Artificial Intelligence and Intelligent Systems:

Logic programming as a framework for Knowledge Representation and Artificial Intelligence

Henning Christiansen
Roskilde University, Computer Science Section
 Version 19 Sep 2006

Contents

1 Introduction and overview

This note gives a practical introduction to logic programming through Prolog and its extension CHR under a knowledge representation and artificial intelligence perspective. Prolog and CHR is interesting for students and researchers in these areas as well as in the databases 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 can be used as a shell for backward-chaining, rule-based systems and simple so-called expert systems can be implemented. CHR, on the other hand, can be used in a direct way for forward-chaining systems.
- 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.
- The combination of CHR and Prolog provides a direct way to implement abductive reasoning under integrity constraints, which is useful for a variety of planning and diagnosis problems.
- 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 note is still under development and several relevant items should be added when or if it is going to be published as a book. Sections will be added on Definite Clause Grammars and their combination with abduction and assumptions, and another section should introduce the a probabilistic version of Prolog extended with machine learning techniques as in T.Sato's PRISM system. The distribution of references is uneven and not satisfactory yet, so the interested reader may need to do some searching.

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:
  `=`, `========`, `@@*@@`, `<=>`,

[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.

- 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'`.

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 {⟨`john`,`mary`⟩}.

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,

[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.

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

[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.

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).

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.

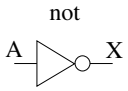
2.2 Example: Logical circuits in Prolog

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.



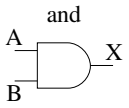
A	X
0	1
1	0

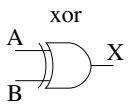
The corresponding definition in Prolog is the following sequence of facts, one for each row in the table.

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

```
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.





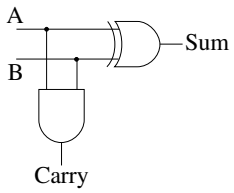
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

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.



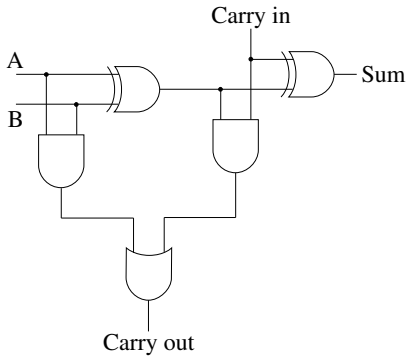
It can be described by a predicate defined as follows.

```
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:

$\texttt{p(X,Y)}\alpha =$`p(a,Y)` 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 $E\mathrm{mgu}(E,F) = F\mathrm{mgu}(E,F)$ is called the *unification* of E and F. If no $\mathrm{mgu}(E,F)$ 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:

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

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`$\neq$`adam` $\wedge$ `X`$\neq$`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

[5] 7-9-13

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 represents the logical condition $\neg(\exists y$`father(`y`,`*name*`)`$)$, 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.

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

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 $\rightarrow$ part of $\leftrightarrow$). 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.

⟨*term*⟩::= ⟨*function symbol*⟩`(`⟨*term*⟩`, ..., `⟨*term*⟩`)`
| ⟨*constant*⟩
| ⟨*variable*⟩

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

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

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

`atomic(`x`)` succeeds 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*`(···)`, 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 Constraint Handling Rules and their applications for rule-based expert systems

The language of Constraint Handling Rules, CHR, is an extension to Prolog intended as a declarative language for writing constraint solvers for CLP systems; here we give a very compact introduction and refer to [10] for a more formal and covering presentation. Examples have been developed using the SICStus Prolog version of CHR and we refer the reader to its manual [14] for remaining technicalities. As textbook on CHR and constraint programming, we refer to [3]. CHR is now integrated in several major Prolog implementations and has gained popularity for a variety of applications due to its expressibility and flexibility, which goes far beyond the traditional applications of constraint programming (such as finite domains, arithmetic, etc.). We refer to the CHR web site [1] for further information on CHR, including its different implementations and applications.

Prolog has its fixed top-down or backward chaining strategy for program execution and CHR extends with bottom-up or forward chaining, and in this way, the combination of CHR and Prolog provides a powerful environment for illustrating the basic mechanisms of rule-based expert systems. In fact, CHR itself allows for quite flexible combinations of the two execution strategies, but we consider here also the combination with Prolog, the latter kept responsible for backward chaining.

Section 7.1 introduces the fundamentals of CHR by means of examples and without too many formal details; references for those who want the full story were given above. In section 7.2, we introduce a little knowledge base and show how it can be written as a Prolog program which makes it behave as a backward-chaining expert system. Section 7.3 shows how forward chaining can be obtained by writing the knowledge base as CHR propagation rules.

7.1 Basic CHR

CHR takes over the basic syntactic and semantic notions from Prolog and extends with its specific kinds of rules. The execution of CHR programs is based on a *constraint store*, and the effect of applying a rule is to change the effect of the store. For a program written in the combination of Prolog and CHR, the system switches between two tow. When a Prolog goal is called, it is executed in the usual top-down (or goal-directed) way, and when a Prolog rule calls a CHR constraint, this will be added to the constraint store — then the CHR rules apply as far as possible, and control then returns to the next Prolog goal.

Technically speaking, constraints of CHR are first-order atoms whose predicates are designated constraint predicates, and a constraint store is a set of such constraints, possible including variables

that are understood existentially quantified at the outermost level. A constraint solver is defined in terms of rules which can be of the following two kinds.

$$\text{Simplification rules: } c_1, \ldots, c_n \texttt{ <=> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$
$$\text{Propagation rules: } c_1, \ldots, c_n \texttt{ ==> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$

The c's are atoms that represent constraints, possible with variables, and a simplification rule works by replacing in the constraint store, a possible set of constraints that matches the pattern given by the *head* $c_1, \ldots c_n$ by those corresponding constraints given by the *body* $c_{n+1}, \ldots, c_m$, however only if the condition given by *Guard* holds. A propagation rule executes in a similar way but without removing the head constraints from the store. What is to the left of the arrow symbols is called the *head*[11] and what is to the right of the guard the *body*. The declarative semantics is hinted by the applied arrow symbols (bi-implication, resp., implication formulas, with variables assumed to be universally quantified) and it can be shown that the indicated procedural semantics agrees with this. This is CHR explained in a nutshell.

CHR provides an third kind of rules, called *simpagation rules*, which can be thought of as a combination of the two or, alternatively, as an abbrevation for a specific form of simplification rules.

$$\text{Simpagation rules: } c_1, \ldots, c_i \setminus c_{i+1}, \ldots c_n \texttt{ <=> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$
$$\text{which can be though of as: } c_1, \ldots, c_n \texttt{ <=> } \mathit{Guard} \mid c_1, \ldots, c_i, c_{n+1}, \ldots, c_m$$

In other words, when applied, $c_1, \ldots, c_i$ stays in the constraint store and $c_{i+1}, \ldots, c_n$ are removed.

In practice, the body of CHR rules can include any executable Prolog expression including various control structures and calls to Prolog predicates. Similarly, Prolog rules and queries can make calls to constraints which, then, may activate the CHR rules.

The guards can be any combination of predicates (built-in or defined by the programmer) that test the variables in the head, but in general guards should not change values of these variables or call other constraints; it these cases, the semantics gets complicated, see references given above if you may have interest in the details. Finally, guards can be left out together with the vertical bar, corresponding to a guard that always evaluates to true.

The following example of a CHR program is adapted from the reference manual [14]; from a knowledge representation point of view it may seem a bit strange, but it shows the main ideas. It defines a a little constraint solver for a single constraint `leq` with the intuitive meaning of less-than-or-equal. The predicate is declared as an infix operator to enhance reading, but this is not necessary (`X leq Y` could be written equivalently as `leq(X,Y)`).

```
:- use_module(library(chr)).
handler leq_handler.
constraints leq/2.
:- op(500, xfx, leq).

X leq Y <=> X=Y | true.
X leq Y , Y leq X <=> X=Y.
X leq Y \ X leq Y <=> true.
X leq Y , Y leq Z ==> X leq Z.
```

[11] Some authors call each constraint to the left of the arrow a head, and with that terminology, CHR has multi-headed rules.

