Chapter 1

Informal introduction to logic databases

The paradigm of logic databases evolved during the 1980s in the form of *deductive databases*, based on the merge of relational databases and logic programming. Deductive databases have been developed with the aim of increasing expressive power of relational query languages, and in particular in connection with the inability of the latter to express recursive queries.

Research in logic databases uses the techniques of logic programming, relational databases, automated theorem proving and mathematical logic in general. In this chapter we give an informal introduction to logic databases.

1.1 What is a logic database?

Databases are intended to handle large amounts of data efficiently. Early database languages and systems used *navigational query languages*, where a part of the query described an explicit way of retrieving data relevant to the query. As a consequence, database languages were of a low level. The development and the success of relational databases have shown the importance of *declarative* query languages. Instead of describing a separate procedure for answering each query, the user has gained the possibility of specifying queries in a nonprocedural language with a precise and transparent semantics based on the relational algebra and the relational calculus.

In declarative query languages the user specifies *what* data should be retrieved from the database. *How* to retrieve the data is left to the database itself. It results in a significant increase in *data independence*, because the query language is only concerned with the logical structure of data. This makes it easier to develop and maintain the database through the years. The precise semantics allows one to implement efficient query optimization techniques.

Logic databases are databases based on logic. The technology of logic databases

extends the relational database technology based on the ideas developed in logic programming. Thus, logic databases inherit ideas and properties its two predecessors: relational databases and logic programming.

Logic programming originally appeared with the intention of providing a *declarative programming* methodology. The practical development of general purpose logic programming languages has seriously declined from this intention. However, at least two fields stemmed from logic programming implemented the declarative programming paradigm: logic databases and *constraint logic programming*.

In logic databases, data are described by logical formulas, usually in a restricted subset of first-order logic. These formulas are intended to specify part of the external world relevant to the application at hand, called the *application world*. Thus, a logic database is a *logical representation* of the application world. Henceforth, the semantics of logic databases is based on mathematical logic.

A user asks a logic database by submitting it a *goal*. Goals is also a logic formulas. A correct query answer provides values for the variables of the goal that make this query *logically follow* from the program. Hence, the semantics of query answering is based on the notion of *logical consequence* developed in mathematical logic.

Besides formulas specifying the database and queries, a logic database can also contain *integrity constraints*: logical conditions which the database must satisfy at any given moment. The semantics of integrity constraints is also based on mathematical logic.

Modern approaches to databases include relational and object-oriented databases. *Relational databases* will be considered in these notes in some detail. *Object-oriented databases* lack declarative query languages. It is expected that relational databases will be developed to include new paradigms, like *recursive queries*, support for *complex objects* and advanced *integrity constraints*. All these extensions will require a serious change in the database languages and design. After including some of the above mentioned features, future relational databases will have very little in common with the current relational database systems. An example of such an extension is *SQL-3* which represents a number of features without a clear foundational concept.

The *logic approach to databases* has a number of advantages for the current and future applications of databases. The logic approach is based on the formalism of mathematical logic developed as a language for foundations of mathematics. Mathematical logic tried to solve many problems similar to those arising in foundations of databases: how to formalize the notion of an application world (logical languages), how to express its properties (model theory) and how to reason about these properties (proof theory). Because of the consistent use of mathematical logic as the basis, the logic approach to databases can handle recursive definitions, integrity constraints, constraint databases, complex values, deduction, induction and abduction *in a unified framework*. This opens a perspective to a wide range of applications

where other database paradigms have limited use. To present the logic approach to databases is the main aim of this monograph.

1.2 Deductive databases

There is no standard definition of deductive databases. In a narrow sense, deductive databases are first-order logic databases consisting of sentences of a special form (usually, so-called normal clauses) and based on *deduction*. In the wide sense, every logic database can be called a deductive database. The notion of a deductive database appeared as a name for the technology combining logic programming with databases. For a long period, deductive databases denoted normal logic programs augmented with integrity constraints. Later, it was realized that such a narrow understanding may not be enough for database applications that may require handling complex objects, aggregate relations, incomplete or inconsistent information, abductive reasoning etc. To express the need for the extended technology, the term *logic databases* has found more use.

