Chapter 9
Applications: Prolog; Relational Databases and SQL; Social Choice Theory

H.C.M. (Harrie) de Swart

9.1 Programming in Logic

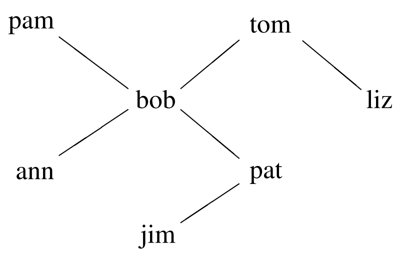
Abstract The language of logic can be used as a declarative programming language, i.e., the programmer has to describe *what* the problem is, not *how* it should be solved. We introduce logic programming by means of an example and explain how the system answers questions given a certain program. The possibility of recursive definitions is one of the cornerstones of logic programming. Prolog is a particular form of logic programming; it has been implemented in a certain way. As a consequence, although declarative in principle, Prolog also has certain procedural aspects. The syntax of logic programming in general and of Prolog in particular is very simple. Although the reasoning mechanism should use unification, many systems work with a simpler form, called matching, for reasons of efficiency. Lists are important terms in logic programming. Cut is a procedural device needed to keep programs efficient. Negation is implemented by means of cut and hence differs from logical negation. Logic programming has many applications in (deductive) databases and in Artificial Intelligence. We discuss the most important pitfalls.

Example 9.1. The best way to introduce the subject of logic programming seems to be to give an example of a concrete logic program. The following example is from I. Bratko [7].

```
parent(pam, bob).        (1)
parent(tom, bob).        (2)
parent(tom, liz).        (3)

parent(bob, ann).        (4)
parent(bob, pat).        (5)

parent(pat, jim).        (6)
grandparent(X,Z) :-
   parent(X,Y), parent(Y,Z).  (7)
```



H. de Swart, *Philosophical and Mathematical Logic*, Springer Undergraduate Texts in Philosophy, https://doi.org/10.1007/978-3-030-03255-5_9

This logic program consists of seven *clauses*. The first six of them are called *facts*; They express that pam is a parent of bob, tom is a parent of bob, etc. The last clause is called a *rule*; it expresses that X is a grandparent of Z if X is a parent of Y and Y is a parent of Z. The symbol :- is to be read as 'if' and the comma between 'parent(X,Y)' and 'parent(Y,Z)' is to be read as 'and'. 'X', 'Y' and 'Z' are called *variables*; it is allowed to replace them by the names of arbitrary individual objects. In the rule 'L :- L_1, L_2', 'L' is called the *head* of the rule and 'L_1, L_2' is called the *body*.

Given a logic program P, the user may ask questions. Given the logic program just presented one might ask who are the parents of bob; in other words, for which X is it true that X is a parent of bob? This question is formulated as follows:

:- parent(X, bob).
or
?- parent(X, bob).

The question whether bob and liz have a common parent is formulated as follows:

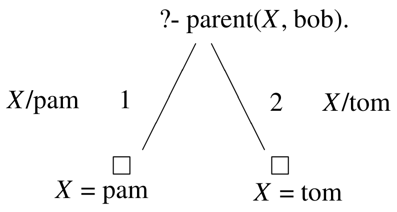
:- parent(X, bob), parent(X, liz).
or
?- parent(X, bob), parent(X, liz).

In *Prolog*, which stands for PROgramming in LOGic and which is just a particular form of logic programming, the answers to these questions are found as follows. Given the logic program mentioned above and given the question

?- parent(X, bob).

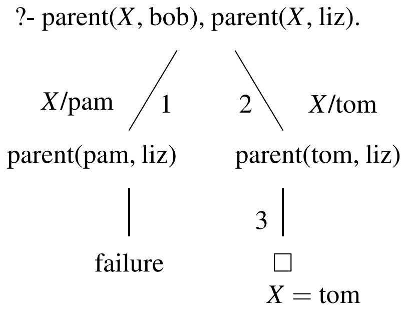
the Prolog system tries to *match* or *unify* the clause 'parent(X, bob)' with the first fact 'parent(pam, bob)' in the given program. This matching or unifying succeeds by replacing the variable 'X' by 'pam'. So, 'X = pam' is the first answer to this question. Next, the Prolog system tries to match or unify the clause 'parent(X, bob)' with the second fact 'parent(tom, bob)' in the given program, yielding the second answer 'X = tom'. Since the Prolog system cannot succeed in unifying or matching the clause 'parent(X, bob)' with the other facts in the program, there are no more answers to this question.

The following picture describes graphically how the Prolog system finds the answers to the question '?- parent(X, bob).' given the program in Example 9.1.



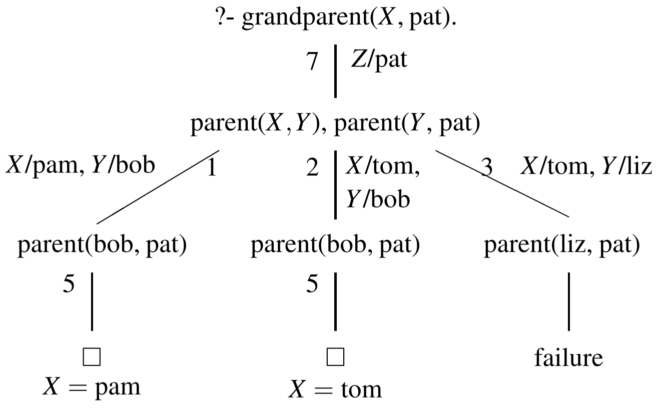
This picture is called the *search tree* for the question '?- parent(X, bob).' given the program above. The numbers 1 and 2 refer to the first and second facts in the program. 'X/pam' indicates that the variable 'X' is substituted by 'pam' in order to match the clause 'parent(X, bob)' with fact (1) in the program. The symbol '□' indicates that no other questions remain to be answered.

The search tree for the question ‘do bob and liz have a common parent?’ given the logic program in Example 9.1 looks as follows.



In this example there are two simultaneous questions or *goals*: parent(X, bob), parent(X, liz). The Prolog system selects the *left-most* goal first and tries to match it with the facts in the program. The first possibility to do so is by matching it with fact (1) in the program via the *substitution* X/pam. Then only the goal ‘parent(X, liz)’ remains with ‘X’ replaced by ‘pam. However, this goal cannot be realized, in the sense that there is no such fact in the program. Hence, this trial to realize the two goals fails. Then the Prolog system *backtracks* and tries to realize the first goal in another way. This can be done by matching it with fact (2) in the program via the substitution X/tom. The second goal ‘parent(X, liz)’ remains with ‘X’ replaced by ‘tom’. Since ‘parent(tom, liz)’ occurs as fact (3) in the program, this goal can be realized by the program and no other goals remain.

Given the logic program in Example 9.1, the search tree for the question ‘who are pat’s grandparents?’, i.e., ‘for which X is it true that X is a grandparent of pat’, looks as follows.



Given the question or goal ‘?- grandparent(X, pat).’, the Prolog system looks for facts of this form in the given program, but does not find any. It also tries to match or unify the goal with the head of a rule in the program. This succeeds: the goal ‘grandparent(X, pat)’ can be unified with the head of clause (7) in the program via the substitution Z/pat. Then the original goal is replaced by two new goals:

parent(X,Y), parent(Y, pat). The *left-most* goal is selected first. Looking at the program, we see that there are six different ways to realize this goal. The first possibility is to use the first clause in the program via the substitution X/pam, Y/bob. The second goal 'parent(Y, pat)' remains with 'Y' substituted by 'bob'. Since 'parent(bob, pat)' is the 5^{th} clause in the program, this goal is realized and no other goals remain. The Prolog system answers 'X = pam'. Next the system *backtracks* and tries to realize the left-most goal 'parent(X,Y)' in another way. This can be done by using the second clause in the program via the substitution X/tom, Y/bob. Then the second goal 'parent(Y, pat)' remains with 'Y' replaced by 'bob'. Since this is the 5^{th} fact in the program, the second branch in the search tree is completed successfully and the system answers 'X = tom'. Again, backtracking takes place and Prolog realizes the left-most goal 'parent(X,Y)' by using the third fact in the program via the substitution X/tom, Y/liz. Then the goal 'parent(liz, pat)' remains. Looking at the program, the Prolog system discovers that this goal cannot be realized. It backtracks and tries to realize the left-most goal 'parent(X,Y)' in a fourth way, and so on.

Summarizing: In logic programming a program consists of *facts* and *rules*. A logic program is a kind of database. A question is a finite sequence of one or more goals. Given a logic program, questions are answered by trying exhaustively to realize the goals by *matching* (or *unifying*) them with the facts and/or the heads of the rules in the program, possibly via *substitution* of the variables. A logic programming system accepts facts and rules as a set of (non-logical) axioms and a question as a putative theorem or conclusion. The logic programming system tries to deduce this putative conclusion logically from the axioms.

Typical features of Prolog are that it has a *left-most selection rule*, that the search is *depth-first* (not breadth-first) and that after successful or unsuccessful termination of a branch in the search tree, *backtracking* takes place in order to find alternative solutions.

The reader is advised to do Exercise 9.1.

9.1.1 Recursion

The use of recursive definitions (of predicates or relations) is typical in logic programming. As an example we might add the following two clauses to the logic program in Example 9.1:

pred(X,Z) :- parent(X,Z).
pred(X,Z) :- parent(X,Y), pred(Y,Z).

$$\begin{tikzcd}
X \arrow[d, dash, "\text{parent}"] \\
Y \arrow[d, dotted, dash, "\text{pred}"] \\
Z
\end{tikzcd}$$

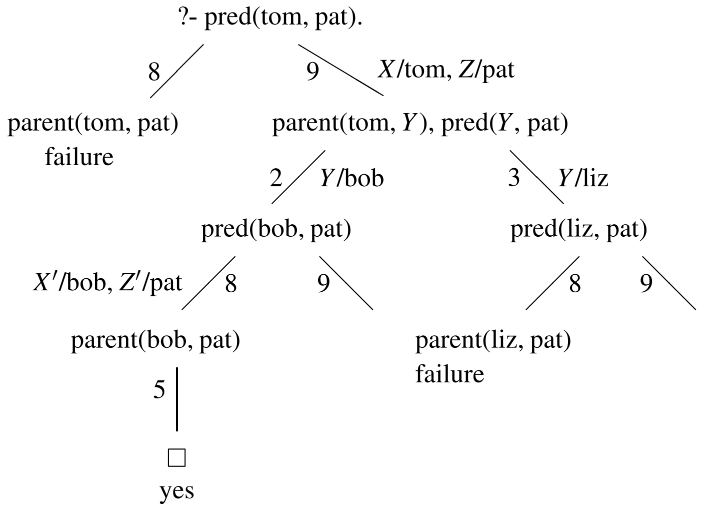
These two clauses define the predecessor relation, abbreviated by 'pred'. The first clause expresses that X is a predecessor of Z if X is a parent of Z. And the second one expresses that X is a predecessor of Z if (for some Y) X is a parent of Y and

Y is a predecessor of *Z*. So, in the second clause the relation 'pred' recurs in the definition of 'pred(X,Z)'. For this reason one speaks of a *recursive definition*.

In order to understand the role of recursive definitions in logic programming, let us ask the question 'is tom a predecessor of pat?', given the program which results from adding the two clauses for 'pred' to the program in Example 9.1. This program then looks as follows:

	parent(pam, bob).	(1)
	parent(tom, bob).	(2)
	parent(tom, liz).	(3)
	parent(bob, ann).	(4)
Example 9.2.	parent(bob, pat).	(5)
	parent(pat, jim).	(6)
	grandparent(X,Z) :- parent(X,Y), parent(Y,Z).	(7)
	pred(X,Z) :- parent(X,Z).	(8)
	pred(X,Z) :- parent(X,Y), pred(Y,Z).	(9)

Given this program, the search tree for the question (or goal) 'is tom a predecessor of pat?' looks as follows:



In order to answer the question 'pred(tom, pat)' the Prolog system tries to unify this goal with a fact or the head of a rule in the program. The first possibility is to unify 'pred(tom, pat)' with the head of clause (8) via the substitution *X*/tom, *Z*/pat. The goal 'parent(tom, pat)' is the result. Since there is no such fact in the given program, the left-most branch in the search tree terminates unsuccessfully and backtracking takes place. The original goal can also be unified with the head of clause (9) in the program via the substitution *X*/tom, *Z*/pat. Then the original goal is replaced by two new goals: parent(tom, *Y*), pred(*Y*, pat). Prolog selects the left-most goal first. It can be unified with clause (2) in the program via the substitution *Y*/bob. Then the goal 'pred(*Y*, pat)' remains with '*Y*' substituted by 'bob'. In order to realize this latter goal, Prolog first matches it with the head of clause (8). Since '*X*' and '*Z*' have already been substituted by ‘tom’ and ‘pat’ respectively, the Prolog system has replaced the variables X and Z in clause (8) by X' and Z' respectively. Clause (8) now looks as follows:

$$\text{pred}(X',Z') \text{ :- } \text{parent}(X',Z'). \qquad (8')$$

The goal ‘pred(bob, pat)’ can now be unified with the head of clause (8′). This yields the new goal ‘parent(bob, pat)’. Because of the 5th clause in the program, the second branch (from the left) in the search tree terminates successfully, and the answer to the original question is ‘yes’. Backtracking takes place in order to see whether the goal ‘pred(bob, pat)’ can be realized in other ways. In our search tree we have not worked this out. But it turns out that the goal ‘pred(bob, pat)’ cannot be realized in another way. Further backtracking takes place in order to see whether the left-most goal in ‘parent(tom, Y), pred(Y, pat)’ can be realized in another way. This may indeed be the case. Applying clause (3) in the program and substituting ‘liz’ for ‘Y’, the goal ‘pred(Y, pat)’ remains with ‘Y’ replaced by ‘liz’, and so on.

9.1.2 Declarative versus Procedural Programming

Logic programming is in principle *declarative*: the programmer only has to describe the problem, in other words, he must formulate *what* the problem is; but he does not have to specify *how* the problem has to be solved. The logic programmer is more concerned with *knowledge* than with *algorithms*.

In order to answer certain questions, the logic programmer has to formulate all relevant information, consisting of facts and rules, in a logic program. Given a certain program, the logic programming system will try systematically to deduce the answer to any question from the facts and rules in the program. This is done by an exhaustive search, which can be represented in a *search tree*.

Given the program in Example 9.2, the answer to the question ‘is tom a predecessor of pat?’ is ‘yes’. This means that pred(tom, pat) logically follows from (is a valid consequence of) the facts and the rules in the program. The answer to the question ‘is pam a predecessor of liz?’ will be ‘no’, meaning that pred(pam, liz) is not a logical consequence of the (facts and rules in the) program. This does **not** mean that $\neg$ pred(pam, liz) is a logical consequence of the program in question! And the answer to the question

$$\text{?- parent}(X, \text{bob}), \text{parent}(X, \text{liz}).$$

was ‘X = tom’. This means that both parent(tom, bob) and parent(tom, liz) logically follow from the given program.

One can prove what is called *soundness*: given any logic program P and question or goal G every computed answer logically follows from P. The converse problem is *completeness*: given any logic program P, is the logic programming system able to compute any goal which logically follows from P? This problem is more difficult and cannot be answered with a simple ‘yes’ or ‘no’.

Programming languages like Pascal, Algol and C are *procedural* languages: the programmer has to specify *how* the problem has to be solved. Although logic programming is in principle declarative and not procedural, the logic programmer has to take into account certain procedural aspects. We have already noticed above that Prolog, being a particular – but most popular – logic programming system, has a *left-most selection rule* and a *depth-first search* strategy. This strategy first develops the left-most branch in the search tree. As long as this branch has not been terminated, no other branches are developed. It also searches for facts and rules in the program in the order they have been programmed. The programmer, who writes a logic program in Prolog, has to take the procedural aspects of the Prolog system into account. The order of the facts and rules in his program may be important and even the order of the goals in the body of a rule may be important. In order to make this clear, look at the following four definitions of the predecessor relation.

pred(X,Z) :- parent(X,Z).
pred(X,Z) :- parent(X,Y), pred(Y,Z).

pred2(X,Z) :- parent(X,Y), pred2(Y,Z).
pred2(X,Z) :- parent(X,Z).

pred3(X,Z) :- parent(X,Z).
pred3(X,Z) :- pred3(X,Y), parent(Y,Z).

pred4(X,Z) :- pred4(X,Y), parent(Y,Z). (I)
pred4(X,Z) :- parent(X,Z) . (II)

In the definition of ‘pred2’ and ‘pred4’ the order of the clauses is reversed with respect to the definition of ‘pred’. And in the definition of ‘pred3’ and ‘pred4’ the goals in the body of the recursion clause are reversed with respect to the definition of ‘pred’. This may have disastrous consequences. Consider the search tree for ?- pred4(tom, pat).

$$\begin{tikzcd}
\text{?- pred4(tom, pat).} \arrow[d, dash, "\text{I}"', "X/\text{tom},\ Z/\text{pat}"] \\
\text{pred4(tom, }Y\text{), parent(}Y\text{, pat)} \arrow[d, dash, "\text{I}"', "X'/\text{tom},\ Z'/Y"] \\
\text{pred4(tom, }Y'\text{), parent(}Y',Y\text{), parent(}Y\text{, pat)} \arrow[d, dash, "\text{I}"'] \\
\vdots \\
\text{ad infinitum}
\end{tikzcd}$$

The left-most branch in this search tree will be infinitely long since clause I will be applied again and again. And the other branches in the search tree will not be developed. So, the Prolog system will give no answer to the question ‘?- pred4(tom, pat).' and although the Prolog system does answer the questions '?- pred2(tom, pat).' and '?- pred3(tom, pat).' positively, Exercise 9.2 makes clear that from a procedural point of view, pred2 and pred3 do not give an adequate description of the predecessor relation.

To summarize, although the definitions of 'pred', 'pred2', 'pred3' and 'pred4' are equivalent from a logical or declarative point of view, they are quite different from a procedural point of view. Consequently, the logic programmer has to take into account the particular procedural aspects of the logic programming system he or she is working with. For recursive definitions there is a simple rule:
i) The more simple clause is put first (supposing that the logic programming system searches through the program from top to bottom). In the case of the predecessor relation this is the clause 'pred(X,Z) :- parent(X,Z).'
ii) The goals in the body of the recursion clause should be ordered from simple to more complex (supposing a left-most selection rule in the logic programming system). So, the recursion clause for the predecessor relation should be

$$\text{pred}(X,Z)\text{ :- parent}(X,Y),\ \text{pred}(Y,Z).$$

since 'parent' is a more simple relation than 'pred'.

Another procedural aspect of Prolog is the cut, denoted by '!'. The cut prunes part of the search tree. This enhances efficiency, but may be dangerous if the pruned part contains successful branches. We will discuss this feature of Prolog further on in Subsection 9.1.6.

9.1.3 Syntax

The syntax of a logic programming language is that of first-order predicate logic (see Chapter 4) with possibly some modifications in notation.

Definition 9.1 (Alphabet of Prolog). The *alphabet* consists of:
a) *Individual variables*, such as $X1, X2, \ldots, X, Y$, Person, ...
b) *Individual constants*, such as pam, bob, liz, ... ; 0, 1, 2, ...
c) *Function symbols*, such as father_of, mother_of, ... ; $+$, $*$, ...
d) *Predicate symbols*, such as parent, male, female; =, $<$, ...

The programmer is free to choose his own symbols in the alphabet. However, in Prolog any expression starting with a capital is a variable. For instance, the expressions 'Pam' and 'Bob' are variables in Prolog, while 'pam' and 'bob' are individual constants. Each function and predicate symbol is k-ary for some k; the arity is chosen by the programmer.

