,B):- r(A,B).
r_union_s(A,B):- s(A,B).
```

Difference $R(A, B) \setminus S(A, B)$:

```
r_minus_s(A,B):- r(A,B), \+s(A,B).
```

Selection $\sigma_{A>B} R(A, B)$:

```
select_r_AgtB(A,B):- r(A,B), A>B.
```

Projection $\pi_A R(A, B)$:

```
project_A_r(A):- r(A,B)
```

Natural join $R(A, B, C, D) \bowtie S(C, D, E, F)$:

```
r_join_s(A,B,C,D,E,F):- r(A,B,C,D), s(C,D,E,F).
```

Compound relational expressions can be represented either by a series of definition (one for each subexpression) or by a larger and more complicated definition in Prolog.

Integrity constraints may be formulated in relational algebra in different ways. In Prolog, integrity constraints can be implemented by means of a query which is tested in a funny way. Once the Prolog program defining the database has been set up with facts for tabular relations and rules for views, we check integrity constraints as follows.

- If a query representing integrity constraints fails, integrity holds.
- If a query representing integrity constraints succeeds, integrity is violated.

The reason for this convention becomes clear when we go through different sorts of integrity constraints. We expect each integrity constraint to be implemented as one clause for a predicate that we call `ic_violated`; if `ic_violated` fails, the database is OK, otherwise not OK. Notice that `ic_violated` failing means that every clause defining `ic_violated` fails.

Key constraints, A is key in $R(A, B, C, D)$:

```
ic_violated:- r(A,B1,C1,D1), r(A,B2,C2,D2), (B1 \== B2 ; C1 \== ; D1 \== D2).
```

Notice that the compound test also can be written `(B1,C1,D1) \== (B2,C2,D2)` where the parentheses-and-commas denote structures of the sort we introduce in section 5.1.

Referential integrity, A in $R(A, B, C, D)$ must reference some tuple $S(A, B)$:

```
ic_violated:- r(A,_,_,_), \+ s(A,_).
```

Assertion with emptiness; example $R(A, B) \cap S(A, B) = \emptyset$:

```
ic_violated:- r_intersect_s(_,_).
```

or directly `ic_violated:- r(A,B), s(A,B).`

Assertion with subset; example $R(A, B) \subseteq S(A, B)$:

```
ic_violated:- r(A,B), \+ s(A,B).
```

Integrity constraints are typically statements about the whole database and checking integrity constraints involves an inspection of the whole database. This is why a formulation of integrity checking as the requirement that something should fail is practical. If the condition `ic_violated` fails, it means that all possible ways it might hold, i.e., all possible combinations of tuples that might violate integrity have been examined. If `ic_violated` succeeds, this means that an unfortunate combination of tuples has been found that violates one of the integrity constraints.

4.1 Exercises

Exercise 4.1 Define a program representing a small database about a collection of persons with the following predicates; invent some test data.

`person(`*registration-number*`, `*name*`, `*street-and-no*`, `*postal-code*`, `*gender*`)`

`married(`*registration-number-for-husband*`, `*registration-number-for-wife*`)`

`offspring(`*registration-number-for-parent*`, `*registration-number-for-child*`)`

`postal_code(`*postal-code*`, `*city*`)`

Define the following views; write them directly as Prolog predicates and sketch the analogous relational algebra expressions.

- `separated_couples(`*registration-number-for-husband*`, `*registration-number-for-wife*`)` which holds for married couples where husband and wife have different address.
- `unmarried_couple(`*registration-number-for-"husband"*`, `*registration-number-for-"wife"*`)` which holds for two persons of opposite gender, however none of which is offspring of the other.
- ... extend the list with your own inventions.
- `address_label(`*name*`, `*street-and-no*`, `*postal-code*`, `*city*`)`; suitable combination of information from `person` and `postal_code` suited for generating an address label.

Define the following integrity constraints by means of `ic_violated` clauses as outlined above; test them one by one as you write them by adding and deleting manually illegal tuples. It is recommended to write them directly in Prolog instead of specifying them in SQL or relational algebra.

- The *registration-number* is key in the `person` relation.
- The *gender* field in the `person` relation can only be one of the constants `male` and `female`.
- Any field in any relation indicating a *registration-number* must refer to an existing `person` tuple.
- Specify yourself natural integrity constraints related to postal codes.

You may continue extending the database with new relations (predicates), integrity constraints, and views.

5 Hacks and features that make Prolog into a general programming language

Up to now we have presented a subset of Prolog that can be used as a database engine answering queries corresponding to relational expressions. As we have noticed, this part of Prolog is a bit more general than relational algebra: Relations can be defined by means of recursion (e.g., the ancestor predicate, page 7) and we noticed also that only those programs that satisfied the range-restrictedness criterion have reasonable interpretations as databases.

No support is given in this subset for updating a database; with facilities described up to now the only possible way is to edit the program and read it into Prolog once again. We return to this topic in section 6 when we have introduced the appropriate tools.

Prolog is not only a database language. It is a fully equipped, general programming language whose expressive power goes far beyond what we expect in a standard database definition and querying language.

A characteristic of the Prolog programming language is its appropriateness for metaprogramming. Metaprogramming means programming about programs, and a typical example is an interpreter: An interpreter is a program that takes as input a program in some object language, analyzes it to its smallest parts and puts together a meaning of that program. For an expression-like language, the meaning is a value of the input expression, e.g., a set of tuples for a relational expression or an integer value resulting from an arithmetic expression. For an imperative language with control structures and side effects, the meaning is given as a simulation of a program execution.

The main advantage of having the semantics of a language defined by means of a program (e.g., a Prolog program) is that we are able to modify the semantics, for example by suggesting alternative evaluation mechanisms in a database.

In the following we go through the remaining elements of Prolog.

5.1 Data structures

The only sorts of data that we have seen so far are constants symbols, but in a general programming language, this is not sufficient.

In general, arguments to Prolog predicates are *terms* built from constants and function symbols (the latter also called *functors* in standard Prolog terminology). We introduce Prolog terms by means of an example inspired by Bratko (2000).

Let us assume that we are interested in geometry and we want to represent in a Prolog program notions of points and line segments in the two-dimensional plane.

To represent a point means to hold a pair of two coordinates; a point with x-coordinate 1 and y-coordinate 2 can be represented by the Prolog structure `point(1,2)`. This is a legal Prolog term, and no declaration as a constructor needs to be made of "`point`" before its use. The following short Prolog program defines a predicate that holds for any structure built with function symbol `point` and two subterms.

```
is_point( point(_,_) ).
```

As for predicates, we identify also function symbols by their name and arity which for the applied `point` is 2. Consider the following queries to the one-line program above:

```
?- is_point( point(1,2) ).
yes
?- is_point( duck ).
no
?- is_point( point(monkey,horse) )
yes
```

The two first ones show an expected behaviour but the last one indicates that it is possible by means of `point` to write structures that exceed what we normally accept as a geometrical points. In strongly typed languages such as Java or Pascal, it is necessary to define the types of the arguments to a given constructor, but in Prolog we can use any function symbol to any sort of arguments.

In general, Prolog terms can be built recursively using any function symbols, constant symbols, and variables. We can give the follow grammar for Prolog terms.

$$\begin{aligned}\langle \textit{term}\rangle ::= {} & \langle \textit{function symbol}\rangle\texttt{(}\langle \textit{term}\rangle\texttt{, } \ldots\texttt{, }\langle \textit{term}\rangle\texttt{)} \\ \mid {} & \langle \textit{constant}\rangle \\ \mid {} & \langle \textit{variable}\rangle\end{aligned}$$

Going back to the geometry example, we may choose to represent line segments in the plane by means of a function symbol `line_segment` of arity 2 so that `line_segment(`p_1`,`p_2`)` represents the segment with end points p_1 and p_2. For example, `line_segment(point(1,1), point(2,2))` represents with the indicated interpretation, a segment between points (1,1) and (2,2).

In geometry, it usually does not matter in which order we mention the two endpoints, but Prolog has no such understanding of geometry as shown by the following query and answer:

```
?- line_segment(point(1,1),point(2,2))=line_segment(point(2,2),point(1,1)).
no
```

Prolog's understanding of equality is purely syntactical, no real meaning is attached to the symbols. It is the responsibility of the programmer to express meanings by means of predicate definitions. So in a program in which comparison of line segments is of interest, the following predicate definition may be relevant.