In this book, we shall usually deal with deductive databases, and use the term "logic databases" when we discuss the extensions of the deductive database technology. In this section we give an informal introduction to deductive databases by discussing their *constituents*.

Formulas of a deductive database describe *relations*. The formulas used in the description are usually called *clauses*. Clauses provide a description of the external world consisting of *objects*. In deductive databases, objects are represented by *terms* of first-order logic or by tuples of terms. The part of the external world described in a database is called the *application world*.

Clauses of a deductive database are usually divided in two parts. The first part is the *extensional database*. In the extensional part of the database relations are defined by the explicit enumeration of all objects satisfying these relations. On the implementation level, it means that tuples of any extensional relation can be stored explicitly. The *intentional database* defines relations in terms of other relations. The intensional relations are defined by so-called *rules* The intensional database relations are similar to *views* in relational databases. They may be stored in the database implicitly as rules or explicitly via the mechanism known as *materialized view*. We shall demonstrate all these notions by examples.

Extensionally defined relations

A natural way to define a relation is to explicitly mention all tuples belonging to this relation. For example a database storing information from a telephone book of a military base can be defined by explicit enumeration of all its entries of the form $\langle$`NAME`, `ADDRESS`, `PHONE_NUMBER`$\rangle$. In relational databases such relations are usually represented by a table:

```
⟨"Andy", "Colonel street", 1234567⟩;
...
⟨"John", "Fifth major street", 7243201⟩;
...
```

To describe this relation in the language of deductive databases we have to assign this relation a name, for example `entry` and represent this relation as a set of *facts*:

```
entry("Andy", "Colonel street", 1234567);
...
entry("John", "Fifth major street", 7243201);
...
```

In mathematical logic, such expressions are called *atomic formulas*, or simply *atoms*. A database clause that is an atom is called a *fact*. The *extensional database* defines relations by sets of facts, for example

```
rank("Andy",colonel);
rank("John",major);
...
higher_rank(major,colonel);
...
```

When writing a set of facts, we shall sometimes separate facts by a dot.

In this book, we shall often consider military examples. In particular, we introduce a military database in this chapter. In order to be powerful and omniscient (as a military database should be), our database should contain many more relations. Let us add the information about all human inhabitants and all dogs living in the division as extensionally defined relations[1]:

```
man(andy).
...
woman(helene).
...
dog(bobick).
...
```

[1] Since the author lived a considerable part of his life in the Soviet Union, the names of dogs in the database are taken from the Russian language. In all military examples the author uses either his scarce knowledge about the organization and regulations of the Soviet Army or his even more scarce knowledge of its enemy armies. The military department of Novosibirsk University back in 1978–1980 is responsible for any errors that can be presented in military examples.

Intensionally defined relations. Rules

Explicit enumeration of all tuples is a very primitive way of defining relations. Very often, we can describe a relation in terms of other relations. For example, the sentence "a higher-ranked officer is an officer having a higher rank" describes the relation "higher-ranked officer" between officers in terms of the relation "higher rank" between ranks. In deductive databases, such sentences are expressed by so-called *rules*. To describe the relation "higher-ranked" by rules, we have to rephrase the above sentence in the "if" form:

$$\begin{array}{c}\text{An officer } A \text{ is higher-ranked than an officer } B \ \textit{if} \\ \text{the rank of } A \text{ is higher than the rank of } B.\end{array} \tag{1.1}$$

Of course we could have defined this new relation also by explicit enumeration of all pairs of officers satisfying this sentence:

```
higher_ranked_officer("Andy","John").
higher_ranked_officer("Andy","Peter").
...
```

However such an explicit definition is dangerous. First, it may require a lot of memory to store all such pairs of officers explicitly. Second, the problem arises when the information about officers changes. Changes in the relation `rank` on which the relation `higher_ranked_officer` depends, require appropriate changes in the `higher_ranked_officer` relation. For example, if the officer `John` becomes a colonel, it would not be enough to only replace the fact `rank("John",major)` to `rank("John",colonel)`. To preserve consistency of the database, we shall have to change many facts about the `higher_ranked_officer` relation, if we do not want to face a total disorder in our military division.