Definition 9.2 (Terms). *Terms* are defined as in Chapter 4:
a) Each individual variable and each individual constant is a term.
b) If f is a k-ary function symbol and $t_1, \ldots, t_k$ are terms, then $f(t_1, \ldots, t_k)$ is a term.

Examples of terms are: 1) Person, tom, bob; 1, 2.
2) father_of(Person), mother_of(bob); +(1, 2), usually written as $1+2$.
3) mother_of(father_of(Person)); $*$(1, +(1, 2)), usually written as $1*(1+2)$.

Definition 9.3 (Atomic Formulas). *Atomic formulas* are defined as in Chapter 4: if p is an n-ary predicate symbol and $t_1,\ldots,t_n$ are terms, then $p(t_1,\ldots,t_n)$ is an atomic formula.

Examples of atomic formulas: parent(tom, bob), parent(X, liz), female(X), male(Y), male(bob); = (X, 1), usually written as $X = 1$, and <(2, 3), usually written as $2 < 3$.

Definition 9.4 (Definite Program Clause, Goal, Program, Clause).
a) A *definite program clause* is any expression of the form B :- $A_1,\ldots,A_m$ $(m \geq 0)$, where B and $A_1,\ldots,A_m$ are atomic formulas (Cf. Definition 2.10). If $m = 0$, one writes simply 'B' instead of 'B :- '. 'B :- $A_1,\ldots,A_m$' is to be read as: B if A_1 and ... and A_m; in the notation of Chapter 4: $B \leftarrow A_1 \wedge \ldots \wedge A_m$. B is called the *head* and $A_1,\ldots,A_m$ is called the *body* of the clause B :- $A_1,\ldots,A_m$. If $m = 0$, one calls the definite program clause a *fact*, otherwise a *rule*.
b) A *definite goal* is any expression of the form :- $A_1,\ldots,A_m$ $(m \geq 0)$, where $A_1,\ldots,A_m$ are atomic formulas. Each A_i is called a *subgoal* of the goal. If $m = 0$, one speaks of the *empty goal* or *empty clause*, denoted by '$\square$'.
c) A *definite program* is a finite set of definite program clauses. See, for instance, Example 9.1.
d) A *definite clause* or *Horn clause* is either a definite program clause or a definite goal.

Definition 9.5 (Literal, Normal Program Clause, Goal, Program).
a) A *literal* is an atomic formula A or the negation 'not A' of an atomic formula A. In the notation of Chapter 4 'not A' was written as '$\neg A$'.
b) A *normal program clause* is any expression of the form B :- $L_1,\ldots,L_m$, where B is an atomic formula and $L_1,\ldots,L_m$ are literals.
Example: sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X), not $X = Y$.
c) A *normal goal* is any expression of the form :- $L_1,\ldots,L_m$, where $L_1,\ldots,L_m$ are literals.
d) A *normal program* is a finite set of normal program clauses.

From a theoretical point of view, in particular with respect to completeness, definite logic programs are to be preferred to normal logic programs. However, for practical purposes the latter ones are often needed. We return to the problems connected with negation in Subsection 9.1.7 and 9.1.9.

Definite and normal program clauses are formulas of a special kind. In order to see this, let us repeat the definition of *formulas* as given in Chapter 4.

Definition 9.6 (Formulas).
a) Any atomic formula is a formula.
b) If A and B are formulas, then so are $(A \rightleftarrows B)$, $(A \rightarrow B)$, $(A \wedge B)$, $(A \vee B)$ and $(\neg A)$, to be read as 'A if and only if B', 'if A, then B', 'A and B', 'A or B' and 'not A', respectively.

c) If A is a formula and x is an individual variable, then $\forall x[A]$ and $\exists x[A]$ are formulas, to be read as 'for all x, A' and 'there is at least one x such that A', respectively. (See Definition 4.5 for more details.)

In Theorem 2.18 we have shown that any formula not containing the quantifiers $\forall$ and $\exists$ is equivalent to a finite conjunction of clauses, where a *clause* is a formula of the form $A_1 \wedge \ldots \wedge A_m \to B_1 \vee \ldots \vee B_n$ or, in different notation, $B_1 \vee \ldots \vee B_n$:- $A_1, \ldots, A_m$, the A's and B's being *atomic* formulas. (See the topic on Knowledge Representation and Prolog in Subsection 2.5.2.) Note that $B_1 \vee \ldots \vee B_n$:- $A_1, \ldots, A_m$ is equivalent to $B_1 \vee \ldots \vee B_n \vee \neg A_1 \vee \ldots \vee \neg A_m$.

However, in Prolog only *definite clauses* are allowed, i.e., clauses with $n \leq 1$, for reasons of efficiency. For instance, suppose we had a program containing the clause $p(1) \vee q(2)$. Now the question ?- $p(X) \vee q(X)$ should be answered as follows: $X = 1$ if $p(1)$ and $X = 2$ if $q(2)$. It is hard to implement a system that is able to give such conditional answers.

In Subsection 4.3.6 on Skolemization and Clausal Form we have defined the *clausal form* $C(A)$ of any formula A, such that 1) $C(A)$ is a conjunction of clauses, and 2) A is satisfiable iff $C(A)$ is satisfiable. And in Subsection 4.7.3 on Logic and Artificial Intelligence we have shown that any (definite) logic program is a formula in clausal form.

9.1.4 Matching versus Unification

In the preceding examples we have seen that a Prolog system, given a certain program and answering a certain question, makes use of what is called matching or unification. In this subsection we want to describe matching and unification more precisely and to point out the difference between them.

Definition 9.7 (Matching). *Matching* is a process that takes as input two terms or atomic formulas and checks whether they match. The rules governing this process are the following:

1. Two individual constants match only if they are syntactically the same.
2. If X is a variable and t a term, then they match and X is instantiated to, or substituted by, t.
3. Two terms match only if they have the same principal function symbol and all their arguments match.
4. Two atomic formulas match only if they have the same principal predicate symbol and all their arguments match.

Example 9.3. a) The pair {parent(X, bob), parent(pam, bob)} can be matched.
b) The pair {parent(X, bob), parent(pat, jim)} cannot be matched, because the second arguments cannot be matched.
c) The pair $\{p(f(X),Z),\ p(Y,c)\}$ can be matched via the substitution $Y/f(X),\ Z/c$.
d) The pair $\{p(f(X),c),\ p(Y,f(Z))\}$ cannot be matched, since the second arguments cannot be matched.

e) The pair $\{p(X,X),\ p(Y,f(Y))\}$ can be matched. The first arguments can be matched via the substitution X/Y. The resulting pair is $\{p(Y,Y),\ p(Y,f(Y))\}$. The second arguments can be matched via the substitution $Y/f(Y)$.

The following logic program shows that in some cases matching yields undesirable results. Let P be the following logic program:

$$\text{parent}(Y, \text{child\_of}(Y)). \tag{1}$$

Note that 'parent(X,X)' is similar to '$p(X,X)$' and that 'parent(Y, child_of(Y))' is similar to '$p(Y,f(Y))$' in Example 9.3 e). Given this program, the search tree for the question '?- parent(X,X).' looks as follows.

$$\begin{tikzcd}
?\text{- parent}(X,X). \arrow[d, dash, "(1)"', "X/Y,\ Y/\text{child\_of}(Y)"] \\
\square \\[-2em]
\text{yes}
\end{tikzcd}$$

However, this result is undesired, since 'parent(X,X)' does *not* logically follow from program P. In the intended interpretation the only formula (1) in P expresses a true proposition, while 'parent(X,X)' expresses a false proposition for any value of X.

For reasons of efficiency, most logic programming systems make use of matching and take for granted that in some cases this may yield the wrong results. What they should do, however, is to replace matching by unification and take for granted that in some cases this may be inefficient.

Definition 9.8 (Unification). *Unification* is characterized by the following slogan:

$$\text{Unification = matching + occur check.}$$

The *occur check* involves checking whether in the substitution of a term t for a variable X (clause 2 in the definition of matching), the variable X does not occur in t. If X does occur in t, then unification fails, while matching may succeed.

In Example 9.3 e) we have seen that the pair $\{p(X,X),\ p(Y,f(Y))\}$ can be matched. However, this pair cannot be unified. After the substitution X/Y the resulting pair is $\{p(Y,Y),\ p(Y,f(Y))\}$. And although the second arguments in this pair can be matched, they cannot be unified since the variable Y does occur in $f(Y)$.

The *unification algorithm* is like the matching algorithm given above, except that the occur check is added to clause 2. Below, we demonstrate how the unification algorithm works in a few examples.

Example 9.4. (Lloyd [16]):
Is it possible to unify $p(f(c),g(X))$ and $p(Y,Y)$?
1) The predicate symbols are identical.
2) The left-most arguments that differ are $f(c)$ and Y. *Occur check*: Y does not occur in $f(c)$. So, replace Y by $f(c)$. Result: $p(f(c),g(X))$ and $p(f(c),f(c))$.
3) The left-most arguments that differ are $g(X)$ and $f(c)$. These terms have different principal function symbols and hence cannot be unified.
Conclusion: $p(f(c),g(X))$ and $p(Y,Y)$ cannot be unified. Nor can they be matched.

Example 9.5. (Lloyd [16]):
Can $p(c,X,h(g(Z)))$ and $p(Z,h(Y),h(Y))$ be unified?
1) The predicate symbols are identical.
2) The left-most arguments that differ are c and Z. *Occur check*: Z does not occur in c. So, replace Z by c. Result: $p(c,X,h(g(c)))$ and $p(c,h(Y),h(Y))$.
3) The left-most arguments that differ now are X and $h(Y)$. *Occur check*: X does not occur in $h(Y)$. So, replace X by $h(Y)$. Result: $p(c,h(Y),h(g(c)))$ and $p(c,h(Y),h(Y))$.
4) The left-most arguments that differ now are $h(g(c))$ and $h(Y)$. The principal function symbols are identical. The arguments $g(c)$ and Y are different. *Occur check*: Y does not occur in $g(c)$. So, replace Y by $g(c)$. Result: $h(g(c))$ and $h(g(c))$.
Conclusion: $p(c,X,h(g(Z)))$ and $p(Z,h(Y),h(Y))$ are unifiable via the substitutions Z/c, $X/h(Y)$ and $Y/g(c)$.

For more details about substitution and unification the reader is referred to Lloyd [16]. See also Exercise 9.3.

Note that both the matching and the unification algorithms result in a *most general unifier* (substitution) in the sense that no more is substituted than strictly necessary. For instance, unifying date(D, M, 1983) and date(D1, may, Y) results in the substitution $D/D1$, M/may and Y/1983. The substitutions D/3, D1/3, M/may and Y/1983 also unify the two terms, but are less general.

One can show that given a logic program P any answer to a question is correct in the sense that the computed answer is a logical consequence of the given program, provided the system uses unification instead of matching. For instance, given the program of Example 9.1 the Prolog system computes two answers to the question ?- grandparent(X, pat): X = pam; X = tom. The theorem just mentioned, called the *soundness* theorem, then says that ‘grandparent(pam, pat)’ and ‘grandparent(tom, pat)’ are logical consequences of the given program. For more details we refer the reader to Lloyd [16] where also the converse problem is discussed whether any logical consequence of a given program can be computed by the Prolog system (*completeness*).

9.1.5 Lists, Arithmetic

Lists are very important terms in the practice of logic programming. For instance, if one wants to represent information about families in a logic program, one has the problem that different families have different numbers of children. By putting the children of any family in a list, one can represent any family in a uniform way:

family(Father, Mother, List_of_Children).

Here, ‘family’ is a predicate symbol taking three arguments, no matter how many children there are.

Lists are terms of a special kind, defined as follows.

Definition 9.9 (Lists).
1) [ ] is a list, called the *empty list*.
2) If t is any term and L is a list, then $[t \mid L]$ is a list.
In $[t \mid L]$, t is called the *head* and L is called the *tail* of the list $[t \mid L]$.

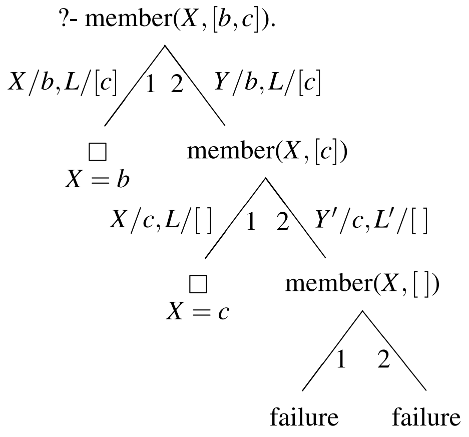
Example 9.6. Examples of lists:

1)	[ ]	2)	$[c \mid [\,]]$, usually rendered as $[c]$.
3)	$[b \mid [c]]$, usually rendered as $[b,c]$.	4)	$[a \mid [b,c]]$, usually rendered as $[a,b,c]$.

Working with lists, one needs a program that determines the elements or members of a given list. Reading 'member(X,L)' as 'X is a member of list L', the membership relation is defined recursively as follows.

Definition 9.10 (Member). member$(X, [X \mid L])$. (1)
member$(X, [Y \mid L])$:- member(X,L) . (2)

In words: X is a member of a given list if (1) X is the head of the list, or (2) X is a member of the tail of the list. Given this program, the search tree for the question '?- member$(X,[b,c])$.' (what are the members of the list $[b,c]$?) looks as follows.



In Exercise 9.4 the reader is invited to define a concatenation relation for lists and in Exercise 9.5 to define a relation for deleting members from a given list.

Prolog contains some built-in arithmetic operations which can be used in the infix notation, i.e., $2+3$ instead of $+(2,3)$, etc. Among them are

$+$	for addition,	$*$	for multiplication,
$-$	for subtraction,	$/$	for division.

When doing arithmetic in Prolog it is important to realize that '=' is a built-in matching operator, while 'is' is a built-in operator that forces the evaluation of the term in question. In order to make the difference clear, consider the following examples.

```
?- X = 2 + 3.     ?- X is 2 + 3.     ?- 2 + 3 = 3 + 2.
X = 2 + 3         X = 5              no
```

Other built-in operators which force the evaluation of the terms in question are:

$>$	is greater than;
$>=$	is greater than or equal to;
$=:=$	the values of the left and right terms are equal;
$=\backslash=$	the values of the left and right terms are not equal.

Example 9.7. ?- $2+3 =:= 3+2$.
yes

?- $2*3 > 5$.
yes

It is important to realize that all arguments must be instantiated to numbers at the time that the evaluation is carried out. Examples:

?- X is $2*3$, $X > 5$.
$X = 6$

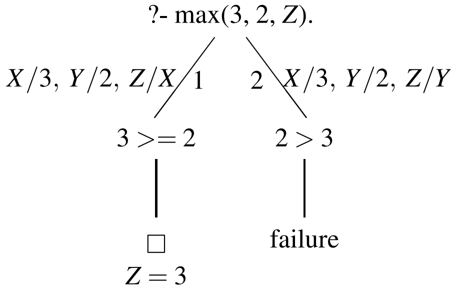
?- $X > 5$, X is $2*3$.
control error

9.1.6 Cut

max(X,Y,Z), to be read as 'Z is the maximum of X and Y', can be defined as follows.

$$\text{max}(X,Y,X) \text{ :- } X >= Y. \qquad (1)$$
$$\text{max}(X,Y,Y) \text{ :- } Y > X. \qquad (2)$$

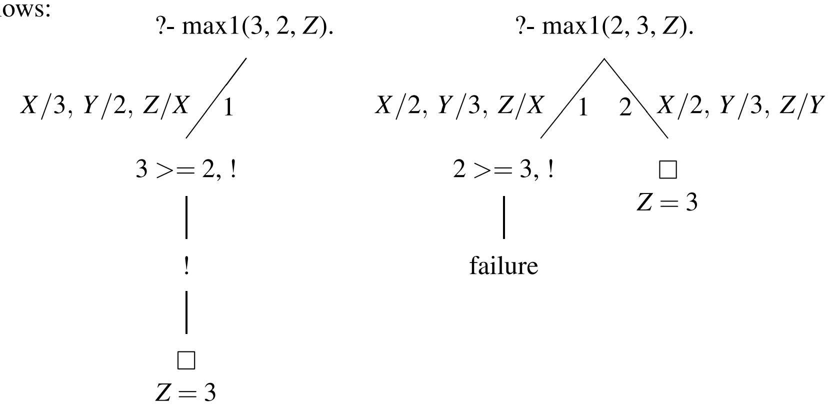
Now the programmer, but not the logic programming system, knows that if the goal $X >= Y$ succeeds, then the goal $Y > X$ is bound to fail. So, given this program and the question ?- max(3, 2, Z), it is a waste of time and energy to try to apply the second clause via backtracking, once the left-most branch in the search tree has terminated successfully.



It is attractive to have a control facility that prunes that part of a search tree that only contains unsuccessful branches. Prolog has such a control facility, called *cut* and denoted by '!'. The cut ! can be conceived of as a true atomic formula or as a goal that always succeeds. However, while the declarative or logical meaning of '!' is 'true', the procedural meaning of '!' is the pruning of the search tree. Given the program

$$\text{max1}(X,Y,X) \text{ :- } X >= Y, !. \qquad (1)$$
$$\text{max1}(X,Y,Y). \qquad (2)$$

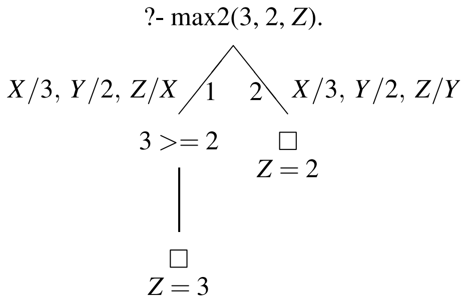
the search trees for the questions ?- max1(3, 2, Z) and ?- max1(2, 3, Z) look as follows:



The right-most branch in the search tree for ?- max1(3, 2, Z) is pruned because first the goal $3 >= 2$ succeeds and next the cut is passed. The goal $2 >= 3$ in the left-most branch in the search tree for ?- max1(2, 3, Z) fails, hence the cut is not passed and backtracking takes place as usual.
From a procedural point of view, the programs for max(X,Y,Z) and for max1(X,Y,Z) yield the same results. However, from a declarative or logical point of view the program for max1(X,Y,Z) is not an adequate description of the maximum relation. Since the declarative meaning of '!' is 'true', from a declarative point of view the program for max1 is equivalent to the following one.

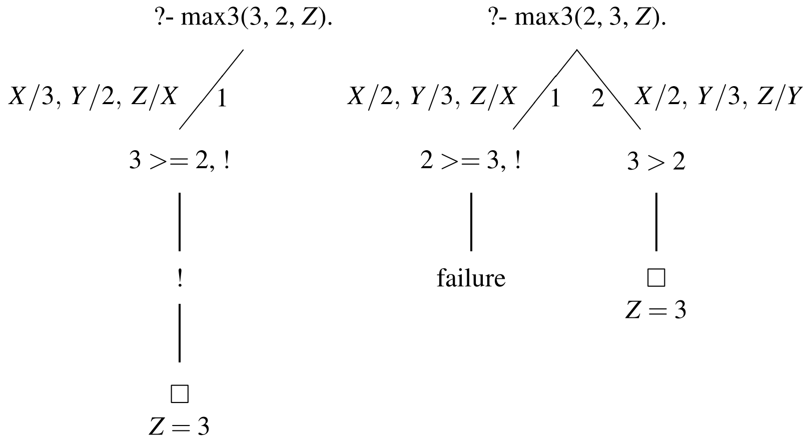
$$\text{max2}(X,Y,X) \text{ :- } X >= Y \qquad (1)$$
$$\text{max2}(X,Y,Y). \qquad (2)$$

But the question ?- max2(3, 2, Z) yields two answers: $Z = 3$ and $Z = 2$.