The first line loads the CHR library which makes the syntax and facilities used in the file available. The `handler` directive is not very interesting but is required. Next, the constraint predicates are declared as such (here only one such predicates) and this informs the Prolog system that occurrences of these predicates should be treated in a special way.

The program consists of four rules, one propagation, two simplifications, and one simpagation. The first simplification describe transitivity of the `leq` constraints. If, for example, the constraints `a leq b` and `b leq c` are called, this rule can fire and will produce a new constraint `a leq c` (which in turn may activate other rules).

The second rule is a simplification rule which will remove the two constraints and unify the arguments. Intuitively, the rule says that if some `X` is less than or equal to some `Y` and the reverse also holds, then they should be considered equal (antisymmetry). With constraint store {`a leq Z`, `Z leq a`}, the rule can apply, removing the two constraints and unifying variable `Z` with the constant symbols `a`.

Consider a slightly different example, the constraint store {`a leq b`, `b leq a`}. Again, the rule can apply, removing the two constraints from the store and calling `a=b`. This will fail as `a` and `b` are two different constant symbols.

Notice that CHR is a so-called *committed choice* language in the sense that when a rule has been called, a failure as exemplified above will not result in backtracking. I.e., in the example, the observed failure will **not** add {`a leq b`, `b leq a`} back to the constraint store so other and perhaps more successful rules may be tried out. However, when CHR is combined with Prolog, a failure as shown will cause Prolog to backtrack, i.e., it will undo the addition of the last of the two, say `b leq a`, and go back to the most recent choice point.

The simplification rule `X leq Y <=> X=Y | true` will remove any `leq` constraint from the store with two identical arguments. This illustrates a fundamental difference between Prolog and CHR. Where Prolog uses unification when one of its rules is applied to some goal, CHR uses so-called matching. This means that the mentioned rule will apply to `a leq a` but not to `a leq Z`. In contrast, the application of Prolog rule `p(X,X):-···` to `p(a,Z)` will result in `a=Z` before the body is entered.

The third rule in the program above is a simpagation rule `X leq Y \ X leq Y <=> true` which serves the purpose of removing duplicate constraints from the store.

We will consider the following query and see how the constraint store changes.

```
?- C leq A, B leq C, A leq B.
```

Calling the first constraint triggers no rule and we get the constraint store {`C leq A`}. Calling the next one will trigger the transitivity rule (the last rule), and we get {`C leq A`, `B leq C`, `B leq A`}. The last call in the query will trigger a sequence of events. When `A leq B` is added to the constraint store, it reacts, so to speak, with `B leq A` and the second rule applies, removing the two but resulting in the unification of `A` and `B`; let us for clarity call the common variable `V1` which is referred to by both `A` and `B`. Now the constraint store is {`C leq V1`, `V1 leq C`}. Now the same rule can apply once again, unifying `C` and `V1`, so that the result returned for the query is the empty constraint store and the bindings `A=B=C`.

In general, when a query is given to a CHR program (or a program written in the combined language of CHR plus Prolog), the system will print out the final constraint store together with Prolog's normal answer substitution. Alternative solution can be asked for as in traditional Prolog by typing semicolon.

We end the presentation of CHR showing a few simple examples taken from CHR web site [1]. This program by Thom Frühwirth evaluates to greatest common divisor of positive numbers.

```
:- use_module( library(chr)).
handler gcd.
constraints gcd/1.

gcd(0) <=> true.
gcd(N) \ gcd(M) <=> N=<M | L is M-N, gcd(L).
```

Here are a few test queries.

```
?- gcd(2),gcd(3).
?- X is 37*11*11*7*3, Y is 11*7*5*3, Z is 37*11*5, gcd(X), gcd(Y), gcd(Z).
```

The following program generates the prime numbers between 1 and n when given the query `?- primes(`n`).` Written by Thom Frühwirth and adapted by Christian Holzbaur.

```
:- use_module(library(chr)).
handler primes.
constraints primes/1, prime/1.

primes(1) <=> true.
primes(N) <=> N>1 | M is N-1, prime(N), primes(M).
prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

7.2 Warming up: A little knowledge base and expert system in Prolog

7.2.1 The example knowledge base

Consider this little knowledge base written as a set of if-then rules (see [12] for an introduction to such rules).

1. If rains and go_out and have_not(umbrella) then get_wet.
2. If shower_bath then get_wet.
3. If need(X) and have_not(X) then go_out and buy(X).
4. If thirsty then need(beer).

We can illustrate backwards reasoning by an investigation of whether a user will get wet. Rule 2 provides one way of getting wet, so the system may go from conclusion to premise and ask the user "Will you take a shower?" and if user replies "yes", the system can reply "Well, then you'll get wet." Trying instead using rule 1 in backwards direction, may result in questions whether it rains, and via backwards application of rule 3 followed by 4 whether the user is thirsty, and finally if he has no umbrella; in that case, the user will also get wet when he goes out to buy beer.

These rules can be rewritten into Prolog by considering each piece of data as a predicate, for example the second rule as `get_wet:- shower_bath.` This would not be very useful, however, as

primitive information such as "thirsty" (etc.) needs to be added as a fact `thirsty.` to the program. A dialogue with the user seems difficult.

Below we will introduce a little expert system shell implemented in a few lines of Prolog that makes it easy to use Prolog for backward chaining expert systems which provides a dialogue with the user

7.2.2 A simple expert system shell in Prolog

The file `expert0` contains Prolog definitions which makes it possible to read in knowledge bases in a format illustrated by sample file below. Rules are written as Prolog rules in the standard way with conclusion to the left and premises to the right.

The system inherits Prolog's goal-directed (top-down, backward chaining) strategy and uses a "generic" predicate called `goal` as a container for all data. Notice that rules, such as rule 3 above, with "and" in the conclusion needs to be rewritten into more that one rule.

To use this system you should load it into Prolog as follows:

```
:- [expert0].
```

Now you can load in your own source files; e.g.,

```
:- [my_kb0].
```

The following shows the full source text for an expert system which could be the file `my_kb0`.

```
:- multifile goal/1.

goal(get_wet):- goal(rains), goal(go_out), goal(have_not(umbrella)).
goal(get_wet):- goal(shower_bath).
goal(go_out):- goal(need(X)), goal(have_not(X)).
goal(buy(X)):- goal(need(X)), goal(have_not(X)).
goal(need(beer)):- goal(thirsty).

can_question(rains).
can_question(thirsty).
can_question(have_not(_)).
can_question(shower_bath).
```

The first line is a little Prolog technicality which is needed since the file `expert0` contains a Prolog clause for the `goal` predicates that provides the dialogue with the user.[12]

The next lines constitute the rules of the knowledge base, and the last bunch of lines defines which data the system can query the user about.

It is possible to query a knowledge base by a standard Prolog query using the `goal` predicate but for some subtle reason you cannot investigate different possible ways that made the given goal succeed. Use instead the predicate `test_goal`; the following exemplifies a dialogue with the system.

[12] Normally, and without the `multifile` directive, all clauses for a given predicate must be in one and the same file.

```
| ?- test_goal(get_wet).
Is it true that rains?y
Is it true that thirsty?y
Is it true that have_not(beer)?y
Is it true that have_not(umbrella)?y
get_wet is true. Do you want me to check other ways?y
Is it true that shower_bath?y
get_wet is true. Do you want me to check other ways?y
no
| ?- test_goal(get_wet).
Is it true that rains?n
Is it true that shower_bath?y
get_wet is true. Do you want me to check other ways?n
yes
```

7.2.3 Adding explanations

In may be complained that the little backward chaining expert system above lacks one important feature in order to be called an expert system, namely that of producing an explanation for how it proved the given goal.

However, this can easily be included by adding an extra argument to the goal predicate, which holds an explanation for that specific goal. When the user is asked for a goal and tell it to succeed, we add the explanation `user`, so for example when asked if he is thirsty, the goal succeeds as `goal(thirsty,user)`.

Each rule in the knowledge base is then extended so that it constructs an explanation for its conclusion from its subgoals and their explanations. For example:

```
goal(go_out,E):- goal(need(X),E1), goal(have_not(X),E2),
    E = [[need(X),E1], [have_not(X),E2]]
```

The explanation returned for, say, `goal(get_wet, Exp)` will then be a huge term with several levels of lists-in-lists that gives the full explanation. We will not study generation of explanations in more detail, and our purpose here has been simply to demonstrate that it straightforward to generate them.

7.3 Forward chaining expert system in CHR

7.3.1 Propagation rules are forward chaining

Propagation rules in CHR provide a natural paradigm for forward chaining. From given sets of input contraints representing given data, the rules will apply as long as possible, deriving all possible consequences thereof.

As with Prolog above, we can in principle defines each possible fact as a constraint, but for simplicity and the developments to follow, we prefer instead to introduce a "generic" constraint to hold facts. A forward chaining version of our sample knowledge base can be written directly as a CHR program that needs no extra definitions, i.e., the file can be run alone.