```
same_line_segment(L,L).
same_line_segment(line_segment(point(X1,Y1),point(X2,Y2)),
                  line_segment(point(X2,Y2),point(X1,Y1))).
```

Notice again, that a predicate defined in this way is a sort of over-specification of the intended predicate. This means that the relation actually specified by this program is much larger than its intended meaning. It includes, for example, the fact

```
same_line_segment(card(hearts,ace),card(hearts,ace))
```

which seems to make no sense since `card(hearts,ace)` does not represent a geometric notion.

Prolog programmers usually do not pay attention to the problem of over-specification, and experienced programmers (unless they make a bug) usually write their programs so that over-specified predicates are applied only to the right sorts of data.

We can put more precise words on the indicated problem. The program defining `same_line_segment` is not range-restricted, i.e., the variables in the head do not occur in the body (which is empty for both clauses). Adding conditions in the body that describe the sort of values that are allowed for the variables will provide range-restrictedness. Consider the following elaboration of the program, where we expect satisfactory definitions of `is_segment` and `is_point`.

```
same_line_segment(L,L):- is_segment(L).
same_line_segment(line_segment(P1,P2), line_segment(P2,P1)):-
    is_point(P1), is_point(P2).
```

The purpose of these examples was to introduce structures in Prolog and we used the occasion to explain once again what range-restrictedness means. We go back to the introduction of Prolog in the way most Prolog programmers use it. The following two definitions of predicates for line segments being vertical or horizontal show the true power of Prolog's unification in the head of clauses.

```
vertical(   line_segment(point(X,Y), point(X,Y1)) ).
horizontal( line_segment(point(X,Y), point(X1,Y)) ).
```

Each of the two clauses describe by means of a pattern what it means for a segment to satisfy a condition, no explicit tests are necessary.[6] As we have learned already, a call to a defined predicate serves not only as a test but in general it may instantiate variables so that the condition is enforced.

```
?- vertical( line_segment(point(1,1), point(A,3)) ).
A = 1
```

We showed in section 4 how expressions of relational algebra could be written by hand as definitions of Prolog predicates. By means of structures, we can also represent relational expressions directly so that we can write evaluators and analyzers for them. The following expression is a legal Prolog term:

```
union( join(r,s), minus( intersect(t,u), join(r,v))) )
```

Prolog assigns no interpretation to this term unless a programmer expresses one by means of predicate definitions. We may define an interpretation as a predicate `evaluate` which as its first argument takes a relational expression and as its second a representation of a set of tuples. An evaluator is typically compositional: A compound expression is decomposed into its parts, these are evaluated, and the results for the parts are combined into a result for the compound. The following clause could be part of an interpreter for relational algebra.

```
evaluate(union(A,B), Result):-
    evaluate(A, ResultA), evaluate(B, ResultB),
    make_union(ResultA, ResultB, Result).
```

A similar clause must be supplied for each operator in the algebra and the "semantic compositions" such as `make_union` should be defined in suitable ways. Here, the idea is that variables `ResultA` and `ResultB` eventually get bound to values representing two relations (i.e., sets of tuples) and `Result` represents the union of these two relations when `make_union` has finished.

5.2 Lists in Prolog

The list data structure is one of the most important data structures in programming and Prolog includes a specialized notation for working with lists. Lists in Prolog are internally represented by structures such as those we have seen above, and the extra notation is an example of syntactic sugar that makes things easier to read and write but does not add any new semantics. The following illustrates Prolog's list notation for representing the list of constants `a`, ..., `d` in that order:

[6] The reader may ponder over how many lines of Java code will be needed in order to implement the functionality embedded in the two-line `vertical-horizontal` program.

```
[a,b,c,d]
```

A list can be written directly in a program or query in this way; no tiresome sequence of calls to constructors is needed as in Java and similar languages.

Internally, a list is represented by means of binary function symbols that combine list head and tail. Most Prolog versions use a function symbol written as a period or dot sign "`.`", and the constant symbol "`[]`" for the empty list. The sample list expression above is just a convenient notation for the following:

```
.(a,.(b,.(c,.(d,[]))))
```

These two expressions are completely equivalent as demonstrated by the following query:

```
?- [a,b,c,d] = .(a,.(b,.(c,.(d,[])))).
yes
```

There is no special semantics built into the list notation. The way Prolog handles it is to translate any expression of form `[..., ..., ...]` into the form with the binary dot before trying to interpret it; when results are printed out, any dot expression is written in the list notation.

The list notation has a special form for matching the tail of a list in one piece, `[`*head*`|`*tail*`]`. It is illustrated by the following query:

```
?- [a,b,c,d] = [H|T].
H = a, T = [b,c,d]
```

The notation extends so that it can match any number of specific elements of the list and the remaining tail:

```
?- [a,b,c,d] = [H1,H2|T].
H1 = a, H2 = b, T = [c,d]
```

The traditional `member` predicate provides a good example of how to work with lists in Prolog:

```
member(X,[X|_]).
member(X,[_|L]):- member(X,L).
```

An element is member of a list if either it is the first element or it is member of the tail consisting of the remaining elements. The predicate can generate on backtracking the different elements of the list:

```
?- member(X,[a,b,c]).
X = a ?;
X = b ?;
X = c ?;
no
```

The reversibility principle (i.e., any argument can be used freely for input or output) means that `member` can also be applied for constructing lists:

```
?- member(a,L), member(b,L).
L = [a,b|_117]
```

The internal variable _117 represents the unknown tail in which any further elements would go as in the following:

```
?- member(a,L), member(b,L), member(c,L).
L = [a,b,c|_118]
```

The last recursive call assigns the structure written [c|_118] to the variable _117. On backtracking, a call of this form produces different results showing an infinity of possible ways that a, b, and c can be placed in a list.

Another example of a standard list predicate is the append predicate. A call append(L_1,L_2,L_3) is satisfied when the list L_3 consists of the elements of L_1 followed by those of L_2. It can be defined by the following two clauses:

```
append([], L, L).
append([X|L1], L2, [X|L3]):- append(L1, L2, L3).
```

The predicate recursively traverses the first list while gradually expanding a new list (in the third argument) with the elements of the first one. When it comes to the point where the first list ends with tail [], the second list is taken as the tail of the new list — and we have the combined list as result in the third argument. For example:

```
?- append([a,b],[c,d],L).
L = [a,b,c,d]
```

In order to understand the Prolog definition of append, the reader is proposed to simulate with paper and pencil the recursive calls and variable bindings involved in execution of this query.

The append predicate can also be used the other way round for splitting a list into two separate parts:

```
?- append(L1,L2,[a,b,c]).
L1 = [], L2 = [a,b,c] ?;
L1 = [a], L2 = [b,c] ?;
L1 = [a,b], L2 = [c] ?;
L1 = [a,b,c], L2 = [] ?;
no
```

The predicates member and append are built into most Prolog systems; in SICStus Prolog, they are part of a library that needs to be loaded explicitly. A source program file that uses these predicates should include the following directive:

```
:- use_module(library(lists)).
```

For playing with append and member without having a source program, one can load the library giving the use_module(library(lists)) directive to Prolog as a query (without the ":-").

5.3 A collection of more or less logical built-in predicates

5.3.1 Procedural test predicates

Prolog comes with a comprehensive collection of auxiliary predicates, many of which can only be understood in a procedural way. However, when used in a proper way they can provide an overall, logical program behaviour.

Some predicates test the actual status of a variable at the time the test predicate are called. An example is the `var` test that succeeds if its argument is an unbound variable and fails otherwise.