So, if one wants a program for the maximum relation that is both correct from a declarative point of view and efficient from a procedural point of view, the following program is to be preferred.

$$\text{max3}(X,Y,X) \text{ :- } X >= Y, ! . \qquad (1)$$
$$\text{max3}(X,Y,Y) \text{ :- } Y > X . \qquad (2)$$



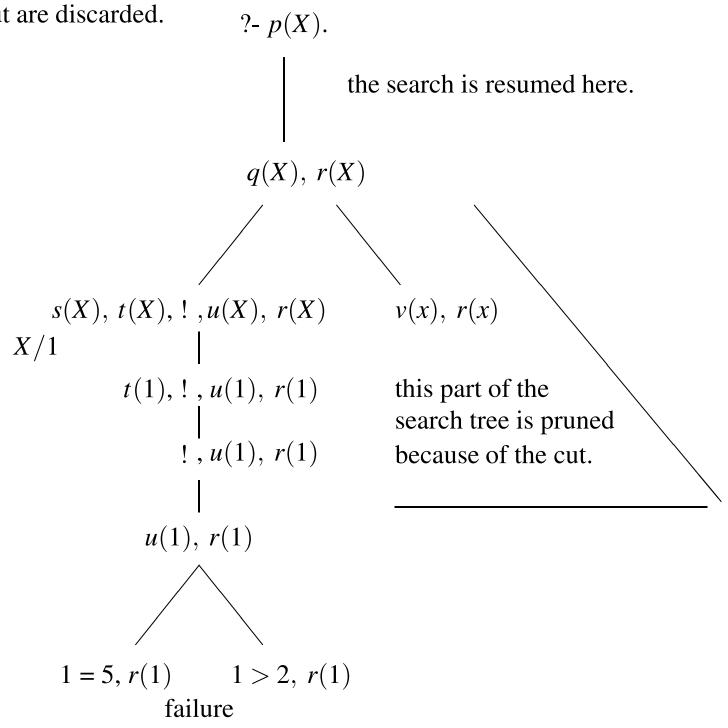
So, a cut prunes the search tree. This is safe if the pruned part contains no successful branches. In that case the cut merely enhances efficiency; it saves time. However, if the pruned part contains successful branches, the use of cut may have disastrous consequences. For that reason the programmer should be very careful in using this control facility. Unfortunately, more complicated programs often require the use of cuts in order to keep the program efficient.

What part of the search tree is pruned by using cut? In order to answer this question more precisely, consider the following program P.

```
p(X) :- q(X), r(X).
⋮
q(X) :- s(X), t(X), !, u(X).
q(X) :- v(X).
⋮
r(1).
s(1).
t(1).
u(X) :- X = 5.
u(X) :- X > 2.
```

The following picture shows the effect of cut on the search tree for the question '?- $p(X)$.', given program P above. Given the program P above, the goal '?- $p(X)$.' will be answered with 'no', even if we add the fact $v(1)$ to P. In that case there is a successful branch in the search tree, namely, the branch with $v(X)$, $r(X)$. However, this branch will be pruned because of the cut. If we add a second rule to P, '$p(X)$:- $X = 2$.', the goal '?- $p(X)$.' will have '$X = 2$' as its only solution.

In order to formulate precisely what the effect of cut on the search tree is, we have to define the notion of *parent* goal. The *parent* goal is the goal that causes the clause containing the cut to be activated. In our example this is $q(X)$. The cut commits the system to all choices made between the time the parent goal was involved and the time the cut was encountered. The remaining alternatives between the parent goal and the cut are discarded.



Exercises 9.7 and 9.8 give some other programs containing cuts.

9.1.7 Negation as Failure

Prolog has a built-in operator ‘not’, which has been defined, using cut, as follows.

not (A) :- A, !, fail.
not (A) :- true.

In order to understand this definition, the reader should know that ‘fail’ and ‘true’ are built-in expressions which always fail or succeed respectively, when they are invoked. From this definition it follows immediately that

(i) the goal ‘not (A)’ fails if the search tree for ‘?- A.’ is finite and has a successful branch, and

(ii) the goal ‘not (A)’ succeeds if the search tree for ‘?- A.’ is finite and has no successful branches.

Note that if the search tree for ‘?- A.’ contains no successful branches and has at least one infinite branch, then the Prolog system cannot answer the question ‘?- not (A).’. In order to see how Prolog handles negation, let us consider the following program P.

```
student(tom).
student(jane).
teacher(mary).
```

It is important to note that neither ‘student(mary)’ nor ‘not student(mary)’ are logical consequences of *P*.

Given this program, the question ‘?- not student(mary).’ is answered by the Prolog system as follows:

$$\begin{tikzcd}
\text{?- not student(mary).} \arrow[r] \arrow[d, dotted, dash] & \text{?- student(mary).} \arrow[d, dash] \\
\text{yes} & \text{fail} \\
 & \text{no} \arrow[ul]
\end{tikzcd}$$

The Prolog system uses what is called the *Negation as Failure (NF)* rule: if the search tree for *A*, given a certain program, is finite and has no successful branches, then conclude ‘not *A*’.

The Negation as Failure rule is *non-monotonic*, i.e., adding new facts and/or rules to the given program may eliminate some former conclusions. For instance, if we add the fact ‘student(mary)’ to the program *P* above, the conclusion ‘not student(mary)’ can no longer be drawn. More information may lead to different (and other) conclusions.

Program *P* above is equivalent to the following one:

student(X) :- X = tom.
student(X) :- X = jane.
teacher(X) :- X = mary.

In most cases what the programmer has in mind is not the program *P* itself, but what is called the *completion* of *P*:

student(X) iff X = tom or X = jane.
teacher(X) iff X = mary.

The completion of *P* is obtained by replacing the if’s in program *P* by iff’s. And although ‘not student(mary)’ is not a logical consequence of *P*, it is a logical consequence of the completion of *P*. Both the Negation as Failure rule and the process of completion capture the idea that information not given by the program is taken to be false.

Exercises 9.9 and 9.10 make clear that for programs which contain negation, the use of cut may affect the soundness of the system.

9.1.8 Applications: Deductive Databases and Artificial Intelligence

In Example 9.1 we have given a very simple application of logic programming to databases. In this example a database is given containing facts or data concerning who is a parent of whom. This database has been extended with rules stating under what conditions the grandparent relation applies. We have seen that one can add other rules such as rules for the predecessor relation, the mother relation, etc. (see also Exercise 9.1). These rules enable the user to derive conclusions from the database which are not explicitly present in the database (as static facts), but which can be logically deduced from the facts in the database by means of application of the rules. For this reason one speaks of *deductive databases*. A logic program can be viewed as a (deductive) database, consisting of facts and rules. *Relational databases* correspond to logic programs consisting only of facts.

Prolog contains a number of facilities for updating databases. For instance, the goal 'assert(C)' will always succeed and will result in adding the program clause C to the database. The goal 'asserta(C)' adds C at the beginning of the database and the goal 'assertz(C)' adds C at the end of the database. The goal 'retract(C)' deletes a program clause that matches C.

Example 9.8. The following non-trivial example of a deductive database is from Bratko [7], Section 4.1. The database or logic program contains facts of the following form:

```
family(
        person(tom, fox, date(7, may, 1950), works(bbc, 15200)),
        person(ann, fox, date(9, may, 1951), unemployed),
        [person(pat, fox, date(5, may, 1973), unemployed),
         person(jim, fox, date(5, may, 1973), unemployed)]
                                                            ).
```

These atomic formulas are built from a ternary predicate symbol 'family', a 4-ary function symbol 'person', a ternary function symbol 'date', a binary function symbol 'works' and a number of individual constants. The overall structure of these facts is: family (Father, Mother, List_of_Children). Now, given a database of the type above, the question 'give name and surname of all married woman who have at least two children' can be formulated in Prolog as follows:

```
?- family(_, person(Name, Surname, _, _), [_, _| _]).
```

In order to understand this formulation the reader should know that '_' is a so-called *anonymous variable*, i.e., a variable whose value is not given when Prolog answers the question. Among the answers to this question would be:

Name = ann,
Surname = fox.

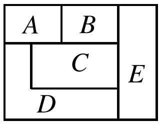
In Exercise 9.11 the reader is invited to add a number of rules to the database such that many other questions can be asked in a straightforward manner.

Since any logic programming system is equipped with a reasoning mechanism, one might say that any such system is able to simulate reasoning and hence disposes of *Artificial Intelligence (AI)*. This makes logic programming a very appropriate tool for solving many problems, which are generally considered to belong to the field of Artificial Intelligence. Many puzzles can be solved by appropriate logic programs. A nice example is *cryptarithmetic puzzles*, such as

$$\begin{array}{r} S\ E\ N\ D \\ \underline{M\ O\ R\ E} \; + \\ M\ O\ N\ E\ Y \end{array}$$

where the problem is to assign decimal digits to the letters of the alphabet such that the above sum is correct. Bratko's book [7] contains in Section 7.1 a Prolog program for solving cryptarithmetic puzzles.

Example 9.9. We give a simple logic program for colouring a given map, such that the colour in each region is different from the colours in all its adjacent regions.



color(X) :- X = red.
color(X) :- X = blue.
color(X) :- X = green.
color(X) :- X = black.
next(X,Y) :- color(X), color(Y), not ($X = Y$).
colormap([A,B,C,D,E]) :- next(A,B), next(A,C), next(A,D), next(B,C),
next(B,E), next(C,D), next(C,E), next(D,E).

Given this program, the appropriate question to ask is

?- colormap(Z).

Example 9.10. Another example of the use of logic programming in the domain of Artificial Intelligence is for *parsing* sentences. The following program is for parsing sentences in a very simple and small fragment of English.

np([john]).	'john' is a noun phrase	tv([loves]).	'loves' is a transitive verb
np([mary]).		tv([hates]).	
np([bill]).		det([a]).	'a' is a determiner.
cn([dog]).	'dog' is a common noun	det([the]).	
cn([woman]).		vp([walks]).	'walks' is a verb phrase.
cn([man]).		vp([talks]).	

np(L) :- conc($L1$, $L2$, L), det($L1$), cn($L2$).
vp(L) :- conc($L1$, $L2$, L), tv($L1$), np($L2$).
s(L) :- conc($L1$, $L2$, L), np($L1$), vp($L2$).
conc([ ], L, L).
conc([$X \mid L1$], $L2$, [$X \mid L3$]) :- conc($L1$, $L2$, $L3$).

In this program 'conc($L1$, $L2$, L)' should be read as 'L is the concatenation of $L1$ and $L2$', and 's(L)' should be read as 'L is a sentence'. Given this program, questions one might ask are:

?- s([john, hates, the, dog]).	?- s([john, hates, the, walks]).
yes	no

The question '?- s(S).' will generate all syntactically correct sentences in the given fragment of English.

When a logic program behaves like an expert in some specific domain such as medical diagnosis or system break-down diagnosis, the logic program is called an *expert system* or a *knowledge-based-system*. By ‘behaving like an expert’ we mean that 1) the logic program contains some expertise information concerning a specific domain, 2) that the program must be able to ask certain questions to the user and 3) that the program must be able to indicate in a user friendly manner how it has derived the answer(s) to a given question. Logic programs which satisfy these conditions become rather complex. Relatively simple examples of such expert systems can be found, among others, in Bratko [7], Chapter 14.

9.1.9 Pitfalls

There are at least four pitfalls the logic programmer should be aware of.

1. We have already mentioned that most actual logic programming systems use matching instead of unification for reasons of efficiency. However, as indicated in Subsection 9.1.4 on Matching versus Unification, it may happen as a consequence that some goal is answered affirmatively, while the goal does not logically follow from the given program. In other words, the lack of the occur check destroys the soundness of the system.

2. The occurrence of a cut in a *definite* program does not affect the soundness of the system, although it may affect the completeness of the system by pruning successful branches. However, Exercise 9.10 makes clear that the use of cut in a *normal* program may even destroy the soundness of the system.

3. Consider the following program P (from Lloyd [16], Section 10).

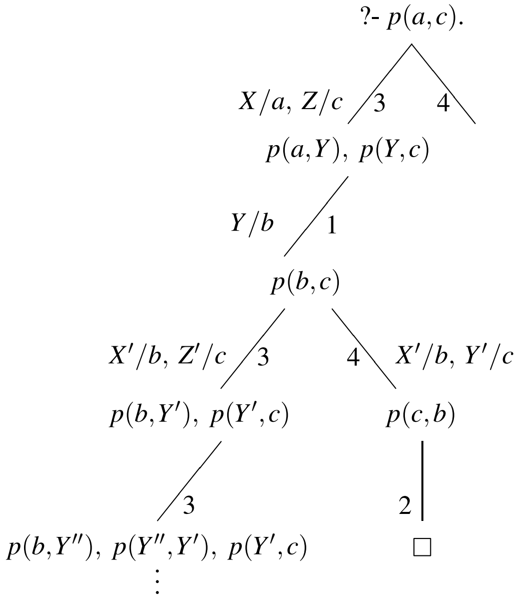
```
p(a, b).                     (1)
p(c, b).                     (2)
p(X,Z) :- p(X,Y), p(Y,Z).    (3)
p(X,Y) :- p(Y,X).            (4)
```

Now it is easy to see that p(a, c) is a logical consequence of P. From (2) and (4) it follows that p(b, c) (5). And from (1), (5) and (3) it follows that p(a, c).

However, given this program, the question ‘?- p(a, c).’ will not be answered by any of the existing Prolog systems. In order to see why, let us consider the search tree for this question.

Any logic programming system that uses a depth-first search, combined with a fixed order for trying clauses given by their ordering in the program, will never find the success branch, because the left-most branch in the search tree is infinite. We have seen that all the clauses (1), (2), (3) and (4) were used in concluding p(a, c) from P. However, in the left-most branch of the search tree for ‘?- p(a, c).’, clause (4) will never be applied. Interchanging clauses (3) and (4) in the program P would result in a left-most branch in which clause (3) is never applied, while all the clauses in P are necessary to deduce p(a, c).

The solution to this problem would be a logic programming system with a breadth-first search rule. However, it is unlikely that such a system can be implemented efficiently.



4. Many logic programming systems do not satisfy the *safeness condition*: negative literals are only allowed to be selected if they do not contain any variables. The safeness condition can be implemented by delaying the treatment of negative subgoals until any variable in the subgoal has been substituted by a term not containing variables.

Violation of the safeness condition affects the soundness of the system. Consider, for instance, the following program P:

bachelor(X) :- not married(X), man(X). (1)
man(bob). (2)
married(alice). (3)

What the programmer actually has in mind is not P itself, but the *completion* of P, consisting of the following formulas:

bachelor(X) $\rightleftarrows$ not married(X), man(X).
man(X) $\rightleftarrows$ X = bob.
married(X) $\rightleftarrows$ X = alice.

From the completion of P it logically follows that for some X, bachelor(X), namely X = bob. A logic programming system that delays the treatment of a negative subgoal, until all variables in the subgoal have been replaced by terms not containing variables, will answer the question '?- bachelor(X).' with 'X = bob'.

$$\begin{tikzcd}
\text{?- bachelor}(X). \arrow[d, dash, "(1)"] & \\
\text{not married}(X),\ \mathit{man}(X) \arrow[d, dash, "X/\text{bob}"', "(2)"] & \\
\text{not married(bob)} \arrow[r] \arrow[d, dash] & \text{?- married(bob).} \arrow[d, dash] \\
\square \ \text{success} \ X = \text{bob} & \text{failure} \arrow[l]
\end{tikzcd}$$

The goal printed in italics is the goal selected by a system satisfying the safeness condition.

However, a logic programming system that does not satisfy the safeness condition will answer the question '?- bachelor(X).' with 'no'.

$$\begin{tikzcd}
\text{?- bachelor}(X). \arrow[d, dash, "(1)"] & \\
\text{not married}(X),\ \text{man}(X) \arrow[r] \arrow[d, dotted, dash] & \text{?- married}(X) \arrow[d, dash, "X/\text{alice}"', "(3)"] \\
\text{failure} & \square \ \text{success} \arrow[l]
\end{tikzcd}$$

Exercise 9.1. Extend the program concerning the parent relation in Example 9.1 with rules which define the offspring, the father, the mother, the sister and the brother relation. It will be necessary to introduce unary predicate symbols 'male' and 'female' and to add some facts about the sex of the persons whose names occur in the program. Given the extended program, construct the search trees for the following questions.

?- mother(tom, liz). ?- mother(X, bob).
?- sister(ann, pat). ?- father(bob, Y).

Exercise 9.2. Let the predecessor relation be added to the program in Example 9.1 (concerning the parent relation) in the following ways.

a) pred2(X,Z) :- parent(X,Y), pred2(Y,Z).
pred2(X,Z) :- parent(X,Z).

b) pred3(X,Z) :- parent(X,Z).
pred3(X,Z) :- pred3(X,Y), parent(Y,Z).

Construct the search trees for the following questions: ?- pred2(tom, pat). ?- pred3(tom, pat). and ?- pred3(liz, jim). Conclude that from a procedural point of view, pred2 and pred3 do not describe the predecessor relation in an adequate way. (The examples are from Bratko [7], Section 2.6.2.)

Exercise 9.3. Determine whether the following pairs can be matched or unified:
a) p($f(X),Z$) and p(Y,c); b) X and $f(X)$ and c) p($f(X),c$) and p($Y,f(Z)$).

Exercise 9.4. Give a recursive definition of the concatenation relation, reading 'conc($L1, L2, L$)' as 'L is the concatenation of the lists $L1$ and $L2$'.

Exercise 9.5. Give a recursive definition of the deletion relation, with 'del($X, L, L1$)' read as '$L1$ results from the list L by deleting one occurrence of X'.

Exercise 9.6. Give a recursive definition for establishing the length of a list, reading 'length(L,N)' as 'N is the number of elements in the list L'.

Exercise 9.7. (Bratko [7]) Let P be the following program:

```
p(1).
p(2) :- ! .
p(3).
```

Construct the search trees for the following goals: a) ?- p(X). b) ?- p(X), p(Y). c) ?- p(X), !, p(Y).

Exercise 9.8. Let P be the following program:

$$\begin{array}{l} p(X, 0) \text{ :- } X < 1. \\ p(X, 1) \text{ :- } X >= 1,\ X < 2. \\ p(X, 2) \text{ :- } X >= 2. \end{array}$$

Using cuts, change P into a program which is declaratively equivalent, but procedurally more efficient.

Exercise 9.9. (Lloyd [16]) Consider the following program P for the subset relation, representing sets by lists, where p(X,Y) expresses that X is not a subset of Y.

$$\begin{array}{ll} \text{subset}(X,Y) \text{ :- not } p(X,Y). & (1) \\ p(X,Y) \text{ :- member}(Z,X), \text{ not member}(Z,Y). & (2) \\ \text{member}(X, [X \mid L]). & (3) \\ \text{member}(X, [Y \mid L]) \text{ :- member}(X,L). & (4) \end{array}$$

Make clear how the Prolog system answers the question: ?- subset([1, 2], [1, 2, 3]).