```
:- use_module(library(chr)).
handler forward_chaining.
constraints fact/1.

fact(rains), fact(go_out), fact(have_not(umbrella)) ==> fact(get_wet).
fact(shower_bath) ==> fact(get_wet).
fact(need(X)), fact(have_not(X)) ==> fact(go_out), fact(buy(X)).
fact(thirsty) ==> fact(need(beer)).
```

Notice that CHR's syntax allows us to have multiple facts as both premise and conclusion. To investigate the consequences of a given situation, we simply enter all known primitive facts as a query, and the system calculates the set of all consequences. Here is an example.

```
| ?- fact(thirsty), fact(rains), fact(have_not(beer)), fact(have_not(umbrella)).
fact(thirsty),
fact(need(beer)),
fact(rains),
fact(have_not(beer)),
fact(go_out),
fact(buy(beer)),
fact(have_not(umbrella)),
fact(get_wet) ?
yes
```

We observe that `get_wet` is reported in the final state oprinted out, so the system have told us that in the situation described, being thirsty, no beer in the house and no umbrella, on a rainy day, we well eventually get wet.

It is, of course, possible to extend the program above so its presents a more user friendly dialogue and only prints out those new facts that are derived, but we leave this out in order not to destroy the clarity of the presentation. Explanations can easily be added similarly to what we described in section 7.2.3, and we shall not consider this topic further.

7.3.2 Conflict resolution

As described in Negnevitsky's book [12], section 2.8, there may be conflicts among the rules of a knowledge base, so that in some situations, two different rules can apply, leading to different conclusions that are considered inconsistent with each other. In such cases, the expert system should have some strategy to decide which of the two rules to apply. Such principles do not fit very well into CHR, as the meaning of a set of CHR rules is defined in terms of first order predicate logic in which there is no sense in consider some rules better that others.

Let us consider a little example for crossing the street, and assume the following four CHR rules.

```
fact(light(green))        ==> fact(action(go)).
fact(light(red))          ==> fact(action(wait)).
fact(car_in_full_speed)   ==> fact(action(wait)).
fact(no_traffic)          ==> fact(action(go)).
```

CHR allows us to specify explicitly what are the inconsistencies, which here could be that the colour of the light is unique and the same for the action.

```
fact(ligth(X)),  fact(ligth(Y))  ==> X=Y.
fact(action(X)), fact(action(Y)) ==> X=Y.
```

Such rules corresponds to what in databases and abductive logic programming are called *integrity constraints*. Consider then the query from which a conflict arises.

```
?- fact(light(red)), fact(no_traffic).
```

These two fact will lead to the derived facts `fact(action(wait))` and `fact(action(go))`; next, these two trigger the second integrity constraint, resulting in an attempt of the unification `wait=go` which obviously fails. Thus the answer to the query is a useless "`no`".

So the conclusion must be that CHR is well suited for representation of consistent knowledge bases and for reasoning about consistent states, but additional complications such as avoidance of inconsistencies by priorities does not fit it.

Of course, CHR is a general programming language and integrated with Prolog so it will be possible to whatever conflict resolution a designer has in mind, but we may loose the transparent reading of the rules. We will, however, indicate some possible strategies which could be developed further in student projects.

Preference to some facts over others

In the example above we could make the integrity constraints less hard. It might be obvious here, in case of conflicts among possible actions to prefer the one which seems the most safe, i.e., prefer `action(wait)` for `action(go)`, and if there were conflicting information about the colour of the light, keep the conflict but issue a warning.

```
fact(ligth(X)),  fact(ligth(Y))  ==>
                          X \== Y | write('Warning: Colour of light not unique').
fact(action(wait)) \ fact(action(go)) ==> true.
```

You should be very careful when using CHR in this way. Assume that `fact(action(go))` happen to be the first fact to be derived, and that it via other rules, leads to the creation of other facts, call them F. If now, later, `fact(action(wait))` is derived, then the modified integrity constraint above will remove `fact(action(go))`, and thus the total program should be written in such a way that the facts of F also are removed.

Ad-hoc weights in CHR knowledge bases

We may consider adding ad-hoc numerical weights to each rule, that are used for determining a weight for each fact. The weight of a fact derived via a rule may then be the product of the two numbers, from the fact to which the rule is applied, and the weight assigned to the given rule. It will be practical to have all numbers to be between 0 and 1. The rule base above may be extended with weights as follows.

```
fact(light(green),V)      ==> W is V * 0.9, fact(action(go),W).
fact(light(red),V)        ==> W is V * 0.9, fact(action(wait),W).
fact(car_in_full_speed,V) ==> W is V * 1,   fact(action(wait),W).
fact(no_traffic,V))       ==> W is V * 1,   fact(action(go),W).
```

We see that the designer has given highest priority to those rule that depends on the actual traffic rather than the traffic lights. Integrity constraints can then be modified so that they resolve conflicts be throwing away the one out of two conflicting facts which has the lowest priority. This can be done by simpagation rules as follows.

```
fact(ligth(X),V)  \ fact(ligth(Y),W)  <=> V >= W | true.
fact(action(X),V) \ fact(action(Y),W) <=> V >= W | true.
```

In this way the query

```
?- fact(light(red),1), fact(no_traffic,1).
```

would lead to the answer that you should walk (with weigh 1).

We have indicated here how priorities may be added to CHR knowledge bases, but we will stress that this was done in a completely ad-hoc manner. If you want to add priorities to knowledge bases written in CHR, it is suggested that base you approach on a firm theoretical basis such as probability theory or fuzzy logic. The remarks made for the previous approach should be repeated also here, that when you explicitly remove an undesired fact, you should take care also to remove those other facts derived from it.

7.4 Combining forward and backward chaining

The text book by Negnevitsky [12] indicates, p. 40 at the bottom, that forward and backward chaining can be combined. Backward or goal directed is the most convenient when asking queries to the currently known set of facts, and forward chaining is used when a new fact is recognized so that the fact base is as complete as possible.

It is possible with a few additional "linking rules" to combine the two versions of a knowledge base that we have shown, backward chaining using Prolog and forward chaining using CHR, to achieve this effect. It does not seem very instructive to show the details here so we shall avoid this, as it amounts only to a small optimization of the backward chaining version (but no essential new functionality).

However, if we consider adding integrity constraits, conflict resolution and/or priorities to the forward chaining part, the combination may become interesting.

7.5 Exercises

Exercise 7.1 The following program generates Fibonacci numbers. When you give the query `?- fib(5,N).`, the value returned for `N` is the `N`th Fibonacci number; in addition, the program produces constraints that stores all Fibonacci up till the `N`th.

Fibonacci numbers are the series 1, 1, 2, 3, 5, 8, 13, ..., i.e., each number is the sum of the two previous ones.