```
?- var(X).
yes
?- var(a).
no
?- X=a, var(X).
no
?- var(f(X)).
no
```

The `var` test is useful for the definition of a predicate where reasons of efficiency suggest that instantiated and uninstantiated arguments should be treated differently. There is a counterpart to `var` called `nonvar` that does exactly to opposite, i.e., `nonvar` fails on an unbound variable and succeeds on anything else.

We will not show sample programs here including these tests as they are more useful when combined with other mechanisms introduced in the following. Here is a collection of other such test predicates.[7] For a full catalogue of what is available, consult your Prolog manual.

`ground(`x`)` succeds if x is a ground term at the time of call, fails otherwise.

`atom(`x`)` succeds if x is instantiated to a constant symbol which is not a number at the time of call, fails otherwise.

`integer(`x`)` succeds if x is instantiated to an integer number at the time of call, fails otherwise.

`atomic(`x`)` succeds when one of `atom(`x`)` or `integer(`x`)` would succeed, fails otherwise.

x`==`y succeeds if x and y are identical terms, fails otherwise. This notion of identity should be taken literally and is stronger than the two terms being unifiable. For example, "`?- X==Y`" fails whereas "`?- X=Y,X==Y`" succeeds.

x`\==`y is the opposite of x`==`y; we already considered its use in section 2.5.

term =.. *list* is for splitting or synthesizing a term. The head of the list represents the function symbol and the tail, the arguments. Example: `f(a,b) =.. [f,a,b]`; To produce the term from the parts, write `Term =.. [f,a,b]` and to get the parts of the term, write `f(a,b) =.. [F|Args]`.

[7] Although these test devices do not conform with a mathematical logic notion of a predicate, they are called so due to their syntactic role in Prolog.

We remind here also about the very useful `dif`/2 built-in predicate (section 2.5) that applies its own principle of execution: It is delayed in the execution state until its arguments are sufficiently instantiated as to make a final judgement. So although `dif` probably is slightly less efficient to use than `\==`, it has the advantage of preserving a logical semantics in all cases, including programs that are not range-restricted.

We have already seen, in section 2.5, a test that compares numbers x`>`y. In case the terms represented by x and y are not instantiated to numbers at the time of call, this results in an error message. Prolog includes also x`<`y, x`>=`y, and x`=<`y. These tests are, in fact, a bit more general than indicated here as they to a certain extent can interpret correctly arithmetic expressions as arguments; we return to this topic in section 5.3.3 where we give a general introduction to arithmetic in Prolog.

5.3.2 Control

As we have described, Prolog executes things in a specific order: Sequences of goals from left to right, and clauses are applied alternatively on backtracking in their textual order.

Prolog has some control devices that make it possible to interfere with this. We have already seen, page 8, the semicolon which logically reads as an "or" and which procedurally means to set up an alternative to be executed under possible backtracking.

A control device that Prolog programmers use all the time is the so-called *cut* written as an exclamation mark.[8] A cut can be placed in a rule as an ordinary call to a predicate. When executed, it succeeds immediately without any noticeable effect, but it strikes in case of backtracking. We show an example:

```
salary(X,0):- student(X), !.
salary(X,1000000).
student(peter).
```

The meaning of the cut is that when trying to re-execute it under backtracking, it not only fails but it enforces the entire call to the predicate in the head of the clause (here `salary`) to fail. Let us examine a few queries to this program:

```
?- salary(peter, S).
S = 0 ?;
no
?- salary(jane, S).
S = 1000000 ?;
no
```

The call `salary(peter, S)` fits with the first clause, the subgoal in the body `student(peter)` succeeds and the cut is executed. When the user ask for possible alternative solutions (by typing semicolon), the interpreter tries to redo the cut with the resulting effect that the original call `salary(peter, S)` fails. For the second test query, with `salary(jane, S)`, the subgoal `student(jane)` fails so the control is transferred to the next clause and the cut is not in effect.

This shows a common use of the cut used for setting up exceptions to a general rule. In the example above, the cut serves to filter out students that are given a special treatment; only the

[8] It has nothing to do with the proof rule in mathematical proof theory that is called the cut rule.

first clause goes for them, they cannot be handled by the general rule that goes for all others. This is a common pattern that occurs often in Prolog programs and we can sketch it as follows.

```
  p(···):-  identify-special-case$_1$ , ! , special-treatment$_1$ .
     ⋮
  p(···):-  identify-special-case$_n$ , ! , special-treatment$_n$ .
  p(···):-  treatment-of-all-other-cases.
```

We may also refer to the last rule as the *default* rule as it is the one to rely on when no special case is identified.

Care should be taken when cut is used as its procedural nature implies some restrictions on how the predicate should be called. As it is defined above, the `salary` predicate works only in a satisfactory way when called with a ground first element. See what happens otherwise:

```
?- salary(X, S).
X= peter, S = 0 ?;
no
```

The first clause is applied, `student(X)` succeeds, binding `X` to `peter`, the cut is activated so that no other results can be produced. This is logically wrong because when we ask directly for the salary of `jane` we get an answer that we do not get when trying to ask for the salary of everyone (under backtracking). So the definition of `salary` is fine only in case the programmer takes care only to apply the predicate with `nonvar` first argument.

If needed, the undesired effect of the hack using cut to get defaults-with-exception can be partly removed by another hack. Consider the following rule added to the program before the other two.

```
salary(X,S):- var(X), !, (X = peter, S = 0 ;  S = 1000000).
```

This partly solves the problem:

```
?- salary(X, S).
X= peter, S = 0 ?;
S = 1000000 ?;
no
```

We got one answer for `peter` and another one which embeds an infinity of solutions, including one concerning `jane`. But this is still not perfect, as there is no way to indicate that the second answer does not concern `peter`.

In most practical cases, these imperfections do not cause problems, because in practice, the problem domain often implies some inhererent restrictions on how it is "natural" to use a specific predicate.[9]

But as our interest in Prolog is motivated by database applications, we will stay with the salary example for a while. The logical database expert immediately identifies the problem of the `salary` program as being lack of range-restrictedness in the second clause. The way to repair it is to introduce a predicate explicitly defining non-students. The following version of the program that does not include any of Prolog's procedural bells and whistles is more satisfactory from a database point of view:

[9] However, the author of this text, being an experienced Prolog programmer, knows by experience that some of those program bugs that are most difficult to locate are when a predicate is called differently than originally thought.