Exercise 9.10. If we replace clause (3) in the program of Exercise 9.9 by the clause

$$\text{member}(X, [X \mid L]) \text{ :- } ! . \qquad (3')$$

then the membership program will generate just one solution and not all possible solutions. Verify that if we do so, the question '?- subset([1, 2, 3], [1]).' will be answered affirmatively, while 'not subset([1, 2, 3], [1])' logically follows from P. So, the use of cut in combination with negation may affect the soundness of the system!

Exercise 9.11. Extend the database in Example 9.8 with appropriate rules, such that the following questions can be formulated in Prolog in an adequate way. (Confer Bratko [7], Section 4.1.)
1. Give the names and surnames of all people in the database.
2. Give all children born in 1973.
3. Give the names and surnames of all employed wives.
4. Give the names and surnames of all unemployed people born before 1960.
5. Give all people born before 1960 whose salary is more than 10000.
6. Give the surnames of all families with at least two children.
7. Give the surnames of all families without children.

Exercise 9.12. Instead of $f(t1, t2)$, Prolog also allows the *infix notation* $t1\ f\ t2$. For that purpose it is necessary to define f as an operator with a given precedence. The *precedence* of an arbitrary term is then defined as follows.
1. The precedence of individual variables and individual constants is 0.
2. The precedence of $f(t_1,\ldots,t_n)$ is the precedence of f.
3. The precedence of (t), t a term, is 0.
In order to ensure that $a+b*c$ is interpreted as $a+(b*c)$ and not as $(a+b)*c$, the operators $+$ and $*$ may be defined as follows.

$$\text{op}(500, yfx,\ +). \qquad (1)$$
$$\text{op}(400, yfx,\ *). \qquad (2)$$

In (1) the operator $+$ is defined as an infix operator (i.e., occurring between its arguments) with precedence 500. 'y' represents an argument whose precedence must be lower than or equal to that of the operator, and 'x' represents an argument whose precedence must be strictly lower than that of the operator.
1. Check that under the definitions (1) and (2) $a+b*c$ is understood as $a+(b*c)$ and not as $(a+b)*c$.
2. Defining '$-$' by 'op(500, yfx, $-$).', check that $a-b-c$ is read as $(a-b)-c$ and not as $a-(b-c)$.
3. Defining 'has' by 'op(600, xfx, has).', check that instead of 'has(peter, information).' the programmer can write 'peter has information.'.

9.2 Relational Databases and SQL

Abstract In this section we shall concentrate on the conceptual schema, i.e., the description of a database on a logical level. Only the relational model of databases will be discussed, because this model is most interesting from a logical and set-theoretical point of view. The description of the logical structure of relational databases in set theoretic terms shows that a Query Language such as SQL is a very natural one. Tuple-, table- and database- *constraints* are discussed. The notion of *key* is introduced and we also discuss the *Boyce-Codd Normal Form*, the *projection* of a table and the (natural) *join* of two tables. The material presented in this section is based on F. Remmen's book *Databases* (in Dutch) and on de Brock [8].

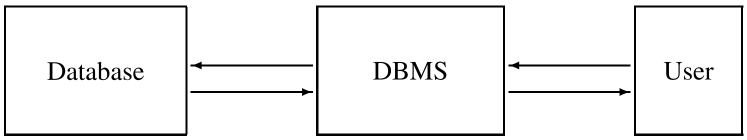
By a database we mean a class of permanent data, which is available to all users of an information system. These data relate to the objects which are relevant to the information system and to the attributes which are relevant to these objects. For instance, the permanent data of a hospital organisation include, among other things, the name, address and residence of each patient.

These permanent data should be available to all users of an information system. This availability for many users has important consequences as different groups of users will be interested in the data in different manners. For instance, an administrator in a hospital organisation will be in need of financial data about persons and rates, while a specialist needs to have at his disposal all medical data of persons and of all treatments to be applied.

Which objects with what properties are relevant to the information system can only be determined by the users. The design and implementation of a database will be a compromise between the different and partly clashing desires of the different users. It is the task of the Data Base Administrator (DBA) to bring about such a compromise.

In current database terminology the difference between species and individual is usually indicated by the difference between *type* and *occurrence*. So one can speak of the (object) type patient, and of the (object) occurrence of a patient in a hospital-organisation. In this example we have one type (species) with – in general – many occurrences (individuals).

Each user communicates with the database via the Data Base Management System (DBMS). In fact, a DBMS can be considered as a special expansion of the operating system.



The users of an information-system are interested in information about individual objects and information about an individual object can only be provided in the form of values of one or more attributes. In general, many possible values will be available for each attribute. For instance, the attribute 'pnr' (short for 'patientnumber') of the object 'patient' may have a value between 1 and 100000, and the attribute 'pnm' (patient-name) may have a value consisting of a combination of at most 25 characters. In general, we demand that the values of an attribute form a set.

The set of attributes of an object together with the sets of values belonging to them is called the *object-characterisation* of that object. In the following examples it is made clear how we shall render an object-characterisation.

obchar patient =

attrib	pnr	: $\{1,\ldots,100000\}$,	patientnummer
	pnm	: chs25 ,	name
	padr	: chs20 ,	address
	pres	: chs20 ,	residence
	db	: $\{18800101,\ldots,19991231\}$,	date of birth
	sex	: $\{m, f\}$,	sex

endobchar

By chs25 (character string 25) we mean the set of all strings of at least one and at most 25 signs (letters, figures). In the following example – an object-characterisation of the object 'admission' (into a hospital) – we use the abbreviation 'dat' for the set $\{19500101,\ldots, 19991231\}$, i.e., the set of all natural numbers between 19500101 and 19991231.

obchar admission =

attrib	pnr	: $\{1,\ldots,100000\}$,	
	pnm	: chs25 ,	
	padr	: chs20 ,	
	pres	: chs20 ,	
	indat	: dat ,	date of admission
	outdat	: dat ,	date of discharge
	reas	: chs25 ,	reason of admission
	snr	: $\{1,\ldots,100000\}$,	number of specialist
	snm	: chs25 ,	name of specialist
	rnr	: $\{1,\ldots,1000\}$,	number of nursing-room
	wnr	: $\{1,\ldots,15\}$,	number of ward

endobchar

Lastly, we give as an example an object-characterisation of the object 'specialist'.

obchar specialist =

attrib	snr	: $\{1,\ldots,100000\}$,	registration-number
	snm	: chs25 ,	name
	sadr	: chs20 ,	address
	sres	: chs20 ,	residence
	wnr	: $\{1,\ldots,15\}$,	number of ward
	nbd	: $\{1,\ldots,100\}$,	number of beds

endobchar

Definition 9.11 (Object-characterization; Tuple). Let O be an object with attributes $A_1,\ldots,A_m$ and let $W_1,\ldots,W_m$ be the sets of values belonging to $A_1,\ldots,A_m$ respectively. Then we define

$$F_O := \{(A_1,W_1),\ldots,(A_m,W_m)\},$$

and call it the *object-characterisation* of O.

Next we define $\pi(F_O) :=$

$$\{t \mid t = \{(A_1,w_1),\ldots,(A_m,w_m)\} \text{ for some } w_1 \in W_1,\ldots,w_m \in W_m\}.$$

The elements of $\pi(F_O)$ are called *tuples* for O. A tuple for O is a function (and hence a relation) with domain $\{A_1,\ldots,A_m\}$. If t is a tuple for O and $(A_i,w_i) \in t$, then we write $t(A_i)$ for w_i.

So a *tuple t* for O is a set $\{(A_1,w_1),\ldots,(A_m,w_m)\}$ with $w_1 \in W_1, \ldots, w_m \in W_m$. We write $w_1 = t(A_1),\ldots,w_m = t(A_m)$. Each tuple for O represents one object-occurrence. By mentioning the attributes at the head of columns, we can list the tuples for a given object O in a table, each row in the table corresponding to a tuple for O. For instance, below we give a partial table for the object 'patient'.

pnr	pnm	padr	pres	db	sex
537	Blunt	36 Evans Drive	Cranbury	19080527	m
498	Kiviat	67 Main Street	Newark	19090730	f

In general it holds that not all possible combinations of values of the attributes $A_1, \ldots, A_m$ will be allowed. In the literature on databases these restrictions are called *constraints*. We distinguish constraints on tuples, constraints on tables, and constraints on databases.

Below we give some examples of tuple-constraints, i.e., constraints on tuples.
$C_1(t)$: if t(pnr) < 200, then t(db) < 19000101; t being a tuple for object 'patient'.
$C_2(t)$: t(indat) $< t$(outdat); t being a tuple for object 'admission' (into hospital).
$C_3(t)$: if t(wnr) $= 9$, then t(sres) = Princeton; and if t(wnr) $= 7$, then t(nbd) ≤ 2; t being a tuple for object 'specialist'.

So a *tuple-constraint* is a condition on tuples for a given object O, such that it can be determined whether the condition holds for a given tuple t or not, completely independent of the other tuples.

Definition 9.12 (Tuple-type). Given an object O and a tuple-constraint C

$$T\text{-}O := \{t \in \pi(F_O) \mid C(t)\}.$$

T-O is called the *tuple-type* for O (determined by the constraint C) and is the set of all tuples t for O satisfying the condition C.

If the number of tuples t for O satisfying a condition C is finite (and in practice not too large), then the tuple-type T-O for O can be rendered by an exhaustive list of all object-occurrences satisfying condition C.

A *table for O* is by definition a subset of T-O. There may also be constraints on such tables. We give some examples below.
$TC_1(D)$: $\forall t_1, t_2 \in D$ [t_1(pnr) = t_2(pnr) $\rightarrow t_1 = t_2$]; D being a table for the object 'patient'. $\forall t_1, t_2 \in D$ stands for 'for all t_1 and t_2 in D'. This table-constraint TC_1 is also formulated as follows: {pnr} is *uniquely identifying*, or: {pnr} uni, for short.
$TC_2(D)$: $\forall t_1, t_2 \in D$ [t_1(pnr) $= t_2$(pnr) $\wedge$ t_1(indat) $= t_2$(indat) $\rightarrow t_1 = t_2$]; D being a table for the object 'admission'. This table-constraint TC_2 is also formulated as follows: {pnr, indat} is *uniquely identifying*, or: {pnr, indat} uni, for short.
$TC_3(D)$: {snr} uni, and {snm, sadr, sres} uni, and the number of specialists at ward 9 is at least 2; D being a table for the object 'specialist'.

So, a *table-constraint* indicates which subsets of a tuple-type are allowed. The set of all tables allowed, given an object O, is called a table-type for O.

Definition 9.13 (Table-type). Let O be an object, T-O a tuple-type for O and TC a table-constraint for O. Then

$$TT\text{-}O := \{D \in P(T\text{-}O) \mid TC(D)\}$$

is called a *table-type* for O. If $D \in TT$-O, we say D is a *table of type O*.

Definition 9.14 (Functional Dependence; Uniquely Identifying; Key).
Let O be an object with attributes $A_1, \ldots, A_m$, and let D be a table of type O. Let $V, W \subseteq \{A_1, \ldots, A_m\}$.

1. $V \to W$ in $D := \forall t_1, t_2 \in D\ [\ t_1 \lceil V = t_2 \lceil V \to t_1 \lceil W = t_2 \lceil W]$, where $t \lceil V$ is the restriction of t to V. In words: *V functionally determines W in D*, or W is *functionally dependent on V in D*.
2. V is *uniquely identifying within D* $:= V \to \{A_1, \ldots, A_m\}$ in D, i.e.,
$\forall t_1, t_2 \in D\ [\ t_1 \lceil V = t_2 \lceil V \to t_1 = t_2]$.
3. $V \to W$ for O := for every table D of type O, $V \to W$ in D.
V is a *key for O* $:= V$ is uniquely identifying within every table of type O.

Example 9.11. {pnr} is a key for 'patient'; {pnr, indat} is a key for 'admission'; {pnm, padr, pres} is a key for 'patient'; {snr} is a key for 'specialist'.

Within the framework of an information-system one usually will be interested in more than only one table-type. For instance, in a hospital-organisation one may be interested in patients, admissions (into the hospital) and specialists and hence also in the table-types TT-patient, TT-admission and TT-specialist belonging to them. At a certain moment the situation of the hospital, at least with respect to patients, admissions and specialists, can be summed up by three tables, one of type TT-patient, one of type TT-admission and one of type TT-specialist. Such a triple of tables is called a *relational database*.

Definition 9.15 (Database-characterisation). A set of objects together with table-types belonging to them is called a *database-characterisation*. More precisely, let $O_1, \ldots, O_n$ be objects, together with table-types $TT\text{-}O_1, \ldots, TT\text{-}O_n$ belonging to them. Then

$$F_{DB} := \{(O_1, TT\text{-}O_1), \ldots, (O_n, TT\text{-}O_n)\}$$

is a database-characterisation. In the following example it is made clear how we shall render a database-characterisation.

Example 9.12. The database-characterisation for the combination of the objects patient, admission and specialist looks as follows:

dbchar hospital =
    obj pat : TT-patient ,
        adm : TT-admission,
        spec : TT-specialist
enddbchar

Note the analogy between an object-characterisation and a database-characterisation: the attributes are replaced by objects and the sets of values by table-types.

Definition 9.16 (Relational Databases).
$\pi(F_{DB}) := \{\{(O_1, D_1), \ldots, (O_n, D_n)\} \mid D_1 \in TT\text{-}O_1, \ldots, D_n \in TT\text{-}O_n\}$. The elements of $\pi(F_{DB})$ are called *relational databases*.

Given a database-characterisation with objects $O_1, O_2, \ldots, O_n$ and table-types $TT\text{-}O_1$, $TT\text{-}O_2, \ldots, TT\text{-}O_n$ belonging to them, in general not all databases in $\pi(F_{DB})$ will be allowed. This brings us to the last class of constraints, the so-called *database-constraints*. The set of all databases satisfying a certain database-constraint is called a *database-type*.

An important subclass of database-constraints is formed by the so-called *subset-requirements*. We make this notion clear by means of the following two database-constraints.

$DC_1(D_1,D_2,D_3) := \forall t_2 \in D_2\ \exists t_1 \in D_1\ [\ t_2(\text{pnr}) = t_1(\text{pnr})\ ]$, in words: for each admission-tuple t_2 there is a patient-tuple t_1 such that the value of pnr in t_2 is equal to the value of pnr in t_1.

$DC_2(D_1,D_2,D_3) := \forall t_2 \in D_2\ \exists t_3 \in D_3\ [\ t_2(\text{snr}) = t_3(\text{snr})\ ]$, in words: for each admission-tuple t_2 there is a specialist-tuple t_3 such that the value of snr in t_2 is equal to the value of snr in t_3.

The constraint DC_1 means that for any database allowed the set of pnr-values in the admission-table is a subset of the set of pnr-values in the patient-table. A similar remark is to be made for DC_2. For these subset-requirements the following notation is used: for DC_1, ssr(adm.pnr, pat.pnr); for DC_2, ssr(adm.snr, spec.snr).

Below we give an example of a database-type. The symbols *tatp, tutp, obchar, attrib* stand for table-type, tuple-type, object-characterisation and attribute respectively. The symbols *tuc, tac, dbc* stand for tuple-constraint, table-constraint and database-constraint respectively.

```
Type nr   : {1,...,100000}
     hoev : {1,...,100}
     dat  : {19000101,...,19991231}
     tatp TT-patient =
               tutp T-patient =
                      obchar patient =
                             attrib pnr  : nr                      ,
                                    pnm  : chs25                   ,
                                    padr : chs20                   ,
                                    pres : chs20                   ,
                                    db   : {18800101, ...,19991231} ,
                                    sex  : {m,f}
                      endobchar;
                      tuc   pnr < 200 → db < 19000101
               endtutp;
               tac       {pnr} uni,
                         {pnm, padr, pres} uni
     endtatp,

     tatp TT-admission =
               tutp T-admission =
                      obchar admission =
                             attrib pnr    : nr    ,
                                    pnm    : chs25 ,
                                    padr   : chs20 ,
                                    pres   : chs20 ,
                                    indat  : dat   ,
                                    outdat : dat   ,
                    attrib reas : chs25          ,
                           snr  : nr             ,
                           snm  : chs25          ,
                           rnr  : {1,...,1000} ,
                           wnr  : {1,...,15}
                endobchar;
                tuc  indat < outdat,
                     reas = 'informaritis' → rnr = 5
           endtutp ;
           tac      {pnr, indat} key
      endtatp,
      tatp TT-specialist =
           tutp T-specialist =
                obchar specialist =
                    attrib snr  : nr          ,
                           snm  : chs25       ,
                           sadr : chs20       ,
                           sres : chs20       ,
                           wnr  : {1,...,15} ,
                           nbd  : hoev
                endobchar;
                tuc   wnr = 9 → sres = 'Princeton',
                      wnr = 7 → nbd ≤ 2
           endtutp;
           tac     keys {{snr}, {snm, sadr, sres}},
                   'at least two specialists at ward 9'
      endtatp,
       dbtype DT-hospital =
           dbchar hospital =
                obj pat  : TT-patient    ,
                    adm  : TT-admission ,
                    spec : TT-specialist
           enddbchar;
           dbc    ssr (adm.pnr, pat.pnr),
                  ssr (adm.snr, spec.snr)
       enddbtype,
  endtype
```

Looking at the database-type given above, we see that some attributes are *redundant*: pnm, padr and pres in the table for 'admission' are uniquely determined by pnr and already occur in the table for 'patient'; snm in the table for 'admission' is uniquely determined by snr and also occurs in the table for 'specialist'; and wnr in the table for 'admission' is uniquely determined by rnr, although a table in which both rnr and wnr already occur is not (yet) available. For these reasons we say that the table-type *TT*-admission given above is not *normal*. In concrete cases this means that in case of a change of address of a patient not only the table for 'patient' has to be updated, but also the table for 'admission'; otherwise, an inconsistent database would result.

Definition 9.17 (Boyce-Codd Normal Form). Let O be an object with attributes $A_1,\ldots,A_m$ and let $TT\text{-}O$ be a table-type for O. $TT\text{-}O$ is in *Boyce-Codd Normal Form* (BCNF) := for all $V \subseteq \{A_1,\ldots,A_m\}$ and for all $A \in \{A_1,\ldots,A_m\}$: if $V \rightarrow \{A\}$ for O and $A \notin V$, then V is a key for O. Informally: $TT\text{-}O$ is in BCNF if every set V of attributes which determines an attribute outside of V is a key for O.

Since {pnr} $\rightarrow$ {pnm} for 'admission', pnm $\notin$ {pnr}, but {pnr} is not a key for 'admission', it follows that TT-admission is not in Boyce-Codd Normal Form. (Remember that {pnr, indat} is a key for 'admission'.)

In the literature one also finds various other normal forms including the first, second, third and fourth normal forms. If $TT\text{-}O$ is in BCNF, then $TT\text{-}O$ is also in 3NF (Third Normal Form).

Definition 9.18 (Normal Database-type). A *normal database-type* is a database-type in which each table-type is normal.

Example 9.13. We can convert the database-type given above into a normal database-type by applying the following two operations to the given database-type:
1. In the table-type TT-admission leave out the attributes pnm, padr, pres, snm and wnr.
2. Add a table-type TT-room as follows:

```
tatp TT-room =
        tutp T-room =
                obchar room =
                        attrib rnr : {1,...,1000},
                               wnr : {1,...,15}
                endobchar;
        endtutp;
        tac                 {rnr} uni
endtatp.
```

The result is a normal database-type, in which redundancies are avoided, while all information has been saved. (However, in practice, redundant *storage* of data may be necessary, for instance, because of the required time of response.)

Of course, a database-type for any actual hospital organisation will be much more complex than the simple example considered here.