A solution to these problems is to store the `higher_ranked_officer` relation implicitly, describing it by a rule. Definition (1.1) of the `higher_ranked_officer` relation in terms of the relations `rank` and `higher_rank` can be rephrased as follows:

$$\begin{array}{l}\textit{For each} \text{ pair of officers } A, B, \\ A \text{ is a higher-ranked officer than } B \ \textit{if} \\ \quad \text{the rank of } A \text{ is } RA \ \textit{and} \\ \quad \text{the rank of } B \text{ is } RB \ \textit{and} \\ \quad RA \text{ is a higher rank than } RB.\end{array} \tag{1.2}$$

In the language of deductive databases this can formally be written as a *clause* of the form

$$\begin{array}{l}\texttt{higher\_ranked\_officer}(A, B) \ \texttt{:-} \\ \quad \texttt{rank}(A, RA), \\ \quad \texttt{rank}(B, RB), \\ \quad \texttt{higher\_rank}(RA, RB).\end{array} \tag{1.3}$$

Such clauses are also called *rules*. In general, *intensional definitions* of relations use rules and facts. In the translation of natural language sentence (1.1) into its clause form (1.3) we had to make several important steps. First, we reformulated the definition of a higher-ranked officer in terms of an "if" expression (1.2) of a special form. Second, we used atoms like `rank`(A, RA) to describe relations among particular objects. Third, we had to introduce some intermediate objects RA and RB for expressing the relation between objects A and B.

Let us consider the anatomy of clause (1.3) in more detail. The part preceding the symbol `:-` is called the *head* of the clause. It contains the defined relation `higher_ranked_officer` with the *arguments* A, B. The symbol `:-` denotes the (reversed) logical implication. Informally, it should be read as "if". The part after the symbol `:-` is called the *body* of the clause. It consists of atoms separated by the symbol "," denoting logical *conjunction* which should be read as "and". The symbols A, B, RA, RB are *variables* ranging over arbitrary objects. Other possibilities for representing objects will be considered later. With this intuitive explanation, we can rewrite the clause less formally in the following way:

The relation `higher_ranked_officer` holds between objects A, B *if*
the relation `rank` holds between objects A, RA ***and***
the relation `rank` holds between objects B, RB ***and***
the relation `higher_rank` holds between objects RA, RB.

This statement can be generally understood in at least two ways, depending on whether we interpret A, B as *variables* ranging over *arbitrary* objects or as *concrete* objects. Concrete objects will be denoted by *constants*. To distinguish variables from constants, deductive databases have special syntax conventions that will be discussed later. Since the intended meaning of A, B, RA and RB is that of variables, it will be more careful to specify the informal meaning of clause (1.3) as

For all objects A, B, RA, RB,
the relation `higher_ranked_officer` holds between objects A, B *if*
the relation `rank` holds between objects A, RA ***and***
the relation `rank` holds between objects B, RB ***and***
the relation `higher_rank` holds between objects RA, RB.

As we shall see later, this statement is equivalent to

For all objects A, B,
the relation `higher_ranked_officer` holds between objects A, B *if*
there exist objects RA, RB such that
the relation `rank` holds between objects A, RA ***and***
the relation `rank` holds between objects B, RB ***and***
the relation `higher_rank` holds between objects RA, RB.

In addition to variables, intensional relations can be described using *constants* to represent particular objects. For example, if we want to represent the fact that president is ranked higher than any human, we can write

```
higher_ranked_officer(president, H) :-
  human(H),
```

where `president` is a constant denoting a particular object (president) and H is a variable ranging over arbitrary objects.