```
salary(X,0):- student(X).
salary(X,1000000):- director(X).
student(peter).
director(jane).
```

In case we prefer to use negation-as-failure for non-studentness, we need to have a predicate that so to speak defines the range of the X in the second clause, e.g.,

```
salary(X,1000000):- person(X), \+ student(X).
```

Prolog includes predicates fail and true that respectively always fails and always succeeds. A common combination in a Prolog program is "!, fail" used to express that a give predicate does not hold for specific categories. As an example of this, we show how Prolog's negation by failure in principle could have been defined:

```
\+(X):- X, !, fail.
\+(_).
```

We use here a feature of Prolog that a term, here passed through the argument X, can be executed as a call as indicated. The definition reads "if X succeeds, \+(X) fails; otherwise it succeeds without doing anything." This is why Prolog's sort of negation is also referred to as *negation-by-default.*

The final control structure that we show is the conditional that we introduce by an example.

```
salary(X,S):-
  student(X) -> S=0
  ;
  director(X) -> S=1000000
  ;
  professor(X) -> S=500000
  ;
  S = 10.
```

When this rule is applied, the conditions (in front of the arrows) are checked from above; the first one that succeeds determines which alternative is chosen. The last case, without an explicit condition, is chosen if none of the tests succeeds. In case of backtracking, no alternative branch in the conditional is tried, but the original salary call does not fail as is the case when cut is used. If there were subsequent clauses for salary they would be tried out as well. Notice, however, that this use of the conditional shares the property of some of the previous formulations that it only works in a sensible way when the first argument is given in the call to salary.

5.3.3 Arithmetic in Prolog

Arithmetic is treated as a stepchild in Prolog. The reversibility principle does not hold for Prolog's way of doing arithmetic. Consider an equation such as $x = y + 2$; if $y = 2$ we would expect a logical interpreter to figure out that $x = 4$ and, the other way round, if $x = 10$ we would expect it to set $y = 8$. Prolog's primary handle to arithmetic is a built-in predicate is/2 written between its two arguments. An example of its use is the following.

```
X is Y+2
```

If Y is instantiated to a number, say 7, when this subgoal is encountered during the execution of a program, it can evaluate and in this case, as expected, instantiate X to 9.

In case Y is uninstantiated, an attempt to execute this goal results in an error message (even if X is instantiated to a number).

Any arithmetic expression using standard notation can be written to the right of "is"; check your Prolog manual for the precise details if necessary. Provided all variables are instantiated to numbers, and no division by zero or the like occurs, the value of the expression is unified with the term in front of "is". This means that programs using arithmetic in Prolog must satisfy a generalized form of range-restrictedness that also specifies a type of numbers for the variables involved in aritmetic.

Arithmetic expressions can also be used in goals formed by comparison operators <, >, =<, and >=. For example, the subgoal X+2<1+Y*3 succeeds if X=10 and Y=4 at the time of call. Special versions of equality and nonequality operators that accept arithmetic expressions are available, written =:= for equality and =\= for nonequality. These predicates share the property of "is" that all variables need to be properly instantiated at the time of call.

The following predicate uses arithmetic in order to calculate the length of a list.

```
length1([], 0).

length1([_|L], N):-
   length1(L, M),
   N is M + 1.
```

The definition is straightforward: When given a list as first argument, it evaluates recursively the length of the tail and adds one as to get the length of the whole list.

However, consider a case where the first argument in a query is a variable, and the second a specific number, e.g.:

```
?- length1(L,3).
```

We would expect the result that L is assigned a value corresponding to a list of length three such as [X,Y,Z] with unknown elements. This is indeed the case. The first call matches the larger of the clauses (the rule), and let us assume that variables in the clause are renamed by putting "1" onto their names. Unification of query with head of goal results in L=[_|L1], N1=3 and the recursive call is thus length1(3, M1); this call matches the first clause which sets L1=[] and M1=0; returning from this call back to the rule, we encounter 3 is 0+1 that fails as 0+1 is 1 that does not unify with 3; now the Prolog system backtracks, trying to redo the aforementioned recursive call by using the rule instead. Renaming next level of variables by putting 2 onto their names, this leads to a recursive call length(L2,M2) which returns after first attempt with M2=0; addition takes place setting M1=1 and control returns to N1 is M1+1 which now is instantiated to 3 is 1+1; it fails and more backtracking is needed until finally the right combination of choices are found so that the pending chain of "is" goals all succeed. Obviously, there is a combinatorial explosion and a query such as ?- length1(L,5000) takes very, very long time to execute, but the predicate is fully reversible — in a logical sense — as it generates the right results independently of how its arguments are instantiated or not instantiated.

We will use this example to demonstrate how Prolog's nonlogical control devices can be applied for optimizing a predicate so it runs much faster and still retains a logically satisfactory behaviour.

The discussion of the `lenght1` predicate suggests that we use a different strategy when the list argument is unknown and the length given. This is what we do in the following alternative definition of a predicate that we call `lenght2`.

```
lenght2(V, N):-
    var(V), nonvar(N), !,
    generate_list(V, N).

lenght2([], 0).

lenght2([_|L], N):-
    lenght2(L, M),
    N is M + 1.

generate_list([], 0):- !.

generate_list([_|L], N):-
    M is N - 1,
    generate_list(L, M).
```

We use the default-with-exception pattern by means of a cut as explained in section 5.3.2 to filter out the special cases (unknown list and known length) and call a specialized predicate, called `generate_list`, specifically optimized for such queries. For all other "normal" cases we do the same thing as in the previous definition of `lenght1`.

The length predicate is standard in most Prolog implementations under the name of `lenght`, and it is easy to check whether it uses a `length1` or a `length2` style of definition.

5.3.4 Finding all solutions to a query

Usually Prolog returns one solution at a time, but there are facilities to collect all solutions to a given call into a list. We give here a simplified presentation in terms of examples that are sufficient for the applications we make in what follows; see your Prolog manual if you need more details. The standard predicate `setof` is called with arguments as follows:

`setof(`*pattern*`,` *goal*`,` *result-list*`)`

Each successful execution of *goal* gives rise to an instance of *pattern*, and *result-list* is the list of all possible such instances of *pattern*. The following shows a call and the answer; notice that the second argument is a compound goal surrounded by parentheses.

```
?- setof(X, (member(X,[1,2,3,4]), X>2), L).
L = [3,4]
```

The `setof` predicate returns a list without duplicates, so if a solution can be generated in two different ways, it is included only once in the result:

```
?- setof(X, (member(X,[1,2,3,4,1,2,3,4]), X>2), L).
L = [3,4]
```

In a database context, we can use it for getting an explicit representation of the contents of a given relation. Consider the following database:

```
father(john, mary).
father(john, karen).
father(paul, john).
```

The following query generates the relation:

```
?- setof(tuple(X,Y), father(X,Y), FatherRel).
FatherRel = [tuple(john,karen),tuple(john,mary),tuple(paul,john)]
```

The use in the goal part of variables that do not occur in the pattern part is a bit complicated. Consider the following query together with three alternative answers generated:

```
?- setof(X, father(X,_), FatherRel).
FatherRel = [paul] ? ;
FatherRel = [john] ? ;
FatherRel = [john] ? ;
no
```

The variable indicated with the underline is not in the pattern, so the three answers demonstrate three different ways to instantiate it (to a child in the example), and each consisting of a of fathers `X` for each such child. As in any database satisfying reasonable integrity constraints, these lists are of length one in this particular example.

A notation for existential quantifier can be used if we want to generate a list of anyone who is father to someone. The following example shows its use:

```
?- setof(X, Y^father(X,Y), Rel).
Rel = [john,paul] ? ;
no
```

In fact, `setof` returns a list of solutions ordered according to a standard ordering on terms denoted "`@<`". How this ordering is defined is not important and is not described here, and in the applications we make of `setof` here we do not make use of the sortedness property. However, for the programmer who wants to write programs optimized for efficiency, this is useful to be aware of.

There is a companion to `setof`, called `bagof`, that works the same way except that solutions appear in the order they are found and may as a consequence contain duplicates:

```
?- bagof(X, (member(X,[1,2,3,1,2,3,4,4,3,2,1]), X<4), L).
L = [1,2,3,1,2,3,3,2,1]
```

One unfortunate property of `setof` and `bagof` as seen from a database viewpoint is that they fail in case the list of solutions is empty. In a database, it is not a failure that a relation is empty, or that some expression evaluates to an empty set of tuples: The empty set of tuples is a perfectly sensible value that can be combined with other relations in different ways. In case you know the result is deterministic, i.e., there is only one solution to be produced by a call to `setof` or `bagof` — as it will be in a database context — we can easily fix this by means of the conditional construct described in section 5.3.2:

```
( setof(pattern, goal, List) -> true ; List = [])
```

Finally, there is the `findall` construct that works similarly to `bagof` but differs in three ways: It can return the empty list of solutions, it treats any variable in the goal part not occurring in the pattern part as existentially quantified, and finally, it does not backtrack for solutions. Example:

```
?- findall(X, father(X,_), FatherRel).
FatherRel = [john,john,paul]
```

The `findall` mechanism is far the most efficient, so if we can live with the duplicates it may be the one to prefer. If we want our programs to be faithful to a set-based semantics of databases, `setof` together with the trick above to handle empty relations is the one to use.

5.3.5 Input/output

Normally, it is not necessary to describe explicit input or output for a Prolog program as all communication is done by asking queries and having the system print out values for variables.

Prolog contains a collection of auxiliary predicates to do all standard things such as reading and writing to/from files or the terminal — everything is there, and you can find it in the Prolog manual.

However, an extremely useful predicate is `write`/1 which can be used for all sorts of test prints. The Prolog system's debugging facilities are quite flexible but sometimes difficult to use. Instead it is often much easier to add calls to `write`. The predicate takes as argument any Prolog term and prints it out in a standard format. Variables are printed using internal numbers, so they are somewhat difficult use. Assume we have a predicate `p` and that we add the following clause to the program following all other clauses for `p`:

```
p(X):- write(p(X)), write(' failed'), fail.
```

In case the original definition of `p` does not use cut, this rule is applied exactly when all other rules have failed, and this rule also fails. Thus it does not change the overall behaviour of the program, but it has printed a report that `p(X)` did fail for a particular `X` during execution. In order to have line breaks in your printouts, you may use the `nl` predicate.

5.4 Having programs to inspect and modify themselves during execution

It is difficult to imagine a version of Java in which a program can consult its own source text and modify itself while running. Well, it will be possible to write a Java program that reads its own source text from a file, uses some standard parsing algorithm and builds a syntax tree. This syntax tree can be modified and transformed, printed to a file which, then, can be compiled and activated. This seems to be a difficult process, at least not very elegant, and it is not a dynamic modification of the program while its actually running. In Prolog things are different:

- Programs and data are in a homogeneous format; it is difficult to see the difference between the program fact `p(a,b)` and the term (i.e., the piece of data) `p(a,b)`.

- Prolog programs were introduced in the previous sections as databases; thus a program is a database, i.e., a particular sort of data structure. Executing a program is explained as an evaluation of the information in that data structure.

- ... and Prolog includes a few handles to access the database≈program directly as we show in the following.

It is quite obvious that such facilities have some inherent semantical problems but are nevertheless very useful when used in a proper way.

- We can implement database updates, and on top of that write elaborate pieces of code that determine which updates should be made.
- We can implement evaluation mechanisms that are different from Prolog's standard evaluation mechanisms.

For reasons of efficiency, it is necessary to inform the Prolog system that a given predicate will be inspected and modified; this allows the Prolog system to apply all sorts of advanced optimization techniques for predicates that do no use these facilities. The following directive given as part of a Prolog source text makes it possible to inspect and modify the `father` predicate.

```
:- dynamic father/2.
```

New clauses can be added to a predicate while it is running by means of two predicates `asserta` or `assertz`. The ...`a` means to add it as a new first clause for the predicate, and ...`z` as a new last clause. Assume that initially the `father` relation consists of a single fact `father(john,karen)`; the following dialogue shows the machinery at work.

```
?- father(X,Y).
X = john, Y = karen ?;
no
?- asserta(father(john,mary)), assertz(father(john,paul)).
yes
?- father(X,Y).
X = john, Y = mary ?;
X = john, Y = karen ?;
X = john, Y = paul?;
no
```

You should be aware of the following:

- Modifying a running program does not change its source text on the file.
- There is no way to insert a new clause in the middle of a predicate definition.[10]
- In case of backtracking, `asserta` and `assertz` do not undo the modification of the program.
- The effect of an `asserta` or `assertz` takes place immediately, so in the evaluation of a complicated query that changes some predicate, evaluation of subqueries to that predicate will reflect the changes.

[10] If, for some strange reason, you want to insert a new clause in the middle of a predicate definition, you have to first take out all clauses (using `retract`) and then `assertz` (or `asserta`) them again in a proper order together with the new clause.

- There is a listing predicate that prints out the current content of the database; "?- listing." gives you the whole database, and "?- listing(father)." only the indicated predicate.

Rules can also be asserted as in this example:

```
?- assertz((father(X,Y):- adopted(X,Y), \+ dead(X))).
```

Notice the extra parentheses which are needed in order to avoid Prolog's parser getting confused by the ":-" operator and the comma.

The use of asserta and assertz implies the same sort of problems caused by implicit quantifications as those experienced for Prolog's version of negation-as-failure. Whether an occurrence of a variable indicates a variable in the new clause or is used by the surrounding program in order to synthesize a clause depends on the instantiation of variables. Consider this example (that does not seem to preserve a good meaning of the father predicate):

```
?- asserta(father(X,Y)), X=mary, Y=karen, asserta(father(X,Y)).
```

It results in the addition of two facts, one telling that anyone is father of anyone and another one telling that mary is father of karen.

To remove clauses, we may use a predicate retract. A call retract(*pattern*) identifies and removes the first clause in the database which, when viewed as a piece of data, unifies with *pattern* possibly binding variables in *pattern*. In case father(john, mary) is the first father clause in the database, it works as follows:

```
?- father(john, mary).
yes
?- retract(father(X,Y)).
X = john, Y = mary?
yes
?- father(john, mary).
no
```

Notice that retract under backtracking removes other clauses if possible (and not undoing previous modification). So the following query is a way of removing all facts about john's children:

```
?- \+ (retract(father(john,X)), fail).
yes
```

This example shows a way that Prolog programmers often write loops and which is a bit confusing for the newcomer. The outermost "\+" instructs Prolog to search for a successful way of execution to query "retract(father(john,X)), fail". It executes the retract removing some clause and continuing with the explicit fail leading to backtracking; Prolog tries to re-execute retract removing yet another clause, and this continues as long as there are clauses matching the pattern father(john,X). When no more such clauses are present in the database, retract fails, and thus the whole "\+" goal succeeds because Prolog could not find a success for "retract(father(john,X)), fail".

When trying to delete a rule for a predicate, we should indicate the structure of that rule. If, e.g., we wanted to remove the father rule shown above, we may attempt retract((father(_,_):-

`_,_))`. If we wrote `retract((father(_,_):- Body))` it is possible that the pattern matches a fact with the result `Body=true`. So when using `retract`, care should be taken as to insure that the intended clause is removed and not another one.

The final predicate in this toolbox is the `clause`/2 predicate that makes it possible to inspect a clause of a dynamic predicate without destroying it. Notice the stylistic asymmetry in that `clause` takes two arguments, one for head and one for body, whereas the modification predicates takes one argument representing a whole clause. To see it at work, assume that the database includes a `grandfather` rule, and consider the following query:

```
?- clause(grandfather(A,B), Body).
Body = father(A,_117),father(_117,B) ?
```

As it appears, the system locates a clause (the first one) that unifies with the pattern; notice that a unification has taken place so that the `A` and `B` variables in the query have been substituted into body returned. In the following example, it is shown how a ground data value is being substituted into the body, so that we achieve a specialized version of that body, effectively specialized so it could be applied in the search for a solution to `grandfather(peter,B)`.