Definition 9.19 (Projection). Let O be an object with attributes $A_1,\ldots,A_m$, $V \subseteq \{A_1,\ldots,A_m\}$ and let D be a table of type O. $D \parallel V$, the *projection* of D on V, is by definition $\{\, t\lceil V \; ; \; t \in D\}$.

Example 9.14. For instance, for the following table D_1:

nr	name	sal	sex	dept
8	Johnson	2200	male	1
7	Johnson	3100	female	2
9	Kiviat	2900	male	1

$D_1 \parallel$ {sex, dept} is the table

sex	dept
male	1
female	2

.

Definition 9.20 (Compatible tuples). Let O_1 be an object with attributes $A_1, \ldots, A_m$ and O_2 an object with attributes $B_1, \ldots, B_n$. Let t_1 be a tuple for O_1 and t_2 a tuple for O_2. t_1 and t_2 are *compatible* := $t_1 \lceil \{A_1, \ldots, A_m\} \cap \{B_1, \ldots, B_n\} = t_2 \lceil \{A_1, \ldots, A_m\} \cap \{B_1, \ldots, B_n\}$.

Definition 9.21 (Join). Let D_1 be a table of type O_1 and D_2 a table of type O_2. $D_1 \bowtie D_2 := \{t_1 \cup t_2 \mid t_1 \in D_1$ and $t_2 \in D_2$ and t_1 and t_2 are compatible$\}$.
$D_1 \bowtie D_2$ is called the (natural) *join* of D_1 and D_2.

Example 9.15. For instance, let D_2 be the table:

anr	name	man
2	planning	7
1	production	9

and let D_3 result from table D_1 in Example 9.14 by replacing 'dept' by 'anr'. Then $D_3 \bowtie D_2$ is the table:

nr	name	sal	sex	anr	name	man
8	Johnson	2200	male	1	production	9
7	Johnson	3100	female	2	planning	7
9	Kiviat	2900	male	1	production	9

9.2.1 SQL

The purpose of a *query-language* is to enable the user to make use of the data stored in the database in an user-friendly manner. In order to give a more concrete idea of a query-language, we shall treat some elements of the query-language SQL (Structured Query Language; 1980) on the basis of some examples. Having understood the logical structure of a relational database, query-languages such as SQL become very perspicuous.

The terminology of SQL is familiar to the terminology of set theory, as will become clear from the examples below. In these examples P, ADM, SP and R stand for the set (or table) of all patients, the set of all admissions, the set of all specialists and the set of all rooms, respectively. The examples all refer to the objects described in the *normalized database* given in Example 9.13.

Example 9.16. Describe the set of numbers, names and addresses of all patients who live in Princeton and were born before 1960.
Answer: a) $\{ t \lceil \{\text{pnr, pnm, padr}\} \mid t \in \text{P} \mid t(\text{pres}) = \text{'Princeton'} \wedge t(\text{db}) < 19600101\}$.
b) Now, in SQL the query 'give number, name and address of all patients who live in Princeton and were born before 1960' is formulated as follows:

```
SELECT   t.pnr, t.pnm, t.padr
FROM     P  t
WHERE    t.pres = 'Princeton'
AND      t.db < 19600101
```

Here t.pnr corresponds to t(pnr).

Example 9.17. Describe the set of numbers, names and addresses of all patients who were admitted into hospital in the period between May 26 and July 11, 1981.
Answer: a) $\{t\lceil\{\text{pnr, pnm, padr}\} \mid t \in \text{P} \mid \exists s \in \text{ADM}$
$[s(\text{pnr}) = t(\text{pnr}) \wedge s(\text{indat}) \geq 19810526 \wedge s(\text{indat}) \leq 19810711]\}$
or, equivalently,
$\{t\lceil\{\text{pnr, pnm, padr}\} \mid t \in \text{P} \mid t(\text{pnr}) \in \{s(\text{pnr}) \mid s \in \text{ADM} \mid s(\text{indat}) \geq 19810526 \wedge s(\text{indat}) \leq 19810711\}\}$.
b) Now, in SQL the query 'give number, name and address of all patients who were admitted into hospital in the period between May 26 and July 11, 1981' is formulated as follows:

```
SELECT t.pnr, t.pnm, t.padr
FROM   P t
WHERE  t.pnr IN
              (SELECT s.pnr
               FROM   ADM s
               WHERE  s.indat ≥ 19810526
               AND    s.indat ≤ 19810711)
```

Example 9.18. Describe the set of names, addresses and residences of all specialists who were responsible for an admission in August 1977 of a patient from Princeton for reason 034.
Answer: a) $\{t\lceil\{\text{snm, sadr, sres}\} \mid t \in \text{SP} \mid \exists s \in \text{ADM}\ [\ s(\text{snr}) = t(\text{snr}) \wedge s(\text{reas}) = 034 \wedge 19770801 \leq s(\text{indat}) \leq 19770831 \wedge \exists u \in \text{P}\ [\ u(\text{pnr}) = s(\text{pnr}) \wedge u(\text{pres}) = \text{'Princeton'}]]\}$
or, equivalently,
$\{t\lceil\{\text{snm, sadr, sres}\} \mid t \in \text{SP} \mid t(\text{snr}) \in \{s(\text{snr}) \mid s \in \text{ADM} \mid s(\text{reas}) = 034 \wedge 19770801 \leq s(\text{indat}) \leq 19770831 \wedge s(\text{pnr}) \in \{u(\text{pnr}) \mid u \in \text{P} \mid u(\text{pres}) = \text{'Princeton'}\}\}\}$.
b) Now, in SQL the query 'give name, address and residence of all specialists who were responsible for an admission in August 1977 of a patient from Princeton for reason 034' is formulated as follows.

```
SELECT t.snm, t.sadr, t.sres
FROM   SP t
WHERE  t.snr IN
           (SELECT s.snr
            FROM   ADM s
            WHERE  s.reas = 034
            AND    s.indat ≤ 19770831
            AND    s.indat ≥ 19770801
            AND    s.pnr IN
                     (SELECT u.pnr
                      FROM   P u
                      WHERE  u.pres = 'Princeton'))
```

Example 9.19. Describe the set of numbers and names of all patients, reason of admission and number of nursing-room, who were admitted into hospital between

September 1 and 5, 1977, in ward number 9.
Answer: a) $\{t\lceil\{\text{pnr, pnm, reas, rnr}\} \mid t \in \text{P} \bowtie \text{ADM} \mid 19770901 \leq t(\text{indat}) \leq 19770905 \wedge \exists s \in \text{R}\ [s(\text{rnr}) = t(\text{rnr}) \wedge s(\text{wnr}) = 9]\}$
or, equivalently,
$\{t\lceil\{\text{pnr, pnm, reas, rnr}\} \mid t \in \text{P} \bowtie \text{ADM} \mid 19770901 \leq t(\text{indat}) \leq 19770905 \wedge t(\text{rnr}) \in \{s(\text{rnr}) \mid s \in \text{R} \mid s(\text{wnr}) = 9\}\}$, where $P \bowtie$ ADM is the join of P and ADM.
b) Now, in SQL the query 'give number and name of all patients, reason of admission and number of nursing-room, who were admitted into hospital in the period between September 1 and 5, 1977, in ward number 9' is formulated as follows.

```
SELECT t1.pnr, t1.pnm, t2.reas, t2.rnr
FROM   P t1, ADM t2
WHERE  t1.pnr = t2.pnr
AND    t2.indat ≥ 19770901
AND    t2.indat ≤ 19770905
AND    t2.rnr IN
          (SELECT s.rnr
          FROM    R s
          WHERE   s.wnr = 9)
```

Example 9.20. Describe the set of all room-numbers, in which no patients from Cranbury were hospitalized in the period between August 11 and 17, 1977.
Answer: a) $\{s(\text{rnr}) \mid s \in \text{R} \mid \neg\exists t \in \text{ADM}\ [\ t(\text{rnr}) = s(\text{rnr}) \wedge 19770811 \leq t(\text{indat}) \leq 19770817 \wedge A(t)]\}$ where $A(t) :=$
i) $\exists u \in \text{P}\ [u(\text{pnr}) = t(\text{pnr}) \wedge u(\text{pres}) = \text{'Cranbury'}]$ or, equivalently,
ii) $t(\text{pnr}) \in \{u(\text{pnr}) \mid u \in \text{P} \mid u(\text{pres}) = \text{'Cranbury'}\}$.
Note that $\neg\exists t \in \text{ADM}\ [t(\text{rnr}) = s(\text{rnr}) \wedge \ldots \wedge A(t)]$ is equivalent to

$$s(\text{rnr}) \notin \{t(\text{rnr}) \mid t \in \text{ADM} \mid \ldots \wedge A(t)\}.$$

b) Now, in SQL the query 'give the numbers of all rooms, in which no patients from Cranbury were hospitalized in the period between August 11 and 17, 1977' can be formulated as follows.

```
SELECT s.rnr
FROM   R s
WHERE  s.rnr NOT IN
          (SELECT t.rnr
          FROM    ADM t
          WHERE   t.indat ≤ 19770817
          AND     t.indat ≥ 19770811
          AND     t.pnr IN
                    (SELECT u.pnr
                    FROM    P u
                    WHERE   u.pres = 'Cranbury'))
```

For further reading, the reader is referred to E. O. de Brock [8].

Exercise 9.13. The following queries all refer to the normalized database given in Example 9.13. Formulate these queries into SQL.
a) Give name, address and residence of all specialists from ward number 9, having more than two beds.
b) Give number and name of all specialists who were responsible for admission on March 3, 1980, because of informaritis.
c) Give number of all rooms in which no patients from Princeton were hospitalized in the period between May 9 and 18, 1980.
d) Give number, name, address and residence of all patients who were hospitalized by a specialist of ward number 9.

Exercise 9.14. Let D_1 be the table

nr	name	sal	sex	dept
8	Johnson	2200	male	1
7	Johnson	3100	female	2
9	Kiviat	2900	male	1

and let D_2 be the table

anr	name	man
2	planning	7
1	production	9

Determine $D_1 \bowtie D_2$. Let D_3 result from D_1 by replacing 'dept' by 'anr' and 'name' by 'wnm'. Determine $D_3 \bowtie D_2$.
Let D_4 result from D_2 by replacing 'man' by 'nr' and 'name' by 'anm'. Determine $D_1 \bowtie D_4$ and $D_3 \bowtie D_4$.

Exercise 9.15. Make clear why the following set does not describe the set of all room-numbers in which no patients from Cranbury were hospitalized in the period between August 11 and 17, 1977 (compare Example 9.20).
$\{s(\text{rnr}) \mid s \in \text{R} \mid \exists t \in \text{ADM}\ [\ t(\text{rnr}) = s(\text{rnr}) \wedge 19770811 \leq t(\text{indat}) \leq 19770817 \wedge \neg\exists u \in \text{P}\ [\ u(\text{pnr}) = t(\text{pnr}) \wedge u(\text{pres}) = \text{'Cranbury'}]]\}$.
Hint: Consider the following tables.

R	rnr	wnr
s1	11	5
s2	12	5
s3	13	6

ADM	pnr	indat	rnr
t1	400	19770812	11
t2	500	19770813	11
t3	600	19770814	12

P	pnr	pres
u1	400	Princeton
u2	500	Cranbury
u3	600	Cranbury

9.3 Social Choice Theory; Majority Judgment

Abstract We show that most well-known and most frequently used voting rules have a number of unacceptable properties. The hope for a voting rule with only nice properties seemed to be vanished when Kenneth Arrow [1] proved his impossibility theorem in 1951. However, in 2010 Michel Balinski and Rida Laraki made clear that – by asking voters for their evaluations of the candidates instead of their preferences over the candidates – a nice voting rule does exist: Majority Judgment (MJ). They show how poorly the existing voting rules perform in the French and American

presidential elections and how Majority Judgment would lead to other and more plausible results.

9.3.1 Introduction

When choosing a mayor, president, chairman, etc., usually the first thought is: most votes count. Many people think that democracy is more or less identical to application of ‘most votes count’, in other words, the Plurality Rule (PR). However, this procedure to choose a winner or a common (or social) preference over the candidates or alternatives has many defects. This rule takes only the top preference of the voters into account, ignores the second, third, etc. preferences of the voters and hence causes serious *loss of information*. In technical terms, this procedure is not Independent of Irrelevant Alternatives (not IIA), as we shall see in Section 9.3.2.

Is then pairwise comparison, in other words Majority Rule (MR), a good alternative? This procedure does take the individual preference orderings of the voters over the alternatives into account and is Independent of Irrelevant Alternatives. However, it is not transitive and hence does not in all cases yield a feasable outcome, as we shall see in Section 9.3.3. By the way, ‘most votes count’ and ‘pairwise comparison’ coincide in the case of only two alternatives, i.e., with only two candidates Plurality Rule and Majority Rule give the same outcome.

In 1951 K. Arrow [1] proved that any voting rule which takes as input the individual preference orderings (over the candidates or alternatives) of the voters and which is transitive and Independent of Irrelevant Alternatives (together with some other natural properties like anonymity and neutrality) is dictatorial, i.e., there will be a voter whose preference is always the outcome of the voting rule, no matter what the preferences of the other voters are. In Section 9.3.6 we shall give a simple proof of (a version of) Arrow’s theorem, due to Balinski and Laraki [3].

Recently, Balinski and Laraki [3] showed that even with only two candidates ‘most votes count’ in many cases may give an unnatural or counterintuitive outcome, i.e., it may select a candidate as winner who in fact has lower evaluations than his competitor. In their words: Majority Rule does not respect domination. Consequently, Majority Rule and Plurality Rule are disqualified as good voting rules for determining a winner or a common preference ordering over the alternatives. We shall elaborate this in Section 9.3.7.

Considering all this, the conclusion seems to be inevitable: there is no ‘good’ voting rule to determine a winner (or a common preference over the candidates) in an election, where we mean by ‘good’ that the voting rule is transitive, Independent of Irrelevant Alternatives and in addition respects domination.

However, already in 2010 Balinski and Laraki [2] presented their Majority Judgment (MJ). This voting rule takes as input not the individual preference orderings (over the alternatives) of the voters, but the evaluations by the voters of the different candidates in sufficiently varied terms, like for instance: excellent (*ex*), very good (*vg*), good (*go*), acceptable (*ac*), poor (*po*) and reject (*re*). It turns out that this voting rule, Majority Judgment, is IIA, transitive and does respect domination and nevertheless is not dictatorial. In addition, this Majority Judgment contains certain safeguards to prevent successful manipulation by the voters. We describe this voting rule in Section 9.3.8.

How is it possible that Majority Judgment escapes the curse of Arrow's theorem? Because MJ takes the evaluations of the candidates by the voters as input and not the individual preferences over the candidates. Here it is important to notice that from the evaluations of the candidates by a voter one may deduce the individual preference ordering of this voter, but that conversely, from the individual preference ordering over the candidates one cannot deduce the evaluations of the candidates by the voter in question. So, an evaluation of all candidates by a voter is much *more informative* than his preference ordering over the candidates. In addition, if two voters say that they prefer candidate A to candidate B, they may mean quite different things: one that he judges A as excellent and B as acceptable, the other that he judges A as poor and B as even more poor. In other words, individual preference orderings over the candidates lead to a babylonian confusion of tongues and one should not be surprised that this yields problems, as becomes evident from Arrow's theorem.

Balinski and Laraki show on the basis of the presidential elections in the USA [4] and in France [5] how poorly our familiar ways of choosing a president may work out and illustrate with these examples from real life how their Majority Judgment would lead to other and more plausible outcomes. We discuss this in Section 9.3.12 (USA) and 9.3.13 (France). In Section 9.3.14 we pay attention to the situation in the Netherlands.

9.3.2 Plurality Rule (PR): most votes count

In the year 2000 there were presidential elections in the USA with Bush, Gore and Nader as the most important candidates. In Florida the result of the ballot was approximately as follows:

41%	Bush
39%	Gore
20%	Nader

Because 'most votes count' or Plurality Rule (PR) is applied, Bush was the winner in Florida (with in fact only a few hundred votes more than Gore). But most votes count? Or rather not? The individual preferences of the voters were approximately as given in the following *profile p*.

41%	Bush	Gore	Nader
39%	Gore	Nader	Bush
20%	Nader	Gore	Bush

Notice that (39 + 20) = 59% of the voters, hence a majority, has Bush as last preference. But Plurality Rule chooses Bush as the winner. How can this be? Because the Plurality Rule causes *loss of information*: only the first preferences of the voters are taken into account, the second, third, etc. preferences of the voters are left out of consideration.

Taking this extra information into account, pairwise comparison, in other words Majority Rule (MR), yields the following result: both Gore and Nader beat Bush with 39 + 20 = 59% against 41. And Gore beats Nader with 41 + 39 = 80% against 20. So, the outcome under pairwise comparison (MR) would be: *Gore Nader Bush*, in this order, while the outcome under Pluraiity Rule was: *Bush Gore Nader*.

PR	Bush	Gore	Nader
MR	Gore	Nader	Bush

In a pairwise comparison Bush loses of every other candidate, and is therefore called a *Condorcet loser*, but he becomes the winner under 'most votes count'. Gore beats every other candidate in a pairwise comparison and is therefore called the *Condorcet winner*.

Candidate Nader was irrelevant in the sense that he did not have a chance to become president. For that reason he could have withdrawn his candidacy. One might think, no problem, because Nader was not chosen anyway. However, without Nader the profile above looks like this:

41%	Bush	Gore
39 + 20 = 59%	Gore	Bush

Now, when applying 'most votes count', Gore would win instead of Bush. So, under 'most votes count' the choice between Bush and Gore is determined by the participation or non-participation of a third (irrelevant) candidate. In other words, 'most votes count' (PR) is *not Independent of Irrelevant Alternatives* (not IIA). Notice that Majority Rule (MR), or pairwise comparison, is (by definition) IIA.

Related to this, the 20% voters with preference ordering *Nader Gore Bush* prefer Gore to Bush. By giving an improper order of preference *Gore Nader Bush* they can ensure that under 'most votes count' Gore becomes the winner with 39 + 20 = 59% of the votes, which is a better outcome for them. In other words, 'most votes count' (PR) is *not strategy-proof*, i.e., cheating may pay off.

Another objection against Plurality Rule (PR) has been pointed out by Donald Saari [13, 14, 15]. Profile p above contains what Saari calls a *reversal portion*:

20	Bush	Gore	Nader
20	Nader	Gore	Bush

These 20 + 20 voters have diametrically opposed preferences and hence cancel each other out. One would intuitively expect that adding a reversal portion to or subtracting it from a given profile does not change the outcome. However, subtracting the reversal portion in question from the original profile p yields:

21	Bush	Gore	Nader
39	Gore	Nader	Bush

Now, under 'most votes count' Gore instead of Bush would become the winner, while one would expect intuitively that the outcome does not change.

9.3.3 Majority Rule (MR): pairwise comparison

As we have remarked earlier, Majority Rule (MR), or pairwise comparison, is Independent of Irrelevant Alternatives (IIA). This follows immediately from the definition of Majority Rule: in a competition between two candidates A and B only the relative positions of A and B in the given profile are compared and a third alternative C has no influence on that. Related to this is that Majority Rule is also strategy-proof; see Exercise 9.16. This might suggest that Majority Rule is a perfect voting rule to aggregate the individual preference orderings of the voters to a common or social ordering of the candidates. However, this is not the case, because in some cases Majority Rule does not yield a feasible outcome, as illustrated by the following so called *Condorcet profile q*:

$$\begin{array}{llll} 1/3 & a & b & c \\ 1/3 & b & c & a \\ 1/3 & c & a & b \end{array}$$

A majority (group 1 and 3) prefers a to b, another majority (group 1 and 2) prefers b to c and again another majority (group 2 and 3) prefers c to a. So, a beats b and b beats c, but not a beats c. On the contrary, c beats a. In other words: Majority Rule is *not transitive*. The outcome under Majority Rule may be cyclic: $a\ b\ c\ a$. This is called *Condorcet's paradox*.