Disjunction

If we want now to define humans as either men or women, the language of conjunctions and implications is not enough. What we have to express is

For each object A,
A is a human *if*
  A is a man *or*
  A is a woman.

To describe relations like `human`, we have to be able to express the *disjunction* of conditions, separated by *or*. A definition of a relation described by disjunction of conditions is represented by a *set* of clauses. The `human` relation is formally described by two clauses

```
human(A) :-
  man(A).

human(A) :-
  woman(A).
```

In general, intensional relations are described by finite sets of rules or facts. Several rules or facts about a relation should be understood as an "or"-definition, or *disjunctive definition*. The informal description of the meaning of a set of clauses is less straightforward. For example, the definition of the `higher_ranked_officer` relation by two clauses

```
higher_ranked_officer(A, B) :-
  rank(A, RA),
  rank(B, RB),
  higher_rank(RA, RB).

higher_ranked_officer(president, B) :-
  human(B).
```

has the intuitive meaning

For all objects A, B,
A is a higher-ranked officer than B *if*
  there exist objects RA, RB such that
    RA is the rank of A *and*
    RB is the rank of B *and*
    RA is a higher rank than RB
  or
    A = president *and*
    B is a human.

Negation

There are many natural definitions of relations that cannot be easily expressed using only conjunctions and disjunctions. For example, if we want to represent the fact that president is ranked higher than any human but himself, we have to write something like

```
higher_ranked_officer(president, H) :-
  human(H),
  non_president(H).
```

Then, in defining the relation `non_president`, we have to include all objects but president in this relation. This is hardly realistic for any database describing many objects. Fortunately, logic databases often allow to use *negation* in the body of a clause. Then the above clause can be replaced by

```
higher_ranked_officer(president, H) :-
  human(H),
  not (H = president),
```

where `not` denotes negation. Another example of a specification using negation is

```
man(H) :-
  human(H),
  not woman(H),
```

expressing that a man is a human who is not a woman.

So far we had no formalism to express *negative information*, like the statement `not woman`(H) asserting that the object H is *not* a woman. In deductive databases, it is not allowed to define negative information explicitly. For example, handling clauses

```
not man(H) :-
  human(H),
  woman(H)
```

is beyond the standard deductive database technology. The main approach used in logic database is to never describe the negative information explicitly. Instead, a negative statement `not` S is considered true if the positive statement S cannot be established. This principle is known as *negation as failure* and is based on the so-called *closed world assumption* discussed below.

All the above mentioned features describe what is usually called *normal databases*. Very often, deductive databases in a narrow sense are identified with normal databases. Some restrictions on the clauses defined using conjunction and negation yield a notion of nonrecursive datalog programs equivalent in expressive power to the relational algebra used in the relational query language SQL.

Goals

Logic databases represent information about the application world in the form of logical sentences. The user communicates with the database by submitting it *goals*. A goal is a logical statement about relations described in the database. Usually, goals have the same structure as clause bodies. An example of a goal is `human(president)`. The meaning of a goal Q is "does Q logically follow from the database?". Thus, the semantics of query answering in logic databases is based on a notion of *logical consequence*. Often, the notion of a logical consequence used in the database theory is different from the notion of a logical consequence in standard mathematical logic. The semantics of logic databases studies the logical consequence relation between databases and goals and related questions, like the constraint satisfiability.

In logic programming, one often says "*query*" instead of a "goal". Some deductive database papers also use this terminology. We use the term "goal" throughout this book in order to avoid inconsistencies with the standard database terminology, where the term "query" has a different meaning. We shall however retain the terminology *query answering*, *query evaluation*, and *query processing* when we refer to the process of finding answers to goals.