```
?- clause(grandfather(peter,B), Body).
Body = father(peter,_117),father(_117,B) ?
```

On backtracking, the `clause` predicate will go through the possible clauses that matched the specified pattern. This behaviour is perfectly suited for writing an interpreter of Prolog. The following little Prolog interpreter written in Prolog is known in the logic programming folklore under the name of Vanilla:

```
solve(true):- !.

solve((A,B)):-
    !,
    solve(A),
    solve(B).

solve(A):-
    clause(A,B),
    solve(B).
```

It just replicates the operations performed by Prolog when executing a program and preserves the order in which they take place. Notice however, that Vanilla does not understand all of Prolog as it is given here; most remaining parts but the cut can easily be added. The following dialogue shows its use and that it returns exactly the same answers as Prolog.

```
?- father(X,Y).
X = john, Y = mary ? ;
X = john, Y = karen ?
yes
?- solve(father(X,Y)).
X = john, Y = mary ? ;
X = john, Y = karen ?
```

The reason for the two cuts in the Vanilla program is to prevent the interpreter on backtracking to attempt to find clauses about "`true`" and comma (in which case the Prolog system complains).

However, with a little care this can be expressed without cut, and if we also tell the interpreter how to handle the `clause` predicate by this rule:

```
solve(clause(H,B)):- clause(H,B).
```

it is even capable of interpreting itself:

```
?- solve(solve(father(X,Y))).
X = john, Y = mary ? ;
X = john, Y = karen ?
```

You have probably asked yourself by now the following question:

Why is this interesting? What can Vanilla be used for?

The answer is simple: You can hack Vanilla!

In more precise words, Vanilla is a program which is an explicit description of how a query is executed, i.e., a specification of the semantics of the language being executed (in this case a Prolog subset). And a program is something that you can modify, which means that *you* can change the semantics. You can change the overall meaning so that different answers will be produced, or you may confine yourself to changing the evaluation strategy. As a simple example, Prolog's left-to-right strategy can be replaced by a right-to-left strategy by changing Vanilla's rule for comma into the following one.

```
solve((A,B)):-
    !,
    solve(B),
    solve(A).
```

The following references have applied modified Vanilla interpreters for describing flexible query evaluation that apply knowledge about the data to change the query. As an example, if you ask your database for a list of dogs but there are no dogs, the interpreter may relax the query to look for any animal (going upwards in a taxonomy) or change it into a query for cats (going sideways in the taxonomy, suggesting another kind of animal which is close to dog as both qualify as pets).

- Andreasen, T., Christiansen, H., Flexible query-answering systems modelled in metalogic programming. *ECAI'96 workshop "Knowledge Representation Meets Databases"*, August 13, 1996, Budapest, Hungary, pp. 1-7.

- Andreasen, T., Christiansen, H., Nonstandard database interaction from metalogic programming. *Flexible Query Answering Systems*, Kluwer Academic Publishers, 1997. pp. 61-78.

- Gaasterland T., Godfrey P., and Minker J., Relaxation as a Platform for Cooperative Answering. *Journal of Intelligent Information Systems*, 1, 3/4, pp. 293-321, 1992.

The following references have used it for implementing a kind of counterfactual reasoning in a database *à la* "If I were rich, should I buy a red or a yellow Ferrari, and whom should I invite on a trip to Barbados?" — or perhaps more useful, if someone is afraid of flying: "How can I plan my travel to go from here to Barbados but avoiding planes" which can be rephrased "If it were the case that there were no flights, ..."

- Andreasen, T., Christiansen, H., Hypothetical queries to deductive databases. Eds. Geske, U., Ruiz, C., Seipel, D., *Proc. of the 5th International Workshop on Deductive Databases and Logic Programming. Workshop in Conjunction with ICLP'97, Leuven, Belgium, July 11, 1997.* GMD-Studien 317, GMD-Forschungszentrum, Informationstechnik GMBH, pp. 37-48, 1997.

- Andreasen, T., Christiansen, H., Counterfactual exceptions in deductive database queries. *12th European Conference on on Artificial Intelligence*, ECAI'96, August 11-16,1996, Budapest, Hungary. pp. 340-344.

In the following textbook we have shown how tracers and debuggers for Prolog and other programming languages can be implemented by extending an interpreter (such as Vanilla for Prolog) in straightforward ways.

- Christiansen, H. *Sprog og abstrakte maskiner, 3. rev. udgave [in Danish; eqv. "Languages and abstract machines"]*, Datalogiske noter 18. Roskilde University, Denmark, 2000.

5.5 Syntactic extensibility

Basically, any expression in Prolog is a term written according to the syntax we indicated at page 28. The familiar `grandfather` rule can in fact be written in a program fully correctly as follows:

```
:-( grandfather(X,Z), ','(father(X,Y), father(Y,Z)) ).
```

Thus "`:-` behaves syntactically as a function symbol of arity two and the same thing can be said about the comma that combines the two goals in the body.

And, in fact, "`:-`" that we have learned is part of the syntax for rules, can be used as a function symbol, which we have seen already in the previous section 5.4.

Prolog's syntax includes a notion of *operators*, and an operator is a function (or predicate) symbol whose arguments can be indicated by positions before or after without bracketing. A binary symbol can be declared to be an *infix* operator meaning that it can be placed between its arguments. We have already seen "`:-`", "`is`", "`+`", "`-`" etc., and "`<`", "`>`" etc. Unary symbols can be declared as prefix or postfix operators meaning that their argument is written after, resp. before the symbol. We have seen prefix operators "`:-`" applied in directives to the Prolog system, and the negation symbol "`\+`"; prefix "`+`", "`-`" can be used as well in aritmetic expressions. Postfix operators are rare, but possible. As it appears, the same symbol may serve as more than one operator.

There is a special directive `op` by means of which you can define your own operators, and the predefined operators have also been defined in this way. A selection of the operators used for arithmetic is defined as follows.

```
:- op(700,xfx,is).
:- op(500,yfx,+).
:- op(500,fx,+).
:- op(500,yfx,-).
:- op(500,fx,-).
:- op(400,yfx,*).
```

The colon-dash operators used for clauses and directives are defined in the following way.