```
:- use_module(library(chr)).
handler fibonacci.
constraints fib/2. % fib(N,F): The Nth Fibonacci number is  F

fib(N,F1)\fib(N,F2) <=> F1=F2.
fib(1,X) ==> X=1.
```

```
fib(2,X) ==> X=1.
fib(N,F) ==> N > 2 | N1 is N-1, fib(N1,F1), N2 is N-2, fib(N2,F2), F is F1+F2.
```

This program illustrates the difficult procedural issues of CHR, and the question in this exercise is to compare execution speed for the program as it is and if you swop the first and list rules. Logically, the two program versions are identical and produces the same result; can you argue why one version takes considerably more time than the other one?

Exercise 7.2 Identify some problem field of which you have good knowledge, and write down some if-then rules about that domain similarly to those of section 7.2.1.

Implement the rules as a forward chaining CHR program as indicated in section 7.3.

Exercise 7.3 Extend the knowledge base that you wrote down in CHR for the previous exercise with integrity constraints and perhaps additional rules as to make conflicts possible. Apply some of the principles sketched in section 7.3.2 in order to resolve conflicts. (You may choose to avoid some of the technical caveats given, and simply test out different approaches and see how they work).

8 Abductive reasoning in Prolog and CHR

8.1 Deduction, abduction, and induction in logic programming

The philosopher C.S.Peirce (1839–1914) is considered a pioneer in the understanding of human reasoning, especially in the specific context of scientific discovery. His work is often cited in computer science literature but probably only few computer scientists have read Peirce's original work. We recommend [9] as overview of Peirce's influence seen from the perspective of computer science.

Peirce postulated three principles as *the* fundamental ones:

- **Deduction**, reasoning within the knowledge we have already, i.e., from those facts we know and those rules and regularities of the world that we are familiar with. E.g., reasoning from causes to effects:
  "If you make a fire here, you will to burn down the house."
- **Induction**, finding general rules from the regularities that we have experienced in the facts that we know; these rules can be used later for prediction:
  "Every time I made a fire in my living room, the house burnt down, aha, ... the next time I make a fire in my living room, the house will burn down too".
- **Abduction**, reasoning from observed results to the basic facts from which they follow, quite often it means from an observed effect to produce a qualified guess for a possible cause:
  "The house burnt down, perhaps my cousin has made a fire in the living room again."

In fact, Peirce had alternative theories and definitions of abduction and induction; we have adopted the so-called syllogistic version, cf. [9]. We can replicate the three in logic programming terms:

- A Prolog system is a purely deductive engine. It takes a program of rules and facts, and it can calculate or check the logical consequences of that program.

- Induction is difficult; methods for so-called inductive logic programming (ILP) have been developed, and by means of a lot of statistics and other complicated machinery, it synthesizes rules from collections of "facts" and "observations". We refer to [4][13] for an overview of different applications. Inductive logic programming has been successfully applied for molecular biology concerned with protein molecule shapes and human genealogy. See [13] for an in-depth treatment of ILP methods.

- Abductive logic programming; roughly means from a claim of goal that is required to be true (i.e., being a consequence of the program), to extend to program with facts so that the goal becomes true. See [11] for an overview. Abduction has many applications; we may mention planning (e.g., goal is "successful project ended" and the facts to be derived are the detailed steps of a plan to achieve that goal), diagnosis (goal is observed symptoms, the facts to be derived comprise the diagnosis, i.e., which specific components of the organism or technical system that malfunction). An important area for abduction is language processing, especially discourse analysis (the discourse represents the observations, the facts to be derived constitute an interpretation of that discourse). We will look into some of these in more detail below and give references.

However, we should be aware that while deduction is a logically sound way of reasoning, this is generally not the case for abduction and induction. We will make a simple analysis for abduction. Assume a logical knowledge base $\{a \rightarrow c, b \rightarrow c\}$ where the arrow means logical implication. If we know c, an abductive argument may propose that a is the case, however, this is not necessarily true as it might that b is the case and not a, or it could even be the case that none of a and b are the case, and that there is another and unknown explanation for c. Abduction is often described as reasoning to the best explanation. i.e., best with respect to the knowledge we have available.

8.2 A definition of abductive logic programming

(This section is adapted from [5]). An abductive logic program [11] is usually specified as a triplet $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ where $\mathcal{P}$ is a logic program, $\mathcal{A}$ a set of *abducible* predicates that do not occur in the head of any clause of $\mathcal{P}$, and $\mathcal{IC}$ a set of integrity constraints assumed to be consistent. Assume additionally that $\mathcal{P}$ and $\mathcal{IC}$ can refer to a set of *built-in* predicates that have a fixed meaning identified as a theory $\mathcal{B}$; a predicate in $\mathcal{P}$ that is neither abducible nor built-in is called *defined*. A typical built-in predicate is the `dif/2` predicate of SICStus Prolog [14]; `dif(X,Y)` means that `X` and `Y` must be different. We assume for simplicity in the following that $\mathcal{IC}$ refers to abducible and built-in predicates only.

Given an abductive logic program $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$, we define for pairs of sets of abducibles and built-in atoms $\langle A, B \rangle$, a *consistent ground instance* to be a common ground instance[14] $\langle A', B' \rangle$ of $\langle A, B \rangle$ so that[15]

- $\mathcal{B} \models B'$ (the instance of built-ins is satisfied)

[13] A bit old; if you are interested, you should search for more recent overview papers and consult proceedings of the recent ILP conferences; see http://www.informatik.uni-trier.de/~ley/db/conf/ilp/index.html.

[14] We recall that an expression is ground if it contains no variable. A common ground instance means that the same grounding substitution is applied to both A and B.

[15] Recall that $\models$ can be read as "entails" or "has the logical consequence".

- $\mathcal{B} \cup A' \models \mathcal{IC}$ (the instance of abducibles respects the integrity constraints)

For simplicity and without loss of generality, we consider only ground queries;[16] an *abductive answer* to a query Q is a pair of finite sets of abducible and of built-in atoms $\langle A, B \rangle$ such that

- $\langle A, B \rangle$ has at least one consistent ground instance $\langle A', B' \rangle$,
- for any such $\langle A', B' \rangle$, we have $\mathcal{P} \cup A' \models Q$.

In other words, the abductive answer should be consistent and, if the abducible atoms were added to the programs as if they were ordinary Prolog facts, the query would succeed.

Minimality and Compaction

It is often required that an abductive answer be minimal measured in the number of abduced literals (or, alternatively, in a subset relation or subsumption ordering). If, for example, the query "`?- happy(me).`" has the set

$$A_1 = \{\texttt{rich(me)}, \texttt{has(me,nice\_students)}, \texttt{moon(full)}\}$$

is an abductive answer, and that

$$A_2 = \{\texttt{rich(me)}, \texttt{has(me,nice\_students)}\}$$

also is an abductive answer, then somehow, the fact `moon(full)` does not seem very interesting, as the smaller answer A_2 can explain the query. The answer A_2 is minimal provided that none of its subsets are abductive answers, i.e., neither $\{\texttt{rich(me)}\}$ nor $\{\texttt{has(me,nice\_students)}\}$ can explain the query.

Most published abduction algorithms try to unify a new abducible with one already produced (as to produce answers of a minimum number of literals), and tries out different alternatives under backtracking. This does not guarantee minimality in cases when, say, one branch of executions produces abducibles a and b but another may produce only a. Minimal answers can be selected by post-processing all answers found in this way. However, we argue that this principle which we call *compaction* is not always obvious or desirable, and we suggest it be optionally specified for selected abducible predicates.

If, for example, someone's car was stolen in Paris and his wallet in New York, it seems over-constrained to assume by default that the thieves are the same one. In other words, if both

$$B_1 = \{\texttt{steel(X,wallet)}, \texttt{steel(Y,car)}, \texttt{thief(X)}, \texttt{thief(Y)}\}$$

and

$$B_2 = \{\texttt{steel(X,wallet)}, \texttt{steel(X,car)}, \texttt{thief(X)}\}$$

are abductive answers to a given query, we claim that it is not obvious that B_2 should be considered the best. In fact, when you write B_2 as $B_1 \cup \{\texttt{X=Y}\}$ seems that B_1 is smallest in some sense.

[16] For example, "`?- happy(me).`" is a ground query but "`?- happy(X).`" is not. Considering only ground queries means that we can ignore the traditional answer substitution; this can easily be added.

8.3 Abduction implemented in Prolog with a little help from CHR

(This section adapted from [8, 5]; see those papers for references to earlier work).

Constraint Handling Rules [10], CHR, is a declarative, rule-based language for writing constraint solvers and is now included as an extension of several versions of Prolog. Operationally and implementation-wise, CHR extends Prolog with a constraint store, and the rules of a CHR program serve as rewriting rules over constraint stores. CHR is declarative in the sense that its rules can be understood as logical formulas. Constraint predicates must be declared as such and can then be called from a Prolog program; see [10] for details. The following example declares a constraint predicate `a` and defines a so-called propagation rule.