In relational databases, a goal is usually a statement of the form "find all tuples $\langle a_1, \ldots, a_n \rangle$ such that ...". In logic databases, such goals are expressed as logical formulas $\varphi(x_1, \ldots, x_n)$ with variables $x_1, \ldots, x_n$. The meaning of such a goal is "find all tuples of objects $\langle t_1, \ldots, t_n \rangle$ such that $\varphi(t_1, \ldots, t_n)$ is a logical consequence of the database. Suppose that a database defines a relation `likes` between objects. An example of a goal with variables is `likes(president,`x`)`: find all objects x such that (the object denoted by) `president` likes x. A more complicated goal is

```
likes(x, y), not likes(y, x):
```

find all pairs $\langle x, y \rangle$ of objects such that x likes y and y does not like x.

1.3 Towards more expressive power

All the previous discussion of deductive databases is related to query languages not more powerful than standard relational databases. In this section we discuss features going beyond the current relational database technology.

Recursion

Logic databases have a facility to define relations by *recursive definitions*. The ability to use recursion increases the expressive power of databases far beyond the expressive power of relational databases and even beyond the expressive power of first-order logic. A *recursive clause* describes a relation in terms of itself. A typical example of a recursive definition useful in database applications is that of the *transitive closure* of a relation. For example, if we design a traffic database containing information about the traffic in a big city, we can recursively define the relation `connected(StartPoint,EndPoint)` expressing that `StartPoint` is connected to `EndPoint`, in terms of the relation `segment(Point1,Point2)` denoting short ways between two points:

```
route(StartPoint, EndPoint) :-
  segment(StartPoint, EndPoint).
route(StartPoint, EndPoint) :-
  segment(StartPoint, NextPoint),
  route(NextPoint, EndPoint).
```

Here the third argument of the relation `route` is the list representing all intermediate points on the route between `StartPoint` and `EndPoint`. This set of two clauses can be informally understood as

`StartPoint` and `EndPoint` are connected *if*
  there is a segment between `StartPoint` and `EndPoint`
or
  there is an intermediate point `NextPoint` such that
    there is a segment between `StartPoint` and `NextPoint` *and*
    `NextPoint` and `EndPoint` are connected

There are more complicated examples of recursive definitions used in logic databases. The presence of recursive definitions makes logic databases extremely powerful. For example, the transitive closure of a relation is not expressible in first-order logic, but it is easily expressible in deductive databases, even with no use of negation.

At the same time recursion creates many problems in understanding the meaning of logic databases and issues a challenge in implementing recursive query processing efficiently. As we shall see later in these notes, semantics of nonrecursive databases is well-defined and unambiguous. Most of semantical problems arise because of the unrestricted use of recursion, and most notably because of the combination of recursion with negation. Query answering in the absence of recursion is well-studied in relational databases. Query processing in logic databases describes methods of recursive query processing.

Integrity constraints

My note 1.1 There is a lot of confusion in terminology. You have to define precisely what is a database, database state etc. Change everything! □

Logic databases, as any other kind of databases, can be *updated*. But even the changing application world at any time moment satisfies some properties. For example, in a military base we often want every platoon to have commander at any time moment. If we design a military database describing the military base, we have to be sure that the description is consistent, in particular we have to be certain that in the database every platoon always has commander. Database updates represent changes in the application world, or sometimes changes in our knowledge about the application world. Arbitrary updates can change the database so that it does not reflect the application world properly. The property of a database to represent the application world correctly is called the *integrity* of the database. The restrictions which the database should satisfy at any given moment are called *integrity constraints*. It is the responsibility of a database engineer to ensure that the integrity of the database is preserved.

In logic databases, the integrity of a database is specified as the set of logical formulas expressing the desirable properties on the database. When we create a database, we can possibly ensure that it reflects the application world correctly. Later, the database can be updated to accommodate changes in the application world. The updates can be specified by users who do not know enough about the integrity of the database. Unrestricted updates may easily *violate the integrity* of the database.