```
:- op(1200,xfx,:-).
:- op(1200,fx,:-).
```

We give a brief sketch of how these declarations should be understood. Each of `fx`, `fy`, `xf`, `yf`, `xfx`, `xfy`, and `yfx` is called the *associativity* of the given operator and it is to be understood as a pictogram. The `f` indicates the position of the actual operator. The number associated with each operator is called its *precedence*. To understand how they are used, we define for each written term, a precedence number.

- Precedence number 0 (zero) is given each term which is: a variable, a constant, a term in standard notation *function*$(\cdots)$, a parenthesized expression (not immediately preceded by a function symbol).
- A term has precedence number n whenever is written by means of an operator with precedence n.

In the pictogram,

- an `x` indicates a term with precedence less than or equal to the precedence of the operator indicated by the `f`, and
- a `y` indicates a term with precedence strictly less than the precedence of the operator indicated by the `f`.

With the definitions for the arithmetic operators above, it appears that the text “`1-2-3`” only can be read equivalently to “`(1-2) - 3`”: Binary infix minus has associativity `yfx` and precedence 500. The arguments of the first minus are constants of precedence 0 and thus associativity obeyed for it; so the first three characters “`1-2` can be read as a term with precedence number 500. The second minus is, thus, preceded by a term of precedence 500 (equal to and thus also less-than-or-equal to 500) and followed by a constant which has precedence 0 (strictly less that 500); this means that associativity `yfx` for the second minus also is satisfied. With similar arguments, it is easy to show that a reading of “`1-2-3`” equivalent to “`1 - (2-3)`” is wrong.

The fact that “`*`” has a lower precedence number that “`+`” and “`-`” means than “`1+2*3`” only can be read as “`1 + (2*3)`”.

There is a very useful predicate called `current_op` that can be used for checking the actual associativity and precedence of an operator, whether it be defined by the programmer or predefined in the Prolog system:

```
?- current_op(X,Y,<).
X = 700,
Y = xfx
```

Here we got the definition for the comparison operator “`<`” and this leads us to a good advice on how to design our own operators. Reason by analogy. Suppose I want to define a comparison operator “`<<`” for defining my own predicate that means “much less than”. A new comparison operator should behave syntactically in the same way as any other comparison operator. To achieve this, I simply copy the declaration for “`<`” and I can define the meaning of my predicate.

```
:- op(700,xfx,<<).
X << Y :- X < Y+1000.
```

The following example program shows how operators can be applied for writing a program in a way so it resembles natural language. The program defines a small taxonomy by means of an "is a" relation. However, Prolog does not allow us to redefine the meaning of "`is`" so the program refers instead to the Danish language; the word "er" applies to all genders and numbers and means "is" or "are". There are undetermined articles, "en" for feminine+masculine and "et" for neuter. Non-Danish speakers may try to guess the meaning of the involved nouns.

```
:- op(700, xfx, er).
:- op(100, fx, [en,et]).

en mand er et menneske.
en kvinde er et menneske.
et menneske er et dyr.
en ko er et dyr.
peter er en mand.
X er Z :- X er Y, Y er Z.
```

The program is queried as follows:

```
?- peter er X.
X = en mand ? ;
X = et menneske ? ;
X = et dyr ? ;
! Resource error: insufficient memory
```

The program responds with the right possible categories to which `peter` belongs, and asking for yet another solution leads to an infinite loop. It is left as an exercise below, to explain and possibly correct the problem.

Finally we mention a predefined predicate `write_canonical`/1 which is very useful for checking how a set of operator declarations works. It takes any term as argument and prints it in the standard syntax. Example:

```
?- write_canonical(peter er et menneske).
er(peter, et(menneske))
yes
```

5.6 Exercises

Exercise 5.1 Evaluate by hand and check on the computer, which answer Prolog will produce for the following query given the simplest version of the `horizontal-vertical` program (consisting of two nonground facts).

```
?- horizontal(L), vertical(L).
```

Exercise 5.2 Extend the geometry program of section 5.1 with predicates concerning triangles and squares (which means you have to devise a representation of these objects):

`identical_triangles(`*triangle*$_1$`,` *triangle*$_2$`)` which holds if and only if the two arguments represent the same triangle.

`identical_squares(`*square*$_1$`,` *square*$_2$`)` which holds if and only if the two arguments represent the same square.

`split_squares(`*square*`,` *triangle*$_1$`,` *triangle*$_2$`)` which holds if and only if the first argument represents a square which can be split into two triangles represented by the two last arguments.

`segment_length(`*line-segment*, *length*`):` First argument is a line segment and second argument is its length.

`area(`*object*, *area*`):` First argument is any point, line segment, triangle, or square, and second argument is its area.

Does it make sense to make the two last predicates reversible?

Exercise 5.3 Evaluate by hand and check on the computer, which possible answers Prolog will produce for the following queries.

```
?- append([a,b],L1,L2).
?- append(L1,[a,b],L2).
?- append(L,L,L).
```

Exercise 5.4 Consider exercise 4.1, page 26, which concerned a small database of personal information. Introduce an integrity constraint that checks that registration numbers for males are odd and that registration numbers for females are even.

Exercise 5.5 Lists can be used for representing sets. Recall that a set cannot contain duplicates, i.e., if some element a is in a set S, then there is no sense in asking whether it is in S one or several times. We consider set representation as lists without duplicates in which the order of the elements does not matter.

- Write a predicate `make_intersection/3` that takes two lists (supposed to represent sets) and produces a list that represents their intersection. Consider two versions, one defined recursively and one defined using only `setof` and `member`.
- Same question for predicates `make_union` and `make_difference` that evaluates the union, resp., difference of two sets; it is sufficient to write the `setof` style definition.
- Consider how a representation of sets by sorted lists could make it possible to provide more efficient implementation of these operations.

Exercise 5.6 The way we have seen until now to represent a database in Prolog is to write each tuple in a tabular relation as a set of facts and having views defined as predicates that can return one tuple at a time. This and the following exercise concern a different representation of a database which explicitly stores and manipulates lists. The solutions to the previous exercise are useful.

Consider your solution to exercise 4.1, page 26. Rewrite this program so that each database predicate, tabular as well as view predicates, take one argument representing a set (i.e., list without duplicates). We indicate the principle by showing a possible definition for `person` by means of a single fact.

```
person( [ tuple(3001121117, 'Peter Jensen', 'Villavej 8', 1170, male),
          tuple(1411190074, 'Mary Jensen', 'Villavej 8', 1170, female),
          tuple(1712653047, 'Jens Petersen', 'Bynkevej 412', 7400, male)
        ]).
```

Exercise 5.7 Use `assertz` in order to write a predicate `materialize`/3 that takes as argument two predicate names and their common arity. The first predicate name indicates an existing predicate, defined perhaps as a view, and the second one a new predicate name that will represent a tabular version of the predicate once `materialize` has finished. For example,

```
?- materialize(grandfather,materialized_grandfather,2).
```

will lead to the creation of a number of facts for the `materialized_grandfather` providing the same solutions as the `grandfather`. Be aware that for the solution to work, that `materialized_grandfather` needs to be declared as dynamic predicates. Notice that the good solution to this exercise is very short.

Exercise 5.8 This question concerns *aggregate values.* Assume we have a database written as a Prolog program which consists of facts of the form `income(`*id*, *integer*`)` and `expense(`*id*, *integer*`)`, where *id* is some key identifying each item, and the second argument an integer value representing some income or expense in a budget. Define a predicate `balance`/1 as a view in terms of the two other predicates that gives the sum of all incomes minus sum of all expenses; define an integrity constraint in Prolog saying that the value of `balance` always should be greater than or equal to zero.

Exercise 5.9 Consider the Danish taxonomy program at page 46. Why did the sample query run into loop when asked for one more solution than the three logically correct ones?

Any suggestion for how to avoid this problem?

Exercise 5.10 Design a set of operator definitions for representing expressions of relational algebra. You may decide to reuse existing operators, e.g., "`+`" for ∪, "`*`" for ∩. For natural join, it may be suggested to use "`><`" and for selection "`where`".

Exercise 5.11 *To be included. Write evaluator for relational algebra expressions. Without explicit schemas and join. Single-tuple at a time version.*

Exercise 5.12 *To be included. Extend previous with schemas and join.*

Exercise 5.13 *To be included. Change two previous to work on sets.*

6 Playing with updates and integrity checking

Here we give a very short introduction to how the Prolog facilities described until now can be put together so that we can produce a functionality that resembles a database application with updating and integrity checking. We only sketch principles and the idea is that the reader should work things out in detail in the exercises.

We consider a database of `father`/2 and `exists`/1 facts. The following integrity constraints are in action.