```
constraints a/1.
a(1), a(2) ==> fail.
```

This rule identifies a state as illegal if it contains the two indicated constraints. As first noticed by [2], there is a clear analogy between abducibles plus integrity constraints and CHR's constraints plus rules.

The implementation of abduction in Prolog with CHR is simple. Abducibles are viewed as constraints in the sense of CHR: the logic program is executed by the Prolog system; whenever an abducible is called it is added automatically by CHR to the constraint store and CHR will activate integrity constraints whenever relevant. The complete hand-coded implementation of an abducible predicate `a/1` is provided by the following three lines.

```
:- use_module(library(chr)).
handler abduction.
constraints a/1.
```

Compaction for `a/1` is implemented by a single CHR rule; the following provides a correct implementation.

```
a(X), a(Y) ==> true | (X=Y ; dif(X,Y)).
```

The implementation used in the HYPROLOG system [5] applies a slightly optimized version of this rule using low-level facilities of CHR.

In addition, the integrity constraints mentioned above can be written directly as CHR rules.

In a comparison with other abductive logic programming systems we may emphasize the following; see [5] for more detailed comments.

- The indicated method is limited with respect to negation. There are important applications (see, e.g., [11]) that requires the creation of new abducibles from within negated calls to predicates. Our method can handle a limited form of so-called explicit negation which is hard-wired into the HYPROLOG system [5], but which cannot handle the indicated applications.

- However, for the applications that this methodology can handle, it is considerably faster that other, known systems for abduction. The reason for this is that we utilize the underlying technology in an optimal and direct way, as no intermediate level of interpretation is involved.

- Important applications such as many cases of diagnosis and natural language analysis can be modeled nicely without negation.

8.4 A first example of abduction in Prolog+CHR

This is extends a standard example from the literature, used by [11] and others. Consider the following Prolog program:

```
grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
```

It has two rules and no facts. Obviously the following query fails when executed in Prolog:

```
?- grass_is_wet.
no
```

On the other hand, an abductive system should not give up that easily. It should do what it can to enforce that the query succeeds, and the only way it can do so is by suggesting which facts to add to the program.

A typical abductive interpreter do not actually modify the program source text but produces an abductive answer (as defined above) consisting of those facts, that if they were added to the program would make the query succeed. With an abductive interpreter, and provided that predicates `rained_last_night` and `sprinkler_was_on` are declared as abducibles, we may experience a dialogue as follows:

```
?- grass_is_wet.
rained_last_night ? ;
sprinkler_was_on ? ;
no
```

We can illustrate the meaning of these two answers by stating that the query `?- grass_is_wet.` would succeed when executed by a Prolog system given one of the following two Prolog programs (no CHR or abduction involved this time).

```
grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
rained_last_night.
```

```
grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
sprinkler_was_on.
```

Basically, an abductive interpreter works in the way that when it enters an abducible goal that otherwise would fail in Prolog, it simply notes that goal as part of the abductive answer. As we indicated above, CHR's constraint store can serve as a container for the abductive answer being built up gradually as the execution goes on.

Consider the following program that combines Prolog and CHR as we indicated above; two predicates are declared as constraints so that they will be treated as abducibles:

```
handler garden_humidity1.

constraints rained_last_night/0, sprinkler_was_on/0.

grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
```

When this program is executed, the constraints entered are added to the constraint store. The declaration `handler garden_humidity.` is not important, but CHR's syntax requires such a declaration.

The resulting constraint store is printed as part of the answer and we get exactly the following when asking a query for `grass_is_wet`.

```
?- grass_is_wet.
rained_last_night ? ;
sprinkler_was_on ? ;
no
```

Integrity constraints can be seen as a way to rule out "weird" abductive answers, and these integrity constraints can be understood intuitively and in their precise semantics as integrity constraints for databases. Let us extend the program above with one more abducible predicate and a straightforward CHR rule that serves as an integrity constraint.

```
handler garden_humidity2.

constraints rained_last_night/0, sprinkler_was_on/0,
            full_moon_last_night/0.

rained_last_night, full_moon_last_night ==> fail.

grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
```

The integrity constraint reads: If the constraint store contains the two constraints indicated by its lefthand side, then its body (following the arrow) is executed and here producing a failure. The following query to the program shows that only one answer is produced, as the potential second one triggers the integrity constraint.

```
?- full_moon_last_night, grass_is_wet.
full_moon_last_night,
sprinkler_was_on ? ;
no
```

Integrity constraints can contain any executable expression a its body, in particular another abducible. Consider the following.

```
handler garden_humidity3.

constraints rained_last_night/0, sprinkler_was_on/0,
            full_moon_last_night/0, mixed_weather_last_night/0.

rained_last_night, full_moon_last_night ==> mixed_weather_last_night.

grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
```

This results in the following:

```
?- full_moon_last_night, grass_is_wet.
rained_last_night, mixed_weather_last_night ? ;
sprinkler_was_on ? ;
no
```

Notice that the additional abducible `mixed_weather_last_night` is only produced by for the first answer, as the integrity constraint is not triggered for the second one.

8.5 Database view update considered as abduction

The following program written in Prolog plus CHR defines a little database with integrity constraints, that express so-called key constraints on the `father` and `mother` relations, i.e., "you can only have one father and only one mother". The `grandfather` predicate corresponds to a view definition in a traditional database.

```
:- use_module(library(chr)).
handler view_update.

constraints father/2, mother/2.

father(X,Y), father(Z,Y) ==> Z=X.
mother(X,Y), mother(Z,Y) ==> Z=X.

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

current_db:-
    father(peter, paul),
    father(paul,jens),
    mother(marie,jens).
```

It may seem a bit confusing that the current database is not defined by means of facts as usual, but this strange way above is needed in order to provide an interaction between new and old database facts.

In order to perform a view update to a given database, we can give it as a query, here using the auxiliary predicate `current_db`. The query below reads informally "In which ways can some `X` be a grandfather of `jens`. Notice that we get the the whole updated database as answer.

```
?- current_db, grandfather(X,jens).
X = peter,
father(peter,paul),
father(paul,jens),
mother(marie,jens),
father(peter,paul),
father(paul,jens) ? ;
```

```
father(peter,paul),
father(paul,jens),
mother(marie,jens),
father(X,marie),
mother(marie,jens) ? ;
no
```

The first answer suggests that `peter` could be grandfather of `jens`, provided the new fact `father(paul,jens)` is added to the database. The second answer does not indicate a specific value for `X` but indicates that any value v for `X` will do provided that `father(`v`,marie)` is added to the database. (For technical reasons that we shall not comment on here, some database facts becomes duplicated; this is easy to avoid by simple techniques in CHR.)

8.6 Simple diagnosis problems formulated as abduction

In a case of medical diagnosis, the patient is explaining various symptoms, "I have pain here, and here, but not here ....", and the doctor's job is to give the diagnosis which may the identification of a particular decease that can explain the observed symptoms. We can consider this as a case of abduction. The doctor's knowledge includes a large collection of rules about which deceases and conditions that may cause which symptoms. In the particular case, he should be able to figure out what are the specific deceases and conditions that can explain this patient's symptoms. Maybe a combination of deceases must be suggested in order to explain symptoms.

It is interesting to notice that the doctor uses his *knowledge* in order to provide statements about the patient's internal state without actually opening the patient. In other words, he is making predictions about hidden information that cannot be immediately checked in reality, and what is available to make the judgment are externally observed symptoms. This little discussion also indicates the potential unsound nature of abduction, in that the doctor may have chosen the wrong out of alternative diagnoses. This may then be corrected by new observations, say, that the suggested medical treatment has no effect, by further questioning of the patient, or from surgical investigations.

We use the notion of diagnosis in a more wide sense for finding explanations for systems that shows certain wrong behaviour. By system, we mean here a structure of interconnected primitive components, and where possible malfunctioning is caused by the malfunctioning of one or more of the primitive components. We will formulate this task as an abductive problem and show how it can be solved with abductive logic programming.

Example: Logical circuits

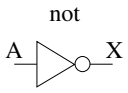
(Copied with a few changes from [7] in order to make this document self-contained). We indicate here how logical circuits can be simulated by Prolog programs, and later we extend the example with abduction for diagnosis.

Logical circuits represent 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. See, e.g., [15], chap. 3, for background and more information.

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 arguments represent the gate's (or circuit's) input and output connectors. A "not" gate, for example, can be specified by the following table.

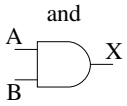


A	X
0	1
1	0

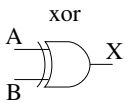
The corresponding definition in Prolog is the following sequence of facts, one for each row in the table.