Integrity constraint are usually specified by unrestricted first-order formulas. For example, the integrity constraint "each platoon has commander" can be specified as the first-order formula

$$\forall x(\texttt{platoon}(x) \supset \exists y \texttt{commander}(y, x))$$

(for any object x, if x is a platoon then there is an object y such that y is commander of x"). The integrity constraint "each platoon has at most one commander" can be expressed by the formula

$$\forall x(\texttt{platoon}(x) \supset \forall y \forall z(\texttt{commander}(y,x) \wedge \texttt{commander}(z,x) \supset y = z)).$$

In addition to integrity constraints imposed on any database state, there may even be constraints imposed on the updates. An example is "on the change of the rank of an officer, the new rank must be higher than the old rank". There are more difficult examples of integrity constraints, including so-called aggregate relations.

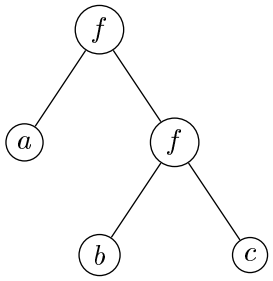
The possibility to handle integrity constraints efficiently is a very important property of databases. It is of paramount importance to check the integrity in real time. In general, integrity checking can be very slow. The possibility to check integrity constraints in real time depends on the application. For example, there are applications of databases where update requests happen nearly as often as goals. For such *update-intensive applications* support of integrity constraints can be very expensive.

There are several approaches to handling integrity constraints in logic databases. *Integrity checking* tries to identify whether the integrity constraints are satisfied on updates. In *declarative updates*, or *view updates* the user specifies the desirable properties of the database after the update and the database system itself tries to find concrete updates forcing the properties. Declarative updates have much in common with a special strategy of reasoning called *abductive reasoning*. Another approach to handling integrity constraints is the design of *update specification languages*. Some of these approaches to handling integrity will be considered in these notes.

Complex values

Traditional relational databases keep information in the form of relations, described as a set of tuples. Tuples consist of simple, unstructured values, like strings or numbers, sometimes called *atomic values*. In all examples of clauses considered above, relations were also defined over atomic values.

Logic programs may handle more complex values, called terms. An example term is the expression $f(a, f(b, c))$. Terms may represent quite complex values known as *trees*. For example, the term $f(a, f(b, c))$ represents the tree



Suppose that we want to modify the traffic database example to also define routes between two points. A suitable representation of a route coming through points $a_1, \ldots, a_n$ would be the list $[a_1, \ldots, a_n]$. To represents lists by terms, we need an operation of adding an element a to a list l. Such operations will be denoted by $[a|l]$. The definition is as follows.

```
route(StartPoint, EndPoint, [StartPoint, EndPoint]) :-
  segment(StartPoint, EndPoint).
route(StartPoint, EndPoint, [StartPoint|Route]) :-
  segment(StartPoint, NextPoint),
  route(NextPoint, EndPoint, Route).
```

This set of two clauses can be informally understood as

R is a route between `StartPoint` and `EndPoint` *if*
  there is a segment between `StartPoint` and `EndPoint`
  `textitand`
  R = `[StartPoint, EndPoint]`
or
  there is an intermediate point `NextPoint` such that
    there is a segment between `StartPoint` and `NextPoint` ***and***
    there is a route `Route` between `NextPoint` and `EndPoint` ***and***
    R is obtained from `Route` by adding `StartPoint` as the first element

The ways of representing objects in mathematical logic are not adequate for some database applications. Modern databases can handle data varying from simple records to objects like images. In order to be competitive on the market, logic databases must be able to represent *complex values* or *complex objects*. The inability to manipulate complex objects is one of the major bottlenecks of relational database languages, for example *SQL-2*. The ability to handle complex objects is the main reason of success of object-oriented databases. Some existing prototypes of logic databases usually provide a separate language for describing complex objects.

Typical examples of complex values that arise in database applications, are trees, *records*, *lists*, *sets*, *multisets* and some other.