Notice that with only two alternatives violation of transitivity cannot occur because transitivity refers to three alternatives. Transitivity of a relation R on a set V means by definition: if aRb and bRc, then aRc for all elements a, b, c in V.

In the case of three alternatives and a great number of voters, supposing that every individual preference ordering is equally likely, the probability of the occurrence of the *Condorcet paradox*, i.e., the probability of a cyclic outcome, is 1 out of 16, a number which is not negligible small; see Gehrlein [11].

As pointed out by Saari [13, 14, 15], the outcome under Majority Rule may change when we add a *Condorcet portion* to, or subtract it from, a given profile. For instance, consider the following profile r:

$$\begin{array}{llll} 1: & a & c & b \\ 2: & b & a & c \end{array}$$

If we apply Majority Rule to this profile r the outcome is: $b\ a\ c$. But if add to profile r the Condorcet portion s:

$$\begin{array}{llll} 2: & a & b & c \\ 2: & b & c & a \\ 2: & c & a & b \end{array}$$

and next apply Majority Rule to the profile $r+s$ the outcome will become $a\ b\ c$. This is counterintuitive: a Condorcet portion represents voters whose collective advice with regard to social choice is confused and hence should be ignored. Note that in a Condorcet portion each candidate is an equal number of times first, second and third choice. So, intuitively, nobody is preferred. A Condorcet portion should give a tie. But it does not necessarily so under Majority Rule, as we have just seen.

9.3.4 Borda Rule (BR)

The French mathematician and political scientist Jean-Charles de Borda (±1750) proposed to count the number of candidates beaten by a given candidate. That is, if a voter gives an order of preference *Bush Gore Nader*, Bush gets 2 (Borda) points, because he beats both Gore and Nader, Gore gets 1 (Borda) point because he beats only one candidate and Nader gets 0 (Borda) points. Given profile p above

41%	Bush	Gore	Nader
39%	Gore	Nader	Bush
20%	Nader	Gore	Bush

the Borda score of Bush is: $(41 \times 2) + (39 \times 0) + (20 \times 0) = 82$,
the Borda score of Gore is: $(41 \times 1) + (39 \times 2) + (20 \times 1) = 139$, and
the Borda score of Nader is: $(41 \times 0) + (39 \times 1) + (20 \times 2) = 79$.
So, the outcome under the Borda Rule (BR) would be: *Gore Bush Nader*, in this order.

Although the Borda Rule takes the individual preference orderings of the voters into account, the Borda Rule still causes *loss of information*: it does not take into account the intensity with which one candidate is preferred to the next one. If a voter indicates that he prefers candidate A to B he may mean quite different things: he may evaluate A as excellent and B as very good, he may evaluate A as excellent and B as poor, or he may evaluate A as poor and B as reject.

Like Plurality Rule, also the Borda Rule is not Independent of Irrelevant Alternatives (not IIA), as illustrated by the following profile:

3:	c	a	b
2:	a	b	c
1:	a	c	b
1:	b	c	a

Given this profile, c is the Condorcet winner, i.e., c beats all other candidates in a pairwise comparison, but a is the Borda winner with $(3 \times 1) + (2 \times 2) + (1 \times 2) + (1 \times 0) = 9$ Borda points against 8 Borda points for c. In a competition between a and c under application of the Borda Rule the third alternative b turns out to be decisive: without the participation of b the Borda winner would become c with 4 Borda points against only 3 for a.

A serious disadvantage of the Borda Rule is that voters can rather easily act strategically: by giving an improper order of preference they may be able to achieve an outcome which is better for them. The three voters with preference $c\ a\ b$ who want c to win, can easily pretend that a is their last preference and pretend that their order of preference is $c\ b\ a$. In this way they achieve that a gets 3 Borda points less, hence $9 - 3 = 6$, the number of Borda points for c remains 8 and the Borda score of b becomes 4 + 3 = 7. So, by giving an improper order of preference these three voters can achieve an outcome c which they prefer to the outcome a when they give their proper order of preference. In other words, the Borda Rule is *not strategy-proof.*

Another objection against the Borda Rule has been pointed out by Balinski and Laraki [2]: if one removes the Borda winner Gore from the given profile p, the order of the remaining candidates may change under the Borda Rule: leaving out the winner Gore from profile p we get:

41	Bush	Nader
39	Nader	Bush
20	Nader	Bush

Applying the Borda Rule to this profile yields *Nader Bush* as social outcome, while with the winner Gore present the social order between these two candidates was just the opposite: *Bush Nader*.

As pointed out by Saari [13, 14, 15], the outcome under the Borda Rule remains unaffected by adding a reversal portion to, or subtracting it from, a given profile. Why is this so? A reversal portion has the following structure:

$$\begin{array}{ccc} a & b & c \\ c & b & a \end{array}$$

Applying the Borda Rule to this reversal portion, a gets $2 + 0 = 2$ Borda points, b gets $1 + 1 = 2$ Borda points and c also gets $0 + 2 = 2$ Borda Points. So, when we add or subtract a reversal portion, the alternatives get the same number of Borda points more or less. A similar result holds for Majority Rule, but not for Plurality Rule, as we have seen in Section 9.3.2.

The outcome under the Borda Rule also remains unaffected by adding a Condorcet portion to, or subtracting it from, a given profile. The reason is simple: all alternatives in a Condorcet portion get the same number of Borda points. A similar result holds for Plurality Rule, but not for Majority Rule as we have seen in Section 9.3.3.

9.3.5 *Outcome depends on the Voting Rule*

In the preceding subsections we have seen that the outcome of an election does not depend so much on the preferences of the electorate, but rather on the voting rule which aggregates the individual preferences of the voters to a common or social order of preference. Given profile p above, the outcome

under Plurality Rule is:	Bush	Gore	Nader
under Majority Rule is:	Gore	Nader	Bush
and under the Borda Rule is:	Gore	Bush	Nader

Notice that with only two alternatives, Plurality Rule, Majority Rule and the Borda Rule are equivalent, i.e., for all profiles they yield the same outcome; see Exercise 9.17.

9.3.6 *Arrow's Impossibility Theorem*

In the preceeding subsections we have seen that Plurality Rule (PR) or 'most votes count' and the Borda Rule are not Independent of Irrelevant Alternatives (not IIA), but they are transitive. On the other hand, Majority Rule (MR) or pairwise comparison is IIA, but not transitive. The question remains whether one can devise a voting rule which is both IIA and transitive. In 1951 K. Arrow made an abrupt end to this hope by publishing his so called *impossibility theorem* [1]: for three or more alternatives every voting rule which takes as input the individual preference orderings of the voters and which satifies IIA and transitivity (together with some other elementary properties like anonymity and neutrality) is dictatorial, i.e., there will be a voter whose preference is always the social or common preference, no matter what the preferences of the other voters are. Such a voter is called a *dictator*.

First some definitions.

Definition 9.22 (Profile). A profile p associates with every voter a (linear or weak) ordering of the candidates or alternatives.

Definition 9.23 (Voting Rule). A voting rule or voting method M assigns to every profile a common (or social) (weak) ordering $\succeq_M$ of the candidates. The ordering $\succeq_M$ may be weak, i.e., indifferences ($A \approx_M B$, i.e., $A \succeq_M B$ and $B \succeq_M A$) may occur.

There are many proofs of Arrow's theorem. Below we present a simple proof of (a version of) Arrow's theorem, recently published by Balinski and Laraki [3]. They start with listing *May's axioms* [12] *for a voting method M in the case of two candidates*:

Definition 9.24 (May's axioms for a voting method M in the case of two alternatives). 1. *Based on comparisons* The input of the voting method M consists of the individual preference orderings of the voters over the candidates or alternatives.
2. *Unrestricted domain* Every vote configuration (profile) is allowed, in other words, the voting method M should assign a social ordering to every profile p.
3. *Anonymity* Interhanging the names of the voters does not change the outcome.
4. *Neutrality* Interchanging the names of the alternatives does not change the outcome.
5. *Monotonicity* If A wins or is socially indifferent to B ($A \succeq_M B$) and one or more voters change their preference in favor of A, then the voting method M will put A above B ($A \succ_M B$).
6. *Completeness* Given a pair of candidates A and B, the voting method M will put A above B ($A \succ_M B$) or B above A ($B \succ_M A$) or declare them indifferent ($A \approx_M B$).

Theorem 9.1 (May [12]). *In the case of only two alternatives the only voting method which satisfies May's axioms is Majority Rule.* (Remember that in the case of two alternatives Majority Rule, Plurality Rule and the Borda Rule are equivalent!)

Proof. (Balinski and Laraki [3]) Suppose two alternatives A and B and the voting method M satisfies May's axioms. Anonymity implies that only the numbers count: the number n_A of voters who prefer A to B, the number n_B of voters who prefer B to A and the number n_{AB} of voters who are indifferent between A and B. Completeness guarantees that there must be an outcome.

Suppose $n_A = n_B$ and $A \succ_M B$. Because of neutrality changing the names of A and B results in $B \succ_M A$. But the new profile is identical to the original profile. Contradiction. Hence, by completeness, $A \approx_M B$ when $n_A = n_B$.

Suppose $n_A > n_B$. Change the preferences of $n_A - n_B$ voters who prefer A to B in indifferences. By May's axiom of unrestricted domain this profile is allowed and given this profile $A \approx_M B$, as we have just seen. Changing this profile back to the original profile yields $A \succ_M B$ according to May's monotonicity axiom. □

For the case of an arbitrary number of candidates Balinski and Laraki [3] add to May's axioms the following two axioms:
7. *Transitivity* If $A \succeq_M B$ and $B \succeq_M C$, then $A \succeq_M C$.
8. *Independence of Irrelevant Alternatives* (IIA) If $A \succeq_M B$ and other candidates are dropped or adjoined, then again $A \succeq_M B$.
Next Balinski and Laraki prove the following version of Arrow's impossibility theorem.

Theorem 9.2 (Arrow's impossibility theorem [1]**).** *For $n \geq 3$ candidates there is no voting method M which satisfies all eight axioms.*

Proof. (Balinski and Laraki [3]) Consider any two candidates A and B. According to IIA it is sufficient to consider only these two. By Theorem 9.1 axioms 1 till 6 imply that the voting method M is Majority Rule. Because of the axiom of unrestricted domain, Condorcet's paradoxical profile is admitted and hence transitivity is violated. Hence, there can be no voting method which satisfies all eight axioms. □

The question whether it is possible to escape from Arrow's impossibility theorem has kept many scientists busy for more than 60 years: mathematicians, economists, political scientists and philosophers. Notice that when two people say that they prefer A to B they may mean quite different things: one may mean that A is excellent and B is (very) good, while the other may mean that A is acceptable and B should be rejected. With many voters a babylonian confusion of tongues is the result and it should not come as a surprise that problems like the impossibility theorem show up.

Already in the first half of last century people like Gerrit Mannoury, L.E.J. Brouwer, David van Dantzig, Frederik van Eeden and some other like minded, unified in the *Signific Circle* in the Netherlands, have pointed to the importance of a careful use of language. We quote Mannoury:

> Who wants to control his feelings must first analyze them and the traditional language forms are utterly insufficient for this purpose. [Mannoury 1917]

> To the further development of philosophical thoughts an impediment stands in the way. ... I know of no image that gives a clearer idea of this impediment than that of the tower of Babel, symbol of the confusion of tongues. [Mannoury 2017]

This is precisely what happens if different people say that they prefer A to B. They all mean something else!

9.3.7 Domination

In their book [2] Balinski and Laraki present a solution: instead of asking voters their preference ordering over the candidates, one should ask them to give an *evaluation* of all candidates in terms which are well understood by everyone involved. For instance in terms of: excellent (*ex*), very good (*vg*), good (*go*), acceptable (*ac*), poor (*po*) and reject (*re*). The range of evaluations should be sufficiently large such that every voter can express his distinction of the candidates.

Notice that evaluations are much *more informative* than preference orderings: from the evaluations of the candidates by a voter one can easily deduce his preference ordering over the candidates, but not vice versa! From a preference ordering over the candidates one cannot deduce the evaluations by the voter in question.

By a more precise use of language, evaluations instead of orderings, Balinski and Laraki [3] do an astonishing, if not shocking, discovery: **Majority Rule does not respect domination!** Let us illustrate what we mean by an example. Consider two candidates A and B who are evaluated by five voters as rendered in the following *opinion profile*:

voter	1	2	3	4	5
candidate A	*go*	*ac*	*po*	*ex*	*vg*
candidate B	*vg*	*go*	*ac*	*po*	*re*

The first three voters slightly prefer B to A, while the last two voters strongly prefer A to B. According to Majority Rule A is beaten by B with 2 against 3: $B \succ_{MR} A$.

However, if we look at the evaluations of A and B, ordered from high till low, then the following *merit-profile* results:

A	*ex*	*vg*	*go*	*ac*	*po*
B	*vg*	*go*	*ac*	*po*	*re*

It is A who has the better evaluations, in other words, the evaluations of A *dominate* those of B. Hence, A instead of B should be the winner! Majority Rule does not respect domination. On the other hand, any reasonable voting rule should respect domination. Question is whether there exists such a voting rule. And yes, there is: Majority Judgment (MJ) of Balinski and Laraki [2, 3]. Let us illustrate how Majority Judgment works by applying it to the situation just given.

There is a majority of 3 voters who think that A deserves at least a *go*, and there is another majority of 3 voters who think that A deserves at most a *go*. For that reason the *majority grade* of A is by definition *go*. For B there is a majority of 3 voters who think that B deserves at least an *ac*, and another majority of 3 voters who think that B deserves at most an *ac*. Hence, the *majority grade* of B is by definition *ac*. The majority grade of A is higher than the one of B and hence, according to Majority Judgment, A is the winner: $A \succ_{MJ} B$.

Majority Judgment (MJ) looks horizontally for majorities in the merit-profile, while Majority Rule (MR) looks vertically for majorities in the opinion-profile. Majority Judgment (MJ) respects domination, however, Majority Rule (MR) does not.

9.3.8 Majority Judgment (MJ)

Balinski and Laraki develop in their book [2] and in their article [3] a theory, called Majority Judgment (MJ), to aggregate the evaluations (instead of the preference orderings) of the candidates by the voters to a common or social (weak) preference ordering $\succeq_{MJ}$ over the candidates. As suggested by the name Majority Judgment, majorities play an essential role in this aggregation method. Majority Judgment (MJ) is Independent of Irrelevant Alternatives (IIA), transitive and does respect domination.

To explain how Majority Judgment works, let us consider an example with three candidates A, B and C and six voters or judges. The evaluations of the candidates by the voters are given in the following *opinion-profile*:

voter	1	2	3	4	5	6
A:	*ex*	*ex*	*vg*	*ex*	*ex*	*ex*
B:	*ex*	*vg*	*vg*	*vg*	*go*	*vg*
C:	*ac*	*ex*	*go*	*vg*	*vg*	*ex*

Anonymity requires that only the judgments or grades count. The number of times that each grade occurs, from high till low, is rendered in the *merit-profile* of the candidates:

A:	*ex*	*ex*	**ex**	**ex**	*ex*	*vg*
B:	*ex*	[*vg*	**vg**	**vg**	*vg*]	*go*
C:	*ex*	[*ex*	**vg**	**vg**	*go*]	*ac*

There is a 4/6 majority of voters who think that C deserves at least a *vg* and there is another 4/6 majority of voters who think that C deserves at most a *vg*. So, for C there is a 4/6 majority for [*vg*, *vg*]. The *majority grade* of C is therefore by definition *vg*. It is the most accurate possible majority decision about the evaluations of C. In a similar way the 4/6 majorities for A and B have been indicated in boldface.

The merit-profile $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)$ of candidate A *dominates* the merit-profile $\beta = (\beta_1, \beta_2, \dots, \beta_n)$ of candidate B iff for every i, $\alpha_i \geq \beta_i$ and for at least one k, $\alpha_k > \beta_k$. Every reasonable voting method should respect domination. In our example the merit-profile of A dominates the one of B and the one of C. Therefore, Majority Judgment (MJ) will make A the winner: $A \succ_{MJ} B$ and $A \succ_{MJ} C$.

How should Majority Judgment (MJ) rank B and C? The 4/6 majorities for B and C are identical: [*vg*, *vg*]. But for B the 5/6 majority (indicated by the square brackets) is for [*vg*, *vg*], while for C the 5/6 majority is for [*ex*, *go*]. Because none of these pairs dominates the other and because there is more consensus in the evaluations of B than in those of C, Majority Judgment (MJ) will rank B above C. So, the social or common preference ordering under Majority Judgment will be: $A \succ_{MJ} B \succ_{MJ} C$. Notice that Majority Rule (MR), applied to the opinion-profile in our example, will rank C above B: C beats B with 3 against 2, so $C \succ_{MR} B$.

More generally, suppose the evaluations of B are $\beta = (\beta_1, \beta_2, \dots, \beta_n)$ and those of C are $\gamma = (\gamma_1, \gamma_2, \dots, \gamma_n)$, both from high till low, and suppose the most accurate majority where the candidates B and C differ is the majority for $[\beta_k, \beta_{n-k+1}] \neq [\gamma_k, \gamma_{n-k+1}]$. We call $[\beta_k, \beta_{n-k+1}]$ B's *middle-most block* with respect to C and $[\gamma_k, \gamma_{n-k+1}]$ C's middle-most block with respect to B. Majority Judgment‘(MJ) ranks B above C iff (a) the middle-most block of B with respect to C dominates the middle-most block of C with respect to B, or (b) the middle-most block of B with respect to C shows more consensus than the one of C with respect to B.

So, $B \succ_{MJ} C$ iff (a) $\beta_k \succeq \gamma_k$ and $\beta_{n-k+1} \succeq \gamma_{n-k+1}$, with at least one $\succeq$ strict, or (b) $\gamma_k \succ \beta_k \succeq \beta_{n-k+1} \succ \gamma_{n-k+1}$. In all other cases the collections of evaluations are identical and $B \approx_{MJ} C$.

9.3.9 Properties of Majority Judgment

From the definition of Majority Judgment (MJ) follows immediately:

Theorem 9.3. *(Balinski and Laraki) Majority Judgment takes as input the evaluations of the candidates by the voters and satisfies all axioms 2 till 8 in subsection 9.3.6.*

In addition, Majority Judgment (MJ) has among others the following properties.