```
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

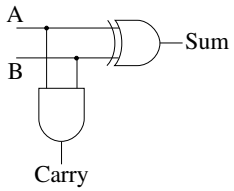


A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

The corresponding predicates `and(A, B, X)` and `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).
```

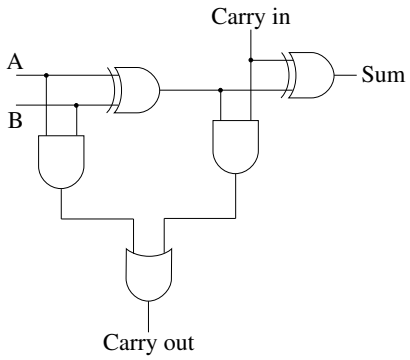
In the graphical language, gates are put together by connecting the components by means of conductors. In Prolog, we can do very much the same thing, except that we use variables instead of conductors. The following picture shows a so-called half-adder circuit.



It can be described by a predicate defined as follows.

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

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(Carryin, A, B, Carryout, Sum):-
   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(1, 0, 1, C, S).
C = 1
S = 0 ? ;
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.

8.7 Diagnosis based on the assumption of periodic faults

"Periodic fault" is a technical term which is a bit misleading, as it refers a fault that occurs now and then with irregular and unpredictable intervals. Thus such periodic faults are very difficult to find, and as we will see, also computationally very complex.

We describe the principles for the example of logical circuits introduced above. First of all, we need to assign an identifier to each occurrence of a gate in the circuit in order to point out (in the abductive answers), which components are defect. We may extend the clauses that define given circuit in the following way.

```
halfadder(A, B, Carry, Sum):-
        and(A, B, Carry, and0),
        xor(A, B, Sum, xor0).

fulladder(Carryin, A, B, Carryout, Sum):-
        xor(A, B, X, xor1),
        and(A, B, Y, and1),
        and(X, Carryin, Z, and2),
        xor(Carryin, X, Sum, xor2),
        or(Y, Z, Carryout, or1).
```

Notice that we have used predicates for the individual gates which take the extra argument for the identifier. We should then define these predicates so that they take into account the possible malfunctioning of the gate. We give the definition for "`and`" and explain it in the following.

```
and(A,B,X,ComponentId):-
   and(A,B,X),
   perfect(ComponentId).

and(A,B,X,ComponentId):-
   and(A,B,Z), disturbe(Z,X),
   defect(ComponentId).

disturbe(0,1).
disturbe(1,0).
```

The `disturbe` predicate is applied for the other types of gates as well in a similar way. The point of using `disturbe` is that we only notice that something is wrong with a given component if it produces the wrong result for given input; `and/3` refers to the previous definition of logical-and.

Before discussing the details of predicates `perfect` and `defect`, let us consider how observations are given. Observations represent observed tests assumed to be made using some actual device, whose structure is modeled by the logic programs shown. More precisely, observations and recorded samples of input-output pairs. For example, if a given half-adder device when given input `A=1`, `B=1` produces `Carry=1` and `Sum=1`, this indicates a wrong result and that something must be wrong inside the circuit.

The purpose of the extended predicates is, then, that we can present them with queries of observed input-and-output in the following ways, for having them analyzed. For example:

```
?- halfadder(1,1,1,1).
```

The output should, then be abducible answers consisting of `perfect` and `defect` facts. Referring to the method for implementing abduction in Prolog plus CHR above, we can make the two to work as abducible predicates by the following declarations.[17]

```
:- use_module(library(chr)).
handler diagnosis.
constraints perfect/1, defect/1.
```

We define the following integrity constraints that govern the interaction among the `perfect` and `defect` constraints.

```
defect(X)  \ defect(X) <=> true.
perfect(X) \ perfect(X) <=> true.
defect(X)  \ perfect(X) <=> true.
```

The first two rules simply remove duplicates, but the third one is interesting. It is a simpagation rule that reads intuitively: if a gate once have shown to be defect, it will be remembered even though the gate in some cases has produced the correct result. In other words, a given gate must expose wrong behaviour at least once in order to be registered as defect. This materializes the periodic fault assumption. Let us test a few queries.

```
?- halfadder(1,1,1,1).
perfect(and0),
defect(xor0) ? ;
no
```

This shows that exclusive-or gate that determines the `Sum` output argument needs to be defect in order to produce this behaviour, and also that there is no reason, from the given observation, to assume the and-gate to be defect.

The problem hight complexity given by the periodic fault assumption is apparent when we give more observations to a more complex circuit. Consider the following query that represents three wrong results.

```
?- fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

It produces the total of 512 possible solutions, including duplicate solutions produced in different ways. This high number arises from a number of reasons.

- The method proposes a lot of explanations that include defects that compensate for each other.
- Combining this with the periodic fault assumption which says that a defect gate may sometimes do the right and sometimes the wrong, the set of possibilities explodes.

[17] Notice that we could have implemented this example with only the `defect` predicate, with the default assumption being perfect behaviour. However, having the two is practical for the modification of the method that we will do in the following subsection.

- Finally, the method is not able to use observations about correct behaviour in any way.[18]

The last point is critical. This corresponds to a doctor who is not able to take into account the observation that "... otherwise the patient is strong and healthy."

8.8 Diagnosis based on the assumption of consistent faults

Consistent faults means that a given primitive component always exposes the same output for the same input. In the example of logical circuits, it may be that case that a particular occurrence of a gate produces a `0` when anding two `1`es. This means that it is not sufficient to record just that the gate is defect, but we must record for which inputs it gives right results and for which inputs it gives wrong results. For this purpose, we extend the abducible predicates with additional arguments so that, say, `defect(and3,1,1)` means that the indicated gate `and3` gives wrong results when given the input of two `1`es. We can implement this in the following way, with integrity constraints that materialize the assumption of consistent faults.

```
:- use_module(library(chr)).
handler diagnosis.
constraints perfect/3, defect/3.

defect(X,In1,In2)  \ defect(X,In1,In2)  <=> true.
perfect(X,In1,In2) \ perfect(X,In1,In2) <=> true.
defect(X,In1,In2)  , perfect(X,In1,In2) <=> fail.
```

As in the previous version, the first two rules remove duplicates, and the last one is important. It indicates that it is not possible to create an abductive explanation that claims two different behaviours for the same component and given input.

The predicate definitions for the entire circuits remain the same, and those for the individual gates are adapted as follows.

```
and(A,B,X,ComponentId):-
   and(A,B,X),
   perfect(ComponentId,A,B).

and(A,B,X,ComponentId):-
   and(A,B,Z), disturbe(Z,X),
   defect(ComponentId,A,B).
```

The following is an example of a query that consists of one wrong and one correct observation.

```
?- halfadder(1,1,0,1), halfadder(1,0,0,1).
defect(and0,1,1),
defect(xor0,1,1),
perfect(and0,1,0),
perfect(xor0,1,0) ? ;
no
```

[18] In fact, even for queries representing entirely correct behaviour, the method proposes a lot of possible defects that mutually compensate for each other.

Only one answer is given. It is interesting to see that the answer also gives information about how throughly the circuit has been tested. It can be seen that the `and0` gate's behaviour on input `0-0` has not been tested, so if the purpose is a thorough test of the circuit, it might be a good idea to try more samples so that `defect(and0,0,0)` or `perfect(and0,0,0)` may show up.

Another interesting query is this one.

```
?- halfadder(1,1,1,0), halfadder(1,1,1,1), halfadder(0,0,1,0).
no
```

How can we interpret this results? Well, it indicates that there is no diagnosis for these observations under the consistent fault assumption. Two things can be wrong, then. Either the actual device does not obey the consistent fault assumptions (in which case we must go back to periodic fault assumption), or that the observations are wrong so that we should redo the tests using the device once again.