Data types representing values that can keep collections of other values are called *collection types* are *bulk types*. For example, we can easily imagine the type of products described by the declaration

```
type product =
  {Id of string;
   price of integer;
   components of product bag;
  }.
```

(Bags, or finite multisets, are similar to sets with the exception that they may contain several copies of the same element). Here the class of objects `project` is given by its structure. Description of complex objects by their structure is not the only way to define such objects. In *object-oriented databases* objects can also contain so-called *methods*. In *constraint databases* objects are described by their properties.

There are other kinds of complex objects: unstructured and semi-structured data. Examples of unstructured data are *multimedia* objects, like *sounds* or *images*. A typical semistructured object is a *Web page*, or an *HTML file*. It may include text, references to images, references to other pages, formatting commands, sectioning commands, color-changing commands and many other features.

Many applications require to handle temporal and/or spatial information. Examples of complex objects useful in such applications are time intervals and three-dimensional figures. In order to deal with temporal and spatial information, we have to be able to operate with concept like "before", "after", "between", "close" and alike. Such concepts may be described using *constraints*: (should not be mixed with integrity constraints) logical conditions which an object (not the database), or a group of objects must satisfy. For example, if we represent temporal information using the time intervals with built-in relations for intervals, an example of a description of the relation "within" between two events

```
within(Event1, Event2) :-
  happens(Event1, Interval1),
  happens(Event2, Interval2),
  Interval1 ⊆ Interval2.
```

Here `Interval1` and `Interval2` are variables ranging over a class of interval, and `happens` and $\subseteq$ are methods of this class. The relation $\subseteq$ expresses that one interval is a subset of another interval. In order to be able to deal with such clauses, databases should have facilities for handling constraints described by built-in relations, like $\subseteq$.

It is an nontrivial problem how to integrate complex objects in the semantics and query answering methods of logic databases. There are several ways of doing this. A relatively simple way is to describe types of objects by their structure, using several *type constructors*, like tuples, arrays, disjoint unions, sets or bags. A more complex problem is to accommodate *recursive type declarations*. The integration of such types into logical query answering requires changing several fundamental notions, especially *unification*. Query answering can become more complex. For example, unification that is a relatively simple operation for trees is NP-complete for finite sets.

1.4 Example: the military database

In order to illustrate the concepts described above, we consider an example of a small military database including many of the above mentioned features. This database will be used in other examples of these notes and referred to as the military database.

My note 1.2 In the military database, there should be an example of built-in relations, for example $\leq$. An example is a promising officer: an officer whose age is relatively young for his rank. □

Example 1.1 In this example, variables start with upper-case letters, while constants and predicate symbols start with lower-case letters. Comments follow the sign %.

% EXTENSIONAL DATABASE

```
% Relation next_rank describes the hierarchy of ranks
  next_rank(lieutenant, captain).
  next_rank(captain, major).
  next_rank(major, colonel).

% the list of officers
  officer(jessep).
  officer(vitali).
  officer(john).
  officer(andy).

% the list of ranks of officers
  rank(jessep,colonel).
  rank(vitali,captain).
  rank(john,major).
  rank(andy,lieutenant).

% the list of dogs living in the base with their owners
  dog(jessep,polkan).
  dog(vitali,bobick).
  dog(john,sharik).
  dog(andy,barbos).

% some dogs bite other dogs, some dogs even bite officers
  bites(polkan,bobick).
  bites(polkan,sharik).
  bites(polkan,barbos).
```

```
  bites(polkan,john).
  bites(barbos,sharik).
  bites(barbos,polkan).

% INTENSIONAL DATABASE

  % Relation higher_rank describes the order on ranks
  % It is defined as the transitive closure of next_rank
  higher_rank(LowerRank, HigherRank) :-
    next_rank(LowerRank, HigherRank).

  higher_rank(LowerRank, HigherRank) :-
    next_rank(LowerRank, MiddleRank),
    higher_rank(MiddleRank, HigherRank).

  % A higher-ranked officer is an officer who has a higher rank
  higher_ranked_officer(Higher, Lower) :-
    rank(Higher, HigherRank),
    rank(Lower, LowerRank),
    higher_rank(LowerRank, HigherRank).

  % likes is a relation among officers.
  % (i) every officer likes himself
  likes(Officer, Officer) :-
    officer(Officer).
  % (ii) every officer likes all higher-ranked officers
  likes(Lower, Higher) :-
    higher_ranked_officer(Higher, Lower).
  % (iii) an officer likes a lower-ranked officer when
  % dogs of the lower-ranked officer
  % do not bite dogs of the officer
  likes(Higher, Lower) :-
    higher_ranked_officer(Higher, Lower),
    not bites_some_dog(Lower, Higher).

  % an auxiliary relation:  some dog of Officer1 bites
  % some dog of Officer2
  bites_some_dog(Officer1, Officer2) :-
    dog(Officer1, Dog1),
    dog(Officer2, Dog2),
    bites(Dog1, Dog2).
```