1. Majority Judgment (MJ) gives a social preference ordering $\succeq_{MJ}$ of the candidates or alternatives and society is indifferent between two candidates A and B, $A \approx_{MJ} B$, precisely when they have the same evaluations. Majority Judgment measures the support of the electorate for the candidates and orders them in proportion to their support. With Majority Rule the voters cannot express their opinions about the candidates, every voter is restricted to supporting one candidate at the expense of all others.
2. From the definitions it is evident that Majority Judgment (MJ) is *Independent of Irrelevant Alternatives* (IIA): whether $A \succeq_{MJ} B$ or $B \succeq_{MJ} A$ does not depend on a third alternative C. As we saw, Plurality Rule and the Borda Rule are not IIA.
3. With more than two candidates, $\succeq_{MJ}$ is *transitive*: if $A \succeq_{MJ} B$ and $B \succeq_{MJ} C$, then $A \succeq_{MJ} C$. As we have seen, Majority Rule (MR) is not transitive.
4. Majority Judgment (MJ) respects domination: if the evaluations of A dominate those of B, then $A \succ_{MJ} B$. Majority Rule (and hence also Plurality Rule and the Borda Rule) do not respect domination.
5. Majority Judgment is *strategy-proof in grading*: a group of voters whose input is higher (respectively, lower) than the majority grade cannot raise (respectively, lower) the majority grade. For instance, suppose candidate A receives the following grades: *good acceptable poor*. The majority grade of A is *acceptable*. The voter who gave A a *good* thinks the majority grade *acceptable* is too low, but he cannot raise the majority grade of A; giving an *excellent* instead of a *good* does not raise the majority grade of A.
   This property certainly does not hold for mechanisms based on adding numbers or taking averages of numbers, neither for the Borda Rule and its variants.
6. Majority Judgment (MJ) is *partially strategy-proof in ranking*: if a voter who prefers B to A, can raise the majority grade of B, then he cannot lower the majority grade of A; and if he can lower the majority grade of A, then he cannot raise the majority grade of B. For instance, suppose voter i gives B a higher evaluation than A and A has the same majority grade as B.

$$\begin{array}{lccc} & \leftarrow & & \\ B & | & \cdot & \\ & \text{majority grade} & i & i \\ A & | & & \cdot \end{array}$$

The only way in which voter i can raise the majority grade of B is by giving B a grade higher than its majority grade instead of a grade lower than B's majority grade. But because i gave a lower grade to A than to B, he cannot lower A's majority grade.

This property certainly does not hold for mechanisms based on adding numbers or taking averages of numbers, neither for the Borda Rule and its variants.

7. The majority grade of a candidate is an important signal both to the candidate and to the electorate.
8. Majority Judgment stimulates candidates to get the highest possible grades of as many voters as possible; every grade contributes to the final judgment.
9. Candidates cannot focus on 51% of the electorate and, once the winner, claim to represent the whole electorate.

9.3.10 Point Summing and Approval Voting

One should notice that voting methods, where voters give points to candidates and where candidates are ordered according to the number of points they have collected, like Majority Judgment also satisfy the axioms 2 till 8 in Section 9.3.6. However, such methods are not strategy-proof neither in grading nor in ranking. In addition, a voting method based on giving points to the candidates is not consistent with Majority Judgment, neither with Majority Rule. Consider the following example:

	1	2	3	4	5	6	7
A:	*ex*	*ex*	*ex*	*ac*	*ac*	*ac*	*ac*
B:	*po*	*po*	*po*	*go*	*go*	*go*	*go*

Looking at this opinion profile vertically, we see that B beats A with 4 against 3, so B is the Majority Rule winner: $B \succ_{MR} A$. Looking at this profile horizontally, we see that the majority grade of B is *go* and the majority grade of A is only *ac*; so, in this example B is also the Majority Judgment winner: $B \succ_{MJ} A$. However, with 5 points for *ex*, 4 for *vg*, 3 for *go*, 2 for *ac*, 1 for *po* and 0 for *re*, A wins with 23 points against 15 for B. So, adding points is not consistent with Majority Judgment, neither with Majority Rule.

The idea of Approval Voting [6] is that every voter gives 1 point to each candidate he or she approves of and 0 points to every candidate he or she disapproves of. With 1 point for *go* or higher, B wins with 4 points against 3 for A. But with 1 point for *ac* or higher, A wins with 7 points against 4 for B. So, Approval Voting yields arbitrary outcomes and is not consistent with Majority Judgment, neither with Majority Rule.

9.3.11 *Majority Judgment with many Voters*

Consider the following merit profile for two candidates A and B:

	ex	*vg*	*go*	$\updownarrow$	*go*	*ac*	*po*	*re*
A:	28.63	16.42	04.95	$\updownarrow$	06.72	14.79	14.25	14.24
B:	12.35	21.71	15.94	$\updownarrow$	09.30	20.08	11.94	08.69

Left and right of the middle one finds 50% of the number of evaluations. For $\varepsilon \leq 4.95$, A and B have a $(50+\varepsilon)\%$ majority for $[go, go]$. But for $\varepsilon < 6.72 - 4.95 = 1.77$, A has a $(54.95+\varepsilon)\%$ majority for $[vg, go]$, while B has a $(54.95+\varepsilon)\%$ majority for $[go, go]$. Because A's middlemost block dominates the one of B, $A \succ_{MJ} B$. This is the case because $4.95 < \min\{6.72, 15.94, 9.30\}$. Finding the smallest of these four numbers is the same as finding the highest percentage of each candidate's grades strictly above and strictly below their majority grades.

Let p_A be the percentage of A's grades strictly above the majority grade α_A of A and q_A the percentage of A's grades strictly below α_A. A's *majority gauge* is by definition (p_A, α_A, q_A). So, in our example the majority gauge of A is (45.05, *go*, 43.28) and the majority gauge of B is (34.06, *go*, 40.71).

The *majority-gauge rule* $\succ_{MG}$ ranks A above B, $A \succ_{MG} B$, iff $\alpha_A \succ \alpha_B$ or ($\alpha_A = \alpha_B$ and $p_A > max\{q_A, p_B, q_B\}$) or ($\alpha_A = \alpha_B$ and $q_B > max\{p_A, q_A, p_B\}$).

In our example: $p_A = 45.05 > \max\{43.28, 34.06, 40.71\}$, therefore $A \succ_{MG} B$. If $\succeq_{MG}$ is decisive (written as $\succ_{MG}$), then its ordering $\succ_{MG}$ is identical to the one of $\succ_{MJ}$. So, in our example it also follows that $A \succ_{MJ} B$, as we already saw above.

9.3.12 *Presidential Elections in the USA*

In [4] Balinski and Laraki give an analysis of the recent (2016) presidential elections in the USA. Their conclusion is unambiguous: the voting method in the USA does not work, more precisely, it does not select the candidate who gets globally the highest evaluation of the electorate. To illustrate this, they use the results mentioned below of a poll by the Pew Research Center in March 2016 among 1787 voters from all political stripes.

candidate	great	good	average	poor	terrible
A	05	28	**39**	13	15 %
B	10	26	**26**	15	23 %
C	07	22	**31**	17	23 %
D	11	22	**20**	16	31 %
E	10	16	12	**15**	47 %

For candidate A there is a majority of 05 + 28 + 39 = 64% who thinks that he deserves at least an *average* and there is another majority of 15 + 13 + 39 = 67% who thinks that he deserves at most an *average*. Therefore, *average* is by definition the majority grade of candidate A. In the table the majority grades of the different candidates have been indicated by bold face letters. Notice that the opinions of the voters are clearly much more detailed than can be expressed by Majority Rule. Also the percentages of voters who think that candidates D and E would be bad presidents is relatively high.

Next Balinski and Laraki determine how, given these judgments, Majority Judgment would rank the candidates. The majority grade of candidate A, B, C and D is *average*, the one of candidate E is *poor*. The majority gauge of candidate A is (33, average, 28), because $p_A = 5 + 28 = 33$ and $q_A = 13 + 15 = 28$. In the table below the majority gauges of all candidates are listed, from which one may derive a ranking of the candidates according to the majority-gauge rule, which is also the Majority Judgment ranking.

	majority grade	majority gauge
1. A	average	(33, average, 28)
2. B	average	(36, average, 38)
3. C	average	(29, average, 40)
4. D	average	(33, average, 47)
5. E	poor	(38, poor, 47)

Because $q_B = 38 > \max\{33, 28, 36\}$ it follows that $A \succ_{MG} B$; because $q_C = 40 > \max\{36, 38, 29\}$ it follows that $B \succ_{MG} C$ and because $q_D = 47 > \max\{29, 40, 33\}$ it follows that $C \succ_{MG} D$. Finally, because the majority grade *average* of D is higher than the majority grade *poor* of E, it follows that $D \succ_{MG} E$.

Amazingly, at election day the two main candidates were D and E, of which E won the election, because he won in most states, although he did not get most votes.

The Majority Judgment ranking is the logical result of majorities which decide about the judgments of the candidates instead of Majority Rule which ranks candidates according to the number of votes they get. *Majority Judgment measures the support of the electorate for the different candidates and ranks them according to their support.* With Majority Rule the voters cannot express their opinions about the candidates; every voter is restricted to supporting one candidate at the same time excluding all others.

Why can Majority Rule work out so poorly? To make this clear, Balinski and Laraki [4] consider the merit-profile of candidates D and E:

	great	good	average	poor	terrible
D	11	22	**20**	16	31 %
E	10	16	12	**15**	47 %

Notice that the evaluations of D dominate those of E. Hence, D should win, as also becomes clear from the following table:

at least	great	good	average	poor	terrible
D	11	33	53	69	100 %
E	10	26	38	53	100 %

Any decent voting method should rank D above E. But Majority Rule can easily fail to make D the winner: suppose that underlying the merit-profile for D and E is the following opinion-profile for these candidates:

	10	16	12	15	14	11	12	04	04	02
D	*go*	*av*	*po*	*te*	*te*	*gr*	*go*	*av*	*po*	*te*
E	*gr*	*go*	*av*	*po*	*te*	*te*	*te*	*te*	*te*	*te*

The individual vote percentages in this opinion-profile are in accordance with the degrees that each candidate received in the merit-profile. For instance, the 22% voters who gave a *good* to *D* are now divided in two groups: a group of 10% voters who gave a *good* to *D* and a *great* to *E* and a group of 12% voters who evaluated *D* as *good* and *E* as *terrible*. Applying Majority Rule to this opinion-profile, *E* will beat *D* with 10 + 16 + 12 + 15 = 53% against 11 + 12 + 4 + 4 = 31%, while *D*'s evaluations dominate those of *E*. Notice that in this opinion-profile the 53% voters who prefer *E* to *D* only slightly do so, while most voters who prefer *D* to *E* do so strongly.

9.3.13 Presidential Elections in France

In [5] Balinki and Laraki take a look at the French presidential elections. Their conclusion is again extremely negative: the French election system can easily select a winner who is rejected by a vast majority of the voters. The French presidential election is in two rounds: 1. If in the first round a candidate has more than half of the votes, then he or she is elected. 2. Otherwise, there is a second round between the two candidates with most votes in the first round.

Let us start with having a look at the presidential elections of 2007 with twelve candidates of which Sarkozy, Royal and Bayrou were the most important ones. The results of the first round were as follows:

31.2%	Sarkozy	Bayrou	Royal
25.9%	Royal	Bayrou	Sarkozy
18.6%	Bayrou		
xy.z%	???	Bayrou	Sarkozy/Royal

In the first round Sarkozy and Royal had most votes, but less than 50%. Therefore, there was a second round between them, in which Sarkozy won. But the polls showed clearly that a majority of 25.9 + 18.6 + xy.z % of the voters preferred Bayrou to Sarkozy and that another majority of 31.2 + 18.6 + xy.z % preferred Bayrou to Royal. As we shall see further on in this subsection applying Majority Judgment would most likely have chosen Bayrou as the winner.

At the French presidential elections of April 21, 2017, there were initially three major candidates, say *A*, *B* and *C*. Suppose the preference orderings of the voters were as follows:

34%	*A*	*B*	*C*
32%	*B*	*A*	*C*
34%	*C*	*B*	*A*

In this case nobody has more than 50% of the votes and *B*, who has least votes, is eliminated. The second round is then between *A* and *C*, in which *A* gets 34 + 32 =

66% of the votes and wins. Next suppose that in the first round *A* gets more support at the expense of candidate *C*:

37%	*A*	*B*	*C*
32%	*B*	*A*	*C*
31%	*C*	*B*	*A*

Then after the first round *C* is eliminated and *B* wins in the second round with 32 + 31 = 63% of the votes. More support for the winning candidate *A* in the first round causes that he becomes a loser instead of a winner. In other words: *the French election mechanism is not monotonic*: more support may mean losing instead of winning.

On April 22, 2007, Balinski and Laraki did an experiment among 1752 voters in three districts of Orsay. These voters were asked to fill in, apart from the official voting ballot, also the following voting ballot.

Pour présider la France, ayant pris tous les éléments en compte, je juge en conscience que ce candidat serait:

	très bien	bien	asses bien	passable	insuffisant	à rejeter
Besancenot						
Buffet						
Schivardi						
Bayrou						
Bové						
Voynet						
Villiers						
Royal						
Nihous						
Le Pen						
Laguiller						
Sarkozy						

Attribuer à chaque candidat une évaluation parmi les mentions.

The results for the three most important candidates were:

	exc	very good	good	acc	poor	reject
Bayrou	13.6	30.7	**25.1**	14.8	8.4	7.4
Royal	16.7	22.7	**19.1**	16.8	12.2	12.6
Sarkozy	19.1	19.8	**14.3**	11.5	7.1	28.2

All three candidates have majority grade *good*. Let *p*, resp. *q* be the percentage strictly above, resp. strictly below the majority grade.

majority rank	*p*	majority grade	*q*	national rank
1 Bayrou	44.3	*good*	30.6	3
2 Royal	39.4	*good*	41.5	2
3 Sarkozy	38.9	*good*	46.9	1

The majority gauge rule yields the ranking: 1 Bayrou, 2 Royal en 3 Sarkozy. One may easily motivate this outcome by looking at the cumulative table below. With the exception of the *exc* column it holds for every column that Bayrou scores better than Royal and Royal better than Sarkozy.

at least	exc	very good	good	acc	poor	reject
Bayrou	13.6	44.3	69.4	84.2	92.6	100
Royal	16.7	39.4	58.5	75.3	87.5	100
Sarkozy	19.1	38.9	53.2	64.7	71.8	100

9.3.14 Elections for Parliament in the Netherlands

Majority Judgment may be used to determine a common or social preference ordering over the candidates. Those candidates may be political parties. But Majority Judgment does not yield a seat distribution among the parties. However, Majority Judgment might be used in the Netherlands to choose a mayor, a prime minister, a chairman, etc. That there is a need in the Netherlands for a better election mechanism may become evident from the following examples.

In the table below one finds the vote and seat distribution after the elections for parliament on September 6, 1989:

Party	% of votes	number of seats
CDA	35.3	54
PvdA	31.9	49
VVD	14.6	22
D66	07.9	12
GL	04.1	06
SR	05.0	07

Suppose the following plausible profile is underlying the seat distribution above:

35.3	CDA	D66	VVD	SR	PvdA	GL
31.9	PvdA	GL	D66	CDA	VVD	SR
14.6	VVD	PvdA	D66	SR	CDA	GL
07.9	D66	PvdA	CDA	VVD	GL	SR
04.1	GL	PvdA	D66	CDA	VVD	SR
05.0	SR	VVD	CDA	D66	PvdA	GL

Notice: VVD beats PvdA with 35.3 + 14.6 + 05.0 = 54.9 against 31.9 + 07.9 + 04.1 = 43.9, but PvdA gets 49 seats and VVD only 22. Similarly: D66 beats CDA with 31.9 + 14.6 + 07.9 + 04.1 = 58.5 against 35.3 + 05.0 = 40.3, but CDA gets 54 seats and D66 only 12. Van Deemen [9] calls this phenomenon: the *more preferred, but less seats* paradox.

The situation may be even worse: a party may beat every other party in a pairwise comparison (Majority Rule) and still get less or no seats at all, as becomes clear from the example below. On September 6, 1989, the Greens (G) were participating, but did not get any seat. Suppose G was for all voters the second choice:

35.3	CDA	G	D66	VVD	SR	PvdA	GL
31.9	PvdA	G	GL	D66	CDA	VVD	SR
14.6	VVD	G	PvdA	D66	SR	CDA	GL
07.9	D66	G	PvdA	CDA	VVD	GL	SR
04.1	GL	G	PvdA	D66	CDA	VVD	SR
05.0	SR	G	VVD	CDA	D66	PvdA	GL

Under pairwise comparison (Majority Rule) G beats every other party and hence is the Condorcet winner. But G gets no seat at all in the Dutch system. At another occasion, a similar fate struck party DS70, which was second or third choice for many voters. Van Deemen [9] calls this phenomenon: the *Condorcet winner, but no or less seats* paradox.

From empirical research [10] it turns out that the 'more preferred, but less seats' paradox occurs abundantly. And from empirical research it also becomes clear that

D66 in 1994 was the Condorcet winner, but got less seats than PvDA, CDA and VVD. In 1982 PvdA was the Condorcet winner, but got less seats than CDA.

Exercise 9.16. Prove that pairwise comparison is *strategy-proof* in the following sense: Let S be a set of voters and p, q profiles such that $p(i) = q(i)$ for all voters i not in S (the individuals in S give in q a dishonest preference). Let x be the Condorcet winner given p and y the Condorcet winner given q. Suppose $x \neq y$. Then there is an individual $i \in S$ who in his honest individual preference ordering $p(i)$ strictly prefers x to y. So, for that individual the strategic change towards $q(i)$ is a disadvantage.

Exercise 9.17. Prove that for two alternatives 'most votes count' (Plurality Rule), pairwise comparison (Majority Rule) and the Borda Rule give the same results. Conclude that Arrow's theorem does not hold for the case of two alternatives.

Exercise 9.18. *Agenda's: Berlin versus Bonn*
At June 20, 1991, the German parliament had to make a choice among the following three alternatives:
(a) the parliament moves to Berlin, but the ministries stay in Bonn;
(b) both the parliament and the ministries move to Berlin;
(c) both the parliament and the ministries stay in Bonn.
The council of elderly had made an agenda, which was essentially as follows: in the first round the representatives have to make a choice between (a) and not (a). In the second round: if (a) is accepted, then the final choice is (a); if not, then the representatives have to choose between (b) and (c).

From a reconstruction it has become pretty evident that the preferences of the 660 representatives were given in the following profile p:

077: a b c
070: a c b
178: b a c
083: b c a
190: c a b
062: c b a

i) Check that the outcome will be (b), in accordance with the real state of affairs. Verify that given profile p there is no Condorcet winner.
ii) Why is the agenda set by the council of elderly not fair?
iii) Check that if the 83 representatives change their preference ordering b c a into b a c and the preference orderings of the other representatives remain the same, then (a) will be the Condorcet winner. Nevertheless, in this case (b) will again be the outcome under the agenda devised by the council of elderly.
iv) A more fair agenda than the one above would be agenda I: in the first round choose between (a) and (b), and in the second round choose between the winner of the first round and (c). Why is this agenda more fair? Check that if a Condorcet winner exists, it will always be the outcome under this agenda I. Check that the outcome under agenda I given profile p will be (c).
v) Devise an agenda II, respectively III, such that the outcome under agenda II, respectively III, given profile p, will be (a), respectively (b).

Exercise 9.19. *District Paradox: more votes, but less seats.*
Suppose there are three districts and two parties, twenty voters in each district and

in each district the Plurality Rule is used to determine the winner. Suppose the ballot yields the following results:

	Candidate of party A	Candidate of party B	Elected candidate
district 1	11 votes	09 votes	A
district 2	11 votes	09 votes	A
district 3	05 votes	15 votes	B

Party A gets a majority in the House of Commons and will form the cabinet. But party B receives more votes (33) than party A (27). So, if the government would be chosen directly, it would be composed by party B. The majority attributed to party A is called a *manufactured majority*: a majority of the seats obtained by a minority of the voters.

Exercise 9.20. *Discursive Paradox in judgment aggregation*
We explain the *discursive paradox* using the following example, due to Saari: A three member faculty committee must determine whether or not a student should be advanced to Ph.D. candidacy. A majority vote is required to advance. Each faculty member's decision is based on the student's performance on both a written and an oral exam. If a faculty member feels that the student failed one or both of these exams, she is instructed to fail the student. The results follow, where a 'yes' or 'no' indicates the judge's opinion on an exam and whether to advance.