To see the difference between periodic and consistent, we recall that the following query produced 512 solution with perodic, and under consistent fault assumption, it produces "only" 144 solutions.

```
?- fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

This is still quite many solutions, but we may improve by adding samples of correct input-out behaviour for our circuit (which, of course, only can be defended if we have performed test using the given device, that actually produced these results).

```
?- fulladder(1,1,0,1,0), fulladder(0,1,0,0,1), fulladder(0,0,1,0,1),
   fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

This limits to now 48 solutions, and a careful study of those indicates that many of them describe combinations of mutually compensating errors which makes the circuit produce the correct samples in wrong ways, so to speak.

To compensate for this, we introduce a principle that we may call the **correct-results-produced-in-correct-way assumption**, which may not always be realistic but may help to bring down complexity. By this principle, we indicate that those inputs that are run through the different gates for the correct observation are assumed always to be correct, i.e., for such combinations we should freeze suitable `perfect` abducibles. This is very easy to implement in the model presented so far. First of all, notice that the clauses for the `and/4` predicate are ordered so that the solution consisting of `perfect` abducibles is tried before any other which includes `defect` abducibles. In other words, the first solution found for observations of correct behaviour will consists of `perfect` abducibles only. (Any other possible solutions with mutually compensating errors are produced afterwards on backtracking.)

Here Prolog's cut can be used effectively as indicated in the following schema for giving the queries.

`?-` ⟨*correct-samples*⟩`, !,` ⟨*incorrect-samples*⟩`.`

This means that backtracking can only occur inside the analysis of the incorrect samples, and the abducibles from the correct ones stay fixed as `perfect`.

We may apply this principle to the qeury above.

```
?- fulladder(1,1,0,1,0), fulladder(0,1,0,0,1), fulladder(0,0,1,0,1),
   !,
   fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

In fact this gives a disappointing `no` which may be a consequence of the correct-results-produced-incorrect-way assumption not being applicable here. Or rather, it indicates that these data have been produced artificially. A more convincing example is the following which without the cut produced 72 solutions, but with the cut only the one solution indicated.

```
?- fulladder(0,1,1,1,0), fulladder(0,1,0,0,1), fulladder(0,0,1,0,1),
   !,
   fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).

defect(xor2,1,1),
defect(and2,0,1),
defect(xor2,1,0),
defect(or1,1,1),
defect(xor1,0,0) ? ;
no
```

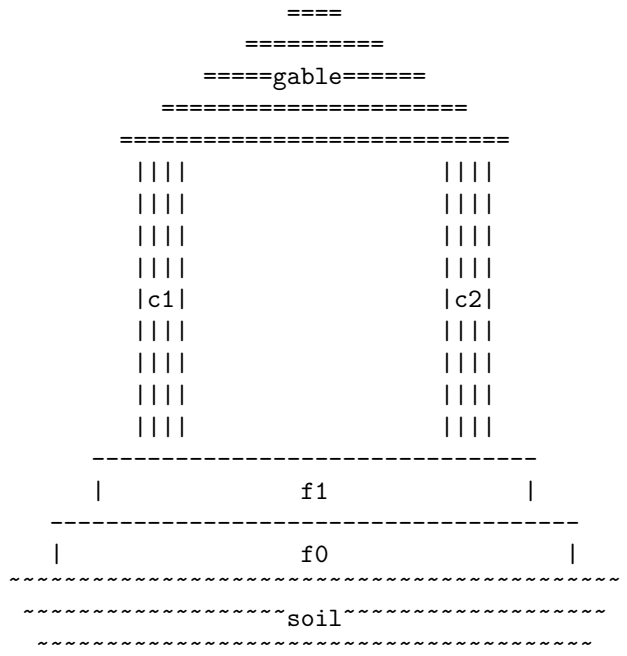
8.9 Discourse analysis as abduction

This section is not expanded in the present version; we refer to the articles [6, 5] for more information; perhaps you may need also to consult an introduction to Definite Clause Grammars which can be found in most textbooks on Prolog.

9 Exercises on Abductive Reasoning in Prolog and CHR

9.1 Exercise: Planning for Construction Works

The purpose of this exercise is to show howabduction can be applied for planning of complex projects. From some initial state (defining available resources, etc.), the goal is to reach a successful final end of project, and the plan is, then, a specification of actions and their order that will lead to that goal. Let us consider the project of building a little two-dimensional Greek temple of the following form.



It is constructed from five parts, identified as `[gable,c1,c2,f1,f2]`, and when building the temple, inherent physical constraints must be taken into account. For example, column `c2` is supported by the top layer `f1` of the foundation, so a plan that suggests to put `c2` in its place before `f1` should be rejected. The `soil` is already there and we can assume for homogeneity that it was placed in the universe at time `0`.

One way of viewing this planning task is that we need to put each part it its place (i.e., we are finished when all parts are in place), provided that inherent constraints for the plan are satisfied. We can assume that the first part is placed at time `1`, the next one at time `2`, and so on.

Having solved this exercise, you should have produced a running planning system using CHR+Prolog using the methods introduced in the course. The separate questions below are intended to guide you gradually to a solution.

A little auxiliary predicate

As you are not expected to have a long experience in Prolog programming, you are offered the following predicate which can be applied to take out the elements of a list in an arbitrary order (so that you can get along with the exercise).

Assume that the planning system's knowledge base includes the following fact, which gives the list of parts for our temple (their order is not interesting).

```
parts([gable,c1,c2,f0,f1]).
```

For the construction of a plan we need, at different times, to be able to take out some arbitrary part from this list, and send on the remaining list of components to further processing. For this purpose, we introduce a predicate with the following arguments,

`takePart(`*WhichPart*`,` *Parts*`,` *RemainingParts*`).`

So if we call, say, `takePart(P,[gable,c2,f1],Rest)`, we may get the different solutions under backtracking (three, in fact, for this example), one which is `P=c2, Rest=[gable,f1]`. ((You may notice that the standard `member` predicate also can be used to generate different members, but it lacks the ability to return the list of remaining elements.))

Here we show a definition of `takePart` which is based on `append` which in SICStus Prolog needs to be imported as part of a library.

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

takePart(X,List0,List1):-
    append(LeftRest,[X|RightRest],List0),
    append(LeftRest, RightRest,List1).
```

9.1.1 Database describing the architect's design

Construct a little database of Prolog facts that describes the aspects of the picture above that are necessary for solving the planning task. (The `parts` predicate shown above can also be considered part of this database.)

Type it into the computer and test it with some simple queries.

9.1.2 Abducibles and their integrity constraint(s)

Abduction means to specify the goal or observation, and the task is to figure out that basic facts ("abducibles") that can explain this goal. For planning tasks, the abducibles are individual steps of the plan, with an indication of at which relative time they should take place (see about time above).

Define using CHR these abducibles as constraints, and write one or more integrity constraints as CHR rules, that you believe are necessary for accepting sensible plans and throwing away the bad ones. As the plan is likely to be built up step by step, it is practical that these constraints works in a correct way also for partial plans. (Hint: It is possible to solve with a single CHR rule, which needs a guard.)

Type the solution into the computer and test it with some simple queries formed by abducibles (that you provide yourself).

9.1.3 The "driver algorithm"

The final component that you need to provide for the temple building planning system, is to define a predicate that describes what is meant by a plan, and how the initial state for the planning is defined and when a plan is finished. We have indicated all information in the introduction to this exercise, and your job is now to write it into Prolog.

Hint: One way to give a solution is to define a Prolog predicate `build(`*Parts*`,` *Time*`)` which makes one step of the plan and then generate a recursive call to solve the remaining problem.

Final remark

This exercise showed abduction applied for planning problems. You should be aware that methods for planning problems has been studied for several decades, and there exist multitudes of systems and algorithms for such purposes.

The method indicated in this exercise has very bad scaling properties unlezz additional optimization techniques are introduced.

9.2 Exercise: Diagnosis of power supply networks

The purpose of this exercise is to apply the abduction-based diagnosis method described in the course to find faults in power supply networks.