% INTEGRITY CONSTRAINTS

% *every officer has a rank*
$\forall x(\texttt{officer}(x) \supset \exists y \texttt{rank}(x, y))$.

% *only colonels may own more than one dog*
$\forall x \forall z_1 \forall z_2(\texttt{officer}(x) \wedge \texttt{dog}(x, z_1) \wedge \texttt{dog}(x, z_2) \supset z_1 = z_2 \vee \texttt{rank}(x, \texttt{colonel}))$.

% *a dog of an officer cannot bite a higher-ranked officer*
$\forall x \forall y \forall z(\texttt{higher\_ranked\_officer}(x, y) \wedge \texttt{dog}(y, z) \supset \neg \texttt{bites}(z, x))$.

1.5 Other developments

Here we discuss some other notions developed in logic databases and related areas.

Disjunctive information

It is quite possible that our knowledge of the application world is indefinite. For example, we may know that Vitali is either captain or major, but not know his rank precisely. Then we have to be able to make conclusions on the basis of such indefinite information. A simplest kind of indefinite information is *disjunctive information* that represents a choice among a finite number of alternatives. *Disjunctive databases* study disjunctive information and methods of query answering in presence of disjunctive information. Disjunctive information can be given by simple *disjunctive facts*, like

```
rank(vitali,captain) ∨ rank(vitali,major)
```

or by more complicated *disjunctive clauses*, for example

```
rank(x,captain) ∨ rank(x,major) :- officer(x)
```

(every officer is either captain or major).

In general, handling disjunctive information is less efficient compared to definite information. Query answering for disjunctive databases uses more complex query answering procedures. Disjunctive databases also have more sophisticated semantics.

Since the same information can usually be represented in several different ways, disjunctive clauses may often be avoided. For most practical applications, one does not need disjunctive databases. Disjunctive databases are useful for several kinds of applications, for example diagnosis.

Concept languages

Objects (and classes of objects) may also be described using *concept languages*. Below we give an example of a description of the concept of a good officer in a concept language. It defines good officers as officers who are married colonels, having at least three children all of whom pay taxes:

```
defconcept good_officer as
  officer and
  married and
  colonel and
  atleast 3 child and
  all child taxpayer.
```

Here `officer`, `married`, `colonel` and `taxpayer` are other concepts and `child` is a so-called *role*, i.e. a relation between pairs of objects.

1.6 Some database problems

In this section we consider some typical problems that require the above mentioned features, like recursion or complex objects.

The anti-trust control problem.

The bill of materials problem.

The anti-trust control and the bill of materials problems have been introduced in (Gozzi & Lugli 1987).

1.7 The logic database system $\mathcal{LAB}$

My note 1.3 Here should be a brief discussion of my database. □

1.8 Applications of logic databases

My note 1.4 Here should be a brief discussion of applications. Continue. This section is connected with Chapter 25 on the systems. □

1.9 Structure of the book

My note 1.5 This section should be changed completely! □

These notes are structured as follows. In Chapters 2 and 3 we introduce the formal notation used in the notes and provide the necessary background in mathematical logic. Our notes are self-contained, i.e. we provide all definitions and