Judge	written	oral	decision
1	yes	yes	yes
2	no	yes	no
3	yes	no	no
Outcome	yes	yes	no

Exercise 9.21. Consider the following Condorcet table of D. Saari:

Ranking	$\{A,B\}$	$\{B,C\}$	$\{A,C\}$
$A > B > C$	$A > B$	$B > C$	$A > C$
$B > C > A$	$B > A$	$B > C$	$C > A$
$C > A > B$	$A > B$	$C > B$	$C > A$
Outcome	$A > B$	$B > C$	$C > A$

Verify that by replacing $A > B$, $B > C$ and $A > C$ by 'yes', $B > A$, $C > B$ and $C > A$ by 'no', the discursive paradox in Exercise 9.20 is a special case of the Condorcet paradox. Notice that the table below, in which the individual preferences are not transitive but cyclic, gives under Majority Rule the same result as the table above:

Ranking	$\{A,B\}$	$\{B,C\}$	$\{A,C\}$
$A > B > C > A$	$A > B$	$B > C$	$C > A$
$B > C > A > B$	$A > B$	$B > C$	$C > A$
$C > B > A > C$	$B > A$	$C > B$	$A > C$
Outcome	$A > B$	$B > C$	$C > A$

So, pairwise comparison ignores the rationality of the voters, i.e., that voters are transitive. Similarly, the IIA condition ignores the rationality of the voters.

Exercise 9.22. *Sen's Paradox: even a minimal form of Liberalism is impossible.*
A voting rule satisfies the *Pareto condition* := if all voters prefer x to y, then also society should prefer x to y.

Assuming that voter 1, respectively 2, is decisive over the pair $\{A,B\}$, respectively $\{C,D\}$ and that the voting rule satisfies the Pareto condition, determine in the table below the outcome for each pair and notice that a cyclic outcome results. This is *Sen's paradox*: even a minimal form of Liberalism is impossible.

Voter	Preference	$\{A,B\}$	$\{B,C\}$	$\{C,D\}$	$\{A,D\}$
1	$D>A>B>C$	$A>B$	$B>C$	–	$D>A$
2	$B>C>D>A$	–	$B>C$	$C>D$	$D>A$
	outcome				

In this table a dash indicates a ranking that is irrelevant for the decision rule because another agent is decisive over that pair.

Notice that for instance for voter 2 it is immaterial whether his $\{A,B\}$ preference is $A>B$ or $B>A$ (because voter 1 is decisive over this pair). But the first choice makes his preferences cyclic, while the second choice makes them transitive - a huge difference. So, the assumptions imposed on the voting rule dismiss the individual rationality assumption (that a voter's preferences are transitive).

9.4 Solutions

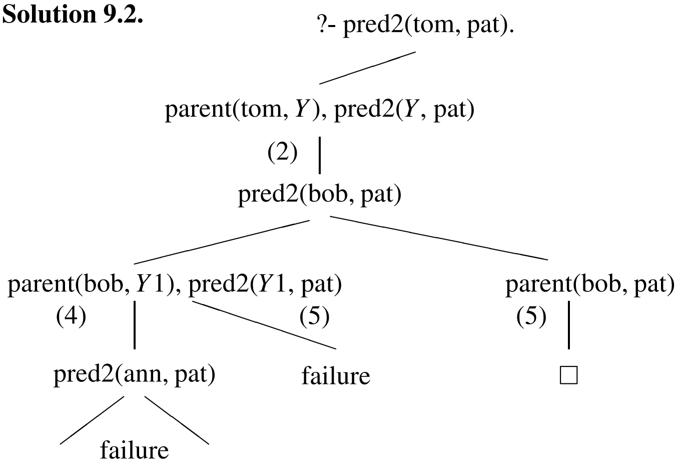
Solution 9.1. Extend the program in Example 9.1 as follows.

(8) male(bob). (11) female(pam). (14) female(ann).
(9) male(tom). (12) female(liz). (15) offspring(X,Y) :- parent(Y,X).
(10) male(jim). (13) female(pat). (16) father(X,Y) :- parent(X,Y), male(X).
(17) mother(X,Y) :- parent(X,Y), female(X).
(18) sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X).
(19) brother(X,Y) :- parent(Z,X), parent(Z,Y), male(X).

```
      ?- mother(tom, liz).                     ?- sister(ann, pat).

         (17) |                                   (18)  |
 parent(tom, liz), female(tom)      parent(Z, ann), parent(Z, pat), female(ann)
         (3)  |                                   (4)   |
        female(tom)                       parent(bob, pat), female(ann)
              |                                   (5)   |
           failure                                female(ann)
             no                                   (14)  |
                                                        □
                                                       yes
```

Solution 9.2.



The system has to backtrack many times before it finds a successful branch in the search tree.

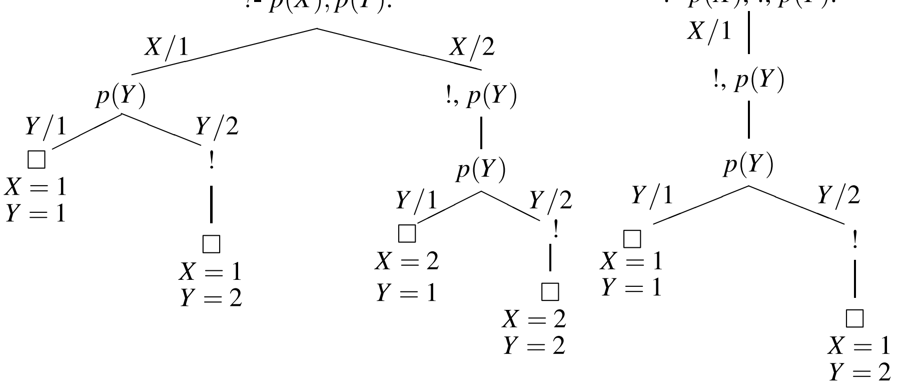
Solution 9.3. a) Replace Y by $f(X)$; Y does not occur in $f(X)$; result: p($f(X)$,Z) and p($f(X)$,c). Next replace Z by c. Consequently, p($f(X)$,Z) and p(Y,c) can be matched and unified. Result: p($f(X)$,c).
b) X and $f(X)$ can be matched, but not unified: replace X by $f(X)$; but X does occur in $f(X)$.
c) Replace Y by $f(X)$. Result: p($f(X)$,c) and p($f(X)$,$f(Z)$). c and $f(Z)$ cannot be matched.

Solution 9.4. conc([ ], L, L). conc([X | $L1$], $L2$, [X | L]) :- conc($L1$, $L2$, L).

Solution 9.5. del(X, [X | L], L). del(X, [Y | L], [Y | $L1$]) :- del(X, L, $L1$).

Solution 9.6. length([ ], 0). length([X | L], N) :- length(L, M), N is $M+1$.

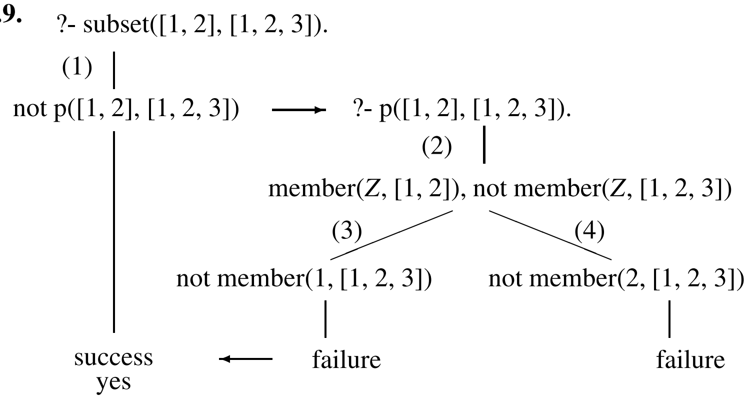
Solution 9.7.



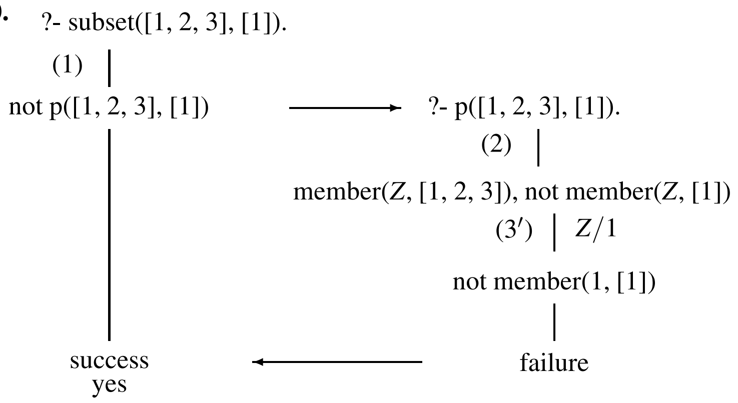
Solution 9.8.

$p(X,0)$:- $X < 1$, ! .
$p(X,1)$:- $X >= 1$, $X < 2$, ! .
$p(X,2)$:- $X >= 2$.

Solution 9.9.



Solution 9.10.

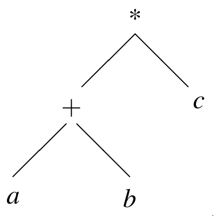


Solution 9.11.

husband(X) :- family(X, _, _).
wife(X) :- family(_, X, _).
child(X) :- family(_, _, L), member(X,L).
exists(X) :- husband(X); wife(X); child(X).

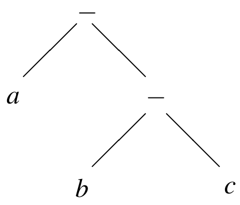
1. ?- exists(person(N,S, _, _)).
2. ?- child(person(N,S, date(_, _, 1973), _)).
3. ?- wife(person(N,S, _, works(_, _))).
4. ?- exists(person(N,S, date(_, _, Y), unemployed)), $Y < 1960$.
5. ?- exists(person(N,S, date(_, _, Y), works(_, Sal))), $Y < 1960$, Sal > 10000.
6. ?- family(person(_, S, _, _),_, [_, _ | _]).
7. ?- family(person(_, S, _, _), _, [ ]).

Solution 9.12. 1. The reading $(a+b)*c$ has the following structure:



Now the precedence of $a + b$ is 500, which is greater than the precedence of $*$. Therefore this reading is rejected.

2. The reading $a - (b - c)$ has the following structure:



The precedence of $b - c$ is 500, which is not strictly smaller than that of the operator $-$. Since the operator $-$ has been defined to be of type yfx, this reading is impossible

3. Since ‘has’ has been defined as an infix operator and the arguments ‘peter’ and ‘information’ have precedence 0, which is strictly smaller than the precedence 600 of ‘has’, ‘peter has information’ will be read as ‘has(peter, information)’.

Solution 9.13.

a)
```
SELECT   t.snm, t.sadr, t.res
FROM     SP  t
WHERE    t.wnr = 9
AND      t.nbd > 2
```

b)
```
SELECT t.snr, t.snm
FROM SP  t
WHERE t.snr IN
     (SELECT u.snr
      FROM ADM u
      WHERE u.indat = 19800303
      AND u.reas = ‘informaritis’)
```

c)
```
SELECT  s.rnr
FROM  R  s
WHERE  s.rnr NOT IN
     (SELECT t.rnr
      FROM ADM t
      WHERE t.indat ≤ 19800518
      AND t.indat ≥ 19800509
      AND t.pnr IN
         (SELECT u.pnr
          FROM  P u
          WHERE  u.pres = ‘Princeton’))
```

d)
```
SELECT t.pnr, t.pnm, t.padr, t.pres
FROM  P t
WHERE t.pnr IN
     (SELECT u.pnr
      FROM ADM u
      WHERE u.snr IN
          (SELECT s.snr
           FROM SP  s
           WHERE s.wnr = 9))
```

Solution 9.14. $D_1 \bowtie D_2$ is the empty set since no tuples agree on the common attribute ‘name’.

$D_3 \bowtie D_2$

nr	wnm	sal	sex	anr	name	man
8	Johnson	2200	male	1	production	9
7	Johnson	3100	female	2	planning	7
9	Kiviat	2900	male	1	production	9

$D_1 \bowtie D_4$

nr	name	sal	sex	dept	anr	anm
7	Johnson	3100	female	2	2	planning
9	Kiviat	2900	male	1	1	production

$D_3 \bowtie D_4$

nr	wnm	sal	sex	anr	anm
7	Johnson	3100	female	2	planning
9	Kiviat	2900	male	1	production

Solution 9.15. The patient with number 500 is from Cranbury and has been hospitalized in the period in question in room number 11. So, 11 should not occur in the set of all room-numbers in which no patients from Cranbury were hospitalized between August 11 and 17, 1977. However, 11 is an element of the indicated set. For consider $s1$; $s1$(rnr) = 11. Then there is $t \in$ ADM, namely $t1$, such that t(rnr) = $s1$(rnr) and 19770811 $\leq t$(indat) $\leq$ 19770817 and $\neg\exists u \in$ P [u(pnr) = t(pnr) = 400 $\wedge$ u(pres) = 'Cranbury'].

Solution 9.16. Let S be a set of voters (who manipulate) and p, q profiles such that for all i not in S, $p(i) = q(i)$. Let x be the Condorcet winner at p and let y be the Condorcet winner at q. Suppose that $x \neq y$.
Because x is the Condorcet winner at p, we know for profile p that x beats y in a pairwise comparison. And because y is the Condorcet winner at q, we know for profile q that y beats x in a pairwise comparison. So, there must be at least one individual i such that
1. i prefers x to y in p, and
2. i prefers y to x in q.
Because only voters in coalition S give another (dishonest) preference order, individual i must be in coalition S. Because in the real (honest) profile p, i prefers x to y, i is punished for the strategic behaviour of the coalition S he or she belongs to.

Solution 9.17. a) Call the alternatives x en y and suppose:

m:	x	y
n:	y	x

Then the Borda score of x equals m en the one of y equals n. Therefore: the outcome under the Borda Rule is $x\ y$ if and only if (iff) $m > n$. But also: the outcome under Plurality Rule is $x\ y$ iff $m > n$. And similarly: the outcome under Majority Rule is $x\ y$ iff $m > n$. Hence, with two alternatives, the Borda Rule, Plurality Rule and Majority Rule yield the same outcome.
b) Because Majority Rule is Independent of Irrelevant Alternatives and in the case of two alternatives trivially is transitive (transitivity says something about 3 alternatives), this makes clear that in the case of two alternatives the theorem of Arrow does not apply: Majority Rule is not dictatorial.

Solution 9.18. i) The first round is between (a) and not (a); 77 + 70 = 147 voters vote for (a), all others vote for not (a) So, the second round is between (b) and (c). 77 + 178 + 83 = 338 representatives vote for (b) and 70 + 190 + 62 = 322 vote for (c). Therefore (b) wins.

Given profile p, (a) beats (b) with 77 + 70 + 190 = 337 votes against 323; (b) beats (c) with 77 + 178 + 83 = 338 votes against 322; and (c) beats (a) with 83 +

190 + 62 = 335 votes against 325. So, given p there is no Condorcet winner.
ii) The agenda set by the council of elderly is not fair, because (a) has to compete with both (b) and (c) simultaneously.
iii) If the 83 representatives change their preference ordering b c a into b a c and the preference orderings of the other representatives remain the same, then one easily checks that (a) beats (b) with 77 + 70 + 190 = 337 against 323 votes and that (a) beats (c) with 77 + 70 + 178 + 83 = 408 against 252 votes. So, in this new configuration (a) is the Condorcet winner. But according to the agenda set by the council of elderly (b) would again become the winner.
iv) Agenda I is more fair than the agenda set by the council of elderly because according to this agenda in every round only two alternatives are compared and every alternative is compared with at least one other alternative. If given a profile there is a Condorcet winner, this Condorcet winner will also win using agenda I, because the Condorcet winner will in the first or the second round be compared with another alternative and from that moment on be the winner in every next round. Given profile p and using agenda I, in the first round (a) beats (b) and in the second round (c) beats (a). So, the outcome under agenda I given profile p will be (c).
v) Agenda II: first round between (b) and (c); second round: between (a) and the winner of the first round. Given profile p, the outcome under agenda II will be (a). Agenda III: first round between (a) and (c); second round: between (b) and the winner of the first round. Given profile p, the outcome under agenda III will be (b).

Solution 9.19. According to Plurality Rule party A wins in district 1 and 2, while party B only wins in district 3. So, party A gets 2/3 of the seats in parliament. But the total number of votes for party A is 11 + 11 + 5 = 27, while party B has 9 + 9 + 15 = 33 votes.

Solution 9.20. A 2/3 majority of the judges gives a 'yes' for the written exam, another 2/3 majority of the judges gives a 'yes' for the oral exam, but another 2/3 majority of the judges gives a 'no' for the final decision. So, judgment aggregation with Majority Rule is problematic.

Solution 9.21. Majority Rule looks only at pairs of candidates. Transitivity concerns three or more candidates. By looking only at pairs of candidates, as required by Independence of Irrelevant Alternatives, transitivity, and hence the rationality of the voters, cannot be taken into account.

Solution 9.22. The outcome is: $A > B$ (1), $B > C$ (2), $C > D$ (3) and $D > A$ (4).
(1) because 1 is decisive over the pair $\{A, B\}$, (2) because of the Pareto condition, (3) because 2 is decisive over the pair $\{C, D\}$ and (4) because of the Pareto condition.

References

1. Arrow, K., *Social Choice and Individual Values*. Yale University Press, 1951.

2. Balinski, M., and R. Laraki, *Majority Judgment; Measuring, Ranking and Electing*. MIT Press, Cambridge; MA, 2010.
3. Balinski, M., and R. Laraki, *Majority Judgment vs Majority Rule*. Cahier 2016-4, Ecole Polytechnique, Paris, 2016.
4. Balinski, M., and R. Laraki, *Trump and Clinton victorious: proof that US voting system does not work*. The Conversation 58752.
5. Balinski, M., and R. Laraki, *Pour éviter un nouveau 21 Avril instaurons le jugement majoritaire*. The Conversation 58178.
6. Brams, S. and Peter C. Fishburn, *Approval Voting*. Springer, 2007.
7. Bratko, I., *PROLOG, Programming for Artificial Intelligence*. Addison Wesley, 1986, 2011.
8. Brock, E.O. de, *The Foundations of Semantic Databases*. Prentice Hall, 1993.
9. Deemen, A. van, Paradoxes of Voting in List Systems of Proportional Representation. *Electoral Studies*, 12:3, 234-241, 1993.
10. Deemen, A. van, Empirical evidence of paradoxes of voting in Dutch elections. *Public Choice* 97: 475-490, 1998.
11. Gehrlein, W.V., Condorcet's paradox and the likelihood of its occurrence: different perspectives on balanced preferences. *Theory and Decision* 52, pp. 171-199, 2002.
12. May, K.O., A set of independent, necessary and sufficient conditions for simple majority decision. *Econometrica* 20, 680-684, 1952.
13. Saari, D., *Chaotic Elections! A mathematician looks at voting*. American Mathematical Society, 2001.
14. Saari, D., *Decisions and Elections; explaining the unexpected*. Cambridge University Press, 2001.
15. Saari, D., *Disposing Dictators, Demystifying Voting Paradoxes*. Cambridge University Press, 2008.
16. Lloyd, J.W., *Foundations of Logic Programming*. Springer Verlag, Berlin, 1987.
17. Sterling, L. and Shapiro, E., *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1986, 1994.