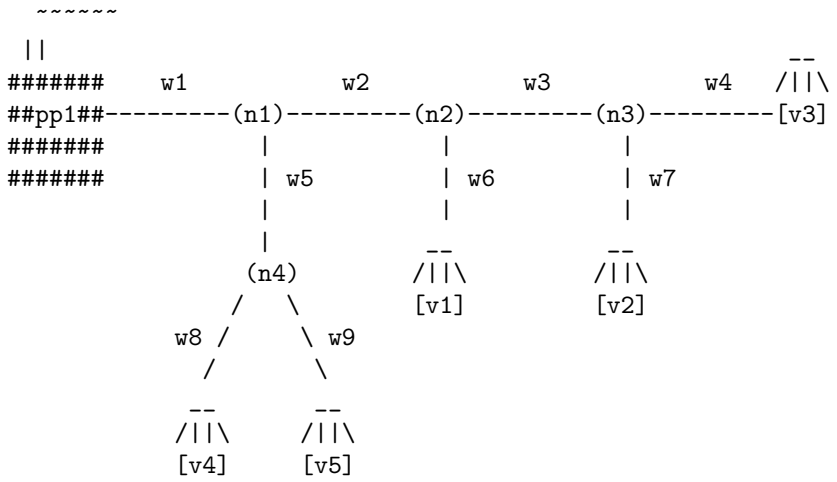
We consider a geographical region which has a power supply control centre that gets informed as soon as any village in the region is without electricity power. In such cases, the centre checks, using mobil phones, whether the remaining villages have electricity power. On the basis of this information, the centre must take the decision of where to send the repair teams. Our job is to develop a diagnosis system that given the information of which villages have or have not electricity power, can give suggestions for which wires or power plants that might be down, and thus are candidates for repair.

For this application, it seems reasonable to take the consistent fault assumption, as if a wire is down, it will stay down until it is repaired.

A given wire w will be in one of two states, `state(`w`,up)` or `state(`w`,down)`, so a diagnosis is a combination of abducibles concerning the different wires in the region. There is also a risk that the power plants in the region can go down, so a diagnosis may also contain abducibles such as `state(`*pp*`,down)` where *pp* refers to a power plant.

Observations are sets of "goals" of the form `has_power(`v`)` or `has_no_power(`v`)` where v refers to a village. In a given situation, we may have such information about all or just some villages.

The electricity power network can be depicted as follows; `pp1` is the only power plant in this region, `w1`–`9` are the wires, `n1`–`4` are nodes in which a number of wires are connected, and `v1`–`v4` are the villages.



For simplicity we assume that nodes always are working (so no `up` or `down` information for those). However, it is convenient to define the `has_power` and `has_no_power` predicates so they take as arguments both villages and nodes.

These predicates should, then, be defined as Prolog predicates (for the set of nodes and villages) that depend on the `state` of incoming wire and the possible `has_power` or `has_no_power` for the previous node.

In order to make the work with the exercise easier, a Prolog source file that models this power supply network is given, file `deductivePower` which is available from the course schedule web page. However, it does not include the diagnosis part. It is written as a purely deductive program[19] which contains as Prolog facts an arbitrary sample of `state(···,{up,down})` information. In this way, you are relieved from putting together the definitions of the `has_power` and `has_no_power` predicates which are a bit tricky, especially if your experience with Prolog is limited.

Before you start working with the questions below, it is strongly suggested that you make yourself familiar with this program, including giving queries to test which villages have power and which haven not. You may also try to change `state` facts and see how it affects the results.

9.2.1 Question 1

Starting from the source file `deductivePower` indicated above, your task is now to implement the indicated diagnosis system by changing `state` into a constraint of CHR (i.e., an abducible predicate). This implies, among other things, that you should remove the particular `state` facts from the program, as they have no meaning now and will only confuse the system.

As a technique to make testing easier, it may be suggested that you define use predicates such as the following (they are already in the source file).

```
observe_all_butV3:-
  has_power(v1), has_power(v2), has_no_power(v3),
  has_power(v4), has_power(v5).
```

[19] Recall for yourself what "deductive" means.

```
observe_all_butV1V3:-
  has_no_power(v1), has_power(v2), has_no_power(v3),
  has_power(v4), has_power(v5).

observe_V4V5:-
  has_no_power(v1), has_no_power(v2), has_no_power(v3),
  has_power(v4), has_power(v5).

observe_total_darkness:-
  has_no_power(v1), has_no_power(v2), has_no_power(v3),
  has_no_power(v4), has_no_power(v5).
```

Then you can simply type query such as `?- observe_all_butV3.` and check if the result is right.

Implement and test you systems.

9.2.2 Question 2

If you have succeeded in getting your solution to the previous question to work, you may try to make the task more complicated by adding an extra power plant `pp2` which is connected by a new wire `w10` to the node `n4`. The following assumptions are made.

- `pp2` is for emergency situations only in which power plant `pp1` goes down. We assume some fault-free (!) communication equipment that makes sure that `pp2` cannot be up if `pp1` is up, and the other way round. (But be prepared for the worst case when both power plants go down).
- Wire `w5` gets the special function, that if `pp1` is up, electricity runs from `n1` to `n4`; if `pp2` is up, it is the other way round.

Take the last assumption carefully into account when you revise the definitions of `has_power` and `has_no_power`; otherwise you may easily program infinite loops.

9.3 Programming project

Extend the temple building in a the following ways (one thing at a time):

1. Each step in the plan takes a certain amount of time to execute. For example, it may take different times to raise each of the two columns; the raising of both can take place at overlapping time intervals after that foundations are finished; however, the gable cannot be added before that last of the columns have been raised. Extend the program so that it take this aspects into account; you should add to the database component of the program, information about the time expected for each possible step.
2. Each step in the plan requires a number of workers to be performed. However, we may assume that we only have a fixed number of workers available, so this limits how many steps can be active at any given moment.

It may be the case that you need to extend the application to a more complex building in order to test your new program(s) in interesting ways.

References

[1] The programming language CHR, Constraint Handling Rules; official web pages. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`

[2] Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series*, pages 141–152. Physica-Verlag (Springer), 2000.

[3] Slim Abdennadher and Thom Frühwirth. *Essentials of Constraint Programming.* Springer, 2003.

[4] Ivan Bratko and Stephen Muggleton. Applications of inductive logic programming. *Commun. ACM*, 38(11):65–70, 1995.

[5] H. Christiansen and V. Dahl. HYPROLOG: a new approach to logic programming with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta, editors, *Proceedings of Twenty First International Conference on Logic Programming (ICLP 2005)*, Lecture Notes in Computer Science, 2005. To appear.

[6] H. Christiansen and V. Dahl. Meaning in Context. In Anind Dey, Boicho Kokinov, David Leake, and Roy Turner, editors, *Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05)*, volume 3554 of *Lecture Notes in Artificial Intelligence*, pages 97–111, 2005.

[7] Henning Christiansen. Introduction to Prolog as a database language. A course note, 2003. `http://www.ruc.dk/~henning/KIIS05/DatabaseProlog.pdf`

[8] Henning Christiansen and Veronica Dahl. Assumptions and abduction in Prolog. In Elvira Albert, Michael Hanus, Petra Hofstedt, and Peter Van Roy, editors, *3rd International Workshop on Multiparadigm Constraint Programming Languages, MultiCPL'04; At the 20th International Conference on Logic Programming, ICLP'04 Saint-Malo, France, 6-10 September, 2004*, pages 87–101, 2004.

[9] Peter A. Flach and Antonis C. Kakas, editors. *Abduction and Induction: Essays on their relation and integration.* Kluwer Academic Publishers, April 2000.

[10] Thom Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.

[11] A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming,* vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pages 235–324, 1998.

[12] Michael Negnevitsky. *Artificial Intelligence, A Guide to Intelligent systems.* Addison-Wesley, 2nd edition, 2004.

[13] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf, editors. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science.* Springer, 1997.

[14] Swedish Institute of Computer Science. SICStus Prolog user's manual, Version 3.12. Most recent version available at `http://www.sics.se/isl`, 2004.

[15] Andrew S. Tanenbaum. *Structured Computer Organization*. Addison-Wesley, 4th edition, 1999.