```
ic_violated:- father(X,Y), father(Z,Y), Z \== X.
ic_violated:- father(X,_), \+ exists(X).
ic_violated:- father(_,Y), \+ exists(Y).
```

It is assumed that the two predicates have been declared as `dynamic`.

6.1 A straighforward and inefficient implementation of integrity enforcement

As discussed already, we can check integrity of a given database by asking the query "`?- ic_violated`". If it fails, the database is consistent, otherwise there is some combination of tuples in the database that violates some integrity constraints.

We should not allow the user to update the database by directly asserting or retracting facts, and we trust that the user only does so by the predicates we devise in the following.

A straightforward way to evaluate a suggested update is to assert it into the database, check integrity, and if there is conflict, remove it again.

Here is a suggestion for a predicate intended to be used for adding tuples to the `father` relation.

```
add_father(X,Y):-
    asserta( father(X,Y) ),
    (ic_violated -> retract( father(X,Y) ), write('Rejected')
     ;
     write('Accepted')).
```

Adding to the other relation can be done in a similar way, and deletion can be done by predicates defined in a quite similar way.

We leave it to the exercises to consider the following issues:

- Adding a tuple that is already in the database should be avoided.
- Only constants symbols should be allowed as arguments.
- The user would appreciate a message more informative than just `Rejected`.

6.2 Simplified integrity constraints

The way for checking integrity shown above does not seem very optimal:

- Each time a single new tuple is suggested, we run a procedure that checks all possible combinations of tuples for a possible violation.
- The fact that the database was checked at the time of the previous update is not utilized at all. Those parts of the database that remain unchanged are checked over and over again.

A principle called *simplification* can be applied to improve this. We give here only a brief introduction and an example, and later in the course we go into more details.

The idea is to assume the invariant property that the database is consistent before an update. For given integrity constraint and update, say "add `father(peter,paul)`", a *simplified* integrity constraint is a specialized one which

- can be checked in the present state of the database without actually performing the update,
- it checks, under the assumption that the present state is consistent, whether or not the suggested update introduces a violation,
- and it considers only the smallest set of tuple combinations that is necessary consider.

It is practical also to assume an invariant that a tuple to be inserted is checked in another way not to be in the database already (and for deletions, that the tuple to be deleted actually is in the database).

For "add `father(peter,paul)`" and these integrity constraints:

```
ic_violated:- father(X,Y), father(Z,Y), Z \== X.
ic_violated:- father(X,_), \+ exists(X).
ic_violated:- father(_,Y), \+ exists(Y).
```

we postulate the following.

```
ic_violated_by_add_father_peter_paul:-
      father(_,paul).
ic_violated_by_add_father_peter_paul:-
      \+ exists(peter).
ic_violated_by_add_father_peter_paul:-
      \+ exists(paul).
```

With the original integrity constraints, in principle $O(n^2)$ combinations of tuples need to be considered for the first one and $O(n)$ tuples for each of other ones; n is the size of the database. With suitable indexing techniques applied by the underlying system, it should be possible to evaluate all in $O(n)$ time. However, as a realistic database may contain millions and millions of tuples, $O(n)$ is far too high.

On the other hand, the simplified version executed by "`?- ic_violated_by_add_father_peter_paul`" obviously runs in constant time when standard indexing is assumed.

The simplified integrity constraint can be lifted so they do not apply to `peter` and `paul` only but to any suggested, new `father` tuple:

```
ic_violated_by_add_father(_,Y):-
      father(_,Y).
ic_violated_by_add_father(X,_):-
      \+ exists(X).
ic_violated_by_add_father(_,Y):-
      \+ exists(Y).
```

These, parameterized and simplified integrity constraints can now be used for improving the bad efficiency of `add_father`/2 predicate shown in section 6.1 above.

6.3 Conversational update routines

Putting all integrity checking into a single predicate such as `ic_violated`/0 in the general case or a specialized one such as `ic_violated_by_add_father`/2, makes it easy to make a check, but noticing that something is wrong by observing a success of one of these predicates does not indicate exactly where the problem is.

In order to provide more detailed information, we may suggest to split the simplified ones into separate predicates that we can test one by one:

```
ic_violated_by_add_father_due_to_old_father(_,Y):-
      father(_,Y).
ic_violated_by_add_father_due_to_nonexisting_father(X,_):-
      \+ exists(X).
ic_violated_by_add_father_due_to_nonexisting_child(_,Y):-
      \+ exists(Y).
```

Testing these one by one for a suggested update, and if one of them fails, it is easy in an update predicate to generate an explanation.

We can even go further as the explanation in each case also suggests a way to achieve consistency but still respecting the user's intention. We concentrate the discussion on the first of the three above, and add an extra argument that returns the name of the existing father that conflicted with the suggested new one.

```
ic_violated_by_add_father_due_to_old_father(_,Y,Old):-
      father(Old,Y).
```

Consider as an example the case where an update request "add `father(peter,paul)`" leads to a success of this predicate that indicates a problem due to a previous father, say, `john`. From this information, it is straightforward to generate the following cooperative reply that includes proposals for modifying the update request:

> *The suggested update is not possible as John is already registered as the father of Paul.*
> *You can chose one the following:*
>
> - *Replace John by Peter as father of Paul.*
> - *Try with another child of Peter (if Paul is a mistake).*
> - *Give up the update request.*

6.4 Exercises

Exercise 6.1 Consider the database and integrity constraints that you have produced as solution to exercises 4.1 and 5.4 (pages 26 and 47).

Write simplified integrity constraints and use them in specialized update predicates for adding and deleting tuples.

Exercise 6.2 In exercise 5.7, we considered a predicate `materialize` that produced a tabular version of a predicate otherwise defined as a view. Extend your solution to the previous exercise 6.1 with a materialized view, e.g., for the `separated_couple` relation. This introduces a maintenance problem as the view predicate may be changed indirectly when an underlying tabular predicate is changed. Extend the update routines from previous exercise so that they also maintain the materialized view. A solution that re-evaluates the materialization from scratch using the `materialize` predicate for each update is not accepted; suggest a mechanism that works analogously to the simplified constraints.

Exercise 6.3 In exercise 5.8, you had to define a database of `income` and `expense` facts together with a `balance` predicate defined as an aggregate over the other two, and also an integrity constraint requiring the value of `balance` to be nonnegative. Design an efficient way to maintain this integrity constraint and include it in update routines for the `income` and `expense` predicates.

7 Extensions to Prolog

7.1 Constraint logic programming

Not included in present version

7.2 Constraint Handling Rules

Not included in present version

7.3 Grammar notations in Prolog and in CHR

Not included in present version

8 Abduction in logic programming and its application to databases

Not included in present version

9 *A few thoughts about a revision and extension of this note*

For an extended version, we consider taking out section 2.4 on logical semantics and expand it into a full section that follows present section 6. It should give more details on propositional and first-order logic, model theory, and formalize resolution (Grant & Minker's CACM paper from 1992 "The impact of..." is good inspiration); truth tables, fixpoints, and proof systems should be illustrated by running Prolog programs. Simplification of integrity constraints is introduced informally in section 6.2; this should remain in this way, but an additional full section should provide precise definition and a detailed procedure for deriving simplified integrity constraints (should follow present section 8 on abduction). The ISO standard for Prolog should be referenced, and syntactic and semantic details of Prolog described in the present text should be checked with it.